Parallel View-Dependent Level-of-Detail Control

Liang Hu, Pedro V. Sander and Hugues Hoppe

Abstract—We present a scheme for view-dependent level-of-detail control that is implemented entirely on programmable graphics hardware. Our scheme selectively refines and coarsens an arbitrary triangle mesh at the granularity of individual vertices to create meshes that are highly adapted to dynamic view parameters. Such fine-grain control has previously been demonstrated using sequential CPU algorithms. However, these algorithms involve pointer-based structures with intricate dependencies that cannot be handled efficiently within the restricted framework of GPU parallelism. We show that by introducing new data structures and dependency rules, one can realize fine-grain progressive mesh updates as a sequence of parallel streaming passes over the mesh elements. A major design challenge is that the GPU processes stream elements in isolation. The mesh update algorithm has time complexity proportional to the selectively refined mesh, and moreover can be amortized across several frames. The result is a single standard index buffer than can be used directly for rendering. The static data structure is remarkably compact, requiring only 57% more memory than an indexed triangle list. We demonstrate real-time exploration of complex models with normals and textures, as well as shadowing and semitransparent surface rendering applications that make direct use of the resulting dynamic index buffer.

Index Terms—Level-of-detail, efficient rendering, multiple-GPU techniques.

1 Introduction

Efficient rendering of complex geometric models has been an active research area for over a decade, with many level-of-detail (LOD) representations offering tradeoffs in fidelity and speed. Intuitively, models that are far away are rendered as coarser meshes. The most challenging setting is a large-scale model that cannot easily be partitioned into independent parts. Its surface mesh must be adapted in real-time to selectively refine the nearby visible regions. Several solutions to this view-dependent LOD problem have been explored previously, as reviewed in Section 2.

Many early techniques for view-dependent LOD provide fine granularity via vertex splits and edge collapses (Figure 1a). These techniques require sequential CPU algorithms to update pointer-based data structures with intricate dependencies, and are therefore difficult to implement efficiently on present system architectures. First, the sequential algorithms cannot benefit from the available GPU parallelism. And second, because the mesh is modified in system memory, a duplicate copy must be transferred to video memory at every frame for rendering. As a consequence, more recent LOD research instead focuses on coarsely partitioning the mesh and storing static buffers in video memory. The sacrifice in LOD granularity is offset by an improvement in frame rate.

Contribution: In this paper, we present a framework that allows vertex-level LOD updates to be performed in parallel on the GPU, with all the data residing in video memory. Our approach makes use of the programmable geometry shader introduced in recent GPUs, which is able to process a stream of elements in parallel.

Parallel view-dependent LOD control is a daunting problem. We must perform "surgery" over an entire mesh as a parallel process, to both coarsen and refine it adaptively, while maintaining consistent connectivity, i.e. a watertight surface without holes, T-junctions, or duplicate faces. Moreover, the computational model for GPU streaming has major restrictions. Each stream element is processed in complete isolation from all others, and a streaming pass cannot both read from and write to the same memory buffer. In light of this, we were initially unsure that a practical solution would be realizable.

Approach overview: Previous fine-grain LOD algorithms all use a traditional mesh data structure, in which each triangle face contains references to its 3 neighboring faces. We soon discovered that such a structure poses an obstacle because updating the references in parallel is unwieldy. Face references cannot be read and written in the same pass, and become stale after a single modification. One of our insights is to design new data structures that obviate the need for face neighbor references. In turn, the absence of neighbor information requires new dependency conditions for vertex split and edge collapse operations within the progressive mesh structure (Section 3).

Our runtime representation has two main parts (Section 4). A set of static structures encode a progressive mesh hierarchy for the detailed input mesh. In addition, a set of dynamic structures encode the active, selectively refined mesh. A unique aspect is that the active mesh is fully specified by a stream of vertices. This stream contains all vertices "above" the active frontier in the vertex hierarchy, and simultaneously identifies both (1)

- *L. Hu and P. V. Sander are with the Department of Computer Science and Engineering, the Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong.*
  E-mail: nickyhu@cse.ust.hk, http://www.cse.ust.hk/~psander/
- *H. Hoppe is with Microsoft Research, One Microsoft Way, Redmond, WA 98052-6399, USA.*
  E-mail: see http://research.microsoft.com/~hoppe/

the active vertices, which are on the frontier, and (2) the active faces, which are created by vertices split above the frontier.

We perform a set of parallel streaming passes to update the vertex stream as the view parameters change, and to create an index buffer for rendering (Section 5). All per-frame computations are performed in time proportional to the complexity of the active mesh rather than the fully detailed input mesh. Additionally, we automatically partition this computation across multiple frames to maintain a fixed frame rate (Section 6).

2 Previous work

There is vast literature describing different simplification and LOD management strategies, as well as different error metrics and criteria. For a comprehensive overview, including the original CPU-based geometry update methods, refer to Luebke et al. [1].

Early CPU methods for view-dependent LOD over arbitrary meshes use fine-grain updates based on vertex splits and edge collapses [2],[3] or octree-based vertex clustering [4]. As the GPU has become more powerful, more recent methods typically maintain a fixed set of static buffers in video memory, and switch or geomorph among them [5],[6],[7].

An interesting special case is that of terrain, in which vertices lie on a regular 2D grid. Methods specialized for terrain grids are based on both fine-grain mesh updates [8],[9],[10], and coarse-grain updates [11],[12]. More recent work renders nested regular grids about the viewer [13]. Our work differs in that it handles the general case of arbitrary meshes.

Programmable graphics hardware has allowed many surface tessellation approaches to migrate to the GPU, including isosurface extraction [14],[15], parametric patches [16],[17],[18],[19], subdivision surfaces [20],[21], and procedural detail [22],[23]. Whereas these approaches are typically used to amplify coarse geometry, our refinement framework is designed to exactly reproduce an arbitrary detailed mesh.

Two recent GPU-based LOD techniques are most closely related to our goal of faithfully preserving detail of an arbitrary input mesh. DeCoro and Tatarchuk [24] present a scheme for simplifying arbitrary meshes using octree-based vertex clustering. This clustering strategy avoids precomputation and storage of a vertex hierarchy, but the resulting approximating meshes are less accurate. Ji et al. [25] also perform LOD computations on the GPU. Their technique first resamples the input model onto a regular remesh over a polycube map. A vertex shader is used to displace inactive vertices to infinity. One key difference of our work from these two techniques is that we do not require traversing the entire representation (e.g. the fully detailed input mesh) at every frame. Instead, our per-frame computation is only proportional to the (selectively refined) rendered mesh.

Our system is the first to perform real-time vertex-granular LOD over arbitrary triangle meshes on the GPU [26]. This work extends the presentation of our approach and provides additional usage scenarios for this level-of-detail control. We describe how our scheme is particularly beneficial for the rendering of shadows, semi-transparent surfaces, and translucent surfaces, because in these multi-pass rendering approaches the same simplified mesh is rendered multiple times per frame.

3 Hierarchy and refinement dependencies

3.1 Vertex hierarchy

Our view-dependent LOD algorithm uses the vertex hierarchy of a progressive mesh (PM) [2],[3]. The construction is based on that of Hoppe [3], with modifications tailored to our parallel LOD update algorithm.

Given an arbitrary mesh M^n, a hierarchy is built by applying a prioritized sequence of edge collapses:

$$M^n \underset{spl_{n-1}}{\overset{col_{n-1}}{\rightleftharpoons}} \cdots\cdots \underset{spl_1}{\overset{col_1}{\rightleftharpoons}} M^1 \underset{spl_0}{\overset{col_0}{\rightleftharpoons}} M^0$$

where col_v is the collapse that creates vertex v, spl_v is the reverse operation that splits v, and M^0 is the base mesh that results after all the collapses. Our method restricts the collapses to *half-edge collapses* to produce a compact read-only vertex buffer [27].

Figure 1a shows a collapse and the split that inverts the operation. The pair of faces f_l^v and f_r^v, which are adjacent to v_t and v_u, are removed by col_v, and are created by spl_v. During PM construction, we reorder the 3 vertices of every face in M^n such that when performing col_v, the vertices in f_l^v and f_r^v are $\{v_t, v_u, v_l\}$ and $\{v_u, v_t, v_r\}$ respectively.

Figure 1b shows the subtree rooted at v in a vertex hierarchy. The leaves are the vertices in M^n, and each non-leaf vertex v results from the collapse of its two children v_t and v_u. Since half-edge collapses are used, v has the same attributes (e.g., position, normal) as its left child v_t and transitively its leftmost descendant leaf, denoted as v_{t^*}. Thus we need only store vertex attributes for the leaves, and access them for any vertex v using v_{t^*}. We let v^p denote the parent of v, i.e. $(v_t)^p = v$. The vertex hierarchy is linearized in memory, with vertices assigned indices in the reverse order that they were collapsed. Thus the leaf vertices are consecutive and last, and can be distinguished from non-leaf vertices solely by their index. The ordering also implies that $v > v^p$ for any vertex v.

A selectively refined mesh, denoted M, corresponds to a frontier of active vertices within the hierarchy, as illustrated in Figure 1b. This frontier partitions the (non-leaf) vertices into split and collapsed states.

3.2 Refinement dependency structure

To prevent foldovers of the triangles in the mesh, the splits and collapses must adhere to dependency rules. In order to produce a compact vertex hierarchy and perform efficient runtime tests on the GPU, we introduce a new dependency structure. The explicit rules [3] incur

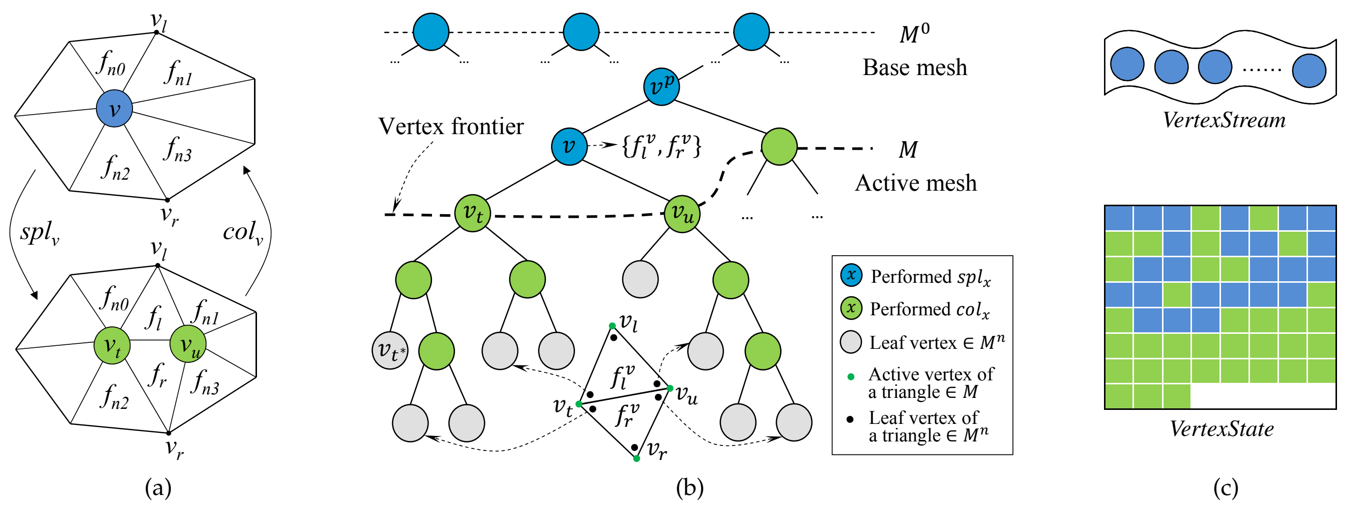


Fig. 1. (a) The neighborhood around a split/collapse operation on a vertex v. (b) the vertex hierarchy showing a subtree rooted at vertex v. Splitting v generates vertices v_t, v_u and the pair of faces $\{f_l^v, f_r^v\}$. A selectively refined mesh M corresponds to a frontier of active vertices. All vertices above this frontier, colored blue, have been split, whereas all vertices colored green are in their collapsed state. The leaf vertices of the triangles f_l^v and f_r^v are shown with their locations in the hierarchy. (c) The set of split (blue) vertices that are processed during selective refinement are stored in the *VertexStream* stream, while the state of all vertices is kept in the *VertexState* texture.

extra memory, whereas the implicit rules [28] are too restrictive and would require many more unnecessary vertex splits to meet the view-dependent criteria. Additionally, they all require relatively complex runtime tests. Our proposed approach follows the same refinement flexibility as the explicit rules, but with a more compact representation inspired by the implicit rules, and most importantly it is well adapted to GPU stream processing due to its simplicity. The memory footprint is smaller than both earlier representations. The splits are only dependent on a smaller mesh neighborhood, while still conforming to the old dependency rules (Figure 2).

During the PM construction, for each col_v, each removed face f_l^v and f_r^v is adjacent to two other mesh faces, $\{f_{n_0}^v, f_{n_1}^v\}$ and $\{f_{n_2}^v, f_{n_3}^v\}$ respectively, as shown in Figure 1a. At runtime, the explicit rules [3] check for the presence and adjacency of these four faces in the current selectively refined mesh. Specifically the rules are as follows:

(i) A split spl_v is legal if the faces $\{f_{n_0}^v, f_{n_1}^v, f_{n_2}^v, f_{n_3}^v\}$ all exist in the current selectively refined mesh.
(ii) A collapse col_v is legal if f_l^v is currently adjacent to $\{f_{n_0}^v, f_{n_1}^v\}$ and f_r^v is currently adjacent to $\{f_{n_2}^v, f_{n_3}^v\}$.

Unfortunately, test (ii) involves maintaining face adjacencies, which is difficult in a parallel algorithm. Our approach is to perform a simpler check that involves storing two vertex indices instead of four face indices. Specifically, we precompute $v_{lmax} = \max(c_v(f_{n_0}^v), c_v(f_{n_1}^v))$ and $v_{rmax} = \max(c_v(f_{n_2}^v), c_v(f_{n_3}^v))$ where $c_v(f)$ is a non-ancestral vertex split that creates f. More precisely, $c_v(f)$ is the vertex x whose split creates face f, unless $f \in M^0$ or x is an ancestor of v, in which cases $c_v(f) = 0$. At runtime, given the side vertices v_l and v_r in M, we can check legality as follows:

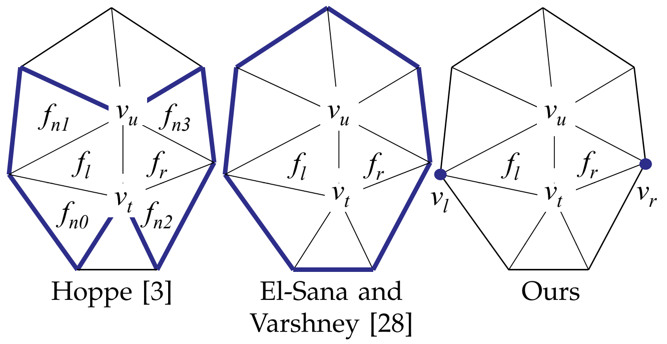


Fig. 2. To apply the dependency rules of Hoppe [3], each vertex split considers the four adjacent faces (left), whereas the rules of El-Sana and Varshney [28] use information on the entire 1-ring vertex neighborhood is used. Our representation is more compact and simply requires checking against two adjacent vertices v_l and v_r.

(I) A split spl_v is legal if $v_l > v_{lmax}$ and $v_r > v_{rmax}$.
(II) A collapse col_v is legal if $(v_l)^p < v$ and $(v_r)^p < v$.

As proven in the appendix, our dependency rules conform to the explicit rules.

4 Data Structures

Our framework maintains a static *VertexBuffer* holding the vertex attributes, and a dynamically updated *IndexBuffer* holding 3 vertex indices per triangle face. These structures are used to render the selectively refined mesh using a generic vertex shader. To maintain the *IndexBuffer*, several static and dynamic data structures are required. These data structures, which are listed in detail in Table 1, are stored on the GPU using sequential buffers. Note that we keep a double-buffered *IndexBuffer* to allow asynchronous update of one index buffer, amortized over multiple frames (Section 6), while rendering the mesh using the other index buffer.

4.1 Static structures

The vertex hierarchy is stored in the *VertexTree* buffer. For each vertex v we store the indices v_t, v_u, v_{t^*} and v^p. Vertex v is also associated with the two triangle faces f_l^v and f_r^v that are removed from the mesh by the collapse col_v. However, we do not keep their indices explicitly with v, since they are simply $2v$ and $2v+1$. To guarantee consistency on meshes with boundaries, where col_v may remove only one face f_l^v at the boundary, we duplicate f_l^v as f_r^v and insert it into M^n. Note that this does not incur any additional rendering cost, and requires negligible memory since there are typically only a small fraction of boundary faces. Additionally, we store parameters used for view-dependent refinement of the non-leaf vertices in the *RefineCriteria* buffer.

The *OrigFaces* buffer stores for each face in the original mesh M^n its vertex indices $\{v_0, v_1, v_2\}$ (i.e. 3 leaves in the vertex hierarchy). These indices are normally stored as 32-bit integers to support large models, so each face in *OrigFaces* would require 12 bytes. However, since meshes with more than 2^{24} vertices (33.5 M triangles) would occupy more than the 1 GB of available video memory, our implementation uses 24 bits per index (9 bytes per face), and uses the saved 3 bytes per face to encode the indices v_{lmax} and v_{rmax} from Section 3.2.

Buffer *BaseFStream* stores the indices of the faces in the base mesh M^0. And, the indices of the vertices in M^0 (i.e. the root vertices in the hierarchy) are stored in a buffer *BaseVStream* to better parallelize the update of *VertexStream*, as described in Section 5.

4.2 Dynamic structures

The dynamic structures are shown in Figure 1c. The *VertexState* texture stores for each vertex in the hierarchy one of three possible states: {*collapsed*, *split*, *splitting*}. As their names suggest, the *split* and *collapsed* states identify vertices that have or have not been split, respectively. The state *splitting* is needed by the algorithm to identify vertices that are currently being split. For details, refer to Section 5. Note that all vertices in the hierarchy that are above the frontier are in either *split* or *splitting* state, while the others are in *collapsed* state, as shown in Figure 1bc using blue and green vertices, respectively. We do not have to associate states with the leaves since they cannot split. Due to GPU hardware constraints, we store *VertexState* as a 2D texture instead of a buffer.

Finally, the main data structure of our algorithm is the *VertexStream* buffer, which contains a dynamically updated list of all the vertices that have been split (i.e., those that are above the frontier). This buffer is useful for identifying both the active vertices and active faces in the current selectively refined mesh. Specifically the set of active vertices (i.e., those on the frontier and in

TABLE 1
Buffers used in runtime selective refinement.

Buffers	Elements	Cardinality	Memory
Static structures			
VertexBuffer	Position	L	$12n$
	Texcoord		$4n$
	Normal		$4n$
VertexTree	$\{v_t, v_u, v_{t^*}\}$	I	$12n$
	v^p	L I	$8n$
RefineCriteria	$\{\delta_v, r_v, \sin^2 \alpha_v\}$	I	$4n$
OrigFaces	$\{v_0, v_1, v_2\}$	M^n	$18n$
	$v_{lmax}(v_{rmax})$		$6n$
BaseVStream	v	M^0	$-^\dagger$
BaseFStream	f	M^0	$-^\dagger$
Dynamic structures			
VertexState	v_{state}	I	n
VertexStream	v	M	$4m \times 2$
IndexBuffer	$\{v_0, v_1, v_2\}$	M	$24m \times 2$
Total			$69n + 56m$

Labels **L** and **I** denote leaf and non-leaf nodes of the vertex hierarchy. Memory usage is in bytes, where n and m are the number of vertices in the original mesh M^n and active mesh M respectively. †Typically the base mesh M^0 has negligible complexity.

collapsed state) and the set of active faces are defined as follows:

$$\bigcup_{v \in M} v = \bigcup_{v \in \mathit{VertexStream}} \; \bigcup_{x \in \{v_t, v_u\}} \{x \mid x_{state} = \mathit{collapsed}\}, \quad (1)$$

$$\bigcup_{f \in M} f = \mathit{BaseFStream} \cup \left(\bigcup_{v \in \mathit{VertexStream}} \{f_l^v, f_r^v\} \right). \quad (2)$$

Note that we keep two *VertexStream* buffers, and ping-pong between them to avoid read-modify-write hazards (Section 5).

5 Runtime Algorithm

The runtime algorithm dynamically updates *IndexBuffer*, which stores the triangles in the selectively refined mesh. The update consists of three processing steps as outlined in Figure 3. The first step checks for desirable edge collapses and vertex splits, and updates the vertex states accordingly; the second step updates and maintains the stream of active vertices based on the updated states; and the final step generates the index buffer using the set of split vertices and the updated frontier implied from the states. Next, we describe each of the steps. Listing 1 provides detailed pseudocode.

UpdateVertexState: We check for splits and collapses according to the three view-dependent criteria – view frustum, surface orientation and screen-space geometric error – from Hoppe [29]. Additional splits may be required to enforce the dependency rules. The algorithm starts by first setting the states of all elements in *VertexStream* to *collapsed*. Next, it traverses the vertices in *VertexStream* to update their states. This is accomplished by generating point primitives in the *geometry shader* and

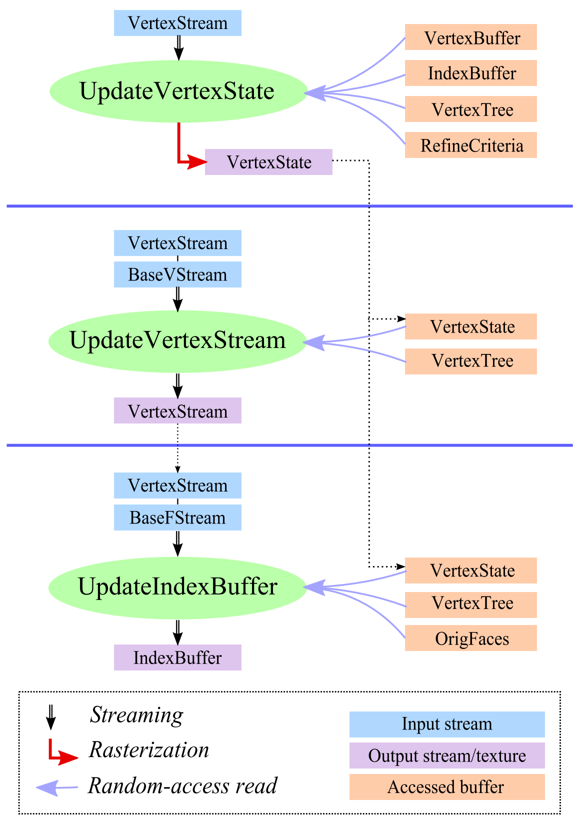


Fig. 3. The 3-step selective refinement algorithm.

rendering the updated state to the *VertexState* texture, which is addressable by vertex index. In the traversal, we set the state of the vertices that do not collapse to *split*. For the children of the vertices in *VertexStream* that are active in M (following (1)), we test if they should split, and if so, update their states to *splitting*. In addition, we also test their children. By doing so, we allow up to *two* hierarchy levels to split simultaneously within each update iteration, thereby resulting in a more efficient update.

Satisfying all split dependencies for the new splits would require performing recursive updates by forcing splits on potentially remote vertices in the mesh. Such splits can be arbitrary in number and in nesting within the hierarchy. Hoppe [3] makes use of a stack to record and then force the chain of required splits. Unfortunately, such an approach is infeasible on the GPU due to the limitations on the output size of a geometry shader instance. A naïve approach would be to mark dependent splits and wait multiple iterations until all dependencies are satisfied before splitting a vertex. However, this would cause a significant temporal lag in LOD refinement.

To overcome this problem we introduce a *cascaded update* method that updates new splits without respecting their dependencies, and forces the adjacent constraining vertices to split in subsequent updates. Specifically for each spl_v, we continue splitting the active side vertices v_l and v_r in f_l^v and f_r^v every iteration as long as they still constrain spl_v. This scheme lets the level-of-detail

Listing 1 Pseudocode for the 3 steps of our algorithm

```
// Step 1
procedure UpdateVertexState
  for $v \in VertexStream$ in parallel do
    $v_{state} \leftarrow collapsed$
  for $v \in VertexStream$ in parallel do
    $v_l \leftarrow (f_l^v)_{v_2}$, $v_r \leftarrow (f_r^v)_{v_2}$
    if Active($v_t$) and Active($v_u$) and not VDCoarsen($v$) then
      $v_{state} \leftarrow split$
    for $x \in \{v_t, v_u\}$ do
      if Active($x$) then
        for $y \in \{x, x_t, x_u\}$ do
          if VDRefine($y$) then
            $y_{state} \leftarrow splitting$
    if $v_l < v_{lmax}$ then
      $(v_l)_{state} \leftarrow splitting$, $((v_l)^p)_{state} \leftarrow split$
    if $v_r < v_{rmax}$ then
      $(v_r)_{state} \leftarrow splitting$, $((v_r)^p)_{state} \leftarrow split$

function bool Active($v$)
  // return $v_{state} = collapsed$   // leads to read-write hazard
  return $v = (v^p)_t$  ?  $v = (f_l^{v^p})_{v_0}$  :  $v = (f_l^{v^p})_{v_1}$

function bool VDRefine($v$)
  Test view-dep. refinement criteria using $\{\delta_v, r_v, \sin^2 \alpha_v\}$.

function bool VDCoarsen($v$)
  return $(v_l)^p < v$ and $(v_r)^p < v$ and not VDRefine($v$)

// Step 2
procedure UpdateVertexStream
  for $v \in BaseVStream$ in parallel do
    Output $v$
  for $v \in VertexStream$ in parallel do
    for $x \in \{v_t, v_u\}$ do
      if $x_{state} \neq collapsed$ then
        Output $x$
      for $y \in \{x_t, x_u\}$ do
        if $x_{state} = splitting$ and $y_{state} = splitting$ then
          Output $y$

// Step 3
procedure UpdateIndexBuffer
  for $f \in BaseFStream$ in parallel do
    OutputFace($f$)
  for $v \in VertexStream$ in parallel do
    OutputFace($f_l^v$), OutputFace($f_r^v$)

function OutputFace($f$)
  for $v \in OrigFaces[f]$ do
    while $(v^p)_{state} = collapsed$ do
      $v \leftarrow v^p$
    Output $v$
```

adapt more quickly, as it takes many fewer algorithm iterations to fully propagate the splits. The drawback is that it may result in temporary foldovers in the mesh surface. However, these temporary foldovers introduced by the view-dependent algorithm are rare (e.g. no more than 0.1% of the faces even for fast camera motions in the Dragon sequence), typically involve thin sliver triangles that are nearly invisible, and are corrected quickly by the LOD update. Consequently they are not noticeable in our experiments.

Forcing the splitting of adjacent vertices may be in conflict with desired collapses on the same vertices, and it is impossible to explicitly identify these conflicts in one pass since they arise in parallel execution paths. We resolve these conflicts by performing additional compensating splits. Specifically, for the side vertices v_l and v_r that are involved in the dependent splits from v, we also update the states of their parents $(v_l)^p$ and $(v_r)^p$ to *split*, in case the parents are collapsing in their own executing instances.

UpdateVertexStream: The second step outputs an updated *VertexStream* by using the *stream out* functionality. In this step, the algorithm traverses the vertices in *VertexStream* and outputs all child vertices that are not in *collapsed* state. We output the children rather than the vertices themselves to save one state check per instance. We output the root vertices in *BaseVStream* directly in a quick preceding pass. Newly split vertices that are two levels deeper in the hierarchy are identified and output as well.

UpdateIndexBuffer: Finally, we update *IndexBuffer* by writing out the active triangle faces of M (following (2)). More specifically, the indices of the active vertices in every pair of faces $\{f_l^v, f_r^v\}$ associated with each split vertex v in the updated *VertexStream* are streamed out to *IndexBuffer*. We obtain the indices of the active vertices in M by retrieving the indices of the leaf vertices in the same faces in M^n from *OrigFaces*, and searching up the hierarchy for the coarsest vertices in *collapsed* states. An additional preceding pass streams out the triangle faces in M^0 by traversing *BaseFStream*.

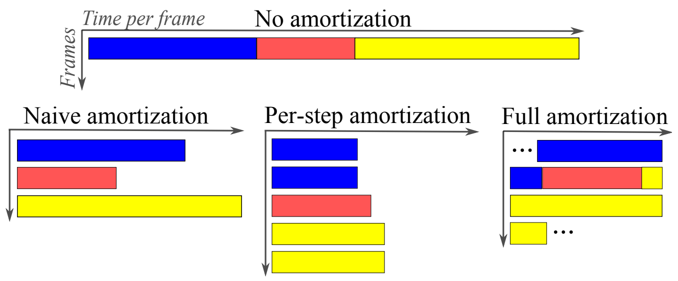


Fig. 4. Amortization mechanisms. The width of each bar represents the time incurred by that step of the update algorithm. Full amortization manages to keep frame times low and consistent.

6 Amortized computation

Geometry LOD algorithms often amortize the refinement computation over multiple frames to maintain a desired framerate [3]. Our runtime update algorithm involves three update steps that require different amounts of time. A naïve amortization that performs one update step per frame leads to oscillations in frame times (see Figure 4). To try to maintain an upper bound T^* on the frame time (e.g., 33.3 ms), at the beginning of each update iteration, we dynamically partition the update steps. We explore two such mechanisms.

TABLE 2
Statistics for the original meshes used in our experiments and the number of faces used in the selectively refined meshes shown in Figure 7.

Input meshes			*Meshes in Figure 7*	
Model name	*Total # faces*	*Memory (MB)*	*Rendered # faces*	*Rendering time (ms)*
Lucy	2,000,000	65.8	88,432	4.6
Terrain	2,097,147	69.0	179,013	7.5
Dragon	7,218,906	237.5	290,892	15.2
Statue	10,000,000	329.0	389,566	22.2

Per-step amortization: The first mechanism partitions each step $i \in \{1, 2, 3\}$ individually. Specifically, at algorithm iteration k, it partitions the input stream S_i^k for step i uniformly into N_i^k segments (see Figure 4). The number of segments N_i^k is estimated based on the previous iteration as

$$N_i^k = \left\lceil \frac{|S_i^k|}{|S_i^{k-1}|} \cdot \frac{T_i^{k-1}}{T^*} \right\rceil$$

where T_i^{k-1} is the total time spent in step i in the previous iteration. The mechanism automatically adapts to the update load of the selectively refined mesh at runtime, but still suffers from frame time oscillations due to the absence of load balancing across different steps of the algorithm.

Full amortization: The second mechanism alleviates this problem by allowing multiple algorithm steps to be executed within the same frame (see Figure 4). As in the previous approach, this method also keeps an estimate of the cost of each step based on the previous update. The algorithm determines what fraction of the current step can be executed within the current frame time budget T^*. If the step can only be executed partially, the remainder is carried over to the following frame. If the step can be executed fully, then the algorithm again determines what fraction of the upcoming step fits within the remaining budget and can be executed. The algorithm proceeds in this fashion until it exhausts the time budget.

7 Results

We implemented our algorithm in Microsoft DirectX 10 using an Intel Core2 CPU with 2 GB memory and an NVIDIA GeForce 8800 GTX graphics card. All shaders are written in HLSL using shader model 4.0. For all examples we use a window size of 1280×800 pixels and a screen-space error tolerance of 1 pixel.

Frame rate: Figure 7 shows selectively refined meshes for the models in Table 2.

Figure 5 graphs the relationship between the number of faces in the current active mesh M and the time it takes to perform all of the algorithm passes. With a static viewpoint, the processing time is nearly linear on M. This linear-time computation has been observed for all

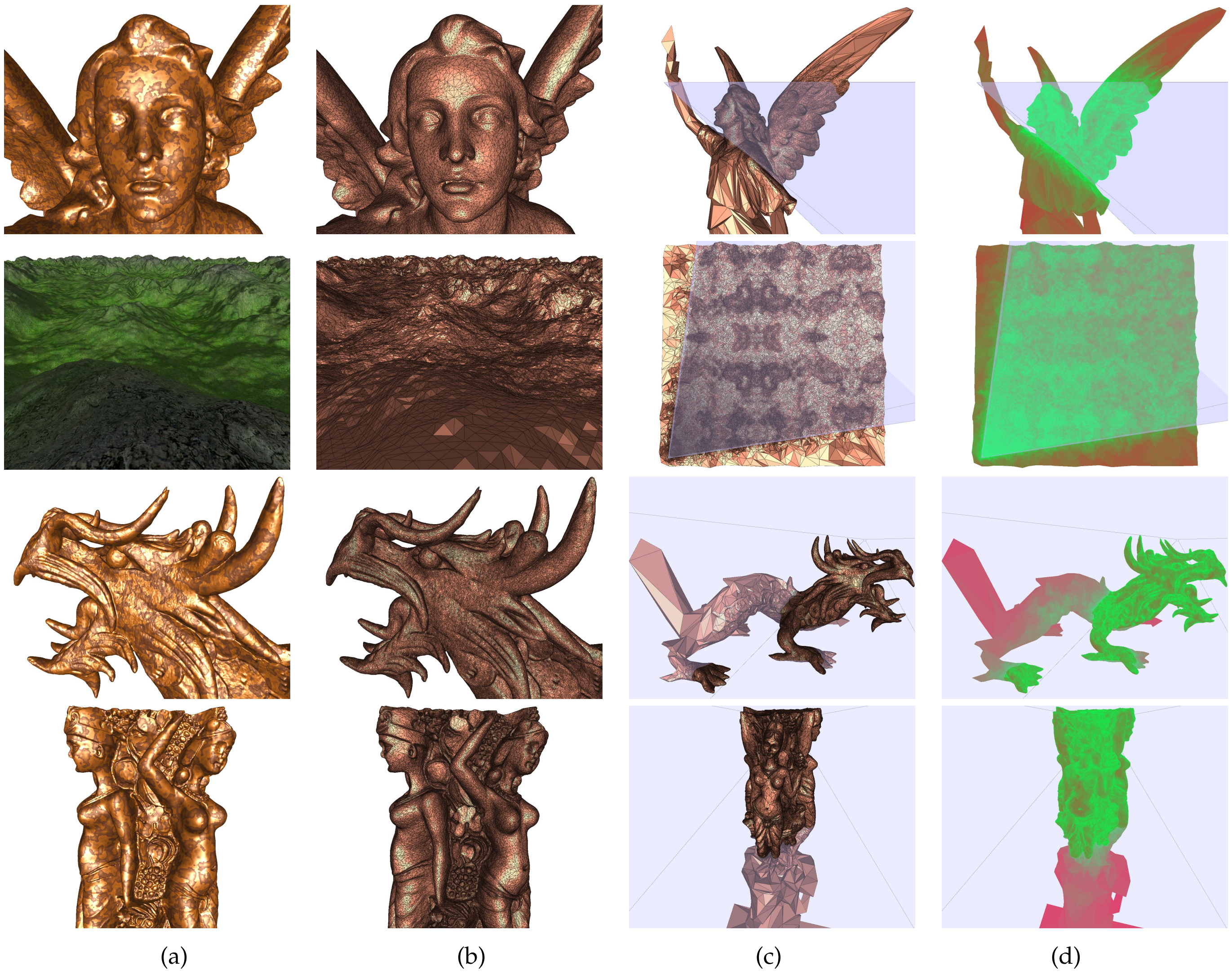


Fig. 7. Renderings of our selectively refined meshes with a procedural texture (a) and in wireframe mode (b). The same mesh is then visualized from a different viewpoint in wireframe mode (c), and with color-coded LOD (d), where green represents higher detail than red. Note that mesh regions that lie outside the frustum, farther away, or facing away from the camera, have lower LODs. The error tolerance is less than one pixel.

of the models in our test suite. Figure 5 also graphs the processing time when using a typical camera motion (dynamic viewpoint). Note that the introduced splits and collapses cause additional processing and geometry amplification, thus slowing down the update. The added GPU data amplification cost is not prohibitively expensive and the relationship remains roughly linear on the size of M. The variation is due to the differing number of splits and collapses for the measurements.

Figure 6 (top) graphs the frame time of each amortization method for a *Statue* rendering sequence using a target frame time upper bound T^* of 33.3 ms. Without amortization, the cost incurred by the update is clearly above the desired frame time. Naïve amortization partitions the update into 3 frames thereby improving frame time significantly. However, due to the high relative costs among the update passes, the frame time still oscillates heavily. Per-step amortization further partitions each step to ensure that frame times are no higher than T^*. Note, that, while it reduces frame times, varying step costs may still cause significant oscillation. Finally, the full amortization method is able to better maintain the target frame rate by allowing multiple passes to be processed within the same frame, as described in Section 6. Note, however, that a completely stable frame time cannot be guaranteed since the GPU algorithm cannot predict drastic changes in M, which would result in immediate slowdowns in some of the passes. Therefore, occasional performance spikes are unavoidable. However, even with the fast camera motions shown in the accompanying video, the spikes are rare and do not significantly harm the observed frame rate across this entire sequence.

Both per-step and full amortization manage to reduce frame time to the desired target bound T^*. When only considering these two satisfactory options, full amorti-

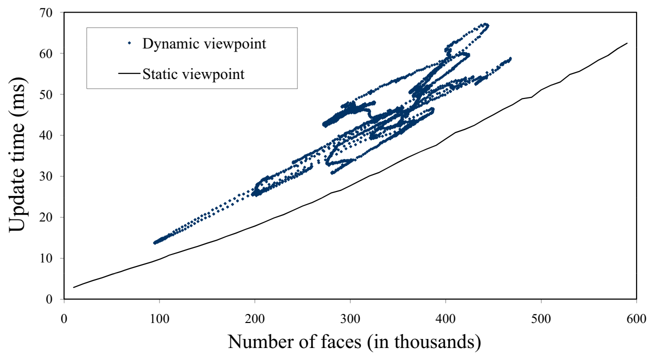


Fig. 5. Update times as a function of the number of faces in the active mesh M when no splits and collapses are necessary (static viewpoint), and when splits and collapses are applied as part of typical viewer motion (dynamic viewpoint).

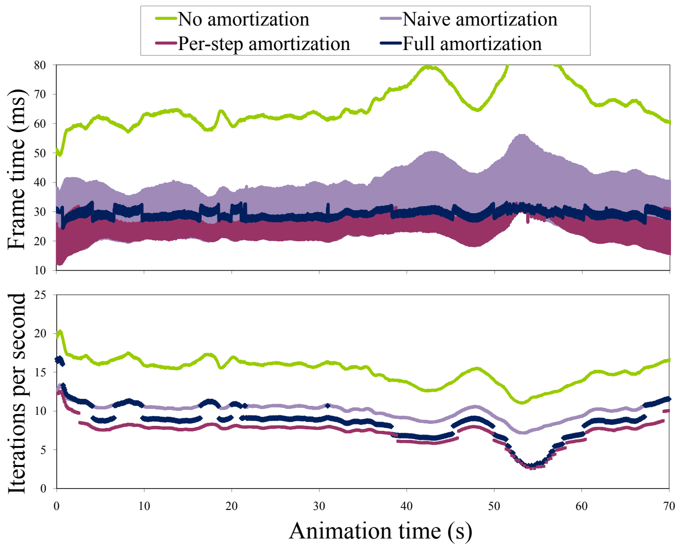


Fig. 6. Timings measurements (top) and algorithm iterations per second (bottom) for the *Statue* rendering sequence using different amortization strategies.

zation manages to accommodate the entire update in fewer frames, as predicted in Figure 4 and evidenced in Figure 6 (bottom), which graphs the total number of iterations per second.

Memory analysis: Our data structures use a total of $69n + 56m$ bytes, where n and m are the numbers of vertices in the original and active meshes respectively. For sufficiently detailed models, it is generally impossible to view all surface regions at high resolution within a frame, so typically $m \ll n$. Thus the memory bottleneck is the size $69n$ of the static portion of our structures. Storing the original mesh as a traditional indexed triangle list requires $44n$ bytes ($20n$ for the *VertexBuffer* and $24n$ for the *IndexBuffer*). Thus, in comparison, only 57% more memory is required for our static data structures. Also as shown in Table 3, ours compares favorably with previous view-dependent LOD schemes, which is

TABLE 3
Comparison of memory size with prior schemes.

View-dependent LOD scheme	*Memory size (bytes)*
VDPM [3]	$216n$
SVDLOD [29]	$88n + 100m$
MT [30]	$75n$
VDT [28]	$90n$
FastMesh [31]	$88n + 6m$
Our scheme	$69n + 56m$

The variables n and m denote the numbers of vertices in the original and active meshes respectively.

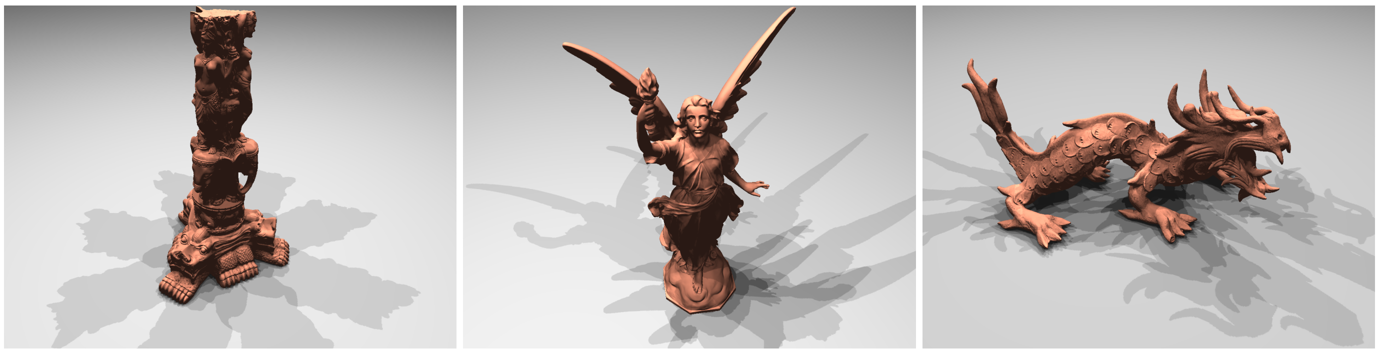
surprising given our challenging parallelism restrictions. In addition, most prior schemes perform immediate-mode rendering (e.g. with glVertex() calls), whereas our dynamic data structures ($56m$ bytes) maintain explicit index buffers that permit more efficient rendering on present graphics systems.

Limitations: Our index buffers define indexed triangle lists rather than strips, so take more space than an optimized static mesh. Also, the ordering of faces within this list is not optimized for vertex cache locality. Finally, our system does not yet support geomorphs [32] for smooth temporal transitions. However, the LOD updates are fast and the screen-space error tolerance is small enough that popping is nearly imperceptible.

8 Usage scenarios

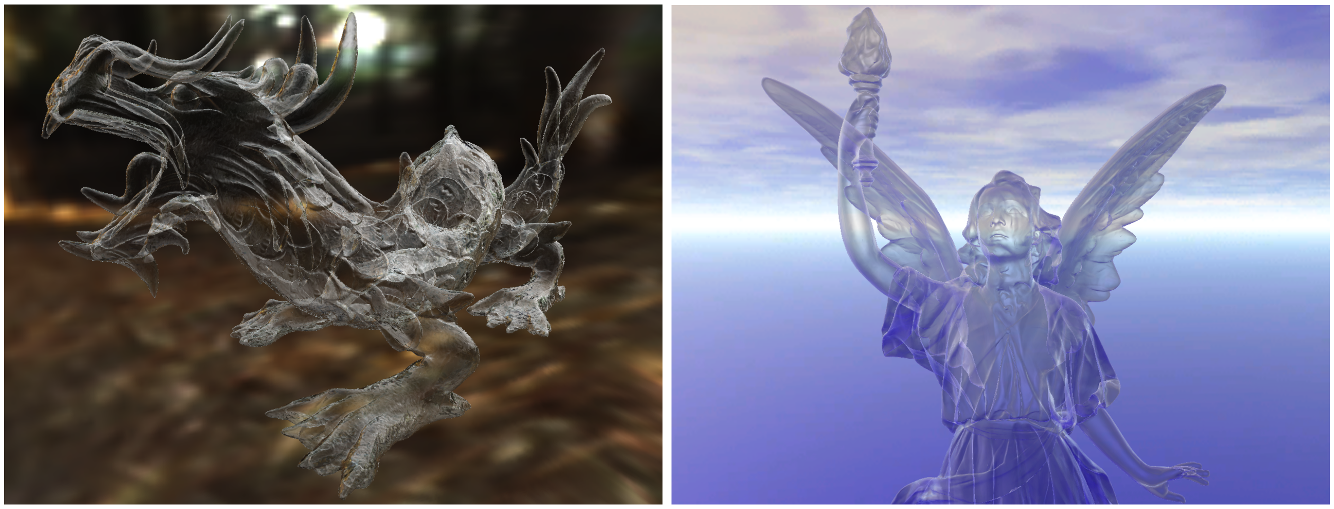
As opposed to cluster-based approaches, our vertex-granular level of detail technique produces a single generic index buffer. This facilitates rendering in traditional graphics systems since the buffer can be used directly by any standard rendering algorithm. More importantly, it introduces no additional processing overhead. This is particularly advantageous for multi-pass rendering techniques, where the index buffer needs to be traversed multiple times per frame. We therefore apply our algorithm to direct implementations of semitransparent surface rendering and shadow mapping techniques. These techniques do not need to be modified at all and can immediately take advantage of the geometry level of detail provided by our approach in order to improve rendering times.

Shadow mapping from multiple light sources: Shadow mapping [33] is the most prevalent technique for real-time shadow generation and display. The algorithm first renders the scene from the point of view of the light source, storing the depth information in a *shadow map*. On a subsequent pass, when the scene is rendered from the observer's point of view, a distance comparison is performed per pixel to determine whether there is an object occluding the pixel's view of the light. If multiple light sources are used, multiple rendering passes are needed in order to generate the shadow maps. For this application, we use a standard implementation of shadow mapping and accumulate shadows from the



Statue, 371 K out of 10 M triangles Lucy, 114 K out of 2 M triangles Dragon, 322 K out of 7.2 M triangles

Fig. 8. Renderings of our selectively refined meshes using shadow maps from multiple light sources.



Dragon, 433 K out of 7.2 M triangles Lucy, 128 K out of 2 M triangles

Fig. 9. Semitransparent renderings of our selectively refined meshes using a multi-pass approach based on depth peeling.

multiple light sources. Figure 8 shows the results, which directly apply shadow mapping to our active index buffers. Rendering speed is improved by a factor of 6–11× when compared to rendering the original meshes from the same viewpoints. The resulting images have no visible quality degradation.

Order-independent transparency and translucency: In the presence of semitransparent surfaces, physically correct rendering is order-dependent. The difficulty lies in the fact that the geometric primitives composing the scene do not have the proper view-dependent depth ordering. Depth peeling [34] allows processing fragments at each screen pixel location in depth order. This is accomplished by *peeling* layers of the geometry in successive rendering passes. The algorithm can be efficiently implemented in commodity graphics hardware [35]. We use a direct implementation of the multi-pass depth peeling algorithm and render the scene in back-to-front order, properly alpha blending the different layers of the geometry. As with the shadowing application, we can easily apply the algorithm directly to our index buffer. The renderings in Figure 9 yield speedups of 5–8× when compared to rendering the original meshes, without visible quality loss. The multi-pass depth-peeling approach can also be used to simulate translucent effects with Fresnel shading [36] at each surface intersection, as shown in Figure 10. This technique resulted in speedups of 7–10×.

9 Conclusion and Future Work

We present the first view-dependent LOD algorithm for vertex-level mesh refinement that operates entirely on the GPU. Our scheme performs general vertex splits and edge collapses to incrementally and selectively refine an irregular hierarchy as a sequence of parallel streaming steps. The approach is highly parallel, processing many splits and collapses simultaneously. Because the mesh is updated and rendered using a constant number of draw calls, CPU utilization is near zero. The cost of these draw calls is further amortized over multiple frames using a simple feedback mechanism.

One of our contributions is to overcome the difficulties imposed by GPU parallel processing. This involves a new, compact dependency structure and a cascaded

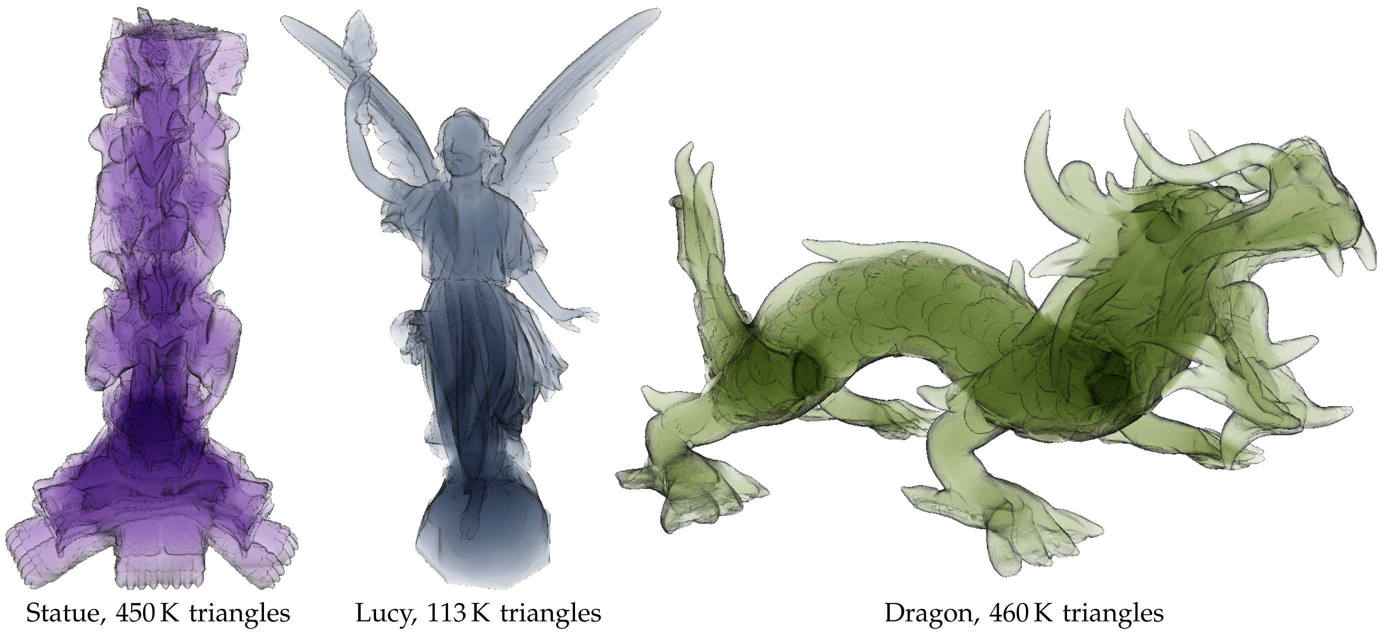


Fig. 10. Translucent renderings of our selectively refined meshes, also using multi-pass depth peeling.

update method. In future work, it would be interesting to consider more general parallel processing APIs. For instance, NVIDIA's CUDA and AMD's Stream Computing interfaces expose local shared memory and allow scattered reads and writes to the same memory buffer. Note however that the current version of CUDA requires computing a prefix sum [37] in order to compactly emit a variable number of elements per thread, which is essential to our approach. But it is likely that such APIs will continue to improve in programming flexibility.

The parallel nature of our algorithm lets it scale with the number of stream processors in the GPU. Thus we expect greater performance gains as the GPU evolves to include more processors. The stream-based approach may also be applicable to other parallel architectures, as it provides an elegant way to decouple an irregular refinement algorithm into dependency-free passes. Finally, we hope that our ideas may inspire new algorithms for handling irregular data structures within parallel architectures.

Appendix

Lemma 1. *The dependency rules (I) and (II) conform to (i) and (ii) respectively, i.e., (i)⇒(I) and (ii)⇒(II).*

Proof: (i)⇒(I): Suppose $v_l > v_{lmax}$. We first prove that $f^v_{n_0}$ exists. By definition of v_{lmax}, $v_l \geq c_v(f^v_{n_0})$ and $v_l \geq c_v(f^v_{n_1})$. We denote $x = c_v(f^v_{n_0})$. The face $f^v_{n_0}$ exists in the current selectively refined mesh if $x = 0$ by definition, otherwise $x \neq 0$ and $f^v_{n_0}$ is created by spl_x, which means the vertex x is an ancestor of two vertices in triangle $f^v_{n_0}$. Since $x \neq 0$, x is not an ancestor of v (and therefore v_t). And because v_l is adjacent to $f^v_{n_0}$ and $v_l \geq x$, x must be an ancestor of v_l. Therefore $f^v_{n_0}$ also exists since spl_x has been performed otherwise v_l would not be active. The same argument holds for the existence of $f^v_{n_1}$, and similarly supposing $v_r > v_{rmax}$, then $\{f^v_{n_2}, f^v_{n_3}\}$ also exist. Since (i) is true, it follows that whenever $v_l > v_{lmax}$ and $v_r > v_{rmax}$, then $\{f^v_{n_0}, f^v_{n_1}, f^v_{n_2}, f^v_{n_3}\}$ all exist, and spl_v is legal, i.e. (I) is true.

(ii)⇒(II): Since v_t and v_u are active, spl_v has been legally performed. According to Hoppe [3], $\{f^v_{n_0}, f^v_{n_1}\}$ and $\{f^v_{n_2}, f^v_{n_3}\}$ must be pairwise adjacent at the time of spl_v. Now suppose $(v_l)^p < v$. We first prove that f^v_l is adjacent to $f^v_{n_0}$. We denote f' as the face adjacent to v_t, v_l and f_l, and denote x' as the vertex such that $spl_{x'}$ creates f'. Suppose $f' \neq f^v_{n_0}$. x' cannot be an ancestor of v_t since $f^v_{n_0}$ and $f^v_{n_1}$ would otherwise not be adjacent when performing spl_v. Then x' must be an ancestor of v_l because $spl_{x'}$ creates f' and x' is an ancestor of two vertices in f'. Since f' is adjacent to f_l, $spl_{x'}$ must be performed after spl_v to be considered legal, i.e. $x' > v$, since otherwise f_l would not be present at the time of $spl_{x'}$. However as x' is an ancestor of v_l, $x' \leq (v_l)^p < v$, which is a contradiction. Therefore $f' = f^v_{n_0}$ and f^v_l is adjacent to $f^v_{n_0}$. By the same argument, it can be shown that f^v_l is also adjacent to $f^v_{n_1}$, and similarly supposing $(v_r)^p < v$, then f^v_r is adjacent to $\{f^v_{n_2}, f^v_{n_3}\}$. Since (ii) is true, it follows that if $(v_l)^p < v$ and $(v_r)^p < v$, then f^v_l is adjacent to $\{f^v_{n_0}, f^v_{n_1}\}$ and f^v_r is adjacent to $\{f^v_{n_2}, f^v_{n_3}\}$. As a result, col_v is legal, i.e. (II) is also true. □

References

[1] D. Luebke, B. Watson, J. D. Cohen, M. Reddy, and A. Varshney, *Level of Detail for 3D Graphics*. New York, NY, USA: Elsevier Science Inc., 2002.

[2] J. C. Xia and A. Varshney, "Dynamic view-dependent simplification for polygonal models," in *Proceedings of the IEEE Conference on Visualization (VIS '96)*, pp. 327–334, 1996.

[3] H. Hoppe, “View-dependent refinement of progressive meshes,” in *Computer Graphics (Proceedings of SIGGRAPH 1997)*, pp. 189–198, 1997.

[4] D. Luebke and C. Erikson, “View-dependent simplification of arbitrary polygonal environments,” in *Computer Graphics (Proceedings of SIGGRAPH 1997)*, pp. 199–208, 1997.

[5] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, “Adaptive tetrapuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models,” *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 796–803, 2004.

[6] L. Borgeat, G. Godin, F. Blais, P. Massicotte, and C. Lahanier, “GoLD: interactive display of huge colored and textured models,” *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 869–877, 2005.

[7] P. V. Sander and J. L. Mitchell, “Progressive buffers: View-dependent geometry and texture for LOD rendering,” in *Proceedings of the Third Eurographics Symposium on Geometry Processing (SGP ’05)*, pp. 129–138, 2005.

[8] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner, “Real-time, continuous level of detail rendering of height fields,” in *Computer Graphics (Proceedings of SIGGRAPH 1996)*, pp. 109–118, 1996.

[9] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-weinstein, “ROAMing terrain: real-time optimally adapting meshes,” in *Proceedings of the IEEE Conference on Visualization (VIS ’97)*, pp. 81–88, 1997.

[10] P. Lindstrom and V. Pascucci, “Terrain simplification simplified: A general framework for view-dependent out-of-core visualization,” *IEEE Trans. Visualization and Computer Graphics*, vol. 8, no. 3, pp. 239–254, 2002.

[11] J. Levenberg, “Fast view-dependent level-of-detail rendering using cached geometry,” in *Proceedings of the IEEE Conference on Visualization (VIS ’02)*, pp. 259–266, 2002.

[12] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, “BDAM: batched dynamic adaptive meshes for high performance terrain visualization,” *Computer Graphics Forum*, vol. 22, no. 3, pp. 505–514, 2003.

[13] F. Losasso and H. Hoppe, “Geometry clipmaps: terrain rendering using nested regular grids,” *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 769–776, 2004.

[14] V. Pascucci, “Isosurface computation made simple: hardware acceleration, adaptive refinement and tetrahedral stripping,” in *Joint Eurographics - IEEE TVCG Symposium on Visualization (VisSym ’04)*, pp. 293–300, 2004.

[15] L. Buatois, G. Caumon, and B. Lévy, “GPU accelerated isosurface extraction on tetrahedral grids,” *International Symposium on Visual Computing*, pp. 383–392, 2006.

[16] M. Guthe, A. Balázs, and R. Klein, “GPU-based trimming and tessellation of NURBS and T-Spline surfaces,” *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 1016–1023, 2005.

[17] A. Patney and J. D. Owens, “Real-time Reyes-style adaptive surface subdivision,” *ACM Transactions on Graphics*, vol. 27, no. 5, pp. 143:1–143:8, 2008.

[18] C. Eisenacher, Q. Meyer, and C. Loop, “Real-time view-dependent rendering of parametric surfaces,” in *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games (I3D ’09)*, pp. 137–143, 2009.

[19] M. Schwarz and M. Stamminger, “Fast GPU-based adaptive tessellation with CUDA,” *Computer Graphics Forum*, vol. 28, no. 2, pp. 365–374, 2009.

[20] L.-J. Shiue, I. Jones, and J. Peters, “A realtime GPU subdivision kernel,” *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 1010–1015, 2005.

[21] A. Patney, M. S. Ebeida, and J. D. Owens, “Parallel view-dependent tessellation of Catmull-Clark subdivision surfaces,” in *Proceedings of High Performance Graphics 2009*, pp. 99–108, 2009.

[22] T. Boubekeur and C. Schlick, “Generic mesh refinement on GPU,” in *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (HWWS ’05)*, pp. 99–104, 2005.

[23] M. Bokeloh and M. Wand, “Hardware accelerated multi-resolution geometry synthesis,” in *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games (I3D ’06)*, pp. 191–198, 2006.

[24] C. DeCoro and N. Tatarchuk, “Real-time mesh simplification using the GPU,” in *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D ’07)*, pp. 161–166, 2007.

[25] J. Ji, E. Wu, S. Li, and X. Liu, “View-dependent refinement of multiresolution meshes using programmable graphics hardware,” *The Visual Computer*, vol. 22, no. 6, pp. 424–433, 2006.

[26] L. Hu, P. V. Sander, and H. Hoppe, “Parallel view-dependent refinement of progressive meshes,” in *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games (I3D ’09)*, pp. 169–176, 2009.

[27] L. Kobbelt, S. Campagna, and H.-P. Seidel, “A general framework for mesh decimation,” in *Proceedings of Graphics Interface*, pp. 43–50, 1998.

[28] J. El-Sana and A. Varshney, “Generalized view-dependent simplification,” *Computer Graphics Forum*, vol. 18, no. 3, pp. 83–94, 1999.

[29] H. Hoppe, “Smooth view-dependent level-of-detail control and its application to terrain rendering,” in *Proceedings of the IEEE Conference on Visualization (VIS ’98)*, pp. 35–42, 1998.

[30] L. D. Floriani, P. Magillo, and E. Puppo, “Efficient implementation of multi-triangulations,” in *Proceedings of the IEEE Conference on Visualization (VIS ’98)*, pp. 43–50, 1998.

[31] R. Pajarola and C. DeCoro, “Efficient implementation of real-time view-dependent multiresolution meshing,” *IEEE Trans. Visualization and Computer Graphics*, vol. 10, no. 3, pp. 353–368, 2004.

[32] H. Hoppe, “Progressive meshes,” in *Computer Graphics (Proceedings of SIGGRAPH 1996)*, pp. 99–108, 1996.

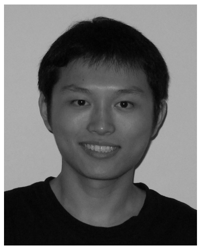
[33] L. Williams, “Casting curved shadows on curved surfaces,” in *Computer Graphics (Proceedings of SIGGRAPH 1978)*, pp. 270–274, 1978.

[34] A. Mammen, “Transparency and antialiasing algorithms implemented with the virtual pixel maps technique,” *IEEE Computer Graphics and Applications*, vol. 9, no. 4, pp. 43–55, 1989.

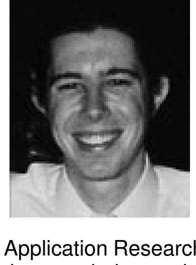
[35] C. Everitt, “Interactive order-independent transparency,” *Tech. rep., NVIDIA Corporation*, 2001.

[36] L. Bavoil, S. P. Callahan, A. Lefohn, J. L. D. Comba, and C. T. Silva, “Multi-fragment effects on the GPU using the k-buffer,” in *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D ’07)*, pp. 97–104, 2007.

[37] M. Harris, S. Sengupta, and J. D. Owens, “Parallel prefix sum (scan) with CUDA,” in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley Professional, August 2007.



Liang Hu received the BS degree in Computer Science from Huazhong University of Science and Technology, China, in 2006 and the MPhil degree in Computer Science and Engineering from the Hong Kong University of Science and Technology (HKUST) in 2009. His research interests include real-time rendering and geometry processing.



Pedro V. Sander is an Assistant Professor in the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology. His research interests lie mostly in real-time rendering, graphics hardware, and geometry processing. He received a Bachelor of Science in Computer Science from Stony Brook University in 1998, and Master of Science and Doctor of Philosophy degrees from Harvard University in 1999 and 2003, respectively. After concluding his studies, he was a member of the Application Research Group of ATI Research, where he conducted real-time rendering and general-purpose computation research with latest generation and upcoming graphics hardware. In 2006, he moved to Hong Kong to join the Faculty of Computer Science and Engineering at The Hong Kong University of Science and Technology.

Hugues Hoppe is a principal researcher and manager of the Computer Graphics Group at Microsoft Research. His primary interests lie in the multiresolution representation, parameterization, and synthesis of both geometry and textures. He received the 2004 ACM SIGGRAPH Computer Graphics Achievement Award for pioneering work on surface reconstruction, progressive meshes, geometry texturing, and geometry images. His publications include 30 SIGGRAPH/TOG papers, and he is currently editor-in-chief of ACM Transactions on Graphics. He obtained a BS summa cum laude in electrical engineering in 1989 and a PhD in computer science in 1994 from the University of Washington.