Automatic Impostor Placement for Guaranteed Frame Rates and Low Memory Requirements

Stefan Jeschke [†, *] Michael Wimmer[†] Heidrun Schumann[*] Werner Purgathofer[†]

[†]Vienna University of Technology [*]University of Rostock

Abstract

Impostors are image-based primitives commonly used to replace complex geometry in order to reduce the rendering time needed for displaying complex scenes. However, a big problem is the huge amount of memory required for impostors. This paper presents an algorithm that automatically places impostors into a scene so that a desired frame rate *and* image quality is always met, while at the same time not requiring enormous amounts of impostor memory. The low memory requirements are provided by a new placement method and through the simultaneous use of other acceleration techniques like visibility culling and geometric levels of detail.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing Algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality

1 Introduction

While the performance of graphics hardware keeps increasing at an enormous pace, the geometric and visual complexity of virtual environments is growing even faster. In many applications, such as flight or driving simulators, virtual reality, or games, it is desirable to *guarantee* interactive or even real-time frame rates.

To achieve a reduction of the time required to render a complex scene, three major strategies have been proposed. *Visibility culling* techniques determine those parts of a scene that are potentially visible. However, the geometry that is actually visible may still overwhelm the capabilities of the graphics hardware. Second, *geometric simplification* techniques substitute complex objects with coarser geometric representations for more distant views without loss in image quality. Unfortunately, geometric simplification is still not useful for arbitrary models. Third, *image-based rendering* techniques can be used where geometric simplification fails. Arbitrary scene parts can be represented by so-called *impostors* made up from simple textured quads, polygon meshes, LDIs, or point clouds. Impostors provide a correct representation only when displayed for a bounded viewing region called *view cell*.

However, impostor techniques suffer from a number of drawbacks. Dynamic impostors (which are generated on the fly) place a high burden on the runtime system, whereas techniques relying on precalculation suffer from one or more of the following problems:

- image artifacts (e.g., image gaps through disocclusions),
- very high memory consumption, and
- long preprocessing times.

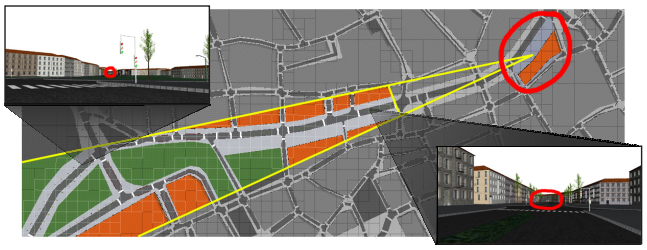

Figure 1: Example of automatic impostor placement (with the impostor in the circle, shown together with its shaft of validity.

In this paper, we present an *automatic impostor placement algorithm* that addresses all of these problems. Given a static scene and a view space, an *impostor placement* defines for every view position which objects should be displayed as impostors, so that both a minimum frame rate and a minimum impostor image quality are guaranteed. The optimization problem is to minimize the amount of memory for all impostors.

Our algorithm first identifies those views that cannot be rendered sufficiently fast. Second, a large set of *impostor candidates* is generated. An impostor candidate is a scene part together with a view cell from where the impostor meets the desired image quality. Finally, a greedy optimization algorithm selects candidates to serve as impostors based on the rendering acceleration they provide and their required memory. This is done until all views can be rendered with the desired frame rate, providing a (soft) frame rate guarantee.

Three main contributions are provided by the new approach in comparison to previous work: First, our algorithm avoids the generation of similar impostors for adjacent viewing regions, keeping the memory requirements so low that they often fit completely into graphics memory. This is achieved because instead of using a fixed set of view cells and a separate set of impostors for each cell, we allow arbitrary combinations between impostors and viewing regions. Second, we show how additional rendering acceleration techniques like visibility culling and geometric simplification can be used together with impostors to make best use of all approaches in a single framework. Third, the use of a generic rendering time estimation allows the algorithm to adapt to the actual rendering bottleneck. In summary, our approach makes impostors accessible for a much wider range of applications.

2 Overview

2.1 Formal Problem Definition

The input model is assumed to consist of a set O of discrete objects o. The space of all possible viewpoints and view directions is called

[*]{stefan.jeschke|schumann}@informatik.uni-rostock.de

[†]{wimmer|wp}@cg.tuwien.ac.at

view space V. An element $v \in V$ is called a *view* and consists of a 3D *viewpoint* v_{3D} and a 2D *view direction* v_{2D} (encoded as azimuth and elevation angles), so that $V = V_{3D} \times V_{2D}$. The field of view and image resolution is assumed to be fixed for all views.

The *problem view space* $V_p \subseteq V$ is one of the fundamental concepts of our approach. It encodes for which so-called *problem views* $v_p \in V_p$ the rendering time exceeds a desired maximum rendering time t_{max}, i.e., where a "problem" occurs.

An *impostor* i can be created given three input parameters: a *view cell* $VC \subseteq V_{3D}$, an *object cluster* $OC \subseteq O$, and an *image quality criterion* IQ. This means for any view within VC, rendering i instead of OC will meet the image quality criterion IQ. The image quality criterion IQ specifies that the impostor resolution meets at least a certain output image resolution (to avoid "blocky" artifacts), and that no visible "image cracks" caused by disocclusions occur. Methods for ensuring this criterion exist for most impostor techniques, like textured quads [Schaufler and Stürzlinger 1996; Shade et al. 1996; Schaufler 1998; Jeschke et al. 2002], textured depth meshes [Decoret et al. 1999; Jeschke and Wimmer 2002; Wilson and Manocha 2003] and point clouds [Shade et al. 1998; Wimmer et al. 2001].

Each impostor is associated with a set $V_i \subseteq V_p$ of problem views with 3D view positions inside VC. Note that the impostor is *created* for a 3D view cell VC but *used* for a 5D view space (in short, i *serves* V_i). This distinction is made because an impostor might increase the rendering time for some views in VC, which happens when most of the objects it represents are actually not visible for a particular view direction. Figure 2 illustrates this definition of an impostor.

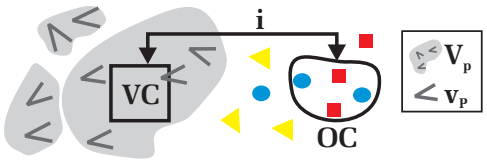

Figure 2: An impostor i is *generated* for a 3D view cell VC and *used* for a 5D problem view space subset of V_p .

An *impostor placement* for a view space V and an object set O is a set I of impostors, each representing a set of objects OC_i in the associated problem views V_i. The *impostor placement problem* can be cast as an optimization problem for finding an impostor placement I that

- satisfies the constraint that the rendering time t_v for each view $v \in V$ does not exceed the user-defined maximum frame time t_{max}. Such a placement is called *valid*:

$$\forall v \in V : t_v \leq t_{max} \tag{1}$$

  The time t_v is obtained assuming that the original objects are replaced by the impostors. If constraint (1) is met for a problem view, we say it is *solved*.

- minimizes the memory required for all impostors. Such a placement is called *optimal*:

$$\sum_{i \in I} m_i \rightarrow \min \tag{2}$$

In order to solve this problem, an impostor placement algorithm has to decide on the object clusters for which impostors should be generated, and for every impostor on the size and position of the view cells, as well as the problem views it should serve. Note that minimizing impostor memory typically also reduces the preprocessing time needed to generate the impostors.

2.2 Observations

The problem in finding a good impostor placement mainly lies in the huge number of valid impostor placements: even for a single impostor, the view cell may have an arbitrary size and position in the non-discrete view space. The goal is to limit the search space while still providing good placements. Three essential observations guide this process:

Observation 2.1. *If multiple objects are adjacent in object space, a common impostor for all objects is likely to require less memory than separate impostors for each object.*

This observation is especially true for distant objects that share many pixels on screen. Consequently, with increasing distance, larger clusters of objects should be represented by single impostors. This observation is addressed, for example, by environment map impostors [Aliaga et al. 1998; Wilson et al. 2001; Jeschke et al. 2002]. A beneficial side effect is that this also reduces the number of rendering calls, and thus improves rendering acceleration on current graphics cards.

Observation 2.2. *If the appearance for a scene part hardly changes when seen from a given viewing region, a single impostor for the whole region is likely to require less impostor memory than splitting the region into smaller view cells and generating an impostor for each such cell.*

Note that previous approaches (refer to Section 7) generate impostors for a fixed set of view cells, and separately for each cell. Because no distinction is made between nearby (apparently changing) and distant (hardly changing) scene parts, many similar impostors are generated for distant scene parts, which constitutes a waste of memory.

Observation 2.3. *If additional rendering acceleration techniques like visibility culling and geometric levels of detail are applied before using impostors, this may dramatically decrease the required amount of impostor memory.*

This observation was only addressed by portal impostors in architectural models [Aliaga and Lastra 1997; Popescu et al. 1998] as well as in urban models [Sillion et al. 1997].

The main contribution of the approach presented in this paper is that it successfully addresses all three issues mentioned above, thus providing a general method for placing impostors in various scenes with moderate memory requirements, something which was not possible until now.

2.3 Algorithm Outline

The impostor placement algorithm consists of four main steps:

Object set hierarchy generation: The object set is subdivided hierarchically, clustering close objects first. This will address observation 2.1. Possible clustering techniques include bounding box hierarchies, octrees and kd-trees. A finer subdivision may produce a better impostor placement, but also increases preprocessing time.

Problem view space approximation: For approximating V_p, the 3D positional view space V_{3D} as well as the 2D view direction

space V_{2D} are subdivided hierarchically up to a user-defined accuracy. The result is a set of *conservative problem views* (in short, *CPVs*). These 5D subsets of V are conservative in the sense that they include one or multiple problem views (Section 3).

Impostor candidate generation: A set of view cells is generated for every node of the object hierarchy, defining a set of *impostor candidates* for each node (Section 4).

Optimization: Finally, an optimization algorithm selects an impostor candidate subset so that constraint (1) is met and the optimization criterion (2) is approximated (Section 5).

In the runtime system, the current CPV (if applicable) is looked up, and all impostors associated with this CPV are rendered instead of the original scene parts.

3 Problem View Space Approximation

In order to find which views cannot be rendered sufficiently fast, the 5D view space V is approximated using a hierarchical subdivision. The approximation will be conservative in the sense that it will be a superset of V_p. Subdivision is done first along the axes of the 3D view space V_{3D}. Then, for each resulting node, the view direction space V_{2D} is subdivided. The result of the 2D subdivision is used as a termination criterion for the 3D subdivision. This means that a 3D region is only subdivided further if its associated 2D subdivision includes a problem view, which allows fast removal of areas where no problem views exist. Both subdivisions proceed to a user-defined minimal size.

3.1 3D View Space

Possible subdivision techniques include octrees, BSP-trees or kD-trees. The minimum region size for the subdivision must be chosen with care: smaller regions lead to better problem view space approximations but the preprocessing time is also increased.

If the rendering system provides from-region visibility culling and/or geometric simplification, these techniques can be performed for each node during the 3D view space subdivision in order to address observation 2.3. Only the visible objects at the correct level of detail are then passed on to the 2D view direction subdivision.

3.2 2D View Direction Space

Given a node from the 3D view space hierarchy together with the (visible and simplified) objects, the question is how to get the rendering times for all possible views within that region. This can be answered by extending the concept of *enclosing frusta* [Aliaga and Lastra 1999]. Figure 3 (left) illustrates this concept for a single view direction. The enclosing frustum for a 3D region contains all objects visible from viewpoints in that region with the same view direction. A rendering time estimation [Wimmer and Wonka 2003] applied to the enclosing frustum is then a conservative estimation for the rendering time of every *enclosed view frustum*. This concept is extended to a range of view directions, which we call a *view direction interval* (in short, *VDI*). The enclosing frustum of a VDI is chosen so that it encloses all views with a view direction within its range (Figure 3 (right)).

The view direction space subdivision starts from a 360° VDI. Figure 4 illustrates a 90° VDI subdivision together with the resulting enclosing frusta for the 1D case.

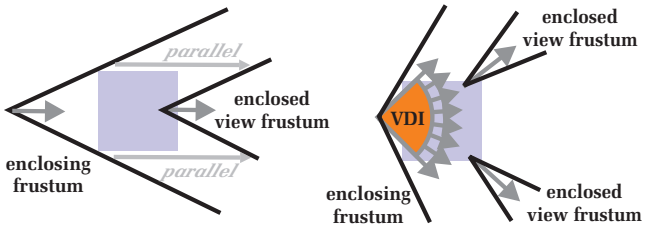

Figure 3: Enclosing frustum for a single view direction (left) and for a VDI (right), both for a 3D view region (light blue area).

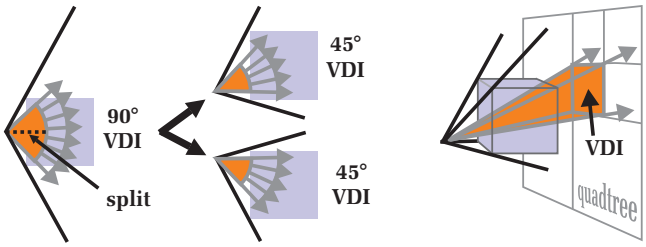

Figure 4: A 90° VDI (left) is subdivided in two 45° VDIs (middle). Right: 2D VDI subdivision using a quadtree.

For the general 2D case, we start with an initial view direction subdivision into six rectangular regions along every axis of the world coordinate system, thus forming a cube. View directions are then separately subdivided for each side of this cube (using a quadtree) in order to obtain a reasonably uniform subdivision. This is shown in Figure 4 (right) for one cube side.

In each subdivision step, the rendering time of every view within a VDI is estimated by the rendering time estimation t_{enc} for its enclosing frustum. If t_{enc} exceeds t_{max}, the VDI possibly contains a problem view and is further subdivided. This is done recursively until either t_{enc} is lower or equal to t_{max}, or the user-defined minimum VDI size is reached. In the latter case, the VDI is assumed to contain at least one problem view. If the 3D region has also reached its minimum size (see Section 3.1), the VDI together with the 3D region is stored as a CPV, representing all problem views starting in the associated 3D region with a view direction within the VDI.

Because the 3D and the 2D view space subdivision always terminate at the same level for problem views, the result of the problem view space approximation is a set of equally sized CPVs.

4 Impostor Candidate Generation

In order to "solve" the conservative problem views generated in the previous step, impostor candidates have to be created so that the optimization algorithm can make a good placement. Every candidate is specified by an object cluster $OC \subseteq O$ and a view cell $VC \subseteq V_{3D}$, so that it serves each CPV with a 3D region enclosed by VC. The main problem to be solved is the huge number of possible candidates. Therefore, observations 2.1 and 2.2 are used for a selective candidate generation. First, the nodes of the object hierarchy are used to define *OCs* for the candidates. This allows the optimization process to address observation 2.1, because larger nodes can be selected with increasing distance to the view cell. Second, for every OC, a set of view cells has to be found so that observation 2.2 can be met by the optimization. This means that with increasing distance between OC and VC, candidates with a larger view cell size

must be provided.

Many previous impostor approaches rely on rectangular view cells in order to fulfill the image quality criterion [Aliaga et al. 1999; Wilson et al. 2000; Darsa et al. 1997; Jeschke et al. 2002]. For these techniques, the 3D view space hierarchy from the problem view approximation (see Section 3.1) can be used directly, so that every combination of a node of the object and the 3D view space hierarchy defines a candidate.

Another view cell shape that is often used is a *shaft* implicitly [Jakulin 2000; Aubel et al. 1999; Schaufler and Stürzlinger 1996; Shade et al. 1996]. A shaft allows viewing an object from a limited angular range (defined by an *apex angle*) and from a minimum distance. Figure 5 shows an example for a shaft. For each object hierarchy node, a set of shafts in different directions and with different minimum view distances is generated. The advantage of shafts compared to rectangular view cells is that they perfectly address observation 2.2 because the view cell extent grows with increasing distance to the object. In Section 6 we will show that shafts provide better results compared to rectangular view cells. Note also that view cells for *view-independent* impostors (e.g. billboard clouds [Décoret et al. 2003]) are shafts with a 360° apex angle.

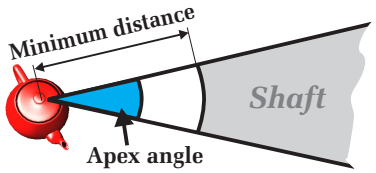

Figure 5: A shaft with apex in the object center.

Candidates are not considered if they serve no CPV within their view cell, or the combination of *VC* and *OC* allows no impostor generation (e.g., if *OC* intersects *VC*). This greatly reduces the overall number of candidates, so that only the most promising remain.

5 Optimization

The optimization algorithm calculates an impostor placement from the set of candidates *IC* generated in Section 4 by applying the following steps:

- select a good subset $I \subseteq IC$ to be generated and used as impostors,
- associate with every CPV the set of impostors that serve it (used during runtime for selecting the impostors to display).

To find an optimal solution, all possible subsets of *IC* would have to be tested, which is prohibitive even though *IC* is already of moderate size compared to the original problem. Instead, we adopt a greedy approach: at every choice the candidate with the best ratio between the rendering acceleration it provides for the CPVs in relation to its memory cost is selected.

5.1 Rendering Acceleration

Figure 6 shows for a single viewpoint how the exact *rendering acceleration function* of an impostor i maps to the approximated version we will present below. The exact *rendering acceleration* $\Delta t^i_{V_p}$ of an impostor i is defined as the integral of the rendering acceleration $\Delta t^i_{v_p}$ for every served view v_p:

$$\Delta t^i_{V_p} = \int_{v_p \in V_p} \Delta t^i_{v_p} \, dv, \tag{3}$$

for some measure dv.

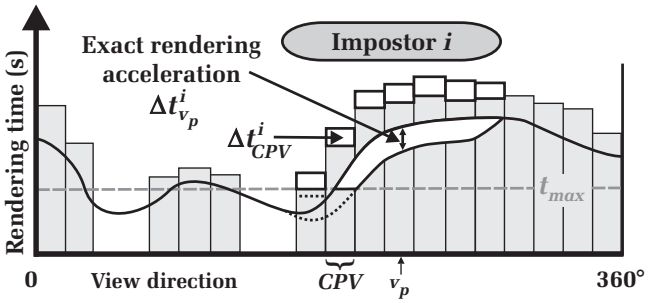

Figure 6: The rendering time for a view point for every view direction and the respective CPVs.

The approximate rendering acceleration Δt^i_{CPV} of an impostor candidate i for a CPV is defined as

$$\Delta t^i_{CPV} = \max(0, t^o_{CPV} - \max(t^i_{CPV}, t_{max})), \tag{4}$$

where t^o_{CPV} is the original rendering time of CPV and t^i_{CPV} is the rendering time of CPV where impostor i is rendered instead of o. The two maximum-terms in this definition reflect the fact that

- a candidate that takes longer to render than the original geometry should never be selected for p, and
- any reduction of the rendering time to less than t_{max} is useless (note how the rendering acceleration is clipped to t_{max} in Figure 6), so candidates with lower rendering acceleration, but also lower memory costs, should be preferred in this case.

In practice, t^i_{CPV} can be approximated by $t^o_{CPV} - (t_o - t_i)$, where t_i and t_o are the times needed to render the impostor and the original object, both obtained by using a rendering time estimation function [Wimmer and Wonka 2003]. Note that Δt^i_{CPV} varies over different CPVs for the same impostor i due to different level-of-detail selections, size on screen, visibility culling etc. Furthermore, Δt^i_{CPV} is defined to be 0 for any CPV with a 3D region that is not enclosed by the view cell of i.

5.2 Candidate ranking

In order to select the "best" candidate in every greedy choice, each candidate i is ranked according to its *score* s_i. This score corresponds to the ratio of the overall rendering acceleration Δt_i obtained in all CPVs, and its required memory m_i:

$$s_i = \frac{\Delta t_i}{m_i} \tag{5}$$

Since all CPVs have the same size, Δt_i is defined by the sum:

$$\Delta t_i = \sum_{CPV \in CPVs} \Delta t^i_{CPV} \tag{6}$$

This sum can be calculated efficiently by traversing the 3D problem view space hierarchy and exploiting the fact that Δt^i_{CPV} can only be non-zero if the view cell for i encloses the 3D region of CPV.

Since the impostor for the candidate to be ranked has not actually been built yet, neither its exact memory requirements m_i nor its rendering time t_i is known exactly. Both have to be estimated based on the object cluster, the view cell, the CPV and the underlying impostor technique. However, because this estimation is only used for candidate ranking, the accuracy is not crucial.

5.3 Greedy Choices

After all impostor candidates i have been ranked with a score s_i, the candidate with the highest score is chosen as an impostor. Such a choice entails the following steps:

- The impostor is generated. This allows a more accurate estimation of Δt^i_{CPV} based on the actual impostor geometry.
- For every CPV served (i.e., for which Δt^i_{CPV} is non-zero), add i to the set of impostors to display. The impostor can now be treated as belonging to the input scene, so that Δt^i_{CPV} can be subtracted from the rendering time t_{CPV} for those CPVs.
- Since a new value t_{CPV} is now available for all CPVs served by i, the scores s_i have to be recalculated for *all* impostor candidates that serve the affected CPVs. This operation is accelerated using a lazy recalculation scheme (Section 5.4).

The optimization algorithm proceeds by selecting the next candidate. This is repeated until no candidate is left.

5.4 Lazy Recalculation

Recalculating the scores s_i after every step of the greedy optimization might be a very costly operation. Fortunately, since no score s_i can increase for any remaining candidate (because Δt^i_{CPV} never increases if an impostor is added to a CPV), a *lazy recalculation* scheme can be applied: after every greedy choice, the candidates with the highest score are recalculated and re-ranked according to their new score. This is repeated until a candidate remains the best choice even after being re-ranked, in which case it is accepted as the next greedy choice. Candidates with a new score of 0 can be deleted, because all CPVs they serve have already been solved.

Lazy recalculation greatly reduces the number of operations needed for the optimization algorithm. Furthermore, for any candidate, only its score, view cell geometry and a link to its object hierarchy node have to be stored, as all other required information can be extracted on demand in reasonable time.

5.5 Overlapping Impostors

Some impostor candidates partially represent the same geometry for some common CPVs. However, displaying multiple impostors for the same geometry in any view is not desired, because image quality problems (z-fighting) might occur. Fortunately, this problem can easily be avoided due to the object set hierarchy used for candidate generation. The hierarchy implies that if two object clusters OC_1 and OC_2 overlap, either OC_1 and OC_2 are identical, or one encloses the other, i.e., $OC_1 \subseteq OC_2$ or $OC_2 \subseteq OC_1$. Note that this consideration constitutes no special case for the algorithm, because already created impostors are treated just as if they *are* the objects they represent (see Section 5.3).

During the course of greedy optimization, an impostor may be removed from all CPVs it is associated with. For that case, the impostor can be deleted and its memory regained.

6 Implementation and Results

6.1 Test Models

The automatic impostor placement algorithm was tested on the freely available model of the city of Vienna (`www.cg.tuwien.ac.at/research/vr/urbanmodels/`). We enhanced it with numerous street objects, so that the final model consists of 5287 objects with 10.4 million polygons in total. The rendering bottleneck for this model in our test machine was identified to be the number of *rendering calls* (a CPU bottleneck [Wimmer and Wonka 2003]), so the impostor placement algorithm was tuned to reduce this number for every output view. Because collapsing several textured objects is still not efficiently possible using geometric levels of detail, impostors are actually the only way for accelerating the rendering process for this type of scene without loss of image quality. Also note that our experiments concentrate on mid-range scenes, where the whole impostor database fits into graphics memory. This means that no restrictions on user movement speed are necessary for texture fetching tasks, which is important for instance in computer games. However, the impostor placement algorithm is not restricted to such cases.

We ran additional tests on the UNC Power Plant model (`www.cs.unc.edu/~geom/Powerplant/`), consisting of 12.7 million polygons. Details about these tests are discussed in Section 6.4.

6.2 Test Setup

The test machine was a PC with an Intel Pentium 4 3.2 GHz and 1 GB memory. The graphics board was an NVIDIA GeForce Quadro FX 3000 with 256 MB of memory. The API was the OpenGL graphics API under the Windows XP operating system.

The object set hierarchy for the Vienna model is a bounding volume hierarchy with 9 levels. For all tests, we selected a view space of 1500x1000 m, which covers the whole city. For the problem view space approximation, the 5D view space subdivision was based on a regular binary space partition for the 3D view space and a view direction space subdivision as presented in Section 3.2. The 3D part of the cubic CPVs had a side length of 23m, and the view direction space was subdivided to intervals of 11.25°. Visibility culling was done for the model using a conservative from-region visibility algorithm [Wonka et al. 2000].

For every node of the object hierarchy, a number of shaft-shaped view cells form the impostor candidates. Shaft apex angles of 11.25°, 22.5°, 45° and 90° were used, and shaft directions in steps of 11.25°. The minimum allowed view distances for every shaft are from 1 to 2^{10} times the object size, doubled in each step. This setup has been found to provide a good tradeoff between a sufficiently high number of candidates and reasonable preprocessing time.

We have chosen *layered impostors* [Jeschke et al. 2002], which represent objects with alpha-textured polygon layers that are spaced so as to eliminate disocclusion artifacts, as is required for our image quality criterion IQ. The rendering time estimation for an impostor is based on its number of screen pixels, since layered impostors have been found to be fill-rate limited [Wimmer and Wonka 2003]. Impostors were generated for an output image resolution of 512x512 pixels and 45° field of view, while the image quality criterion was set to a maximum error of one pixel.

6.3 Test Results

In order to show how the parameters of the algorithm influence its behavior, we first ran a "reference test" with the parameters described in the previous section and a target frame time of 16ms. Afterwards, we successively changed various parameters. Table 1 shows the resulting number of CPVs, impostor candidates and final impostors, and most importantly, the resulting impostor memory. The corresponding preprocessing times for the individual steps of the algorithm are summarized in Figure 7.

Test	Parameter	CPVs	Cands.	# Imps.	MB
1	Refer. test (16ms)	9078	314659	6650	14.1
2	$t_{max} = 30ms$	2270	154696	1504	1.01
3	$t_{max} = 25ms$	3562	204110	2502	2.3
4	$t_{max} = 20ms$	5846	260891	4357	5.7
5	$t_{max} = 15ms$	10172	327600	7425	18.4
6	$t_{max} = 10ms$	18880	393202	14270	103.1
7	256x256 Pixels	9078	426944	6107	3.6
8	1024x1024 Pixels	9078	214345	7548	70.2
9	V_{3D} Approx: 46m	4509	310598	7749	30.6
10	V_{3D} Approx: 11.5m	24685	301537	6306	11.6
11	Less Candidates	9078	33864	6152	28.6
12	More Candidates	9078	1544822	7022	13
13	Rectangular VC	9078	812248	16530	18.9
14	Per rectangular VC	9078	537776	137145	60.4
15	Environment maps	-	-	1545	233.1
16	No visibility (30ms)	43386	1467564	64017	170.9

Table 1: Statistics for the tests with the Vienna model.

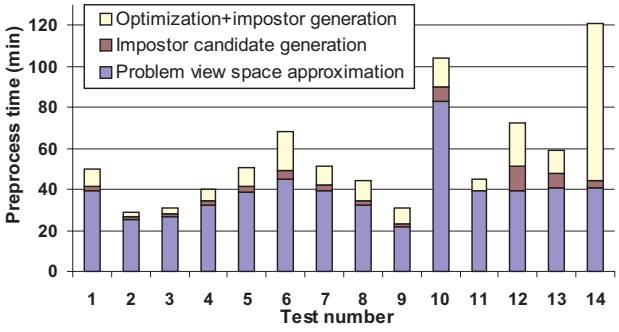

Figure 7: Times needed for the main steps of the algorithm.

Influence of target frame time: For tests 2–6, the target frame time was varied. As was expected, the more acceleration the impostors have to provide, the more memory and preprocessing time is needed. The required memory grows more than linearly with decreasing target frame time, since more and more closer objects have to be represented as impostors. This illustrates that impostors generated in preprocess are most suitable for small and/or for distant objects.

Figure 8 shows the frame times for a sample walkthrough for these tests. While the target rendering time was met in all tests, a general over-conservativeness can be observed. The reason for this is that in our implementation, if a candidate has been chosen as an impostor, it is assigned to *every* problem view it accelerates, even if a problem view has already been solved. This induces no additional memory, but higher acceleration factors for many views.

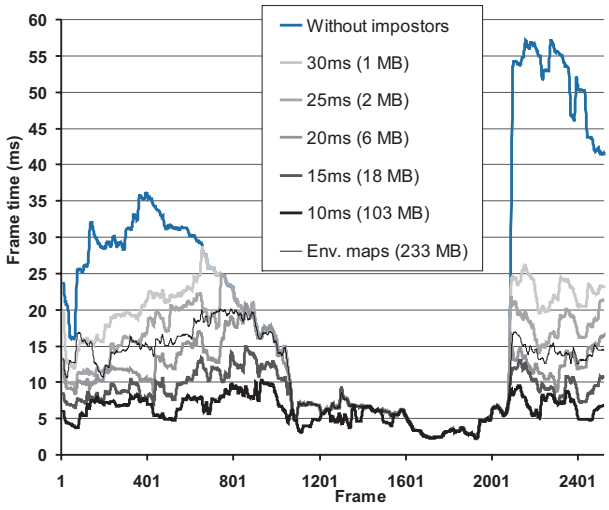

Figure 8: Rendering times for different target frame times.

Influence of output resolution/image quality criterion: Tests 7 and 8 analyze how the output resolution affects the necessary amount of impostor memory in comparison to the reference test. It can be seen that when doubling the output resolution, the memory increases by a factor of about 4 to 5. This can be explained by the fact that the number of impostor texels grows roughly quadratically with increasing output resolution. Furthermore, in our case, a higher resolution also results in a higher number of impostor layers. Note that changing the output resolution is actually equivalent to changing the image quality criterion IQ for the layered impostor technique.

Influence of problem view space approximation accuracy: Tests 9 and 10 analyze how the problem view space approximation (i.e., different CPV sizes in V_{3D}) influences the result. Doubling the CPV size also doubles the required impostor memory, setting it to half the size shows diminishing returns, especially when taking into account the more than twofold increase in preprocessing time. This shows that the approximation accuracy should be chosen with care.

Influence of the number of candidates: The number of candidates is a tradeoff between a sufficiently large basis for a good optimization and reasonable preprocessing times. In test 11, only one tenth of the candidates compared to the reference test resulted in twice the amount of impostor memory. Test 12 shows that a three times increase of the candidate number barely improved the result, but increased the time for the candidate generation and optimization step.

Rectangular view cells: For test 13, nodes of the problem view space approximation were directly used for the impostor candidate generation, as was also described in Section 4. This results in a memory increase of roughly one third compared to the shaft-shaped view cells of the reference test, which illustrates the advantage of shaft-shaped view cells.

Per view cell placement: This test demonstrates the influence of observation 2.2, i.e., the benefit of using large view cells for distant impostors. As for test 13, we used rectangular view cells that were directly obtained from the view space hierarchy. But only leaf nodes were allowed to serve as view cell for a candidate. We tried several sizes for the view cells (i.e., different view space hierarchy levels) with a best result of 60 MB for the impostors, which is more than a three-fold increase compared to test 13 and a more than 4-fold increase compared to test 1. Also note the much longer

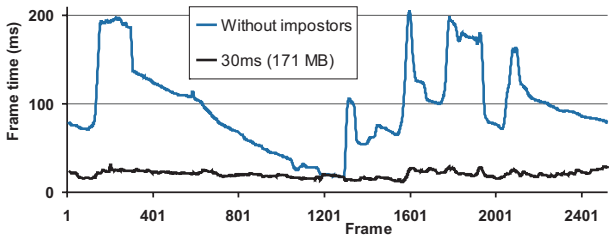

Figure 9: Rendering times without occlusion culling.

preprocessing time for generating the impostors. The results show that the observation 2.2 is an important basis for a good impostor placement.

Environment map impostors: It is interesting to compare the results to a straight-forward approach (test 15), where impostors represent the whole scene from a certain distance (the so-called *far field*) for every view cell. Therefore, we implemented the method of Jeschke et al. [Jeschke et al. 2002] by dividing the view space into a regular grid of view cells. For every cell, visibility culling was applied, and the impostors were arranged as environment map layers, representing the whole visible scene from a certain distance. We tried several combinations of view cell sizes and far field distances and ended up with 25m view cell side length and 300m far field distance as a good tradeoff. This resulted in slightly more than 4.5 hours for impostor generation and 233MB of impostor memory. Note that no frame rate guarantee is given by using impostors in this way, but Figure 8 shows that a frame time of 20ms is not exceeded during the walkthrough. In order to guarantee such a frame time, our new algorithm only needs 5.7MB of impostor memory, which is more than 40 times lower. This shows impressively that impostors should not be placed indiscriminately, but focussed on the scene parts that provide the best rendering acceleration.

Impact of visibility: In test 16, we turned off occlusion culling in order to see how impostor memory requirements increase for providing a frame rate of 30ms. The preprocess needed about 11 hours (3 hours for the problem view space approximation, 36 minutes for the candidate generation and 7.4 hours for the optimization and impostor generation). Figure 9 shows the rendering times for the same walkthrough as above. The results show that visibility culling is a significant factor for our test scene (compare to test 2), and impostors should be used in conjunction with other acceleration techniques in order to make best use of all available techniques.

6.4 Power Plant Results

In order to show how different view spaces influence the impostor placement result, we applied the algorithm to the power plant model. The object hierarchy was an 8-level deep octree of the power plant model, where larger triangles were stored in interior nodes and the remaining ones in the leaf nodes.

We targeted the placement algorithm to limit the number of primitives in every output view to 100,000 polygons. A 40x40m view space was calculated for two side-by-side positions in the model. The resulting impostor memory differs by almost a factor of 4 in this particular case, namely 146MB to 501MB. This clearly shows how the required impostor memory may vary significantly for different view spaces, which makes it clear that comparisons between different impostor placement algorithms are only possible when using exactly the same parameters and view spaces for all tests.

7 Comparison to Related Work

In the past, impostors have been used in several ways, including environment-map-like structures [Darsa et al. 1997; Jeschke et al. 2002], along street segments in urban environments [Sillion et al. 1997], in portals [Aliaga and Lastra 1997; Popescu et al. 1998], per-object (e.g., humans [Aubel et al. 1999], trees [Jakulin 2000], clouds [Harris and Lastra 2001], etc. [Schaufler 1998]), or for nodes of a model hierarchy [Maciel and Shirley 1995; Schaufler and Stürzlinger 1996; Shade et al. 1996]. However, none of these approaches provide a frame rate *guarantee*, and many don't specify exactly for which objects to use impostors. So in many cases, impostors might be placed where they are not necessary (a waste of memory) while in turn they might be missed where needed for rendering acceleration.

One of the first systems exploring rendering with guaranteed frame rates is the predictive level-of-detail management system of Funkhouser and Sequin [Funkhouser and Séquin 1993]. In this approach, image quality is traded for rendering speed. This is done by greedily choosing different object representations based on a cost/benefit metric. Maciel and Shirley [Maciel and Shirley 1995] introduce impostors into this framework, which theoretically allows both, guaranteed frame rates and a minimum image quality. However, they did not specify where exactly to use impostors and experienced high memory requirements.

Wilson et al. [Wilson et al. 2001] use the concept of near field and far field for bounding the scene complexity. They narrow the far field border until the near field complexity falls below a certain threshold. Aliaga et al. [Aliaga et al. 1998] counterbalance the image quality for the near field (displayed applying geometric simplification) and the far field (displayed using impostors) by adapting the size of the near field. Aliaga and Lastra [Aliaga and Lastra 1999] represent distant scene parts with LDIs that are selected using a cost-benefit heuristic, leading to large memory consumption.

All approaches mentioned above share some limitations: first, although many of them aim to maximize the impostor image quality, no guarantee is given for eliminating image gaps caused by disocclusions. As a consequence, visible popping artifacts may occur. Second, only a maximum scene complexity (on the order of 100,000 polygons per frame) is guaranteed, which is hardly related to the output frame rate on current graphics hardware. Third, the placement is done independently for every view cell ([Aliaga and Lastra 1999] even allow only a single LDI per view). This leads to very high memory requirements as was shown in Section 6, because the required impostor memory grows with every view cell. Fourth, none of these approaches use impostors *after* applying visibility culling. In Section 6 we have shown that especially this technique (if available) significantly decreases the amount of required impostor memory. This is the reason why impostors have been successfully used for instance for indoor scenes in portals [Aliaga and Lastra 1997; Popescu et al. 1998]. If no visibility culling is available for a scene, Jeschke et al. [Jeschke et al. 2002] and Wilson et al. [Wilson and Manocha 2003] presented impostor techniques that include visibility calculations within the impostor generation process. However, these visibility calculations require a separate view cell for each impostor, which leads to the same high memory requirements as mentioned above (see test 15 in Section 6).

8 Conclusions and Future Work

We have presented an automatic impostor placement algorithm that can guarantee a specified maximum target frame time and a min-

imum image quality. It was shown that the memory required for impostors can be kept to a tolerable level even for large view spaces and scenes when using impostors carefully and not indiscriminately. For mid-range scenes, as used for instance in computer games, they often fit completely into graphics memory. The main insight that has made this possible is that combinations between *both* objects *and* view space regions have to be taken into consideration, which has not been directly addressed in previous work.

The impostor placement optimization problem can be seen as a multiple choice multiple knapsack problem with partly overlapping items, each having a defined set of knapsacks it can be used for. While greedy optimization is not guaranteed to find an optimal solution, the impostor placements generated by the algorithm are very stable with respect to their input parameters. The algorithm seems to be well adapted to the problem because in normal cases, the items are small compared to the knapsack capacities, which constitutes a good condition for a greedy strategy. Trying an exact algorithm at the end of the optimization phase only barely improved the results.

Furthermore, we have shown that taking visibility calculations into account *before* using impostors greatly reduces the required impostor memory and makes best use of all acceleration techniques at the same time. The approach integrates seamlessly with current real-time rendering systems and is not tied to a particular impostor technique. It can be used with a number of existing techniques if they fulfill the image quality requirements mentioned in Section 2.1. Furthermore, the generic rendering time estimation allows it to easily adapt to the actual rendering bottleneck. Note that the result of the placement algorithm can also be used to generate the impostors on demand at runtime.

A general restriction of any impostor-based approach is that a scene together with the desired frame rate has to be suitable for impostor techniques. For instance, small objects which require a large part of the rendering time budget can easily overload nearby views. In this case, even a very fine view space subdivision might not be sufficient to let impostors provide a desired frame rate. To state it more generally, the combination of the scene, the specified rendering budget and the view space should allow for most of the nearby objects to be rendered using geometry.

In terms of future work, we plan to adaptively choose the degree of problem view space subdivision in order to better adapt to different scene configurations. This will reduce the number of problem views and allow processing extremely large view spaces. It is conceivable to automatically choose between different impostor techniques depending on the scene part to be represented and the problem view space to be served. Finally, we are looking into dynamic impostor techniques to make impostors suitable for scenes with dynamic lighting and various shading effects, which is especially important for computer games.

9 Acknowledgements

This project was supported by the EU in the scope of the GameTools project (IST-2-004363) and the Austrian Science Fund contract no. P17261-N04.

References

ALIAGA, D. G., AND LASTRA, A. A. 1997. Architectural walkthroughs using portal textures. In *Proceedings IEEE Visualization '97*, 355–362.

ALIAGA, D., AND LASTRA, A. 1999. Automatic image placement to provide a guaranteed frame rate. In *SIGGRAPH 99 Conference Proceedings*, 307–316.

ALIAGA, D., COHEN, J., WILSON, A., ZHANG, H., ERIKSON, C., HOFF, K., HUDSON, T., STUERZLINGER, W., BAKER, E., BASTOS, R., WHITTON, M., BROOKS, F., AND MANOCHA, D. 1998. A framework for the real-time walkthrough of massive models. Tech. Rep. TR98-013, Department of Computer Science, University of North Carolina - Chapel Hill.

ALIAGA, D., COHEN, J., WILSON, A., BAKER, E., ZHANG, H., ERIKSON, C., HOFF, K., HUDSON, T., STÜRZLINGER, W., BASTOS, R., WHITTON, M., BROOKS, F., AND MANOCHA, D. 1999. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *1999 Symposium on Interactive 3D Graphics*, 199–206.

AUBEL, A., BOULIC, R., AND THALMANN, D. 1999. Lowering the cost of virtual human rendering with structured animated impostors. In *WSCG'99 Conference Proceedings*, Univ. of West Bohemia Press.

DARSA, L., SILVA, B. C., AND VARSHNEY, A. 1997. Navigating static environments using image-space simplification and morphing. In *1997 Symposium on Interactive 3D Graphics*, 25–34.

DECORET, X., SILLION, F., SCHAUFLER, G., AND DORSEY, J. 1999. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum 18*, 3, 61–73.

DÉCORET, X., DURAND, F., SILLION, F. X., AND DORSEY, J. 2003. Billboard clouds for extreme model simplification. *ACM Transactions on Graphics 22*, 3, 689–696.

FUNKHOUSER, T. A., AND SÉQUIN, C. H. 1993. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *SIGGRAPH 93 Conference Proceedings*, 247–254.

HARRIS, M. J., AND LASTRA, A. 2001. Real-Time cloud rendering. *Computer Graphics Forum 20*, 3, 76–84.

JAKULIN, A. 2000. Interactive vegetation rendering with slicing and blending. In *Proc. Eurographics 2000 (Short Presentations)*.

JESCHKE, S., AND WIMMER, M. 2002. Textured depth meshes for real-time rendering of arbitrary scenes. In *Rendering Techniques 2002*, 181–190.

JESCHKE, S., WIMMER, M., AND SCHUMANN, H. 2002. Layered environment-map impostors for arbitrary scenes. In *Proceedings Graphics Interface 2002*, 1–8.

MACIEL, P., AND SHIRLEY, P. 1995. Visual navigation of large environments using textured clusters. In *1995 Symposium on Interactive 3D Graphics*, 95–102.

POPESCU, V. S., LASTRA, A., ALIAGA, D. G., AND DE OLIVEIRA NETO, M. M. 1998. Efficient warping for architectural walkthroughs using layered depth images. In *Proceedings IEEE Visualization '98*, 211–216.

SCHAUFLER, G., AND STÜRZLINGER, W. 1996. A three-dimensional image cache for virtual reality. *Computer Graphics Forum 15*, 3, 227–235.

SCHAUFLER, G. 1998. Per-object image warping with layered impostors. In *Rendering Techniques '98*, 145–156.

SHADE, J., LISCHINSKI, D., SALESIN, D., DEROSE, T., AND SNYDER, J. 1996. Hierarchical image caching for accelerated walkthroughs of complex environments. In *SIGGRAPH 96 Conference Proceedings*, 75–82.

SHADE, J., GORTLER, S., HE, L., AND SZELISKI, R. 1998. Layered depth images. In *SIGGRAPH 98 Conference Proceedings*, 231–242.

SILLION, F., DRETTAKIS, G., AND BODELET, B. 1997. Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum 16*, 3, 207–218.

WILSON, A., AND MANOCHA, D. 2003. Simplifying complex environments using incremental textured depth meshes. *ACM Transactions on Graphics 22*, 3, 678–688.

WILSON, A., LIN, M. C., YEO, B.-L., YEUNG, M. M., AND MANOCHA, D. 2000. A video-based rendering acceleration algorithm for interactive walkthroughs. In *ACM Multimedia*, 75–83.

WILSON, A., MAYER-PATEL, K., AND MANOCHA, D. 2001. Spatially-encoded far-field representations for interactive walkthroughs. In *ACM Multimedia*, 348–357.

WIMMER, M., AND WONKA, P. 2003. Rendering time estimation for real-time rendering. In *Rendering Techniques 2003*, 118–129.

WIMMER, M., WONKA, P., AND SILLION, F. 2001. Point-based impostors for real-time visualization. In *Rendering Techniques 2001*, 163–176.

WONKA, P., WIMMER, M., AND SCHMALSTIEG, D. 2000. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques 2000*, 71–82.