Chapter 1

Progressive Buffers: View-dependent Geometry and Texture LOD Rendering

Pedro V. Sander[1]
ATI Research

Jason L. Mitchell[2]
Valve Software

The content of this chapter also appears on Symposium on Geometry Processing 2006

1.1 Abstract

We introduce a view-dependent level of detail rendering system designed with modern GPU architectures in mind. Our approach keeps the data in static buffers and geomorphs between different LODs using per-vertex weights for seamless transition. Our method is the first out-of-core system to support texture mapping, including a mechanism for texture LOD. This approach completely avoids LOD pops and boundary cracks while gracefully adapting to a specified frame rate or level of detail. Our method is suitable for all classes of GPUs that provide basic vertex shader programmability, and is applicable for both out-of-core or instanced geometry. The contributions of our work include a preprocessing and rendering system for view-dependent LOD rendering by geomorphing static buffers using per-vertex weights, a vertex buffer tree to minimize the number of API draw calls when rendering coarse-level geometry, and automatic methods for efficient, transparent LOD control.

1.2 Introduction

Real-time rendering of massive 3D scenes lies at the forefront of graphics research. In this paper we present new algorithm for real-time rendering of large polygonal meshes. To our knowledge, this is the first out-of-core view-dependent mesh renderer that supports texture mapping and continuous smooth transitions between LODs to prevent

[1] psander@ati.com
[2] jasonm@valvesoftware.com

popping. Both of the above features allow our method to faithfully render geometry with high fidelity without requiring sub-pixel sized triangles with Gouraud-interpolated vertex colors. Our method is also applicable to instanced geometry, as we will show in the results section.

Our data structure, the progressive buffer (*PB*), is derived from a progressive mesh (*PM*) [Hop96] and consists of a sequence of static buffers at different levels of detail for the different clusters of polygons that make up the mesh. Each buffer stores an irregular mesh, thus faithfully capturing geometric detail for a given polygon rate. Transitioning between different levels of detail is achieved via geomorphing [Hop96]. Our novel method computes geomorphing weights per vertex in order to ensure consistency between neighboring clusters of triangles and to prevent boundary discontinuities. Figure 1 shows a rendering of a *PB* with a texture and with color-coded LODs.

(a) Texture-mapped rendering with shadows *(b) Color-coded chart LOD*

***Figure 1**. View-dependent geometry and texture LOD on a 16M triangle mesh. The adaptive model being rendered has 800,000 triangles. This scene is rendered at 30fps.*

Due to the usage of static buffers and texture mapping, this system achieves high rendering rates using consumer graphics hardware and scales to previous hardware.

This paper presents a preprocessing method and a rendering system for geometry and texture view-dependent dynamic level of detail that is suitable for a large class of graphics hardware. In order to achieve this objective, we introduce the following techniques:

- A rendering method that geomorphs the geometry in the vertex shader using per-vertex weights. This approach completely prevents LOD pops and boundary cracks, while still using "GPU-friendly" static vertex and index buffers.

- A hierarchical method to more efficiently render geometry that is far from the viewer, thereby reducing the number of API draw calls.

- A scheduling algorithm to load required geometry and texture data on demand from disk to main memory and from main memory to video memory.

- An automatic method that controls and smoothly adjusts the level of detail in order to maintain a desired frame rate. This approach is transparent and

gracefully adapts the rendering quality as a function of the graphics horsepower and the scene's geometric complexity.

The approach presented in this paper has the following drawbacks and limitations:

- On current hardware, the size of the vertex buffer is doubled when geomorphing to a lower level of detail. Note, however, that this secondary buffer only needs to be loaded when a particular cluster is in a geomorphing region (see Section 3). Since, high-detail geometry is only required for regions that are close to the camera, the benefit of a flexible data structure outweighs the overhead on the small subset of buffers that reside in video memory.

- Our method requires a larger number of draw calls than purely hierarchical algorithms. This is required because current graphics hardware does not allow changing texture state within a draw call. Grouping separate textures in unified atlases at higher levels of detail would change the texture coordinates, thus preventing those from being geomorphed appropriately. We believe the advantages of texture mapping are more important than the efficiency gain of having fewer draw calls on clusters near the camera. For clusters far from the camera, we address this problem by grouping the low resolution data in unified buffers, thus reducing the number of draw calls on large scenes, where it matters the most.

- Although this approach does not require a perfect voxelization of space to construct different clusters of adjacent faces, our rendering method achieves better results when there are no clusters significantly larger than the average. This is because the maximum cluster radius restricts the size of the LOD regions as described in Section 4. For best performance, clusters should have similar bounding radii (within each connected component). We address this by first voxelizing space, and then further splitting each cluster into charts that are homeomorphic to discs and thus can be parametrized.

The remainder of this paper is organized as follows. In Section 2, we describe previous work and how it relates to our approach. Section 3 outlines our basic data structure, the progressive buffer, which provides a continuous level of detail representation for the mesh. In Section 4, we describe how we efficiently render progressive buffers. Section 5 presents our preprocessing algorithm, which partitions the mesh into clusters and generates the progressive buffers for each cluster. Finally, we present results in Section 6 and summarize in Section 7.

1.3 Previous Work

Several methods for efficient rendering of large polygon models have been proposed in the past. The earlier works focused on continuous LOD, while more recent research addresses rendering large models that do not fit in video memory, thus opening a number of different issues, such as out-of-core simplification and memory management.

The first approaches developed for view-dependent real time mesh rendering adaptively simplified at the triangle level via edge collapses [Xia96, Hoppe97, El-Sana99]. With the

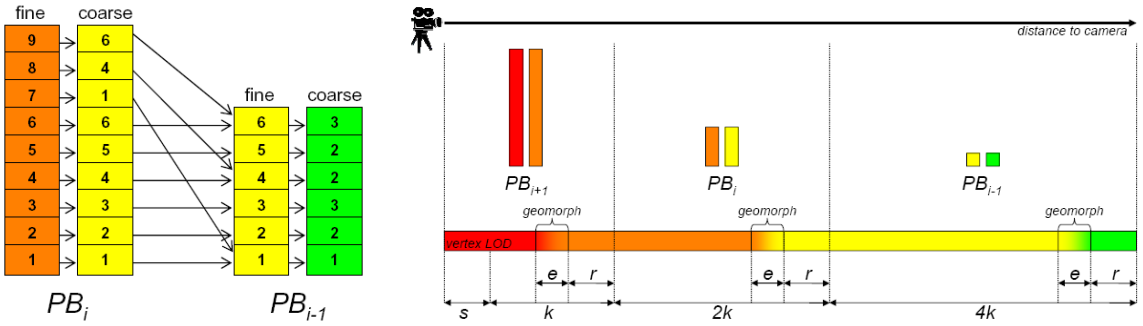
advent of programmable graphics hardware it has become much more efficient to perform larger scale simplification on static buffers. Other methods clustered sets of vertices in a hierarchical fashion [Luebke97]. While these methods are generally good at providing view dependent LOD, none of the above methods are applicable to out-of-core rendering of arbitrary polygonal meshes.

An alternative approach for rendering large meshes was presented by Rusinkiewicz and Levoy [Risinkiewicz00]. Their method converts the input mesh to a vertex tree, which is then rendered using point primitives. However, current graphics hardware is more optimized for rendering triangle primitives with texture/normal maps, which usually produces higher quality results for the same rendering cost. There are several hybrid approaches that use both triangle and point primitives in order to reduce rendering cost (e.g., [Chen01, Dey02]).

Recent out-of-core methods for view-dependent mesh rendering have focused on the fact that graphics hardware is significantly more efficient when rendering static triangle buffers from video memory. These methods use irregular meshes, the most common used path on current graphics cards, making them very efficient for a given fidelity. There are several recent methods for out-of-core rendering based on irregular meshes (e.g., [El-Sana00, Vadrahan02, Lindstrom03, Cignoni04, Yoon04]). These methods address issues of memory management and prefetching to video memory. However, to our knowledge, none of the previously published works for out-of-core rendering of arbitrary polygonal meshes provide a continuous smooth transition between different LODs nor do they support texture mapping. The above methods rely on the fact that, with a very high triangle throughput rate, one can store the required attribute data per vertex and directly switch the rendered triangles to a coarser level of detail before the change becomes visually noticeable (i.e., before a screen space error tolerance is met).

Our novel approach geomorphs between the levels of detail, resulting in a smooth *pop-free* transition, regardless of the screen-space error of the approximation. It does not require pixel-sized triangles, as it can store detail in texture maps and provide LOD control over the texture images. Therefore, since the rendered geometry can be coarser, it allows us to focus the GPU resources on rendering other scene effects with complex shaders while still maintaining real-time frame rates. Furthermore, our method can be used with older generation graphics hardware for a given loss of rendering quality.

Gain and Southern [Gain03] use geomorphing for static LOD within each object of the scene. Our method, however, addresses multiple levels of detail for a single arbitrary object, thus allowing for view-dependent LOD of large meshes. This is accomplished by computing the geomorphing weight per vertex, as opposed to per object, by properly constructing mesh clusters, and by constraining where LOD transitions take place, as described in the next section. [Ulrich02] presents a method designed for terrain rendering and avoids transitions between objects of mismatched LODs by introducing a small vertical ribbon mesh, which would likely produce more significant texture-stretching artifacts for arbitrarily complex geometry. Our approach addresses arbitrary meshes and avoids the need for ribbon meshes by performing the geomorph computation per vertex. The idea of per-vertex LOD was first introduced by the multiresolution rendering algorithm of Grabner in 2003.



(a) Transition between LODs.

(b) Vertex LOD as a function of distance from camera.

Figure 2. *The geomorph depicted on the left occurs when the cluster's bounding sphere's center is near the transition point between rendering PB_i and PB_{i-1}. To ensure continuity, the geomorph must be performed at a distance of r away from this transition point, where r is the maximum cluster radius. This is necessary so that all vertices have finished geomorphing when the cluster switches LOD. The variables k, s, and e determine the size of the LOD bands and the geomorph region.*

1.4 The progressive buffer

As mentioned above, our approach consists of constructing a progressive buffer, which is a series of vertex and index buffers that represent a mesh at different levels of detail (LODs). Figure 2a shows two levels of detail of a progressive buffer: PB_i and PB_{i-1}. Note that there are two vertex buffers associated with each level of detail. Each cell represents a vertex, which is identified by an ID number. Each level of detail also has an index buffer, which is omitted from the figure. We will refer to the set of two vertex buffers and one index buffer at a particular level of detail as a static buffer (PB_i, where *i* is the level of detail), and to the entire sequence of static buffers as a progressive buffer.

Continuous level of detail. The discrete static buffers, coupled with continuous geomorphing weights yield a continuous level of detail representation for the mesh. When rendering a static buffer, geomorphing weights are determined in order to properly blend the vertex data between the fine and coarse buffers based on distance from the camera. The coarse buffer of PB_i contains the same vertex data as the fine buffer of PB_{i-1}. Figure 2b shows a progressive buffer with 3 levels of detail. Note that, as the distance from the camera increases, the cluster is geomorphed to the coarser buffer and subsequently switches to a different static buffer. As long as the geomorph to the coarser buffer is completed before the switch, there will be no difference in the rendered image when switching from one static buffer to the next.

View-dependent level of detail. So far, this method works well for static level of detail, where the entire mesh is rendered using the same static buffer and geomorphing weight. However, in order to enable view-dependent dynamic level of detail, we must be able to assign different levels of detail to different regions of the mesh. To achieve this, we

partition the mesh into multiple clusters and construct a progressive buffer for each cluster. In order to prevent geometric cracks on cluster boundaries, we must meet the following requirements:

- When constructing the progressive buffers, consistently simplify all clusters of each connected component in unison in order to achieve consistent cluster boundary vertex positions at all LODs, as described in Section 5.

- Ensure that the LOD and geomorphing weights of boundary vertices match exactly across clusters, as described next.

Clearly, one cannot assign a constant LOD for the entire cluster; otherwise all clusters of a connected component would need to have the same LOD for all boundaries to match. That would not allow for dynamic level of detail. To address this issue, we compute the geomorph weights per vertex. If the geomorph weight is determined based on the distance from the vertex to the camera, a boundary vertex will have the same LOD and geomorph weight as its mate on the neighboring cluster. This approach avoids boundary cracks and allows the level of detail to vary across the mesh. Note that the discrete static buffer is constant through the entire cluster. It is determined based on the distance from the cluster's bounding sphere center to the camera.

The vertex LOD bar in Figure 2b shows that as long as the proper buffers are used, one can render a cluster by geomorphing each vertex independently, based on its distance to the camera. The distance range in which the geomorph takes place must be at least r away from the LOD boundary, where r is the maximum cluster bounding sphere radius of the mesh. This is necessary in order to ensure that none of the vertices will be in the geomorph range after the cluster's bounding sphere center crosses the discrete LOD boundary and the renderer starts using a different static buffer for that cluster. As shown in Figure 2b, we choose the geomorph range to be as far away from the camera as possible in order to maximize the quality of the rendering.

Coarse buffer hierarchy (CBH). In order to minimize the number of draw calls, we group the static buffer of the coarsest LOD of all clusters in a single vertex buffer with a corresponding index buffer. We then render different ranges of this buffer with the aid of a hierarchical data structure which groups clusters together. This approach, detailed in Section 4.3, also allows us to perform frustum culling at any node of the tree.

Out of core data management. During rendering of an out of core model, the engine keeps track of the continuous LOD determined by the center of the bounding sphere of each cluster. As this number changes, the engine appropriately loads and unloads data to and from the disk, main memory, and video memory. We employ a system that has four priority levels, as shown in Figure 3. Active buffers that are currently being rendered must reside in video memory and have priority 3. Buffers that could become active very shortly if the distance from the camera to the cluster changes slightly have priority 2 and are also loaded to video memory (this buffer prefetching is very important to ensure the buffer is available when needed). Buffers that could possibly be needed in the near future have priority 1 and are loaded to main memory, but not to video memory. Finally, all other buffers have priority 0 and only reside on disk. A least-recently-used (LRU) scheme is used to break ties between buffers that have the same priority level. As shown in Figure 3, the engine can set thresholds to each of these priority levels based on the amount of video and main memory present and how fast it can read from the hard

disk. Methods to automatically adjust the complexity of the scene given fixed memory thresholds or current rendering frame rate are described in Section 4.4.

Priority	System memory	Video memory	Sample thresholds
3 (active)	Yes	Yes	100MB
2 (almost active)	Yes	Most likely	20MB
1 (needed soon)	Yes	No	50MB
0 (not needed)	No	No	Full dataset

Figure 3*. Different priority levels along with where the buffers reside and example maximum thresholds.*

Texture mapping. Progressive buffers can be texture mapped using a consistent mesh parametrization. [Cohen98] described an approach to preserve texture coordinates during simplification. This method extends naturally to progressive buffers. A single texture can be used for the entire progressive buffer. Each mip level of the texture is associated with a static buffer. Thus, the higher the static buffer being used, the higher the maximum mip level. As with the geometry data, texture data is also stored on disk and loaded out of core as the level of detail changes.

1.5 Rendering

In this section, we describe how to efficiently render progressive buffers. We first describe a basic algorithm using the data structure described in the previous section. Then we describe an optimized hierarchical algorithm to reduce the number of draw calls. Finally, we describe how to adjust the level of detail to maintain a stable frame rate.

1.5.1 Computing the level of detail

In order to render the mesh, our rendering algorithm must determine in real-time which level of detail we want to use for each cluster. Our approach determines the level of detail based on the cluster's distance to the camera and tries to maintain a constant triangle size after projection to the screen. Assuming the worst case scenario, in which the triangles in the cluster are all facing the viewer straight-on, this method maintains an approximately constant screen-space area for the triangle as the camera moves. As the distance to the camera doubles, the screen space area of the triangle is reduced by a factor of four. As a result, every time the distance to the camera doubles, we switch to the next coarser level of detail, which has four times fewer vertices. Note that, as shown in Figure 2, this is only true if the parameter ***s*** is set to its default value of 0. The variable s, however, can be set to a positive or negative value in order to further adjust the LOD. One can consider other distance and vertex ratios, but one significant advantage of each LOD having four times more vertices than its parent is that the same factor of four can

be applied to the textures, which is convenient, especially when mipmapping these textures. This way, both vertex and texture detail change by the same factor from one LOD to the next.

The variables *s* and *k* from Figure 2 can be adjusted as a function of several values, such as frame rate, memory and triangle count upper bound. *s* is used as a bias term for the LOD, while *k* is a scaling term. Section 4.4 describes how to automatically adjust these values to maintain a given frame rate.

We set the variable *e*, which represents the length of the geomorph band, to its maximum allowed value of $k - r$. This makes the transitions smoother and does not affect rendering performance since the GPU still processes the same number of triangles.

Given *s*, *k* and *d*, which is the distance from the cluster's center to the camera, the level of detail of a cluster is

$$i = floor\left(\log_2\left(\frac{d-s}{k}+1\right)\right)$$

Prior to rendering the cluster, we must also determine the start distance, d_s, and the end distance, d_e for the geomorph region within that cluster, which is computed as follows:

$$d_e = \left(2^{i+1}-1\right)k + s - r$$

$$d_s = d_e - e$$

These two values must be placed in the GPU's constant store, so that during rendering, the vertex shader can interpolate the position and other attributes based on the vertex distance from the camera.

1.5.2 Basic rendering algorithm

The basic rendering algorithm traverses all of the clusters and, for each cluster, tests the cluster's bounding sphere against the view frustum. Should the cluster be inside the frustum, it then sets the appropriate constant store variables and renders the desired level of detail. The buffers representing that level of detail should already reside on the graphics card, due to their high priority, unless the amount of available graphics memory is not sufficient to render the scene. The following vertex shader pseudo code properly geomorphs between the two positions, p_1 and p_2:

```
d = length(p1 - eye);
w = smoothstep(ds, de, d);
Pos = lerp(p1, p2, w);
```

Normals and texture coordinates are also geomorphed this way. Texture coordinates can be morphed because a consistent parametrization is generated for all LODs. Note that normals need to be renormalized after geomorphing.

The pixel shader performs two texture fetches from the two textures (corresponding to the two LODs), and then interpolates between them using the same interpolation weight w.

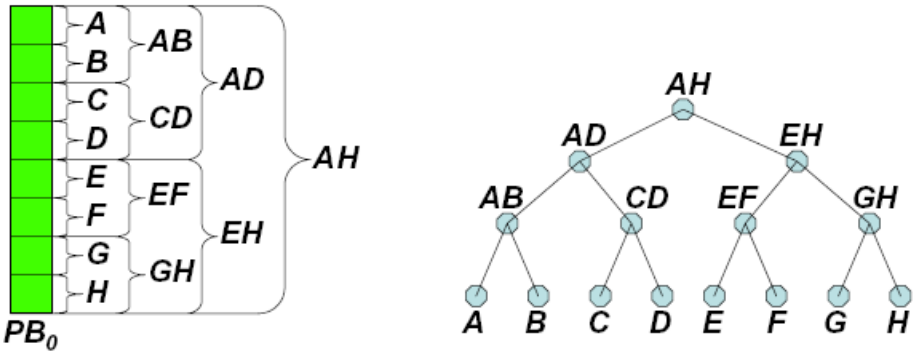


Figure 4*. The coarse buffer for the entire mesh (left) and its accompanying hierarchy, which is used to minimize the number of API draw calls.*

1.5.3 CBH rendering algorithm

In order to reduce the number of draw calls, group clusters at the coarsest LOD level. Since the coarsest buffers take little space, they can always be loaded into video memory.

As shown in Figure 4, the coarse buffers are grouped in an order dictated by the coarse buffer hierarchy (CBH). The CBH is a tree that contains all the clusters of the mesh as leaves. When rendering, the engine parses this hierarchy, and when it finds that all clusters below a certain node in the tree are in the coarsest level (by testing the node's bounding sphere), it renders all those clusters using a single draw call. For instance, node CD in the tree can render clusters C and D with a single draw call, since they are adjacent in the index buffer. The node simply stores the starting index and the number of triangles. The coarsest buffer does not contain parent information, since there are no coarser LODs.

In order to take advantage of early-Z culling hardware, we first render all nodes that are not in the coarsest level, since they represent the front-most geometry. We then render all coarse nodes by performing a depth-first CBH traversal. First, all leaves that are at the coarsest level are tagged "renderable." Then, the recursive algorithm first traverses the children of a particular node and, if they are both tagged renderable, the current node is tagged as renderable. Otherwise, if one of the children is tagged as renderable, it adds that child node to the list of nodes to be rendered. After the tree traversal is complete, all nodes in the list are rendered using a smaller number of API draw calls.

1.5.4 Level of detail control

In this section, we describe how we can automatically control the level of detail. This allows our method to work at satisfactory frame rates on a wide variety of GPUs.

The level of detail is adjusted by increasing and decreasing the *k* variable from Figure 2b. When this variable reaches its minimum value or $r + e_{min}$, the *s* variable can be decreased to a value smaller than 0 if the level of detail must be further decreased.

What remains to be determined is whether, given the current runtime state, we want to increase or decrease the level of detail. Ideally, we would simply use the frame rate to determine whether we want to increase or decrease the level of detail. For instance, if it is above 65 fps, we increase the LOD, and if it is below 55 fps we decrease the LOD. However, oftentimes frame rate only changes after an undesirable event has already taken place. For instance, if the active textures exceed the capacity of video memory, the frame rate suddenly drops. Ideally, we would like to prevent such drops. In order to achieve this, we propose setting a video memory upper bound, a system memory upper bound, a triangle-count upper bound, and a frame rate lower bound. The level of detail is constantly and slowly increased unless one of these bounds is violated, in which case, it decreases. Using these bounds on the best indicators of performance, we prevent such drastic frame rate changes. Naturally, the bounds can be tuned to the system configuration. Preprocessing is not affected by these changes and the same progressive buffer data structures can be used on all systems.

1.6 Preprocessing algorithm

In this section, we describe the steps involved in converting a triangle mesh into a progressive buffer. These preprocessing stages are mainly based on previous techniques for simplification and texture mapping of LOD representations.

1.6.1 Segmentation

To segment an input mesh into clusters, we use a voxelization of space. This ensures that the bounding spheres of all the clusters are bound based on the size of the voxels. This, however, may result in voxels that parameterize with high distortion, have annuli, and are composed of disconnected components. To address these problems, we further split the clusters into charts that are mostly planar and homeomorphic to discs. To achieve this, one can do this chartification manually or use one of several existing chartification algorithms (e.g., [Maillot93, Levy02, Sander03]). These charts can then be parametrized and their attributes can be stored in a single texture atlas for each cluster.

1.6.2 Hierarchy construction

To build the CBH, we start with all of the clusters as leaves of the tree and then perform a bottom-up greedy merge. All possible merging pairs are placed in a priority queue, sorted by smallest bounding sphere of the resulting merged cluster.

The above approach ensures that nearby clusters will be grouped together. Disconnected components can also be merged together, as long as they are using the same LOD band sizes (i.e., the same *k*, *s*, *e*, and *r*).

1.6.3 LOD-compliant parameterization

In order to texture map the mesh, each chart must parametrized into a disc and packed into atlases for each cluster. The parametrization restrictions for chart boundaries are discussed in [Cohen98] and [Sander01]. Any parametrization metric can be used (e.g., [Floater97, Sander01, and Desbrun02]). For our examples, we used the L^2 stretch metric from [Sander01], which minimizes sampling distortion from the 3D surface to the 2D domain.

The computed parametrization and resulting texture mip map will be applicable to all levels of detail. In the next section, we will describe how we simplify the mesh to guarantee this.

1.6.4 Progressive mesh creation

In this step, we must simplify the mesh, ensuring that the edge collapses keep the chart boundaries consistent and do not cause flips in UV space [Cohen98]. We use the half-edge collapse with the memory-less appearance preserving simplification metric as in [Sander01].

Each connected component must be simplified in unison. In order to achieve this, we simplify one cluster at a time until it reaches a specified user defined geometric error threshold. The order in which we simplify the clusters does not significantly affect the results, since we perform the same number of simplification passes on all the clusters between each pair of adjacent levels of detail. In order to consistently simplify the cluster boundaries, when simplifying each cluster, we also load the adjacent clusters and simplify the boundary vertices as well. However, we keep the neighboring cluster's interior vertices fixed. This allows all boundaries to be simplified and prevents boundary cracks. This method is related to that presented by [Hoppe98] for terrain simplification and by [Prince00] for arbitrary meshes.

1.6.5 Vertex and index buffer creation

Now that a *PM* has been constructed, we must extract meshes at different levels of detail and create corresponding vertex and index buffers. As mentioned previously, we chose each level of detail to have four times fewer vertices than the next finer one. For our examples, we picked five levels of detail, as that resulted in a sufficient range of LODs.

After extracting the meshes at the different LODs, we construct a set of vertex and index buffers for each cluster of each LOD. Each vertex will not only contain its own attributes (e.g., position, normal, texture coordinates), but it will also contain all the attributes of its parent vertex in the next coarser LOD. The *PM* hierarchy provides the ancestry tree. The parent vertex is its closest ancestor that is of a coarser level of detail. If the vertex is in the next coarser level, the parent is itself.

After these buffers are created, the vertices and faces are reordered using the method of [Hoppe99] to increase vertex cache coherency and thus improve rendering performance.

Vertex buffer compression. One limitation of this work is the doubled vertex buffer size that is required to render progressive buffers on current graphics architectures. In an attempt to offset this, we store the buffers with 28 bytes per vertex. Each of the two normals is stored with 10 bits per component for a total of 4 bytes per normal. Each set of 2D texture coordinates is stored using two 16-bit integers, thus occupying 4 bytes each. Finally, each of the two sets of 3D position coordinates is stored using three 16-bit integers each, for a total of 6 bytes each.

While the precision for the normals and texture coordinates is sufficient, 16-bit precision for the position in a large scene is not high enough. In order to address this, we initially considered storing the position as a delta from the chart's center position. However, this would result in dropped pixels at chart boundaries because of precision errors when decompressing vertices from adjacent charts whose positions must match exactly. In order to avoid this problem, we store the position modulo r (p % r), where r is the largest cluster bounding sphere. Given the stored position and the cluster's bounding sphere center position (which is made available through the constant store), the vertex shader can reconstruct the original position. Since the stored values (p % r) are identical for matching vertices on adjacent clusters, they are reconstructed to the exact same value.

1.6.6 Texture sampling

Next, we sample the texture images. In our cases, the textures were filled using data from other texture maps that use a different set of texture coordinates, or the textures were filled with normals from the original geometry. Next, we fill in samples near the boundaries using a dilation algorithm to prevent mipmapping and bilinear sampling artifacts. We then compute mipmaps for the textures. Each level of the mipmap corresponds to the texture that will be used as the highest mip level of a particular LOD.

Figure 5. *Visualization of the levels of detail.*

1.7 Implementation and results

We implemented progressive buffers in DirectX 9.0. Our experiments were made on a Pentium 4 2.5GHz machine with 1GB of memory and a Radeon X800 graphics board.

In order to analyze our algorithm, we preprocessed the Pillars model from Figure 7, a 14.4 million polygon textured model with 288 voxels. We used an input model that was split into parametrizable charts. The remaining steps were performed automatically as outlined in Section 5. Each voxel used a 512x512 texture image at the highest level of detail.

Figure 6 shows results for texture-mapped rendering with two different configurations for the Pillar model. The top row of graphs shows statistics using a fixed LOD band size. Note that the frame rate remains at about 60fps until the end of the fly path, when the camera starts moving away from the model. At that point, the frame rate increases, while memory usage, number of faces rendered and draw calls drop. The bottom row shows statistics for the same fly path, except that instead of using a fixed LOD band size, it tries to maximize the band size subject to a target of 60MB of video memory usage. As evidenced by the second chart, memory remains roughly constant around 60-70MB, causing the LOD band sizes to shrink and grow automatically to meet that memory requirement. This is a significantly lower memory footprint than on the first experiment, and therefore the number of rendered faces decreases by a factor of two and the frame rate increases to approximately 90fps.

The vertex caching optimization gave an improvement of almost a factor of three in rendering speed. Peak rates for our system were in the 60Mtri/sec range, which we consider high given that we are decompressing and geomorphing between two buffers in the vertex shader.

Figure 5 shows the different LODs. The lowest level of detail is shown in dark green. Coarse LODs, whose draw calls are grouped together using the CBH, are shown in

white. Figure 7 shows the LODs of the Pillars model from different vantage point. Note that the view point is close to the right-most pillar.

Figure 8 illustrates the importance of prefetching by graphing the number of clusters that were unavailable for rendering when prefetching from disk was disabled (no distinction between priority levels 0, 1 and 2).With prefetching enabled, and loading approximately 30% additional buffer data, all clusters are available.

Figure 1 shows an example of shadow-mapping on our system. A 16M triangle Parthenon mesh was used for this example. Shadows were cast with the coarse geometry by rendering the coarse mesh to the shadow map with just a single API draw call. That scene was rendered at 30fps.

Figure 9 shows examples of using our system for instancing. In this case, the progressive buffer is loaded into memory and instanced multiple times. The total number of virtual triangles is 45 million for the planes scene, and 240 million for the dragon scene. However, less than 1 million triangles are actually rendered when using the LOD system.

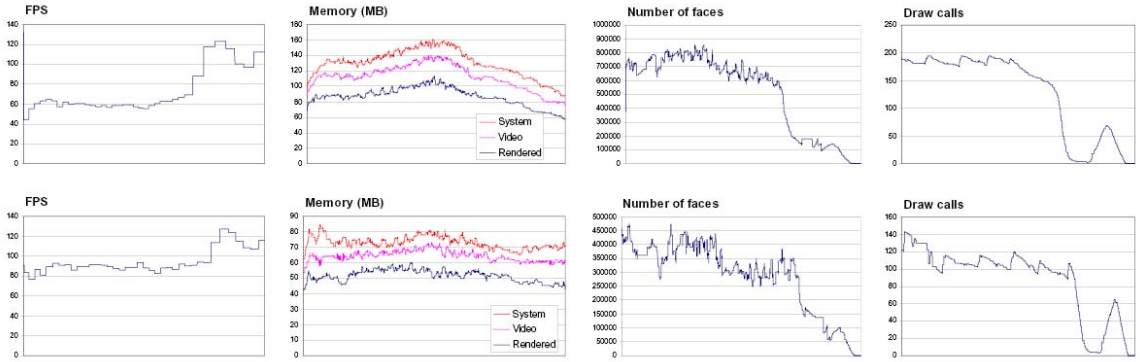


Figure 6. *Results for texture-mapped rendering. The results of using a fixed LOD band size, k, is shown in the top four graphs. The bottom four graphs show the results of automatic LOD control with a target of 60MB of video memory usage.*

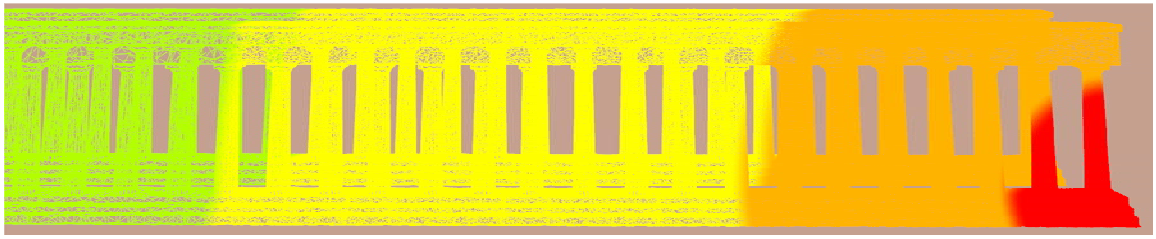


Figure 7. *Wireframe rendering of the color-coded LODs for the Pillars model. Note the significantly lower tessellation to the left, where it is far from the camera.*

1.8 Summary and future work

We presented a new data structure and algorithm for dynamic level of detail rendering of arbitrary meshes. We showed examples with out-of-core and instanced geometry. To our knowledge, our out-of-core view-dependent renderer is the first such system to provide smooth LOD transitions and texture mapping, the latter being a key component of real-time graphics. We presented experiments that demonstrate the viability of such a geometry and texture LOD approach. The method allows for scales well, is suitable for current and previous graphics hardware.

There are several interesting areas of future work:

Deformable models. Although we have not implemented progressive buffers for deformable models, our approach can be adapted to such a setting. The renderer would need to be able to track a bounding sphere for each model, and be aware of the maximum radius *r* of all clusters over all of their possible poses (that is necessary in order to set the geomorph range as shown in Figure 2b). All of these quantities can be preprocessed. The bounding spheres would need to be propagated up the CBH when they change.

Fly path lookahead. If the camera follows a specific known path, such as in a presentation, an architectural walkthrough or a demo, the prefetching algorithm can be adapted based on this fly path, since the application can easily determine which clusters will be needed ahead of time.

Tiled geometry. This technique can also be applied to a streaming world system, commonly used in video games. Only a single copy of each tile needs to be stored in video memory. The different tiles would have to be simplified in a consistent way, so that the vertices would match at the boundaries at all LODs.

Future architectures. In future graphics architectures, with performant texture fetch in the vertex shader, one could consider storing the parent index rather than the parent vertex attributes, thus reducing memory overhead.

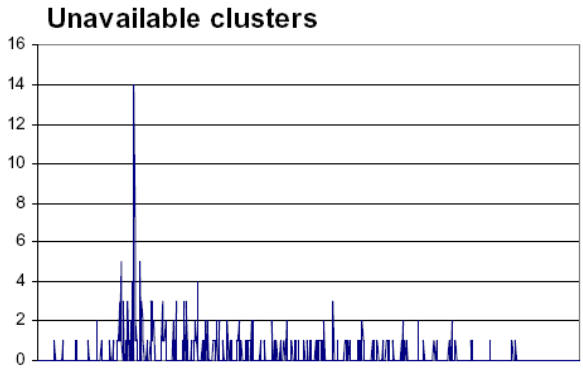
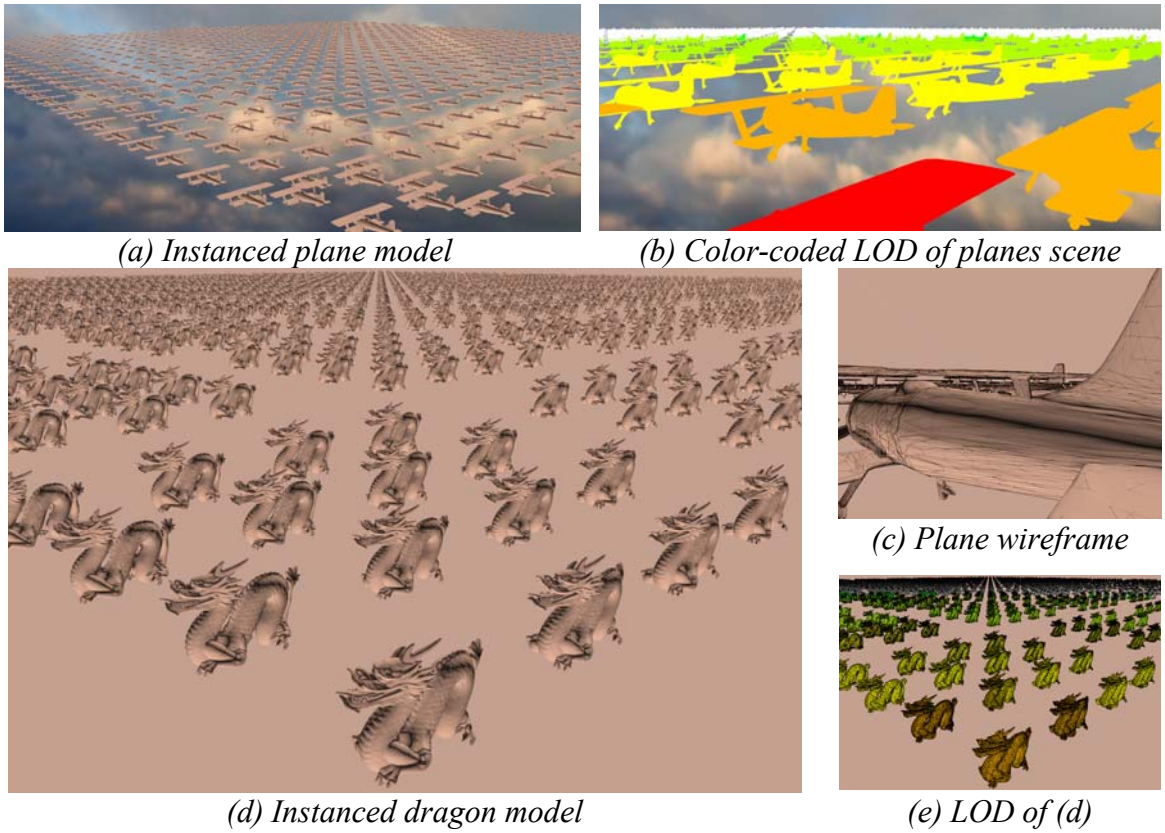


***Figure 8**. Number of unavailable cluster buffers when following a fly path with prefetching disabled. If prefetching is enabled, there were no unavailable buffers for the same fly path.*



(a) Instanced plane model *(b) Color-coded LOD of planes scene*

(c) Plane wireframe

(d) Instanced dragon model *(e) LOD of (d)*

***Figure 9**. Examples of instancing: 900 planes for a total of 45 million triangles and 1600 dragons for a total of 240 million triangles.*

1.9 Acknowledgements

We would like to thank Eli Turner for his help with the artwork. We thank Toshiaki Tsuji for the I/O and thread management code from his library, as well as help with optimizing the code. Finally, we thank Thorsten Scheuermann, John Isidoro and Chris Brennan for interesting discussions about the algorithm, and the reviewers for their comments and suggestions.

1.10 Bibliography

CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.. 2004. Adaptive TetraPuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. In Proc. SIGGRAPH ()

CHEN B., NGUYEN M. 2001. POP: A hybrid point and polygon rendering system for large data. In Visualization 2001

COHEN J., OLANO M., MANOCHA D. 1998. Appearance-preserving simplification. In Proc. SIGGRAPH

DEY T., HUDSON J.: PMR. 2002. Point to mesh rendering, a feature-based approach. In Visualization 2002

DESBRUN M., MEYER M., ALLIEZ P. 2002. Intrinsic parameterizations of surface meshes. Computer Graphics Forum 21, 209–218.

EL-SANA J., CHIANG Y.-J. 2000. External memory view-dependent simplification. Computer Graphics Forum 19, 3, 139–150.

EL-SANA J., VARSHNEY A. 1999. Generalized view-dependent simplification. Computer Graphics Forum 18, 3, 83–94.

FLOATER M. S. 1997. Parametrization and smooth approximation of surface triangulations. Computer Aided Geometric Design 14, 231–250.

GAIN J., SOUTHERN R. 2003. Creation and control of real-time continuous level of detail on programmable graphics hardware. Computer Graphics Forum, March

HOPPE H 1996. Progressive meshes. In Proc. SIGGRAPH

HOPPE H. 1997. View-dependent refinement of progressive meshes. In Proc. SIGGRAPH

HOPPE H. 1998. Smooth view-dependent level-of-detail control and its applications to terrain rendering. In Visualization 1998

HOPPE H. 1999. Optimization of mesh locality for transparent vertex caching. In Proc. SIGGRAPH

LUEBKE D., ERIKSON C. 1997. View-dependent simplification of arbitrary polygonal environments. In Proc. SIGGRAPH

LINDSTROM P. 2003. Out-of-core construction and visualization of multiresolution surfaces. In ACM 2003 Symposium on Interactive 3D Graphics.

LÉVY B., PETITJEAN S., RAY N., MAILLOT J. 2002. Least squares conformal maps for automatic texture atlas generation. In Proc. SIGGRAPH

MAILLOT J., YAHIA H., VERROUST A. 1993. Interactive texture mapping. In Proc. SIGGRAPH

PRINCE C. 2000. Progressive Meshes for Large Models of Arbitrary Topology. Master's thesis, Department of Computer Science and Engineering, University of Washington, Seattle

RUSINKIEWICZ S., LEVOY M. 2000. QSplat: A multiresolution point rendering system for large meshes. In Proc. SIGGRAPH

SANDER P. V., SNYDER J., GORTLER S., HOPPE H. 2001. Texture mapping progressive meshes. In Proc. SIGGRAPH

SANDER P. V., WOOD Z. J., GORTLER S. J., SNYDER J., HOPPE H. 2003. Multi-chart geometry images. In Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry Processing

ULRICH T. 2002. Rendering massive terrains using chunked level of detail control. SIGGRAPH 2002 Course Notes.

VADRAHAN G., MANOCHA D. 2002. Out-of-core rendering of massive geometric environments. In Visualization 2002

XIA J., VARSHNEY A. 1996. Dynamic view-dependent simplification for polygonal models. In IEEE Visualization '96 Proceedings

YOON S.-E., SALOMON B., GAYLE R., MANOCHA D. 2004. Quick-VDR: Interactive view-dependent rendering of massive models. In Visualization 2004.