Topology-driven Progressive Mesh Construction for Hardware-Accelerated Rendering

Pavlo Turchyn* University of Jyväskylä

Sergey Korotov† University of Jyväskylä

Abstract

We present an algorithm for construction of progressive meshes, which are used for rendering from static memory buffers. Compared to previous work our scheme reduces memory storage requirements while maintaining a good vertex cache coherency.

Keywords: continuous level-of-detail, view-independent progressive mesh, vertex cache, sliding window

1 Introduction

Progressive meshes (or continuos level-of-detail) scheme encodes a sequence of meshes with different geometrical complexity [Hoppe 1996]. Originally, algorithm suggests updating memory buffers, which contain mesh's geometry (vertex buffer) and connectivity information (index buffer). Updates of vertex buffer may be avoided by using corresponding simplification operator (e.g. half-edge collapse [Luebke et al. 2002]) at the expense of some geometrical error increase. Sliding-window algorithm [Forsyth 2001] may be used to avoid updates of index buffer at the expense of a large resulting index buffer size. The latter algorithm is optimal in sense of memory management in major 3D APIs (e.g. Microsoft Direct3D) since changing memory buffers, which are located in GPU-accessible memory, imposes CPU overhead and needs additional memory to avoid breaking CPU-GPU parallelism. Moreover, instancing does not require any additional memory.

Since the introduction of FIFO cache for post-processed vertices [Hoppe 1999], it has been widely implemented in the commodity hardware. As the complexity of per-vertex computations grows, the vertex processing performance improvements focused on cache miss rate. Although vertex cache is transparent to the application, a care must be taken to avoid its trashing. [Bogomjakov and Gotsman 2001] presents a *smart update* algorithm for progressive meshes that preserves original optimized rendering order. The idea of preserving original order is also exposed in *skip-strips* scheme [Luebke et al. 2002; Forsyth 2001], which relies on the hardware's ability to efficiently detect degenerate triangles. However, such schemes still require updating index buffer.

Our algorithm uses sliding-window scheme for rendering of progressive mesh, while trying to minimize required memory size. Also, we show that when optimization for vertex cache is incorporated as one of the objectives into the simplification process, the resulting ACMR is nearly identical to discrete LOD optimized with standalone vertex-cache optimizers like NvTriStrip [NVI 2001].

2 Previous work

Mesh M is defined as a set of vertices and a set of indices that define their connectivity. We seek to construct such a sequence $r_1, ..r_n$ of parameters for simplification operator S, where each r_i minimizes simplification metric E

$$r_i = \min_{r \in R} E(S^{r_{i-1}} S^{r_{i-2}} ... S^{r_0} M)$$

Throughout this paper we use the vertex removal operator, but the same reasoning may apply to other operators. We call a set of triangles, which are incident to the removed vertex, a patch p. The latter vertex is called center of patch.

Assume that we have a mesh on which a list P of non-overlapping patches is defined. The index buffer is initialized with P. Progressive mesh construction algorithm iteratively takes patch from the top of the list, apply simplification operator and then append resulting triangulation to the the end of index buffer.

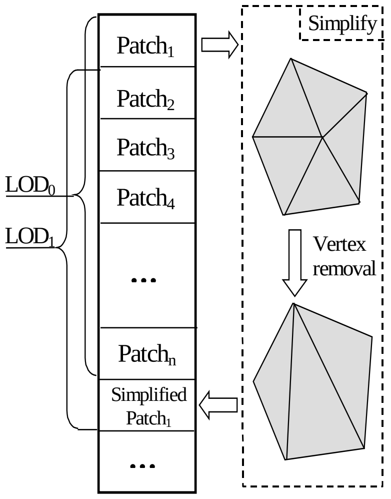

Figure 1: Index buffer in sliding window algorithm

When all patches in the list are processed, one has to perform a next pass (restart algorithm) and build a new P from the simplified mesh. Then new P is processed in the same manner. In such scheme we can render a mesh with the required LOD simply by choosing appropriate window in the index buffer. The index buffer is remaining static, which gives many desired properties to the algorithm: index buffer can be shared among all mesh instances; no CPU overhead

*e-mail: pturchy@cc.jyu.fi

†e-mail: korotov@mit.jyu.fi

occurs due to locking or copying data; no additional memory is required for *rename buffers*, which hold data still being processed by GPU.

The major problem of the algorithm is a size of the index buffer, since approximately 4/6 of indices have to be duplicated within a single algorithm iteration, and also when next pass occurs we have to rearrange processed P into a new collection of patches, which effectively duplicates P.

3 Topology-driven progressive mesh construction

Typically, mesh simplification process is guided by the simplification metric E, which accounts geometrical and attributes error. However, to construct optimal progressive mesh for sliding-window algorithm we follow such objectives, which take into account mesh connectivity:

- Minimize size of static index buffer. This factor mainly depends on the number of passes we perform, which in turn depends on how many collapses we perform each pass. In order to maximize the number of collapses we have to maximize the size of P.

- Maximize vertex cache coherency during rendering. This may be done by rearranging patches such that cached vertices are reused among patches.

3.1 Patch coverage problem

First, we describe a greedy algorithm to find P of maximal size. Assume that we have a mesh with initially uncolored vertices. When we choose a patch, we paint all patches' vertices into some color. Note that any colored vertex cannot be center of patch since patches cannot overlap. Thus, the strategy of a greedy algorithm is to choose a patch, which introduces the least amount of new colored vertices, since this will leave the biggest number of unpainted vertices that can be patch centers. This strategy can be formulated as the following heuristic

$$E_{patch}(p_i) = \frac{\sum_{v_j \in V} F(v_j)}{|V|}, \tag{1a}$$

$$V = \{v_j\},\ v_j \in p_i, \tag{1b}$$

$$F(v_j) = \begin{cases} 0, & \text{if } v_j \text{ is colored} \\ 1, & \text{otherwise.} \end{cases} \tag{1c}$$

The problem of building P is a maximal clique problem on a graph, which complements vertices connectivity graph. This problem is NP-complete. We have compared performances of the greedy algorithm and QUALEX-MS package, which is based on Motzkin-Straus quadratic programming formulation with the complexity $O(n^3)$ [Busygin 2002].

Data	Triangles	Patches found	
		QUALEX	Greedy
Venus10	6718	1015	984
Venus	67170	9891	9739
Bunny	69451	10316	10301

Table 1: Performance of QUALEX-MS and greedy algorithm

Two algorithms demonstrate nearly identical performance. On the other hand, greedy search took several seconds, while QUALEX required up to several minutes to perform its first iteration.

3.2 Optimization for vertex cache

We define the heuristic for a greedy construction of rendering sequence. Each iteration algorithm looks for a patch that minimizes functional

$$E_{vcache}(p_i) = \frac{\sum_{v_j \in V} K(v_j)}{|V|}, \tag{2a}$$

$$V = \{v_j\},\ v_j \in p_i, \tag{2b}$$

$$K(v_j) = \begin{cases} 0, & \text{if } v_j \text{ is in cache} \\ 1, & \text{otherwise.} \end{cases} \tag{2c}$$

Such heuristic chooses a patch that introduces the least amount of cache trashing. The convenience of greedy solution is that (2) can be easily incorporated into the functional E. The alternative approach is to construct universal rendering sequence using recursive cut or minimum linear arrangement algorithms [Bogomjakov and Gotsman 2001]. In practice, average cache miss rate per triangle (ACMR) demonstrated by all these methods is nearly identical.

3.3 Progressive mesh construction

Using weighting method both (1) and (2) may be easily incorporated into the simplification metric E (in theory, along with geometrical error, but it is not used for purely topologically-based algorithm). However, since patch coverage problem is more important for practical use, it is advantageous to optimize for this criterion first, and then for the vertex cache.

```
While |M|>0
  P=FindPatches(M)
  IndexBuffer.Append(P)
  While |P|>0
    p=GetPatch(P)
    P.Remove(p)
    If IsSimplificationValid(p) then
       Simplify(p)
    IndexBuffer.Append(p)
```

FindPatches builds a list of patches using greedy search as described in Section 3.1. *GetPatch* returns a patch, which minimizes (2). *IsSimplificationValid* determines if simplification of the patch results into some undesired effects, e.g. face-flipping in half-edge collapse operator. Choosing good criteria for *IsSimplificationValid* is important in order to avoid destruction of important geometrical features.

4 Numerical results

In our numerical experiments we have compared our method with two alternative approaches of LOD construction, which also offer advantages of static memory buffers. First approach is discrete LOD (DLOD) where each level is created using quadratic error metric (QEM)-based simplification algorithm with sharp features preservation [Garland and Heckbert 1998], and then optimized using NvTriStrip library. The other approach is directly taken from [Forsyth 2001]. It is QEM-guided construction of view-independent progressive mesh (VIPM) for sliding-window algorithm. We used vertex contraction operator for DLOD construction. For both VIPM and topology-driven progressive mesh (TD-VIPM) algorithms we used half-edge collapse operator.

Figures 2–3 show geometrical error (Hausforff distance between the original mesh and simplified mesh) of different LOD. Measurements were done using Metro tool [Cignoni et al. 1998]. Expectedly, the best results are demonstrated by DLOD algorithm since it is using unconstrained geometry-guided simplification. VIPM is only marginally better than TD-VIPM. The explanation of this fact is that we kept the size of index buffer as small as possible by trying not to start new pass until it is absolutely required. However, the size of resulting index buffer in case of VIPM algorithm is still larger than the one in case of TD-VIPM (see Table 2).

Data	Triangles	Index buffer, %	
		TD-VIPM	VIPM
Bunny	69451	550	656
Hugo	16374	510	560
Venus	67170	556	655

Table 2: Static index buffer size in % of index buffer of the original model

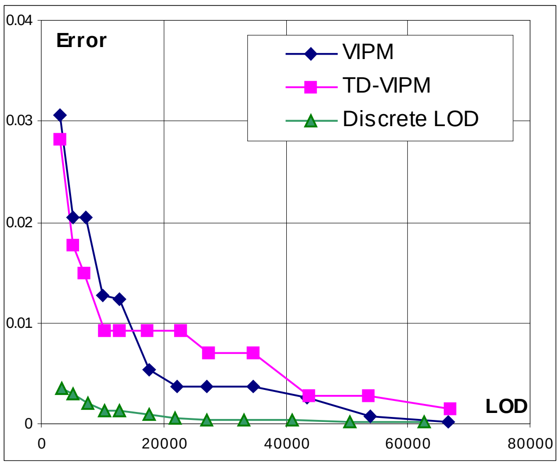

Figure 2: Normalized Hausdorff distance for Bunny

In terms of vertex cache coherency, VIPM method demonstrates moderate results with ACMR that is close to one (see Figure 4). Both TD-VIPM and DLOD algorithms show nearly identical performance in the region of 10-20 cache entries (see Figures 4–5).

Summarizing, topology-driven method demonstrates significantly better performance than geometry-driven one when sizes of index buffer are of the same order. On the other hand, DLOD offers (arguably) better visual appearance while TD-VIPM offers better flexibility. We claim that TD-VIPM can replace DLOD in applications where rendering speed is the priority factor.

5 Conclusions

Simplification based solely on the topology is naturally suitable for animated meshes, where exact geometry is unknown

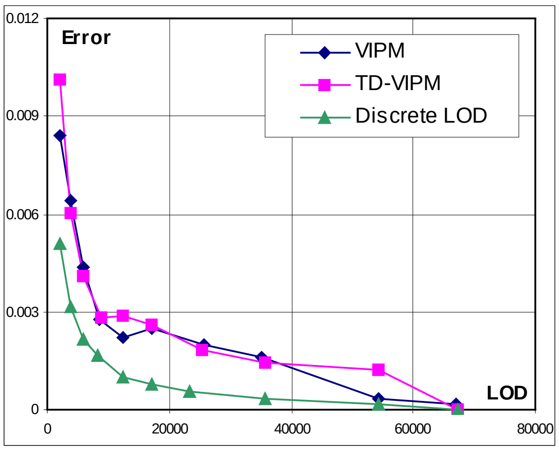

Figure 3: Normalized Hausdorff distance for Venus

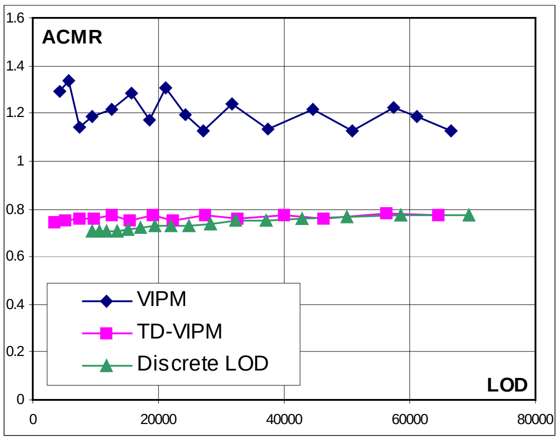

Figure 4: ACMR for Bunny, vertex cache size 16

a priory. Although geometrical error is mostly ignored in our scheme, the numerical experiments show that simplification results are still acceptable, particularly for semi-regular grids. Because of a good vertex cache coherency and static nature of involved memory buffers, our scheme is aimed at real-time applications, such as video games, architectural walkthrough systems etc, where precise load-balancing, which is offered by progressive meshes, is important in order to maintain interactive frame rates.

Acknowledgements

The Bunny and Happy Buddha models are courtesy of the Stanford Scanned Data Repository.

References

Bogomjakov, A., and Gotsman, C. 2001. Universal rendering sequences for transparent vertex caching of progressive meshes. In *Proceedings of Graphics Interface 2001*, B. Watson and J. W. Buchanan, Eds., 81–90.

Busygin, S., 2002. A new trust region technique for the maximum weight clique problem.

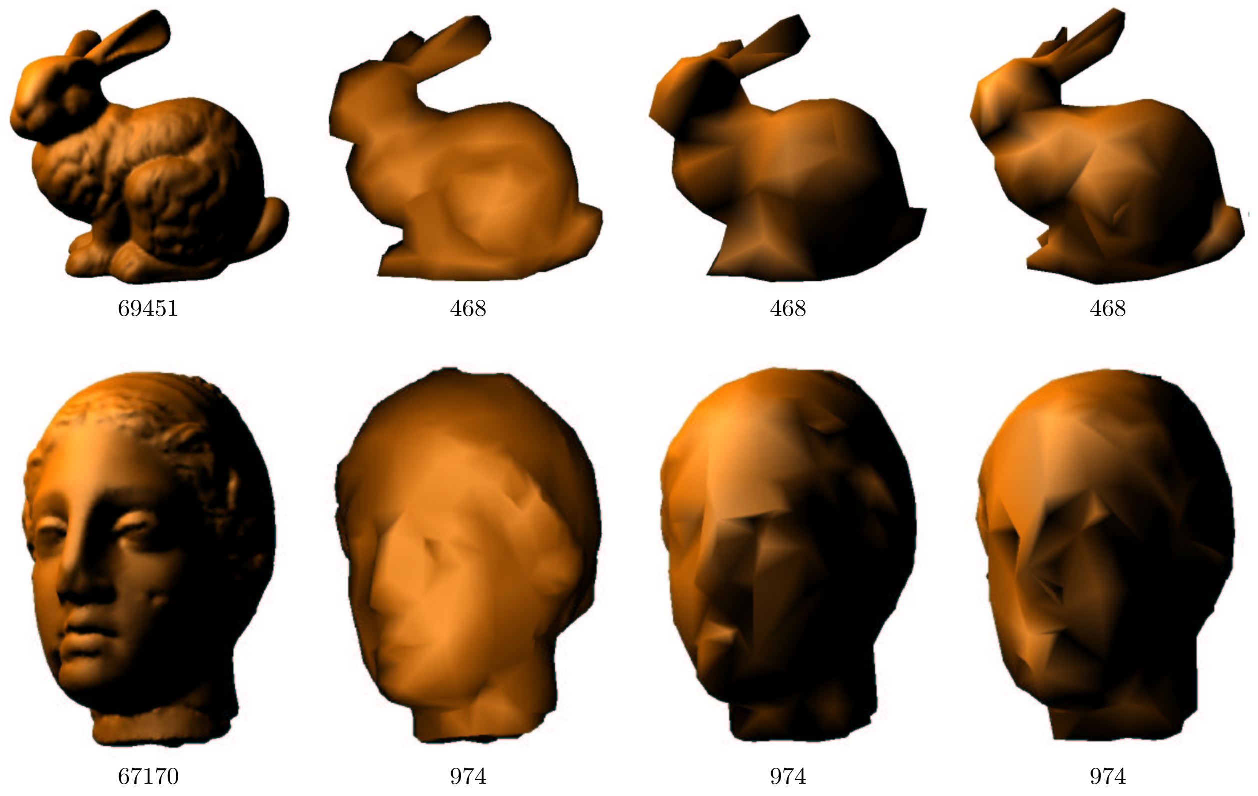

Figure 6: From left to right: original mesh, DLOD, VIPM, TD-VIPM

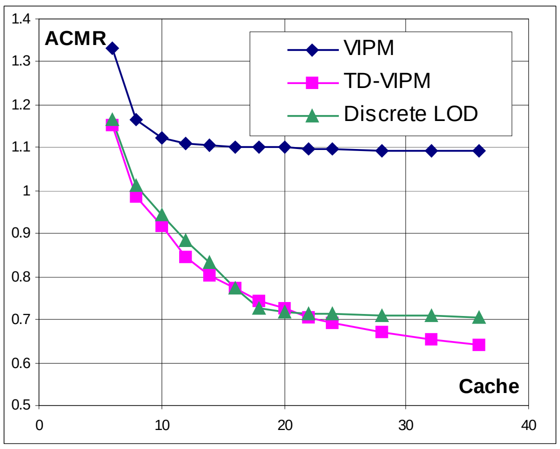

Figure 5: ACMR for Bunny, 50% LOD

Cignoni, P., Rocchini, C., and Scopigno, R. 1998. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum 17*, 2, 167–174.

Forsyth, T. 2001. Comparison of vipm methods. In *Game Programming Gems 2*, Charles River Media, M. DeLoura, Ed., 363–376.

Garland, M., and Heckbert, P. S. 1998. Simplifying surfaces with color and texture using quadric error metrics. In *IEEE Visualization '98*, D. Ebert, H. Hagen, and H. Rushmeier, Eds., 263–270.

Garland, M., 1998. Quadric-based polygonal surface simplification.

Hoppe, H. 1996. Progressive meshes. *Computer Graphics 30*, Annual Conference Series, 99–108.

Hoppe, H. 1999. Optimization of mesh locality for transparent vertex caching. In *Siggraph 1999, Computer Graphics Proceedings*, Addison Wesley Longman, Los Angeles, A. Rockwood, Ed., 269–276.

Luebke, D., Reddy, M., Cohen, J., Varshney, A., Watson, B., and Huebner, R. 2002. *Level of Detail for 3D Graphics*. Computer Graphics and Geometric Modeling. Morgan Kaufmann.

NVIDIA Corporation. 2001. *NvTriStrip: a library for vertex cache aware stripification of geometry.*