Real-time Mesh Simplification Using the GPU

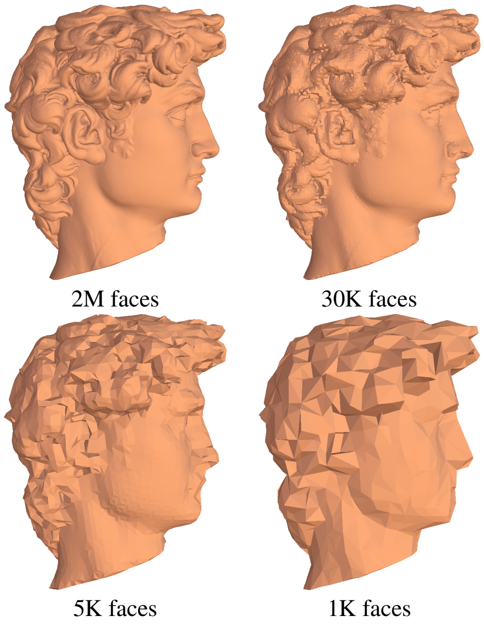
Christopher DeCoro*
3D Application Research Group, AMD / Princeton University

Natalya Tatarchuk†
3D Application Research Group, AMD

Abstract

Recent advances in real-time rendering have allowed the GPU implementation of traditionally CPU-restricted algorithms, often with performance increases of an order of magnitude or greater. Such gains are achieved by leveraging the large-scale parallelism of the GPU towards applications that are well-suited for these streaming architectures. By contrast, mesh simplification has traditionally been viewed as a non-interactive process not readily amenable to GPU acceleration. We demonstrate how it becomes practical for real-time use through our method, and that the use of the GPU even for offline simplification leads to significant increases in performance. Our approach for mesh decimation adopts a vertex-clustering method to the GPU by taking advantage of a new addition to the rendering pipeline - the *geometry shader* stage. We present a novel general-purpose data structure designed for streaming architectures called the *probabilistic octree*, which allows for much of the flexibility of offline implementations, including sparse encoding and variable level-of-detail. We demonstrate successful use of this data structure in our GPU implementation of mesh simplification. We can generate adaptive levels of detail by applying non-linear warping functions to the cluster map in order to improve resulting simplification quality. Our GPU-accelerated approach enables simultaneous construction of multiple levels of detail and out-of-core simplification of extremely large polygonal meshes.

Keywords: mesh decimation, mesh simplification, level-of-detail, real-time rendering, GPU programming



2M faces

30K faces

5K faces

1K faces

Figure 1: *Using our GPU-based mesh decimation algorithm we are able to generate* all *of these multiple levels of detail for this high resolution model of David's head an order of magnitude faster than producing even a single simplified level of detail on the CPU.*

1 Introduction

Advances in data acquisition (eg. [Levoy et al. 2000]) have resulted in the wide availability of massive polygonal datasets. Popularity of content authoring tools such as ZBrush® [Pixologic 2006] provide easy methods for creating extremely detailed art content with polygon counts in excess of several hundred million triangles. However, despite the tremendous leaps in GPU performance, interactive rendering of such massive geometry in computer games or other applications is still impractical due to the performance penalty for vertex throughput and the associated large memory storage requirements.

As a result, mesh simplification algorithms have been an active area of research for nearly a decade. Simplification of massive datasets demands computational efficiency as well as effective use of available memory. Current methods developed for mesh simplification and decimation ([Garland and Heckbert 1997], [Lindstrom and Turk 2000]) are designed with the CPU architecture in mind, and to this date polygonal simplification has not been adapted for the GPU compute model.

However, recent methods for interactive visualization of large multiresolution geometric models at interactive rates (such as [Sander and Mitchell 2005]) perform geomorphing on the GPU to render the objects. These methods require building hierarchical level-of-detail structures, which would benefit from dynamic mesh simplification algorithms. Games have been increasing the complexity of their massive worlds as well as employing a wider variety of lighting and shading techniques. Often, these interactive environments require multiple renderings of the same objects from different viewpoints in a single frame. The most common applications include rendering low-resolution versions of objects into a dynamic environment map for reflective or refractive effects; as well as shadow map rendering for multiple shadowing lights or cascading or omnidirectional shadow maps ([Gerasimov 2004]). Additionally, the introduction of streaming computational models such as [Buck et al. 2004] and [Peercy and Derstmann 2006] enables physics computations for in-game objects directly on the GPU. Low-resolution meshes and level-of-detail computations provide convenient optimization for GPU-based collision detection.

Traditionally, mesh simplification has been a slow, CPU-limited operation performed as a pre-process on static meshes. With increasing programmability in modern graphics processors, especially with the introduction of the latest GPU pipeline with geometry shaders ([Blythe 2006]), mesh simplification becomes amenable to the GPU compute model. In this paper we describe a method for mesh simplification in real-time including the following contributions:

- Reformulation of mesh simplification based on the vertex clustering algorithm of [Lindstrom 2000] adopted to the novel GPU pipeline.
- A general-purpose GPU octree structure
- Adaptive mesh simplification with constant memory requirements
- Importance-based detail preservation using non-linear warping

*e-mail:cdecoro@cs.princeton.edu

†e-mail:natalya.tatarchuk@amd.com

With our mesh simplification method we achieve equivalent quality to the CPU-based mesh decimation algorithms. We demonstrate that the use of our novel probabilistic octree data structure on the GPU effectively increases the grid resolution during simplification and thus yields resulting quality improvements while maintaining an identical memory footprint. Our application of mesh simplification enables processing of massive data sets directly on the GPU with over an order of magnitude increase in performance.

We discuss the existing work on mesh decimation in Section 2.1, and describe the new GPU programmable pipeline in section Section 2.2. Section 3 contains the details of the GPU-friendly mesh simplification algorithm, including the description of data structures. We present quantitative results of our algorithm (Section 4), and conclude with a discussion of potential future work (Section 5).

2 Background

2.1 Mesh Simplification

A wide range of algorithms have been presented to decimate a triangle mesh (introduced in [Schroeder et al. 1992]); that is, given an input mesh containing some number of triangles, produce a mesh with fewer triangles that well-approximates the original. Some of the earliest algorithms fall under the classification of *vertex clustering* [Rossignac and Borrel 1993]. In these, the bounding box of the mesh is divided into a grid (in the simplest case a rectilinear lattice of cubes), and all of the vertices in a given cell are replaced with a single representative vertex ("clustered"). Faces that become degenerate are removed from the resulting simplified mesh.

Other algorithms take an iterative approach, in which a series of primitive simplification operations are applied to an input mesh through intermediate simplification stages. The operations are usually chosen so as to minimize the incremental error incurred by the operation, though this is not always the case. An overview of mesh decimation algorithms can be found in [Luebke et al. 2002].

Perhaps one of the most commonly applied iterative decimation techniques is the QSlim algorithm [Garland and Heckbert 1997]. This algorithm iteratively applies the *pair collapse* operator, which replaces two vertices with one, causing neighboring faces to become degenerate. In order to select a collapsed pair of vertices from the potential candidates, QSlim defines the quadric error metric, which for a vertex v is defined as the point-plane distance from v to a set of associated planes:

$$
\begin{aligned}
f(v) &= \sum_{p \in \text{planes}(v)} (p^T v)^2 && (1)\\
&= v^T \left(\sum_{p \in \text{planes}(v)} p^T p \right) v && (2)\\
&= v^T Q_v v. && (3)
\end{aligned}
$$

Initially, planes(v) consists of the triangle faces adjacent to v in the original mesh. Applying the pair collapse $(a,b) \to c$, we assign planes(c) = planes$(a) \cup$ planes(b). We can remove the need for explicit representation of the set of associated planes by the use of the symmetric 4×4 matrix Q_v, known as the *error quadric*. The set-union operator then reduces to quadric addition.

In [Lindstrom 2000], the author observes that the vertex clustering operation is equivalent to performing the pair collapse operation of QSlim to each vertex in a cluster simultaneously. Thus, the quadric error metric can be used as a measure of mesh quality for such algorithms and, more importantly, can be used to directly compute the representative vertex of a cluster that minimizes the quadric error. The authors showed how this can be used to generate higher quality results then previously shown in a vertex clustering framework. The algorithm is as follows. For each triangle F,

```
1. Compute the face quadric $Q_F$
2. For each vertex $v \in F$
   (a) Compute the cluster $C$ containing $v$
   (b) Add $Q_F$ to the cluster quadric $Q_C$
3. If $F$ will be non-degenerate, output $F$
```

The algorithm acts on each face independently, and stores only the cluster grid as a representation of the intermediate mesh. Importantly, each vertex will access a single location in the grid. This locality and data-independence allows the algorithm to be efficient in the context of out-of-core simplification. For the same reasons, such algorithms are also ideal for the stream computing architecture of the GPU, and our work implements this approach.

While a uniform grid enforces uniform level-of-detail across the output mesh, later work has relaxed this restriction in favor of adaptive approaches using octrees [Schaefer and Warren 2003] and BSP trees [Shaffer and Garland 2001]. While such data structures are not directly suitable for GPU implementation due to their requirements of dynamic memory and sequential writes, we demonstrate a similar GPU structure that maintains much of their advantages.

2.2 GPU Programmable Pipeline

The programmable vertex and pixel engines found in recent GPUs execute shader programs, containing arithmetic and texturing computations, in parallel. The *vertex shader* is traditionally used to perform vertex transformations along with per-vertex computations. Once the rasterizer has converted the transformed primitives to pixels, the *pixel shader* can compute each fragment's color.

This pipeline is further extended in the upcoming generation of DirectX®10 hardware, introducing an additional programmable *geometry shader* stage. This stage accepts vertices generated by the vertex shader as input and, unlike the previous stage, has access to the entire primitive information as well as its adjacency information. This enables the per-face computation of face quadrics required by the vertex clustering algorithm of [Lindstrom 2000].

The geometry shader also has the ability to cull input triangles from the rendering stream and prevent their rasterization, clearly a necessary component for mesh decimation. Finally, the DirectX 10 *stream-out* option allows reuse of the result of geometry processing by storing output triangles in a GPU buffer. This buffer may be reused arbitrarily in later rendering stages, or even read back to the host CPU. Our method implements the simplification algorithm taking advantage of the novel geometry shader stage functionality for computation of the quadric map for each face, and using the stream-out feature for storing and later rendering of the simplified geometry.

3 Algorithm

We will first present the basic structure of our GPU simplification system (Section 3.1). However, this leaves flexibility in several components, most notably the structure of the clustering grid. The most straightforward implementation will use a uniform rectangular grid; however, there are advantages to both using a grid with non-uniform geometry, as deformed using a warping function (Section 3.2) and a non-uniform connectivity as specified with a probabilistic octree structure (Section 3.3).

3.1 GPU Mesh Simplification Pipeline

Our algorithm proceeds in 3 passes and requires the input mesh to be submitted twice through the rendering pipeline. We encode the mapping from cluster-cell index to cluster quadric in a render target (an off-screen buffer), used as a large 2-dimensional array which we will refer to as the *cluster-quadric map*. The quadric

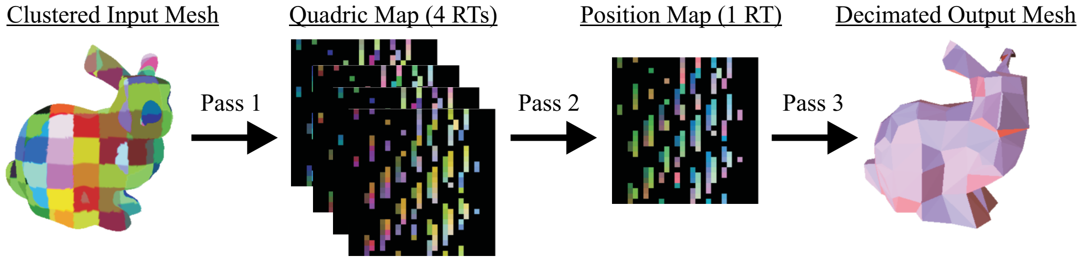


Figure 2: GPU Simplification Pipeline. *The original mesh (left) is subdivided into clusters according to a* $9\times9\times9$ *grid. In Pass 1, we compute the cluster quadrics for each grid cell; the output (shown) is a set of render targets that contain the quadrics in the pixel values. Pass 2 minimizes the quadric error function to compute the optimal representative positions for each cluster. Finally, Pass 3 uses this to output the final, decimated mesh.*

accumulation operation (Equation 2) can be mapped to the highly efficient additive blend. Because the algorithm accesses each mesh triangle only once per pass, it is not necessary to store the entire input mesh in GPU-resident memory (the storage requirements are a function of the output mesh size only), allowing our algorithm to efficiently process meshes of arbitrary size. The passes are as follows, as illustrated in Figure 2:

(Pass 1) Cluster-quadric map generation. Given the source mesh and its bounding box as input, as well as a user-specified number of subdivisions along each dimension, we render the input mesh as points. We then assign a unique ID to each cluster cell, and we treat the render target as a large array that is indexed by cluster ID. Each array location stores the current sum of the error quadric for that cell (10 floats for the 4x4 symmetric matrix), the average vertex position within that cell (3 floats) and the vertex count.

The vertex shader computes the corresponding cluster for each vertex, and its implied position in the render target. The geometry shader, which has access to all vertices in the triangle, uses the world positions to compute the face quadric for the triangle, and assigns that value to each output vertex to be accumulated in the texture map by the pixel shader, which simply propagates the computed colors, with additive blending enabled.

(Pass 2) Computation of optimal representative positions. Using the cluster-quadric map from Pass 1, we compute the optimal representative vertex position for each cluster. Note that we could do this on the next pass (generation of the decimated mesh) but we choose to do this in a separate pass so that the relatively expensive computation can be performed exactly once per cluster with higher parallelism.

We render a single full-screen quad the size of the cluster map render targets into another render target of equal size. In the pixel shader, we retrieve the values of the error quadric from the render target textures, and compute the optimal position by solving the quadric error equation with a matrix inversion ([Garland and Heckbert 1997]). If the matrix determinant is below a user-specified threshold (currently $1e-10$) we assume that the quadric is singular and fall back to using the average vertex position. The position is saved into a render target, and used in the next pass.

(Pass 3) Decimated mesh generation. We send the original mesh through the pipeline a second time, in order to remap vertices to their simplified positions, and cull those triangles that become degenerate. The vertex shader again computes the corresponding cluster for each vertex, and the geometry shader determines if the three vertices are in different clusters, culling the triangle if they are not. Otherwise, the geometry shader retrieves the simplified positions from the output of Pass 2, using these as the target positions of the new triangle, which is streamed out to a GPU buffer for later use.

Multiple Levels-of-Detail. We can compute multiple levels of detail for the same mesh without repeating all three passes. When the resolution of the sampling grid is reduced by half, we can omit Pass 1, and instead create the quadric cluster map by appropriate downsampling of the higher-resolution quadric cluster map. Pass 2 operates as before; however Pass 3 can use the previously simplified mesh as its input (rather than the full resolution input mesh) as the connectivity will be the same. This allows the construction of a sequence of LODs significantly faster than incurring the full simplification cost for each LOD independently.

3.2 Non-Uniform Clustering Using Warping Functions

We can achieve a higher level of adaptivity in the simplification process using a smooth, non-rigid warping function to deform the cluster grid. Applying such function during cluster map generation leads to a higher sampling rate in the desired regions. We can apply arbitrary non-linear functions as the warping guide during decimation. For practical purposes, we in fact apply the inverse warp function to the vertices themselves when computing cluster coordinates, which is equivalent. The only change in our simplification pipeline is the computation of cluster ID from vertex position. The positions used for the computation of error quadrics need not be altered, nor the storage of the grid.

One application of this approach is for *view-dependent simplification*, whereby the algorithm preserves greater detail in regions of the mesh closer to the viewer, as defined by the provided warping function. The simplest and most efficient function we can apply is the current frame's world-view-projection transformation into screen space. This is equivalent to performing a projective warp on the underlying cluster grid. We show a comparison of view-dependent and view-independent simplification in Figure 3. Application of this warping function can be meaningful for simplification on animated meshes in real-time scenarios.

Another application is for *region-of-interest simplification*, where the user (such as an artist) selects regions to be preserved in higher detail (as in the user-guided simplification approach of [Kho and Garland 2003]). In Figure 5, the model is simplified using both a uniform and adaptive grid. In order to preserve detail around a particular region (for our example, the head), we simplify a warped version of the mesh, which provides higher sampling around the region of interest.

In order to guide the region of interest simplification, we use a Gaussian weighting function $f(x)$ centered at the point of interest. We are seeking to derive a warping function which respects the weights specified by $f(x)$. Thus we would intuitively prefer a function $F(x)$ such that points with larger values of $f(x)$ are spaced farther from their neighbors. Additionally, $F(x)$ is one-to-one, and spans the range $(0,1)$. We can derive functions as follows, and show examples of varying parameters in Figure 4:

$$f_{\mu,\sigma,b}(x) = (1-b)G_{\mu,\sigma}(x)+b \tag{4}$$

$$\hat{F}_{\mu,\sigma}(x) = \int_{-\infty}^{x} G_{\mu,\sigma}(t)dt \tag{5}$$

$$= \frac{1}{2}\left(1+\operatorname{erf}\frac{x-\mu}{\sigma\sqrt{2}}\right) \tag{6}$$

$$F_{\mu,\sigma,b}(x) = \frac{\hat{F}_{\mu,\sigma}(x)-\hat{F}_{\mu,\sigma}(0)}{\hat{F}_{\mu,\sigma}(1)-\hat{F}_{\mu,\sigma}(0)}(1-b)+bx \tag{7}$$

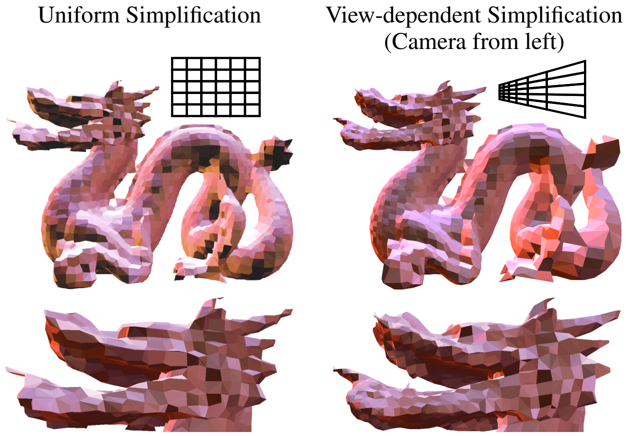


Figure 3: View-independent vs. View-dependent Simplification. *The dragon model is simplified using both methods, with the camera position to the left of the object in the view-dependent case. Note that regions closer to the camera are preserved in higher detail; note especially the detail preserved on the face in the callout.*

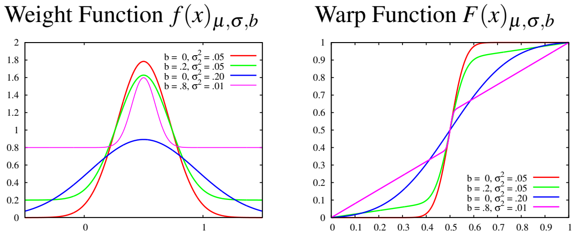


Figure 4: Warping Functions. *We show example weighting functions at various parameters, and their corresponding warping functions. An input vertex coordinate (x-axis) will be mapped to a location in the warped mesh (y-axis). Note how values near the mean ($\mu = 0.5$) are mapped to a wider range in the output than those points farther away.*

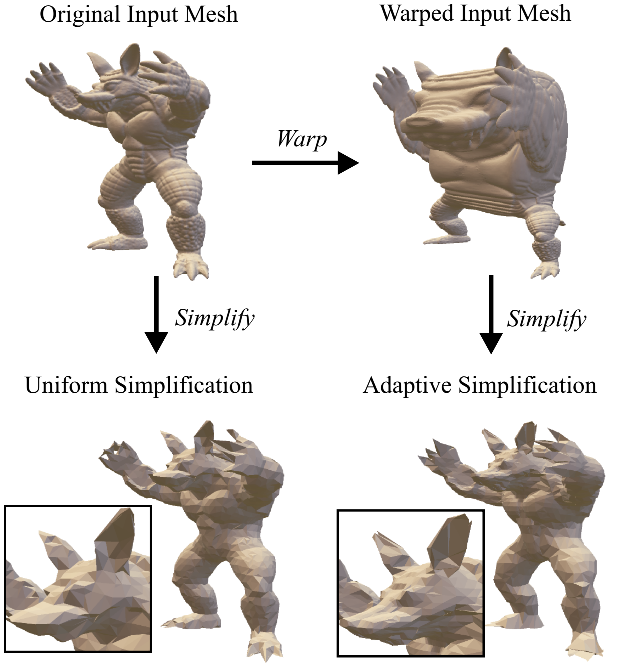


Figure 5: Area-of-interest Simplification. *We adaptively simplify the model to preserve detail around the head, which is performed by warping the mesh appropriately, and clustering the result.*

In this definition, $G_{\mu,\sigma}(x)$ is the standard normal distribution, and $\text{erf}(\cdot)$ is the Gauss error function. We define a bias parameter b, which sets a minimum weighting for regions outside the area of interest; setting $b = 1$ is equivalent to uniform sampling. Note that $\hat{F}$ can be viewed as the cumulative distribution function corresponding to f (see [Ross 2003] for an overview), which we then translate and scale to the unit square to produce the warping function $F(x)$. Note that the function in Equation 7 is currently limited to warps that are separable in x, y and z. However the method supports more general warps, such as those defined by arbitrary splines or radial basis functions.

3.3 Probabilistic Octrees

The use of a uniform grid requires that the user fix the resolution before simplification, and does not easily allow for uneven levels of detail across the resulting simplified mesh (notwithstanding the previously discussed use of warping functions). Additionally, because of the need for direct, constant-time access to the grid, the data is stored in a large fixed-allocation array so that the address can be computed directly, regardless of the number of clusters that are actually occupied.

We propose to address these concerns using a multi-resolution grid with multiple levels, from lowest to finest resolution, where each level has twice the detail of the previous in each dimension ("octree subdivision"). Adaptive octrees were introduced for vertex clustering mesh simplification in [Schaefer and Warren 2003], in the context of out-of-core simplification. Each grid cell will then store the estimate of the error quadric for a cluster of a certain scale. When mapping an input vertex to a cluster in the decimation pass, the representation allows the algorithm to use finer scales in areas with greater detail.

Additionally, rather than allocating all of the potential grid cells for a given level, we allocate a fixed amount of storage, and use a spatial hash function to access the elements in constant time. This implies that not all clusters will be stored, but that there is only a probability of storage, which is expected to be the ratio of stored clusters to allocated space. However, the hierarchical structure allows for a graceful degradation by maintaining a lower resolution estimate.

As this is similar to the commonly-used octree, we will refer to our structure as a *probabilistic octree*. This structure avoids the sequential read-modify-write access and dynamic memory used in traditional octrees, and is well-suited for the GPU. Note that this general-purpose structure is not limited to our application of vertex clustering, as we discuss in Section 5.

Operations. The octree defines the high-level ADDVERTEX(v) and FINDCLUSTER(v) operations, used in Pass 1 and 3, respectively, which act on vertex positions. These use the low-level operations WRITE(k,d) and $d =$ READ(k) to write or read the data value d into the array render-target at location k. We write to the render targets with additive blending enabled, so as to accumulate the quadric values in a cluster.

Probabilistic Construction. When creating the tree (Pass 1), we use the ADDVERTEX operation on each vertex v to insert its quadric into the octree. In a tree with maximum depth l_{max}, a vertex has l_{max} potential levels in which it can be placed. One implementation of ADDVERTEX(v) makes l_{max} passes to assign v to each possible level, resulting in the most accurate construction of the entire tree. However, the decimation time will grow proportionally.

Instead, we can think of the cluster quadric Q_C as being the result of integrating the quadrics Q_x at each point x on the surface contained in C, scaled by the differential area dA. In performing the vertex clustering algorithm on a finitely tessellated mesh, we approximate this quantity by taking a sum of the vertex quadrics

Q_v contained in C, which themselves are computed from their adjacent face quadrics Q_f and corresponding areas A_f.

$$Q_c = \int_{x\in C} Q_x \, dA \approx \sum_{v\in C} \sum_{f\in \mathrm{adj}(v)} Q_f \frac{A_f}{3} \tag{8}$$

However, we can make this approximation with fewer samples than the entire set of those available. In a highly tessellated mesh, each cluster will have many samples with which to estimate the cluster quadric; therefore, we propose to *randomly* select the level of each vertex, and assign it to the array accordingly using WRITE. Due to the hierarchical nature of the tree, the higher levels (larger scales) contain more samples, and a better estimate of the cluster quadric can be made with a smaller fraction of the total vertices than for lower levels. Instead of a uniform random distribution, we choose the level according to a probability mass function that grows exponentially with increasing level. As there are exponentially fewer nodes at lower levels, the sampling rate remains roughly equal. As with any Monte Carlo approximation, more samples (equating to more passes per vertex) will lead to a better approximation, but this is not necessary for highly tessellated models, and an octree can be constructed in a single pass.

Probabilistic Storage. As with the uniform grid, we store the octree levels in render targets, using them as an array; we divide the array into sections for each level. Once ADDVERTEX(v) has selected the level in which to store v, it can compute the appropriate array index k as if the cluster was densely stored, invoking WRITE(k,v) to store the value. To achieve sparse storage, we allocate fewer nodes than would be necessary for storage of the entire level. WRITE uses a uniformly distributing hash function to assign a storage location to k. Therefore, the probability that WRITE(k,d) will be successful is expected to be equal to the percentage of occupied nodes in that level, and this probability can be a parameter to the algorithm, with the allocation size adjusted accordingly. Note that if the sparse storage property of the octree is not important for the application, we can allocate the array such that the storage probability at each level is 1.

Accessing the tree. After the tree is created in Pass 1, we use FINDCLUSTER(v) in Pass 3 to determine a corresponding cluster and scale for v, which is then mapped to the representative vertex. FINDCLUSTER uses a user-specified error tolerance to select a the appropriate scale. We can implement this by performing a traversal from the root of the tree (or from a node of user-specified depth $l_{min} > 0$, to avoid traversing very low detail regions of the tree). The function must keeping in mind that a cluster at any given scale may be unoccupied (no vertex was assigned; indicated by initializing the render target to a flag value) or that there may be another cluster assigned to the same position as a result of a hash collision (discussed shortly). By varying the error threshold, we can produce multiple LODs without creating a new octree.

By using a multi-resolution structure, we mitigate the effect of missing values. The probabilistic implementation of ADDVERTEX maintains the property that each point in space is represented by a node in the structure; only the scale is uncertain. If a node is absent at a particular scale, there is a high probability that the parent node will be available, causing the algorithm only to fall back to a slightly less detailed approximation of that point.

We can accelerate traversal by using a binary search across the different scales. As the tree depth is $O(\log N_C)$, where N_C is the total number of clusters, a (probabilistic) binary search over the depth reduces lookup time complexity to $O(\log\log N_C)$.

Detecting hash collisions. Because we implement each tree level with a hash table, there exists the possibility of hash collisions, where two nodes map to the same address in the array. A common solution is for the WRITE(k,d) operation to record the key k along with the data in storage, allowing READ(k) to determine whether or not it has encountered a collision by a comparison. In our application, we are not able to use this direct approach due to limitation on fixed function additive blending required to accumulate the quadrics. Therefore, we use the max blending mode for the alpha component only and write k to one render target, and $-k$ to the other (effectively using the second render target to perform a *min* operation). The READ(k) operation will check that the values are equal to k and $-k$, respectively. We propose hardware extensions to enable accurate hash collision processing in Section 5.

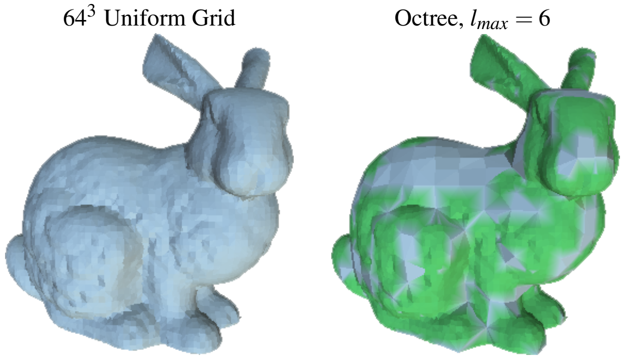


Figure 6: Uniform Grids vs. Adaptive Octrees *The image on the right is simplified with a uniform* 64^3 *grid, resulting in 13K triangles. The image on the left is simplified using a probabilistic octree with a depth of up to 6 (equivalent to the* 64^3 *grid). Through adaptive simplification, we are able to preserve the same detail as the uniform grid in critical regions, such as around the edge of the leg, the ear, and the eye, while reducing the total triangle count to 4K triangles, and using less memory at runtime. Green areas in the right image are those in higher detail.*

4 Results and Applications

The most significant contribution of our system, as opposed to in-memory, CPU-based simplification, is the dramatic increase in speed. We show the timing results on a set of input meshes for both the CPU and GPU implementations of the algorithm in Table 1. Results are shown for a PC with dual Intel®Pentium®4 CPUs (3.20GHz), 1GB of RAM and a preproduction ATI Radeon DirectX®10 generation GPU, collected on Windows Vista®. Note that the CPU implementation was implemented in an efficient manner for optimal performance. The results show that the GPU implementation is able to produce simplification rates of nearly 6 million triangles per second, as compared to a throughput of 300K triangles per second on the CPU.

Model	Faces	CPU	GPU	CPU:GPU
Bunny	70K	0.2s	0.013s	15:1
Armadillo	345K	1.2s	0.055s	22:1
Dragon	879K	1.9s	0.117s	16:1
Buddha	1M	2.5s	0.146s	17:1
David-head	2M	6.8s	0.322s	21:1
Atlas	4.5M	14.8s	0.741s	20:1
St-Matthew	7.4M	24.6s	1.18s	21:1

Table 1: *Performance results for GPU real-time mesh simplification versus CPU mesh simplification. Note that all but the largest results on the GPU were rendered at highly interactive real-time rates, achieving a simplification throughput of nearly 6M faces/second*

In our experiments with probabilistic octrees, we have found that there is a slight penalty to their use; running at about 80% of the full speed. Collisions are not a major factor. For a tree with half the total storage allocated, we found approximately 0.1% of the clusters used with the bunny caused a hash collision. While this does increase as the size of allocated memory decreases, it is not highly significant. For about 8% of the total storage allocated, we have less than 10% collisions.

We can mitigate the speed impact of octrees when creating multiple LODs. We have also found that the most significant overhead in the simplification process is Pass 1, which tends to take 60% to 75% of the total simplification time. Once the octree has been created, we have observed that the generation of a new mesh skips this amount of time overhead (it also avoids recomputing the optimal positions in Pass 2, but this does not use significant time; less than 10%). We note also that the implementation of octrees in our system has not been as highly optimized as the regular grids, and we expect that the performance could be nearly at parity.

5 Conclusions and Future Work

We have presented a method for mesh simplification on the novel GPU programmable pipeline and demonstrated how mesh decimation becomes practical for real-time use through out approach. We have adopted a vertex-clustering method to the GPU and described a novel GPU-friendly data structure designed for streaming architectures called the probabilistic octree. Our approach can be used to simplify animated or dynamically (procedurally) generated geometry directly on the GPU, or as a load-time algorithm, in which geometry is reduced to a level of detail suitable for display on the current user's hardware at the start of the program or change in scene. Simplified meshes can be used for collision detection and other purposes. Additionally, our technology allows an artist to rapidly create multiple levels of detail and quickly select those appropriate for the application.

The use of non-uniform grids has a much broader range of uses than those presented here. One potential application is the use of multi-pass simplification, such as that presented in [Shaffer and Garland 2001], which first clusters vertices on a uniform grid to make an estimate of surface complexity, then uses this estimate to produce an adaptive grid (represented with a BSP tree). This same approach could be used to generate the warping function for a non-uniform grid, achieving adaptive simplification while remaining applicable to the streaming architecture.

We also expect that the probabilistic octree structure would be useful in other applications. This would allow for dynamic octree generation that could be used in the context of collision detection, ray tracing, frustum and back-face culling, and other applications for which octrees are commonly used. While we were unable to provide more than a cursory overview of probabilistic octrees in this paper, we plan on producing a more formal analysis of the structure in future work.

Hardware extensions: geometry shader stage. Current design of the geometry shader stage can only generate individual primitives or lists via stream out. Once generated, there is no vertex reuse due to lack of associated index buffers for GPU-generated data. This affects performance for post-stream out rendering passes and triples required resulting vertex memory footprint. Decimated mesh rendering performance would be improved if the API were extended to allow indexed stream out, i.e. the ability to stream out primitives and their indices from each GS invocation. This can be accomplished by providing an additional mode for geometry shader stage - *indexed stream out* which is fully orthogonal to the regular stream out path. Each geometry shader will have to specify the actual number of output vertices at the beginning of each shader prior to emitting. Thus the hardware would be able to allocate appropriate storage for each invocation, as well as allocate the number of indices generated by this invocation.

Hardware extensions: Programmable blend stage. We utilize fixed function additive blend for accumulating cluster quadrics during quadric map computations. However, fixed function additive blending prevents us from implementing accurate hash collision handling for probabilistic octrees. We would like to see a programmable blend stage extending current functionality beyond simple fixed function stage, similar to simple pixel shader functionality. With flow control and simple ALU computations we would be able to handle hash collisions accurately. Thus, if the octree node being stored has lesser priority than the node currently stored in the destination buffer, it could be culled by the blend shader. An octree node with greater priority would overwrite the value already stored, and an equal priority octree node would simply accumulate.

Acknowledgments

We would like to thank the Stanford Computer Graphics Lab for providing geometry data, including models from the Digital Michaelangelo project. We would also like to thank Szymon Rusinkiewicz of Princeton University for providing useful feedback and advice on early versions of this paper, as well as providing additional funding.

References

BLYTHE, D. 2006. The Direct3D 10 system. *ACM Trans. Graph. 25*, 3, 724–734.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph. 23*, 3, 777–786.

GARLAND, M., AND HECKBERT, P. S. 1997. Surface simplification using quadric error metrics. *Proceedings of ACM SIGGRAPH 1997*, 209–216.

GERASIMOV, P. 2004. Omnidirectional shadow mapping. In *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, R. Fernando, Ed., vol. 20. Addison-Wesley, ch. 12, 193–204.

KHO, Y., AND GARLAND, M. 2003. User-guided simplification. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, ACM Press, New York, NY, USA, 123–126.

LEVOY, M., PULLI, K., CURLESS, B., RUSINKIEWICZ, S., KOLLER, D., PEREIRA, L., GINZTON, M., ANDERSON, S., DAVIS, J., GINSBERG, J., SHADE, J., AND FULK, D. 2000. The Digital Michelangelo Project: 3D scanning of large statues. In *Proceedings of ACM SIGGRAPH 2000*, 131–144.

LINDSTROM, P., AND TURK, G. 2000. Image-driven simplification. *ACM Transactions on Graphics 19*, 3, 204–241.

LINDSTROM, P. 2000. Out-of-core simplification of large polygonal models. In *Siggraph 20000, Computer Graphics Proceedings*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., 259–262.

LUEBKE, D., WATSON, B., COHEN, J. D., REDDY, M., AND VARSHNEY, A. 2002. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA.

PEERCY, M., S. M., AND DERSTMANN, D. 2006. A performance-oriented data parallel virtual machine for gpus. In *ACM SIGGRAPH sketches*, ACM Press, New York, NY, USA.

PIXOLOGIC, 2006. ZBrush. http://www.pixologic.com/zbrush/home/home.php.

ROSS, D. 2003. *Probability Models, 8th Edition*. Elsevier Academic Press, San Diego, CA USA.

ROSSIGNAC, J., AND BORREL, P. 1993. Multi-resolution 3D approximations for rendering complex scenes. *Modeling in Computer Graphics: Methods and Applications* (June), 455–465.

SANDER, P. V., AND MITCHELL, J. L. 2005. Progressive Buffers: View-dependent geometry and texture for lod rendering. In *Symposium on Geometry Processing*, 129–138.

SCHAEFER, S., AND WARREN, J. 2003. Adaptive vertex clustering using octrees. In *Proceedings of SIAM Geometric Design and Computation 2003*, SIAM, New York, NY, USA, vol. 2, 491–500.

SCHROEDER, W. J., ZARGE, J. A., AND LORENSEN, W. E. 1992. Decimation of triangle meshes. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 65–70.

SHAFFER, E., AND GARLAND, M. 2001. Efficient adaptive simplification of massive meshes. In *VIS '01: Proceedings of the conference on Visualization '01*, IEEE Computer Society, Washington, DC, USA, 127–134.