Eurographics Symposium on Rendering 2008
Steve Marschner and Michael Wimmer
(Guest Editors)

Volume 27 (2008), Number 4

Frame Sequential Interpolation for Discrete Level-of-Detail Rendering

Daniel Scherzer and Michael Wimmer†

Vienna University of Technology

Abstract

In this paper we present a method for automatic interpolation between adjacent discrete levels of detail to achieve smooth LOD changes in image space. We achieve this by breaking the problem into two passes: We render the two LOD levels individually and combine them in a separate pass afterwards. The interpolation is formulated in a way that only one level has to be updated per frame and the other can be reused from the previous frame, thereby causing roughly the same render cost as with simple non interpolated discrete LOD rendering, only incurring the slight overhead of the final combination pass. Additionally we describe customized interpolation schemes using visibility textures.

The method was designed with the ease of integration into existing engines in mind. It requires neither sorting nor blending of objects, nor does it introduce any constrains in the LOD used. The LODs can be coplanar, alpha masked, animated, impostors, and intersecting, while still interpolating smoothly.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image GenerationDisplay algorithms; I.3.3 [Computer Graphics]: Picture/Image GenerationViewing algorithms;

1. Introduction

The idea of discrete level-of-detail (LOD) techniques is to use a set of representations with different levels of detail for one model and select the most appropriate representation for rendering at runtime. This selection can be based on the size of the model on screen, distance to the camera or a more elaborate metric like importance [SDC04].

The individual representations of one model can differ in a number of properties, like geometrical complexity or complexity of lighting. This can go as far as using impostors or point-based approaches for some of the representations. Because of this diversity, automatic generation of LOD levels can often produce non-optimal results. In the industry, artists often handcraft each of the representations. This results in small numbers of LODs being used, where switching from one representation to another can lead to noticeable popping artifacts. In practice, even if the difference between two LOD levels is very small, switching is usually perceptible, because a pixel-exact match does not occur.

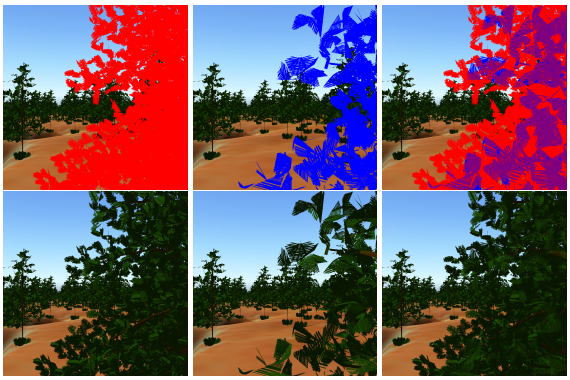

Figure 1: *Our technique combines two buffers containing the discrete LODs to create smooth LOD transitions.* First and second column*: buffers;* last column*: combination. The top row shows the two LODs in red and blue respectively.*

† e-mail: scherzer | wimmer@cg.tuwien.ac.at

The straightforward solution to this problem is to replace the hard switch by a transition phase. In this transition phase the two adjacent discrete levels of detail are blended [GW07]. This introduces a number of drawbacks:

1. The geometry of both representations has to be rendered in this transition phase, thereby generating a higher rendering cost than the higher quality level alone would incur.
2. Possibly complex fragment shaders have to be evaluated for the fragments of both representations.
3. State changes for enabling blending, changing geometry, depth-write mode and changing shaders necessary to blend the two representations are costly on modern architectures.
4. Where the silhouette of the two representations do not match, blending leads to semi-transparent regions of objects that should be opaque.
5. The use of blending can make it necessary to do a depth-sort of the polygons for the objects in transition. Otherwise, popping may occur due to the changing depth order of semitransparent parts (e.g., in objects represented by multiple billboards).
6. Coplanar parts of the two representations will result in z-fighting.

Due to these drawbacks, blending is not well suited for a practical implementation. In this paper we present an algorithm that takes a completely different approach to solve the LOD transition problem. It has none of the mentioned drawbacks while being fast, robust and straightforward to implement.

The algorithm is based on two main ideas: first, the two LODs required during an LOD transition are rendered in *subsequent frames*, thus avoiding to render two representations in a single frame. Second, we introduce the notion of *visibility textures* to replace standard blending, which allows for non-trivial interpolation between objects that can be user controlled or automatically adapted for a given object, providing more plausible transitions between two object representations than blending.

The remainder of the paper is structured as follows: After the discussion of previous work in Section 2, Section 3 explains our algorithm in detail. Results are given in Section 4, and conclusions and ideas for future work in Section 5.

2. Previous Work

Level-of-detail methods are generally divided into two categories: *Continuous LOD* and *Discrete LOD* [LRC*02].

Continuous LOD methods work by creating an object representation specific for each viewing condition for each frame. Since similar viewing conditions result in similar representations, the change of one representation into another is perceived as smooth. Due to the fact that there are infinitely many representations, it follows that the creation of the different representations has to be done by an automatic method, dependent on some input parameters (i.e. error metrics) [GH98]. The problem with such automatic methods is that they do not give satisfactory results for every possible model and all viewing conditions [Hop96]. Therefore most of the continuous LOD literature is specialized on methods for a certain type of models, like in terrain rendering [LKR*96].

Discrete LOD methods on the other hand use a small set of often hand-crafted representations and try to select the most appropriate one for a given viewing condition. These sets are small because of two reasons: memory consumption would explode for many representations, and the cost of an artist designing many representations would be enormous. Due to this small number of representations, switching from one representation to another becomes visible and causes *popping artifacts* [CS06]. One method to avoid these popping artifacts is *late switching*. Here switching is performed only if the two representations have exactly the same appearance for the current viewing condition. In practice this is quite unfeasible: First of all we often do not know when two representations will be undistinguishable, because this depends on numerous factors, like lighting and surrounding objects, which we probable do not know beforehand. And secondly, from the point-of-view of performance we want to switch as early as possible to speed up rendering as much as possible. *LOD blending* [GW07] avoids popping artifacts for discrete LODs by alpha blending and will be discussed in the following section.

This paper focuses on the LOD blending problem, For an overview and in-depth discussion of individual level-of-detail techniques that can be used as a basis for our algorithm, we refer the reader to the comprehensive text by Luebke et al. [LRC*02].

3. Frame Sequential Interpolation

Our new algorithm consists of two main contributions: First, the two LODs required for a transition are rendered in subsequent frames, thus avoiding the cost of having to render two representations in one frame. Second, we propose visibility textures as an alternative for alpha-blending in order to avoid many of its inherent problems at overlapping silhouette regions, coplanar regions etc.

3.1. LOD Transitions

An LOD transition serves the purpose of smoothly replacing LOD_k with LOD_{k+1} (or LOD_{k-1}) in order to avoid popping artifacts. While this is straightforward for continuous LOD methods through *geomorphing*, for discrete LODs it was only shown recently that it is possible to use alpha-blending for the transition [GW07]. The idea is to first "fade in" (using blending) LOD_{k+1} until both objects are completely visible, and then "fade out" LOD_k. The visibility function (which

corresponds to the alpha-function in [GW07]) used for this is depicted in Figure 2 in the bottom row.

This approach always keeps one of the two LODs at full visibility, therefore avoiding both LODs being semi-transparent, because this would lead to the situation that we could look through both LODs onto objects that should not be visible.

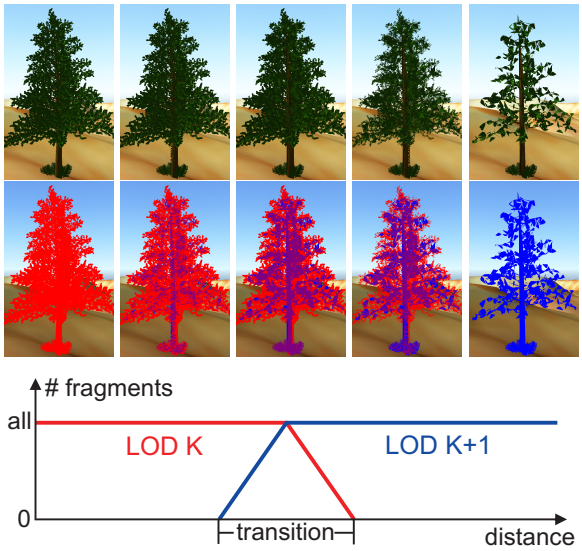

Figure 2: *Transition phase from LOD_k to LOD_{k+1}:* left:*LOD_k;* middle: *midway in the transition all fragments of both LODs are drawn;* right: *LOD_{k+1};* Below: *First LOD_{k+1} is gradually introduced till all its fragments are drawn. Then LOD_k is gradually removed by rendering fewer and fewer fragments. The top two rows show the result of our method and a false color illustration.*

3.2. Frame Sequential *LOD* Transitions

One of the main problems of using LOD transitions as shown above is that both LOD levels have to be rendered during the transition phase, causing a performance drop due to the additional geometry (which can be significant for animated meshes) and rasterization required by the second LOD. This is in contrast to the actual aim of LOD rendering, i.e., improving performance. In this paper, we show how to render the two LODs in subsequent frames.

For the means of our algorithm we will distinguish two states of object instances: instances in the transition phase between two adjacent LODs (called *t*-instances), and all other objects (and their instances) that have either no LOD or are not currently in a LOD transition phase (called *n*-instances).

The central point of our algorithm is to formulate the instance transition in a way that makes it possible to split the rendering of the two LODs of every *t*-instance into sequential frames. Each is rendered along with the rest of the scene into one of two off-screen buffers (the *LOD buffers*) and afterwards combined to form the final image. What makes this approach fast is that we do the LOD level renderings in an *alternating fashion*: only one *LOD buffer* is rendered each frame, containing all the *t*-instances with one of LOD_k or LOD_{k+1}, and the rest of the scene. Note that for the combination pass, we also require a depth value and an object id per pixel for both buffers.

A sample of this is show in Figure 3: In this scene only two different LODs are present – a red sphere (LOD_0) and a blue box (LOD_1). Two instances are in transition (the instances where the LODs differ in the LOD buffers *left* and *middle*), one instance is at LOD_0 (the closest red sphere) and a lot of instances are at LOD_1 (all other blue boxes). The two LOD buffers (*left* and *middle*) are identical except for the differences in the LODs used for the two *t*-instances.

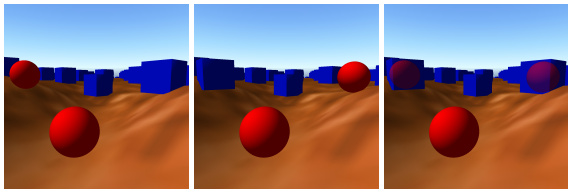

Figure 3: *The LOD buffers of two consecutive frames only differ in the used LODs for t-instances (*left *and* middle*). They are combined to create the final image (*right*).*

One problem with frame-sequential rendering is that if the camera moves between these two consecutive frames, the older of the two LOD buffers will be one frame "out of date". In this case the two frames will not only differ in the *LOD*s of the *t*-instances, but also slightly in the view-space. We account for that possibility by accessing the older LOD buffer via back-projection in the combination pass (see later). Similarly, if a *t*-instance moves between adjacent frames, its two LODs will not overlap completely. However, the artifact caused by this fact is not objectionable since it corresponds to a simple motion blur, which is expected in moving objects anyway.

Frame-sequential rendering also allows easy integration into existing rendering engines: The scene can be rendered each frame as usual. The only difference is we now render to an off-screen buffer, and the concerned parts, the *t*-instances, have to be rendered differently for our algorithm. The additional combination pass discussed in the following is also self contained and only needs access to the two LOD buffers and the reprojection matrix.

3.3. Blending with Visibility Textures

Using alpha-blending for LOD transitions faces several problems. Blending both LODs in the same frame buffer

as in [GW07] requires depth-sorting all t-instances. Objects with high depth complexity or objects that are semi-transparent themselves (e.g., billboard clouds) even require depth-sorting individual polygons in order to avoid popping due to changes in depth order.

These problems do not appear in frame-sequential rendering of t-instances when alpha-blending is used to combine the finished LOD buffers. However, alpha-blending LOD buffers leads to problems when the silhouettes of the two LODs of a t-instance do not match for several objects appearing behind each other: in this case, several semi-transparent regions should be created, but the LOD buffers do not contain any information about the objects behind the first semi-transparent region (each LOD buffer stores only a single depth layer). See, for example, Figure 4. In frame-sequential LOD blending, this problem can lead to severe popping when the transition state of objects change.

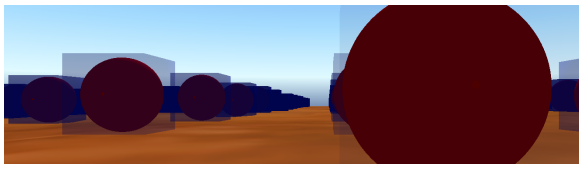

Figure 4: *Blending can create an arbitrary number of LOD blends per pixel.*

We solve this problem by using *visibility textures*. The idea is to change the visibility of an object by changing the amount of rasterized fragments instead of alpha-blending. This is quite similar to the well-known screendoor transparency, where transparency is simulated using a stippling pattern. However, visibility textures are applied in object space instead of screen space, and allow more flexible visibility patterns. Note that visibility textures are already applied when rendering to the LOD buffers, since the resulting visibility is order independent.

We store a 3D visibility function per object (the *visibility texture*) and compare it to a visibility threshold to decide which t-instance fragments to discard. The visibility threshold $\tau \in [0..1]$ is given by the function depicted in Figure 2. Written as an equation: $\lambda : R^3 \times [0..1] \rightarrow \{true, false\}$

$$\lambda : (\vec{p}, \tau) \mapsto visTex(\vec{p}) > \tau \qquad (1)$$

is the function that evaluates for each fragment if it should be discarded. Here $\vec{p}$ is the object-space coordinate (before any animation is applied) of the fragment, τ the visibility threshold and *visTex* is the lookup into the visibility texture. Note that even though the visibility function may be continuous, the thresholding operation gives a binary result and therefore no semi-transparent pixels appear.

The visibility texture is applied in object space because in this way, the noise is perceived as part of the object rather than a world-space effect. If the lookup into the visibility texture were to take place in world space, the visibility texture would stay fixed and not move along with the object when it is animated (see Figure 5).

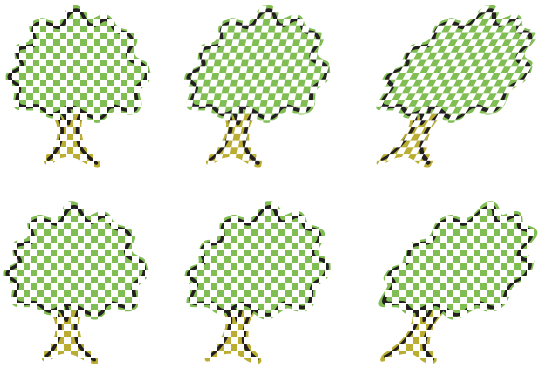

Figure 5: *Visibility textures (here a checkerboard pattern) are applied before transforming an object* (upper row)*. If applied afterwards the pattern would not follow the movement* (lower row)*.*

By using different visibility textures one can control in which way the fragments of a t-instance become visible. Examples include a uniform noise pattern, a function that decreases from the center outward, or any other function best suited to a given object. This has the effect that the amount and distribution of the visible fragments of an object can be controlled (see Figure 6). Note that in practice, we use a 2D noise texture and access it with $(\vec{p}.x + \vec{p}.z, \vec{p}.y + \vec{p}.z)$.

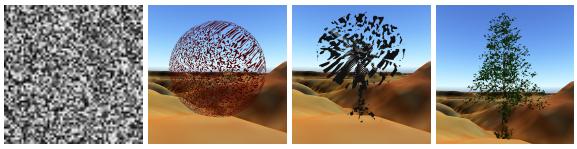

Figure 6: *A uniform noise visibility texture* (left) *applied to three different models with visibility* $\tau = 0.5$*).*

3.4. Combination

Frame-sequential LOD rendering with visibility textures generates two LOD buffers with t-instances that appear "stippled" according to their visibility as discussed in the previous section. These two buffers need to be combined to give the final frame buffer. This is done in a simple screen-space rendering pass.

We need to determine for every pixel, which of the two LOD buffer input pixels to use. Note that the older buffer is accessed using reprojection [SJW07]. This is done by back-projecting each 3D pixel (we have its depth stored in the

LOD buffer) to its position in the older LOD buffer by applying

$$\vec{p}_{old} = \mathbf{P}_{old} * \mathbf{V}_{old} * \mathbf{V}_{new}^{-1} * \mathbf{P}_{new}^{-1} * \vec{p}_{new} \quad (2)$$

where $\vec{p}_{old|new}$ are the old and new pixel positions, $\mathbf{V}_{old|new}$ are the view matrices and $\mathbf{P}_{old|new}$ are the projection matrice of the old and new LOD buffers.

Pixels that back-project outside the LOD buffer have no corresponding old LOD buffer pixel and therefore we treat them as a special case and use only the new color. Note that this can create minor artifacts at the viewport borders when moving the camera. We found that these artifacts were not noticeable in practice.

In addition to color, the LOD buffers also store the depth and an id for all the t-instances (for n-instances we take some default id like 0). Note that different LODs of the same instance have the same id.

Three cases can occur:

1. both pixels are from n-instances
2. one pixel is from an t-instance and the other from a different t-instance or from an n-instance
3. both pixels are from the same t-instance (and thus from different LOD levels)

The first is the most frequent case. No interpolation should happen and therefore the pixel from the newer LOD buffer is used. In the second case, the fragment is chosen according to the stored z-values. Thus the result is the same as if the objects had been rendered to one single buffer instead of two. The third case is the most interesting one. While it could be handled similar to the second case using z-testing, which would correspond to evaluating the screendoor-transparency between the two LOD levels of the t-instance, this is not advisable. Coplanar polygons in the two LOD would lead to z-fighting. We therefore avoid z-testing and use the average of the two colors. This has the interesting effect that halfway through the transition, when both LODs are fully visible, the colors of the two LODs will be mixed where they overlap, and thus correspond to a cross-dissolve between the two LODs.

Note that the older LOD buffer is used solely in the areas where a transition between two LODs is taking place, so most of the final image will consist of the new frame.

3.5. The Algorithm

In summary the algorithm works as follows:

1. First we render the scene into an off-screen LOD buffer containing color, object id and depth.
   a. All n-instances are rendered normally, and with object id 0.
   b. All t-instances are rendered using alternating LODs and an id that is unique for this instance, while using Equation 1 to determine for each fragment if it should be rendered.
2. In the second pass we combine the two LOD buffers in screen space. For the two fragments:
   a. If both fragments are from n-instances, the newer fragment is kept
   b. If one fragment is from a t-instance and the other is not from the same t-instance, the closer fragment is kept
   c. If both fragments are from the same t-instance, the average of the two fragment colors is kept

4. Implementation and Results

We implemented the proposed algorithm on an Intel Core 2 Duo E6600 CPU and NVIDIA 8800GTX graphics card. We used a uniform noise visibility texture of 64×64 pixels for all the images shown in this paper. We used two screen-sized LOD buffers each consisting of color ($24bit$ RGB), depth ($16bit$ float and id ($16bit$ float), 7 bytes per pixel in total. For a 1024^2 view port this amounts to $14MB$ of additional memory required and for a 512^2 viewport to $1.75MB$. We noticed that in practice back projection of the older LOD buffer is often not needed because errors are hidden by the LOD interpolation.

We evaluated our algorithm on three different test scenes (see Figure 9). TREES is a scene with 4,000 trees, LODs ranging from $5,000 - 93,000$ triangles. WINDMILLS has 4,000 windmills, LODs ranging from $100 - 18,000$ triangles. SIMPLE consists of objects with two LODs: a sphere and a cube of different color. The results can best be judged from the accompanying video, which shows side-by-side comparisons of our algorithm to both discrete LOD switching (showing popping) and LOD blending [GW07]. SIMPLE is in a way the worst case for LOD transition algorithms in general because the objects match neither in color nor in silhouettes. While our algorithm handles it correctly, noise artifacts from the interpolation are clearly visible. WINDMILLS is a scene that represents a common case and is well behaved for most LOD transition algorithms. Both blending and our algorithm give smooth results, while our algorithm has a higher frame rate. Note that close-up views would still reveal artifacts in the blending approach. TREES is a scene that shows significant artifacts for the blending approach because polygons in the semi-transparent LOD, which is not z-tested, are not sorted correctly. Our approach works very well for this scene. Noise artifacts are practically invisible in this scene. In general, we conclude that the quality of the blend is sufficient even if slight noise can be visible in pathological cases (e.g. SIMPLE). Furthermore, the transition is always smooth (unlike other approaches) and does not suffer from artifacts.

Figure 8 shows performance measurements for the three methods with varying viewport sizes for WINDMILLS. Our

method is always faster than LOD blending. Note however that in higher resolutions, when the application becomes fillrate limited, our method incurs a certain performance penalty versus discrete switching because of the final combination pass. However, the fragment shaders of the objects in this example are extremely simple and thus the penalty looks exaggerated in comparison to the more common case of realistic fragment shaders.

Given two LODs, one will have a higher rendering cost. As such, in theory each alternating frame could take longer to render (if more of the costly LODs happen to be rendered in this frame) , causing uneven frame rates. However, during our tests such a case never occurred. We evaluated the number of triangles rendered each frame and encountered only differences of up to 2% between consecutive frames.

4.1. Integration into Applications

Integrating the algorithm into an existing rendering pipeline is not very demanding. First, we must be able to change LOD levels each consecutive frame. This is similar to standard discrete LOD switching, where LOD switches could also occur every frame. We need to insert some code into the routine that decides which LOD level to display, for example by deriving from an existing LOD object class. The code checks if the current object is in a transition phase (currently a t-instance). If not, it behaves like a standard LOD object. If it is in a transition phase, we use a toggle counter to decide which of the two LODs will be rendered in this frame and calculate τ by using the function shown in Figure 2. As instance id for the item buffer we use the memory address of the object. Please note that any parameters (like skinning matrices) that have to be calculated on the CPU are needed only once per frame, not for both LODs that take part in the transition phase.

Second, we must be able to add the shader code needed for our approach, namely the calculation of $\vec{p}$ in the vertex shader and the evaluation of the visibility texture (with a potential discard of the fragment) in the fragment shader. In modern shading languages, these routines can be encapsulated into subroutines that can be easily added at the beginning of all relevant shaders in the application. Ideally, the engine already provides a mechanism to link subroutines conforming to a predefined interface dynamically, otherwise the subroutine call needs to be added manually. See the listing at the end of this section for our implementation of these two functions in the OpenGL shading language.

Third, we need the ability to render the scene into offscreen buffers using multiple render targets (MRT). One render target holds the color, while the second holds id and depth. We need these two buffers twice: in each frame, only one color and id-depth buffer is used to write into, while the other is accessed by the shaders. We switch between the two buffers in a ping-pong fashion.

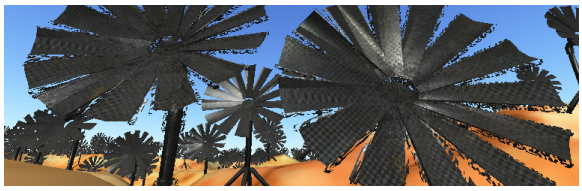

Figure 7: *Where the silhouettes of the LODs do not match, noise artifacts can become visible (over emphasized).*

```
varying vec4 coord_OS;
/*vertex*/ void prepareAlternating() {
  coord_OS = gl_Vertex;
}

uniform sampler2D visTexture;
uniform float tau;
/*fragment*/ void checkVisibility() {
  vec2 coord_ts = coord_OS.xy+coord_OS.z;
  float vis = texture2D(visTexture,coord_ts);
  if(vis > tau) {
    discard;
  }
}
uniform int id;
gl_FragData[1].xy = vec2(gl_FragDepth,id);
```

4.2. Limitations

The main artifact of our method is that the pattern of the visibility texture becomes visible if the silhouettes of the different LODs don't match well (see Figure 7 for an example). This happens either because the chosen LOD levels differ too strongly, especially in the silhouette, or if the silhouettes are transformed due to object movement (motion blur, see Section 3.2). In general, these artifacts are relatively unobtrusive as long as the mismatch is not too strong (e.g., fast object movement).

Another type of artifact occurs when previously unseen pixels appear and therefore no respective pixel in the old LOD buffer exists, for example at the viewport borders (see Section 3.4) or for disocclusions (if new scene parts appear behind occluders). These artifacts are typically very difficult to spot.

5. Conclusions and Future Work

In this paper we presented a new method of creating smooth transitions between discrete levels of detail. Our method has several advantages over previous approaches: It is faster because it avoids rendering transition objects twice through frame-sequential rendering (see also drawbacks 1-3 mentioned in the introduction). It avoids alpha-blending artifacts because it requires no sorting, which is achieved through visibility textures (drawbacks 4-5). Even co-planar LOD levels

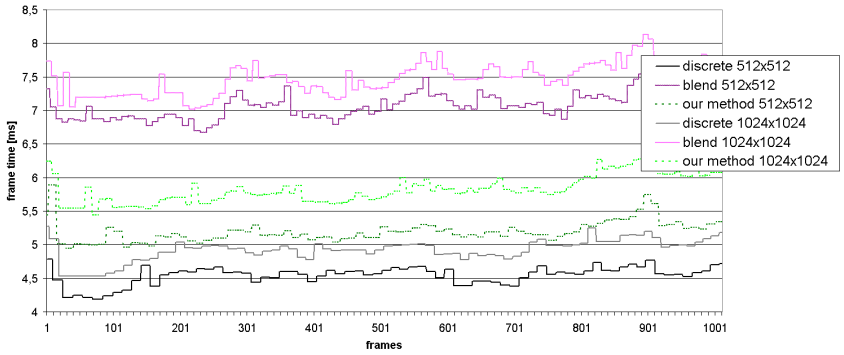

Figure 8: *Frame times for discrete LOD rendering, LOD blending and our method.*

pose no problem due to the use of object ids (drawback 6). The algorithm is also more robust, straightforward to implement and easier to integrate into common rendering engines.

A possible extension for future work is the reduction of the noise we introduced for our interpolation scheme, for instance with a history buffer as in [SJW07] or by using multi-sampling. Another promising avenue for further research would be the topic of visibility textures, i.e., the automatic generation of an optimized visibility texture for a given object.

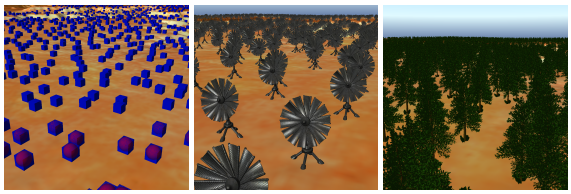

Figure 9: *We used three different test scenes to evaluate our algorithm. Left: SIMPLE, middle: WINDMILLS, right: TREES.*

Acknowledgements

Thanks to Alexander Kusternig for textures and shaders. This project was supported by the EU under the project no. IST-014891-2 (Crossmod).

References

[CS06] CLEJU I., SAUPE D.: Evaluation of supra-threshold perceptual metrics for 3d models. In *APGV '06: Proceedings of the 3rd symposium on Applied perception in graphics and visualization* (New York, NY, USA, 2006), ACM, pp. 41–44.

[GH98] GARLAND M., HECKBERT P. S.: Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the conference on Visualization '98* (Oct. 1998), Ebert D., Hagen H., Rushmeier H., (Eds.), IEEE, pp. 263–270. ISBN 1-58113-106-2.

[GW07] GIEGL M., WIMMER M.: Unpopping: Solving the image-space blend problem for smooth discrete lod transitions. *Computer Graphics Forum 26*, 1 (Mar. 2007), 46–49.

[Hop96] HOPPE H.: Progressive meshes. In *SIGGRAPH 96 Conference Proceedings* (Aug. 1996), Rushmeier H., (Ed.), Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 99–108. held in New Orleans, Louisiana, 04-09 August 1996.

[LKR*96] LINDSTROM P., KOLLER D., RIBARSKY W., HUGHES L. F., FAUST N., TURNER G.: Real-Time, continuous level of detail rendering of height fields. In *SIGGRAPH 96 Conference Proceedings* (Aug. 1996), Rushmeier H., (Ed.), Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 109–118. held in New Orleans, Louisiana, 04-09 August 1996.

[LRC*02] LUEBKE D., REDDY M., COHEN J. D., VARSHNEY A., WATSON B., HUEBNER R.: *Level of Detail for 3D Graphics (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, July 2002.

[SDC04] SUNDSTEDT V., DEBATTISTA K., CHALMERS A.: Selective rendering using task-importance maps. In *APGV 2004 - Symposium on Applied Perception in Graphics and Visualization* (August 2004), ACM SIGGRAPH, pp. 175–175.

[SJW07] SCHERZER D., JESCHKE S., WIMMER M.: Pixel-correct shadow maps with temporal reprojection and shadow test confidence. In *Rendering Techniques 2007 (Proceedings Eurographics Symposium on Rendering)* (June 2007), Kautz J., Pattanaik S., (Eds.), Eurographics, Eurographics Association, pp. 45–50.