Morton Integrals for High Speed Geometry Simplification

Hélène Legrand & Tamy Boubekeur
Telecom ParisTech - CNRS LTCI - Institut Mines-Telecom

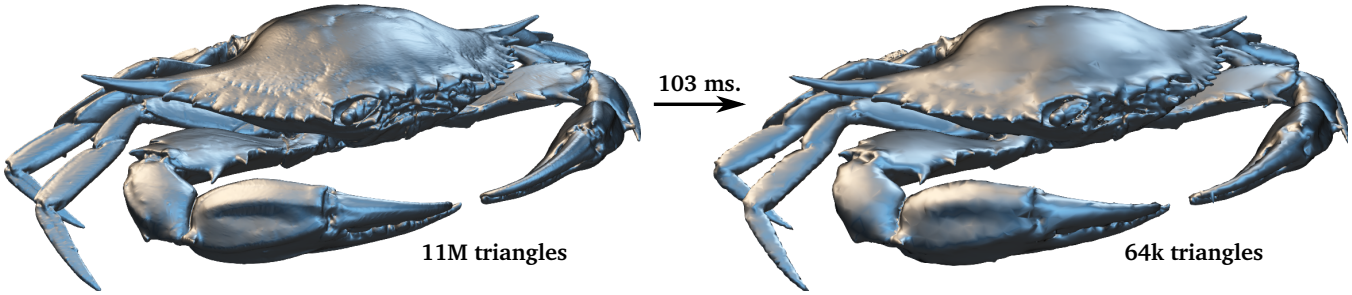

Figure 1: *Adaptive simplification using our algorithm: even beyond 10M tri., our parallel approach remains nearly interactive.*

Abstract

Real time geometry processing has progressively reached a performance level that makes a number of signal-inspired primitives practical for on-line applications scenarios. This often comes through the joint design of operators, data structure and even dedicated hardware. Among the major classes of geometric operators, filtering and super-sampling (via tessellation) have been successfully expressed under high-performance constraints. The subsampling operator i.e., adaptive simplification, remains however a challenging case for non-trivial input models. In this paper, we build a fast geometry simplification algorithm over a new concept: Morton Integrals. By summing up quadric error metric matrices along Morton-ordered surface samples, we can extract concurrently the nodes of an adaptive cut in the so-defined implicit hierarchy, and optimize all simplified vertices in parallel. This approach is inspired by integral images and exploits recent advances in high performance spatial hierarchy construction and traversal. As a result, our GPU implementation can downsample a mesh made of several millions of polygons at interactive rates, while providing better quality than uniform simplification and preserving important salient features. We present results for surface meshes, polygon soups and point clouds, and discuss variations of our approach to account for per-sample attributes and alternatives error metrics.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry & Object Modeling—Geometric Algorithms I.3.5 [Computer Graphics]: Computational Geometry & Object Modeling—Hierarchy and Geometric Transformations; I.3.1 [Computer Graphics]: Hardware Architecture—Parallel Processing; I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors;

Keywords: mesh simplification, GPU algorithms, adaptive clustering, Morton code.

1 Introduction

Modern 3D capture pipelines allow acquiring real world geometry quickly and accurately. The increasing data generation speed has motivated a large number of research projects, to provide visual computing systems with geometric operators able to process this data in real time. In these constrained scenarios, approximations and a controlled loss of quality are preferable to exceeding the time limit. This trend in high performance geometry processing had a number of successes, with now solutions for performing some of the most critical processing steps on-the-fly, for non-trivial input and on commodity hardware. This includes for instance adaptive feature-preserving filtering [Adams et al. 2009] and adaptive smooth tessellation [Pixar 2013] (i.e., upsampling) which are now supported by high-performance data structures and even dedicated hardware and programmable graphics stages.

The case of adaptive geometry simplification (e.g. mesh simplification or point cloud subsampling) remains however challenging. The key problem in this case lies in the data compaction mechanism, which does not cope naturally with a fine-grained parallel computing environment. Although two decades of research have progressively led to simplification algorithms offering a controllable trade-off between output surface quality and computational effort, we are still far from high quality, real time simplification for non-trivial (e.g., multi-millions) sample sets, and current aggressive simplification methods have a limited range of applications. However the interactive and real time scenarios we target may benefit from higher quality simplifications, with applications such as interactive display on mobile devices, dynamic multiview rendering or live broadcast of 3D captured data (e.g., 3D camera). In such applications, the resulting geometry may be discarded after a short amount of time, required immediately after full resolution data generation or needed on-demand, in multiple versions under a wide variety of simplification ratios. In this case, it is crucial to provide instantly visually good simplifications.

Indeed, a key aspect of high performance mesh processing, largely exploited for filtering and tessellation, is the parallel scalability of the operators and their ability to run on the graphics processor unit (GPU). In the context of adaptive simplification/downsampling, we make two observations. First, fast adaptive sampling often relies on an underlying hierarchical data structure which is costly to generate and maintain in a parallel environment. Second, with such a hierarchy in hand, simplifying the mesh often means extracting a particular, error-driven "cut" in the tree structure. This operation, either with a bottom-up of a top-down approach, does not map triv-

ially on a fine-grained parallel architecture.

In this paper, we propose a step toward higher quality real time surface optimization, in the form of a fast simplification method which provides adaptive error-driven mesh samplings, while staying within the real time rates required by our target applications for typical input/output sizes (see Fig. 1). We address the hierarchy problem using the *Morton order* of the input surface samples, over which we compute a one-dimensional sum of the geometric cost associated with each surface sample. This intermediate representation allows for an efficient kd-tree construction and later for concurrent evaluation of all the geometric errors to be estimated on all the nodes of the tree. It also enables the parallel processing of both leaves and inner nodes, for a constant and predictable per-node cost. We use the Quadric Error Metric [Garland and Heckbert 1997] as our basic cost measure for two reasons: first, it remains the state-of-the-art measure for general simplification. Second, quadrics have the nice property to sum to quadrics, which allows us to address the problem of multi-level error computation using the same principle as *integral images* (a.k.a. summed area tables [Crow 1984]). The hierarchy defined by the Morton codes of the samples requires a single sort to be performed at the beginning of the algorithm. Although the resulting space clustering is less accurate than a pure error driven tree refinement/aggregation (e.g., BSP tree), it significantly improves over uniform GPU clustering techniques. As a result, our algorithm can simplify large meshes, polygons soups and even point clouds in real time, accounting for the geometric features, but also for additional attributes on the surface. We present experiments on a collection of models and discuss possible evolutions of our approach.

2 Previous work

Most mesh simplification methods define an objective optimization criterion, with a metric which measures the error caused by the simplification in the form of some distance between the original object and the simplified output. Simplification algorithms usually fall into two categories : *iterative simplification* and *vertex clustering*.

Iterative methods [Hoppe et al. 1993; Garland and Heckbert 1997] progressively reduce the number of primitives of the mesh by performing, at each step, a local simplification operation causing the smallest error according to the chosen metric. These methods usually lead to high quality output meshes but are difficult to parallelize efficiently due to their sequential nature. They also usually require a clean mesh connectivity which, in the context of instantaneous capture and processing, can hardly be guaranteed. Although their method does not reach real time rates, Grund et al. [2011] propose a parallel simplification algorithm based on this approach.

We focus on clustering methods, which optimize for a simplified mesh at a coarser grain, by defining a partition of the mesh, computing a representative vertex/polygon for each cluster and meshing the resulting (smaller) geometric set. The choice of a particular partitioning structure has a strong impact on the overall performance of the process, with solutions including simple grids [Rossignac and Borrel 1993; Lindstrom 2000], octrees [Schaefer and Warren 2003][Lindstrom 2003][Shaffer and Garland 2005], BSP-trees [Shaffer and Garland 2001] and k-means partitions [Cohen-Steiner et al. 2004]. The representative element is again chosen to optimize a certain metric, for which popular choices include the Quadric Error Metric (QEM) [Garland and Heckbert 1997] which models the simplification cost as the sum of the squared distances from the representative point to the planes defined by the triangles of the cluster; or the $L^{2,1}$ [Cohen-Steiner et al. 2004] metric which uses the normal information to grow large flat clusters whenever possible. The final meshing step is performed by either reindexing input triangles

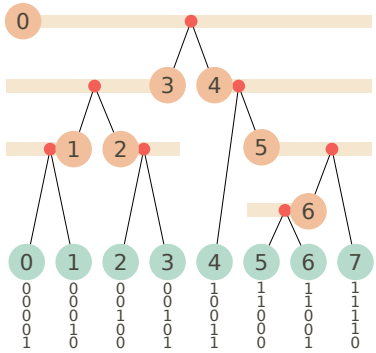

Figure 2: ***Tree layout*** *in the method of Karras. The yellow bars represent the range of leaf nodes (in green, with their Morton code) covered by the internal nodes (in orange). The red dots indicate the binary split position.*

intersecting three different clusters of the partition to the related representatives [Rossignac and Borrel 1993; Boubekeur and Alexa 2009] or generating polygons tracking clusters boundaries.

Since each cluster is (mostly) processed independently, vertex clustering methods are more adapted to parallel computing and GPU architectures. In particular, Decoro and Tatarchuk [2007] have proposed a GPU implementation of QEM-based grid simplification [Lindstrom 2000] as well as a probabilistic octree structure providing adaptivity.

Regarding the type of simplification obtained, our method falls in the same category as [Schaefer and Warren 2003], [Lindstrom 2003] and [Shaffer and Garland 2005], with an axis aligned hierarchy depending on a spatially coherent ordering of the primitives. Most of our contribution is about providing very similar results in terms of quality, while performing a fully parallel simplification at interactive to real time rates.

Hierarchical space subdivision structures provide a good trade-off between full adaptivity and grid clustering to create the initial partition. The efficient generation of such structures has been mostly studied in the context of rendering applications. In particular, Morton curves (or z-order curves) have proved to provide an efficient space parametrization supporting the hierarchy construction [Lauterbach et al. 2009; Pantaleoni and Luebke 2010; Garanzha et al. 2011]. More precisely, a Morton code is calculated for each primitive by interleaving the bits of its binary coordinates. Sorting them by their Morton code then groups the primitives in a spatially coherent manner. This allows for the parallel construction of a binary tree, level by level, starting from the root, each node representing a contiguous range of primitives. A similar idea is exploited by Zhou et al. [2010] to build octress in the context of surface reconstruction.

Our underlying GPU structure is based on the work of Karras [2012], who maximize the tree construction parallelism reaching real time performances on models beyond 1M polygons. Instead of generating one level of the tree at a time, all the nodes are processed in parallel thanks to a particular tree layout which allows finding the range covered by a node and its children independently from the other nodes.

More precisely, the leaf nodes and the internal nodes are stored in two distinct arrays: L (size n) and I (size $n-1$). By assigning the right indices to the internal nodes, Karras finds the range of leaf nodes they cover and the indices of their children, without processing their ancestors or descendants first. In practice, the index of the

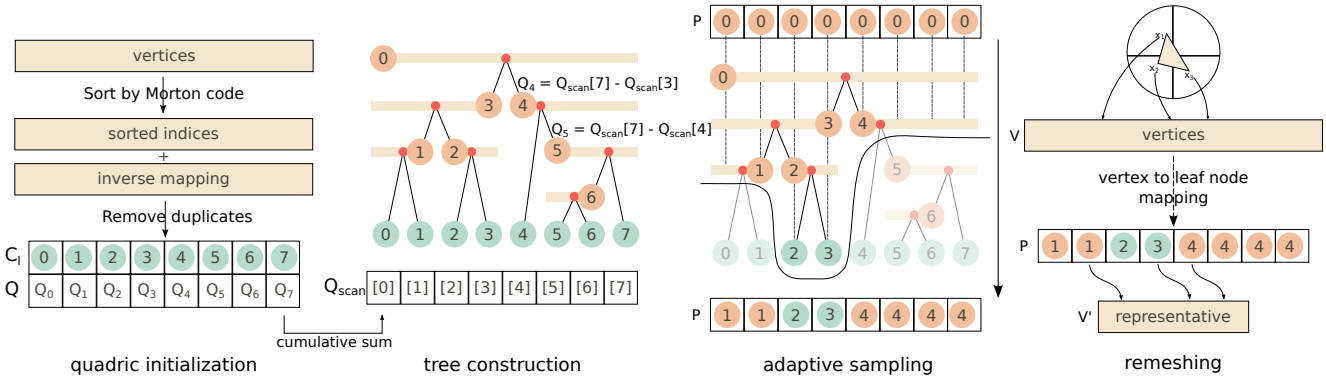

Figure 3: ***Overview****. We start by sorting the input vertices in the Morton order and generate a table of leaves (left) onto which we accumulate per-leaf error quadrics in the Morton order. We then generate a kd-tree, using the Morton integral to compute the internal nodes in parallel (middle left). Finally, an error-driven simplified geometry is generated by extracting an adaptive cut in the tree in parallel (middle right) and meshing its representative vertices using the input connectivity (right).*

root is set to 0. Then, for every internal node, the indices of its children directly depend on the split position: if the range is split at the position p, the index of the left child will be p (in L if it's a leaf, in I otherwise) and $p+1$ for the right child. Consequently, the index of a node corresponds to one end of the range of leaves it covers. The other end of the range and the split position are found using a binary search, detecting the first differing bit of the Morton code in the range (see Fig. 2). We refer to [Karras 2012] for details on the construction of the structure.

3 Algorithm

Overview Our algorithm (illustrated in Fig. 3) takes as input an indexed triangle mesh M with an error threshold θ and runs entirely on GPU. It provides a simplified indexed mesh M' as an output and consists in several stages:

1. we sort the vertices of M by their Morton code and define a list of leaf nodes with error quadrics,
2. we compute a *Morton integral*, which is a cumulative sum of the quadrics in the Morton order,
3. we compute the internal nodes of a kd-tree K in a parallel, non-hierarchical fashion using the Morton integral,
4. we gather the space partition S as the highest nodes of K with an error lower than θ; the simplified mesh is formed by the representative vertices of S computed using the per-node quadric,
5. we re-index the input triangles shared among three *different* clusters on their representative point [Rossignac and Borrel 1993] to define the simplified connectivity.

In the following, we describe the main steps of our algorithm, essentially structured by the tree construction and traversal, and point to relevant parallel primitives.

Morton sorting During the first step of our algorithm, we sort the vertices by their Morton code. Instead of directly sorting the vertex array (as well as eventual color or normal arrays), we sort by key an array of integer indices, ranging from 0 to the number of vertices of the input mesh, using the Morton code of the vertices as key. We also maintain a look-up table of the inverse mapping from the intial vertex order to the Morton one. Once sorted, we eliminate the duplicates in the Morton order – caused by multiple close-by vertices discretized to the same Morton code. As our array is sorted, duplicates are contiguous and we can simply:

1. mark the first occurence of each Morton code,
2. perform a prefix sum over the marking, to generate the mapping from the sorted vertex array to the compact leaf node array
3. allocate a compact array C_l and store the duplicate-free leaf nodes in it.

As a result, we obtain a mapping from the initial vertex array V to the leaf node array: $V \rightarrow C_l$.

Quadrics initialization For every leaf node l, we compute a 4x4 symmetric quadric matrix Q_l following Garland and Heckbert [1997]. To do so, we compute the face quadric Q_t of each triangle t of M and sum it to the leaf quadrics of the vertices of t: $Q_l = \sum_{t \in l} Q_t$. Assuming the Morton code is computed with a fine enough discretization, performing this sum in parallel using *atomics* induces only a low number of actual collisions between the threads. Additionally, we store the mean vertex of each leaf node, its number of vertices and, optionally, the color and/or the normal vectors. As usual with quadric-based optimization, we consider that Q_l locally models the shape through its minimizer, a representative point obtained by either inverting Q_l, or falling back to the previously stored mean position when the determinant of Q_l is below a numerical stability threshold [DeCoro and Tatarchuk 2007].

Morton integrals Beyond the Morton-based hierarchy, one key aspect of our approach lies in the ability to compute concurrently the error and representative point for all internal node, regardless of the current state of their descendants. We solve this issue by computing a *cumulative sum* of attributes (quadric matrix, mean position, etc) along the Morton ordered leaves, defining in particular an additional node attribute Q_{scan} summing all the quadrics stored in the preceding nodes in the Morton order. The computation of this sum is performed in parallel using an *inclusive scan*. Similar to *integral images* [Viola and Jones 2001], this "Morton integral" allows to compute any sum of attributes for consecutive leaves with only two memory accesses; for instance in the case of the quadric of a node n covering leaves r_1 to r_2:

$$Q_n = Q_{scan}[r_2] - Q_{scan}[r_1 - 1]$$

Parallel tree construction We build our kd-tree K by processing internal nodes independently following Karras [2012], using our Morton-ordered leaf node array C_l, enhanced by the Morton

integral. For each internal node n of K, we need to compute its quadric matrix Q_n and its average vertex (position, color, etc) to figure out its own representative point x_n and approximation error E_n w.r.t. the original geometry of M. Q_n is the sum of the quadrics associated with the children of n, which is equivalent to the sum of the quadrics of all the leaves having n as an ancestor. Thanks to the Morton ordering, they are all contiguous and by construction [Karras 2012] trivial to determine (from index r_1^n to index r_2^n). As a result of our Morton integration, this sum is available in constant time, using 2 quadric accesses and one 4x4 matrix subtraction per node. Therefore, it can be performed during the parallel generation of the tree nodes, independently of the node's children. From Q_n, we extract x_n [Garland and Heckbert 1997; Lindstrom 2000] and the quadric error:

$$E_n = (x_n, 1)Q_n(x_n, 1)^T$$

Algorithm 1 Parallel tree traversal.

```
for each leaf node i in parallel do
    c ← 0
    error ← +∞
    while error > θ and c ∈ internal nodes do
        r ← right[c]
        l ← left[c]
        if i > l then
            c ← r
        else
            c ← l
        end if
        error ← errorNode[c]
    end while
    P[i] ← c
end for
```

Adaptive sampling and remeshing Our error-driven simplification gathers an adaptive space partition of M as a cut in K, formed by the highest nodes of K with an approximation error lower than θ (see Alg. 1). In our parallel context, we formulate this cut extraction as the computation of a mapping, associating each element of C_l to its corresponding node in the target cut. To do so, we initialize an array P, as large as C_l, filled with zeros i.e., the root index. Then, we launch a thread i for each element of C_l that performs a top-down traversal of the tree toward the i-th leaf cell in C_l. When passing a node with index $P[i]$, if $E_{P[i]} < \theta$, we stop the traversal. Otherwise, we set $P[i]$ to one of its child nodes, depending on the value of i: again, by construction, the index of the child nodes correspond to the splitting position in the leaf node array (see Fig. 3). Therefore if i is smaller or equal to the index of the left child, we set $P[i]$ to its value, otherwise we set it to the index of the right child. Consequently, the traversal is always done toward the element of C_l (or leaf node) of index i.

At the end of this procedure, any vertex x of V can be mapped to a node of the target adaptive cut of K in constant time using our $V \rightarrow C_l$ mapping, since it is equivalent to $V \rightarrow P$. We collect in parallel the representative vertices for the nodes of the cut to form the vertex set V' of M' by marking the nodes of the tree that appear in the cut and scanning the marking array. This provides the size to allocate for V' as well as for each node of the cut, the index where its representative vertex should be written in V' (mapping $P \rightarrow V'$).

Last, we generate the connectivity T' of the simplified vertex set by classifying all input triangles of M in parallel according to P. We mark the triangles shared by three different clusters and use a parallel prefix sum to allocate the output triangle array T'. We fill this array with the marked triangles, reindexed over the representative vertices thanks to our $V \rightarrow P \rightarrow V'$ mapping. At this stage, for each triangle, we can optionally check if its normal orientation has flipped and reorder its vertices if necessary.

Variations Our simplification method can account for per-vertex attributes, alternative error metrics or input data in different formats.

For instance, the per-vertex color value can be maintained for each node similarly to quadric matrices. Indeed, it is often desirable to weight color averages by the area of incident triangles. This requires two additional arrays: one used to accumulate the weighted color information (Col) and for accumulating areas (A). After the Morton integral computation, the color for any node is given by:

$$Col_n = \frac{Col_{scan}[r_2] - Col_{scan}[r_1 - 1]}{A_{scan}[r_2] - A_{scan}[r_1 - 1]}$$

The exact same process can be used for other attributes, such as normals. In the last part of the algorithm, this extra per-node value influences the cut extraction by, for instance, bounding its standard deviation in all cut nodes. In this case, one more array (Col^2) is needed, to accumulate the squared weighted color information. After computing its Morton integral, the standard deviation for the color of a node is:

$$S_n = \sqrt{\frac{Col^2_{scan}[r_2] - Col^2_{scan}[r_1 - 1]}{A_{scan}[r_2] - A_{scan}[r_1 - 1]} - [Col_n]^2}$$

with Col_n the mean color for the node.

Unorganized point clouds with normals can also be simplified with our approach by (i) computing the quadric matrices directly from point normals, (ii) propagating the normal for each node as we do for color and (iii) omitting the final meshing step. Note however that, unless per-sampled area/radius is provided, this method requires data with relatively uniform sampling.

4 Implementation and results

We implemented our simplification algorithm in C++/CUDA, using the Thrust library [Nvidia 2011] for the prefix sums and sort, and measured performances on a PC equipped with a GeForce GTX 680 and a 3.6 GHz Intel Xeon E5-1620 CPU. We limited our Morton codes to 30 bits as we store them as 32 bits integers in GPU memory. For comparison, we also implemented a GPU regular grid simplification, similar to the method of Decoro and Tatarchuk [2007]. For quality check, we report high quality offline results obtained with QSlim [Garland and Heckbert 1997].

In Table 1, we report the detailed timings of the different steps of our algorithm for a collection of models. We can see that real time performances are reached for meshes up to several millions of polygons and remain interactive beyond 10 millions. While the tree construction and sampling parts add up to a very small part of the total time, the current bottleneck appears at the initial phase. Note however that for application scenarios implying multiple simplifications of the same model (e.g., many-users remote visualization, view-dependent rendering), this stage is performed once for all. In terms of visual quality, as we can see on the Lucy model for instance (Fig. 4), our adaptive space partition preserves visually important features, with small triangles around features and larger ones in flatter parts.

Model	#T	Out #T	Sort	Dupl	Leaves	Scan	Cons	Sampl	Mesh	Total
Bunny	70K	4,300	0.8	0.2	1.3	1.0	1.8	0.1	1.2	6.4
Dragon	100K	9,300	1.1	0.4	3.9	1.2	2.8	0.1	1.5	11.0
Horse	225K	10,000	1.4	0.7	1.9	1.6	3.0	0.1	1.6	10.3
Buste	510K	19,200	1.7	0.3	5.4	1.6	3.5	0.5	1.9	14.9
Caesar	770K	18,500	2.3	0.7	5.6	2.5	4.4	0.9	2.1	18.6
Grog	1M	41,000	2.8	1.1	5.0	3.5	5.1	0.9	2.3	20.8
Gargoyle	1.7M	44,500	3.4	1.0	7.1	3.4	4.1	0.6	2.5	22.1
Raptor	2M	22,500	3.6	1.2	8.8	2.8	1.7	0.2	2.2	20.6
Neptune	4M	24,500	4.6	1.3	30.1	1.8	2.4	0.3	6.6	47.0
Crab	11M	64,200	12.8	3.3	69.8	2.4	6.0	0.9	11.4	106.5
Lucy	28M	116,500	29.7	5.6	151.2	6.1	5.5	0.8	23.9	222.7

Table 1: ***Performance measures*** *in ms, without CPU-GPU memory transfer.* ***Sort****: Morton sorting,* ***Dupl****: duplicates removal.* ***Leaves****: initialization of the leaves attributes (quadrics and mean).* ***Scan****: Morton integration.* ***Cons****: parallel tree construction.* ***Sampl****: error-driven tree traversal and cut extraction.* ***Mesh****: triangles re-indexing.*

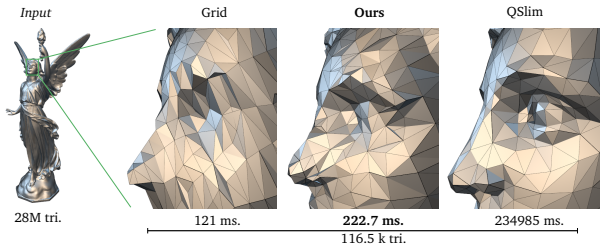

Figure 4: ***Adaptive simplification*** *of the Lucy model. Our approach approximates better features and singular structure than uniform clustering, but still runs in split-second for this large model.*

Model	model size	additional space used	
		27 bits MC	30 bits MC
Bunny	1	7	7
Dragon	2	8	9
Horse	5	19	20
Buste	11	32	37
Caesar	17	61	63
Grog	22	75	80
Gargoyle	39	98	120
Raptor	45	44	89
Neptune	91	83	138
Crab	259	217	275
Lucy	642	502	557

Table 2: ***Memory usage*** *in Mb for 27 bits and 30 bits Morton codes.*

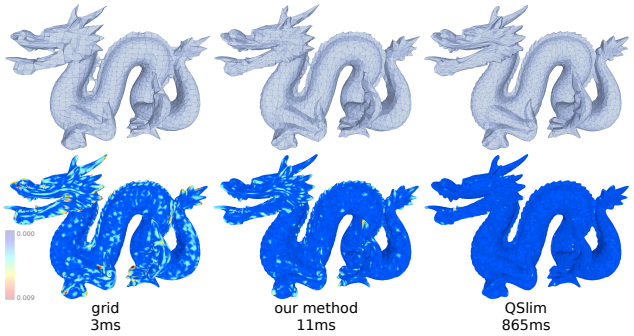

Figure 5: ***Error visualization*** *for the Stanford Dragon model (100K triangles), simplified to 9300 triangles.*

Model	Method	#T	Out #T	Time	H	M12	M21
Bunny	Ours	70K	4,300	6.4	0.013802	0.000500	0.000499
	QSlim			503	0.002425	0.000295	0.000288
	Grid			2.5	0.012880	0.001327	0.001294
Dragon	Ours	100K	9,300	11.0	0.008810	0.000671	0.000585
	QSlim			865	0.009327	0.000370	0.000251
	Grid			3	0.013838	0.001142	0.001039
Horse	Ours	225K	10,000	10.3	0.004485	0.000280	0.000280
	QSlim			1,728	0.001862	0.000104	0.000101
	Grid			4	0.009340	0.000588	0.000555
Buste	Ours	510K	19,200	14.9	0.003257	0.000237	0.000236
	QSlim			4,419	0.001365	0.000095	0.000094
	Grid			6	0.007113	0.000505	0.000490
Caesar	Ours	770K	18,500	18.6	0.013818	0.000220	0.000209
	QSlim			7,544	0.013894	0.000130	0.000124
	Grid			8	0.014448	0.000431	0.000413
Grog	Ours	1M	41,000	20.8	0.005253	0.000255	0.000249
	QSlim			9,101	0.003686	0.000128	0.000124
	Grid			9	0.007022	0.000531	0.000482
Gargoyle	Ours	1,7M	44,500	22.1	0.006374	0.000236	0.000237
	QSlim			15,668	0.004018	0.000120	0.000118
	Grid			13	0.006929	0.000496	0.000469
Raptor	Ours	2M	22,500	20.6	0.012457	0.000280	0.000277
	QSlim			19,057	0.011525	0.000143	0.000137
	Grid			15	0,012629	0.000423	0.000423
Neptune	Ours	4M	24,500	47.0	0.005375	0.000272	0.000282
	QSlim			43,941	0.001851	0.000089	0.000082
	Grid			27	0.008222	0.000487	0.000450
Crab	Ours	11M	64,200	106.5	0.005439	0.000233	0.000244
	QSlim			92,933	0.002617	0.000057	0.000055
	Grid			53	0.004677	0.000319	0.000315
Lucy	Ours	28M	116,500	222.7	0.008423	0.000185	0.000179
	QSlim			234,985	0.000894	0.000035	0.000033
	Grid			121	0.003576	0.000211	0.000203

Table 3: ***Quality and time comparison*** *with* ***H*** *the Hausdorff distance between the original model and the simplification,* ***M12*** *the mean distance from the original model to its simplification and* ***M21*** *the mean distance from the simplification to the original model. Timings are given in ms.*

In Table 2, we present the memory usage for the same set of models. The storage space needed on the GPU only depends on the number of vertices in the input geometry and on the desired Morton code precision. In the worst case scenario, if the Morton code is chosen precise enough to be different for every single input vertex (for example with the bunny model), there will be as many leaf nodes, and thus as many quadrics, average vertices , colors... However, as the size of the model increases, there will be more and more duplicated morton codes, and consequently not as many leaf nodes. For this reason, the GPU memory used by the algorithm does not increase as quickly as the number of triangles in the input model.

We compare our method with GPU grid clustering, which is very fast but not adaptive: for a comparable number of triangles, our method preserves visually important features that disappear with regular clustering while, although slower, keeping timings in a similar range. We also compare our results with QSlim, which favors quality over performances. We report timings and objective error measures in Table 3, with in particular *mean* and *Hausdorff* distances between the original and simplified meshes. Measures are performed using the Metro tool [Cignoni et al. 1998]. We plot visually the simplification error for the three approaches in Fig. 5. While for grid clustering, the error is concentrated around details, our method gives a globally lower error, with lower damages on features. This reflects in the error measures, with an improved Hausdorff distance and a significantly better mean distance to the original model. Of course, the quality of the approximation provided by our algorithm cannot compete with the QEM-based progressive reduction of QSlim. However, as illustrated in the Fig. 11, the visual quality remains overall good, for an execution time which is three orders of magnitude faster.

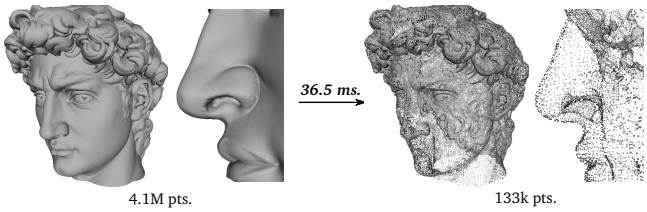

Figure 6: ***Point-based simplification*** *using our approach.*

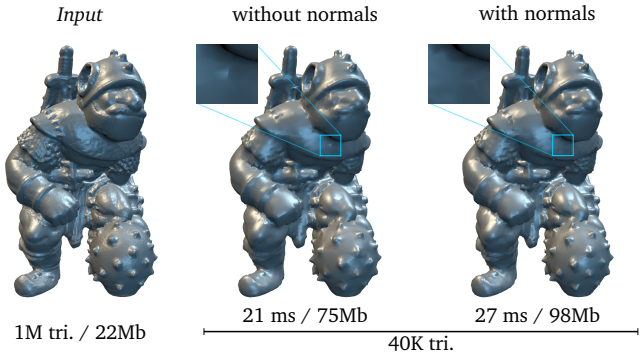

Figure 7: ***Influence of the normals*** *on the visual aspect, timings and GPU memory usage. On the Grog model, small clustering artifacts disappear when the normals are maintained, for example under the beard, on the leg or on the shoulder.*

In Fig. 6, we show an example of point cloud simplification using our approach on the Michaelangelo David's head model. Again, while the geometric resolution is drastically reduced, the important features, in particular the sharp edges, are captured in the simplified point sampling. The generation time is comparable to the mesh case, as the triangle reindexing step is not a bottleneck of our approach. In Fig. 7 and Fig. 8, we show examples of simplifications preserving the normal and color information. As we can observe on the Grog model, maintaining normals improves the visual aspect of the approximation by helping preserving details and visually reducing small clustering artefacts. We also show an example of simplification accounting for the color information provided on a per-vertex basis in the input. Visually important features that mostly exist through the color distribution are better preserved in this case. This is particularly useful for stereovision data, which often exhibits more features in the color than in the geometry. Since no significant change to the algorithm is needed to preserve an additional attribute (we just maintain it along with the quadrics, means, etc), the cost in time and GPU memory is relatively small.

In Fig. 9, we provide an example of simplification for a scene exhibiting a strongly varying vertex density. We can observe a proper behavior of our approach, with large polygons remaining intact with small ones being correctly simplified. Last, in Fig. 10, we show experiments performed on animated data. In particular, we focus on performance capture data [de Aguiar et al. ; Beeler et al. 2011], both for full body (medium resolution) and face (high resolution) models. We illustrate the range of possible applications for our method with on one side a rather simple body model (40k triangles) simplified by a factor of 5, and on the other side a high resolution face model (2.3M triangles) simplified by a factor 100. We show in this case the resulting mesh structure, obtained in real time as well.

Limitations Although adaptive, our approach falls in the *clustering* category, which induces at least two main limitations. First, such methods lack guarantees on the topology preservation At coarse scale, under extreme simplification rates, the input topology frequently changes, collapsing nearby layers. Although this is not always a weakness [Cohen-Steiner et al. 2004], more control on this behavior would be desirable. Second, we do not provide an absolute (polygon-wise) control over the output model size: although the user can set the desired level of simplification by ruling θ, guaranteeing an exact number of polygons is tedious. Our Morton integration is also bounded by the machine precision: when computing cumulative sums on very large arrays, imprecisions accumulate and can cause inaccurate quadric matrices, which can lead to instability in the output mesh. We reduce the impact of this stability problem by scaling the input mesh to the unit cube (scaling back the output), and by using double precision when computing cumulative sums. However, the error may still be significant when processing very large (giga polygons) models. Such data requires a different class of simplification algorithms (out-of-core/streaming), at least as a first pass. Once the model has reached a first appropriate (dense yet in-core) simplification, it is possible to chain one or several other in-core algorithms. Addressing these issues while preserving high performances is clearly one of the main direction for future work.

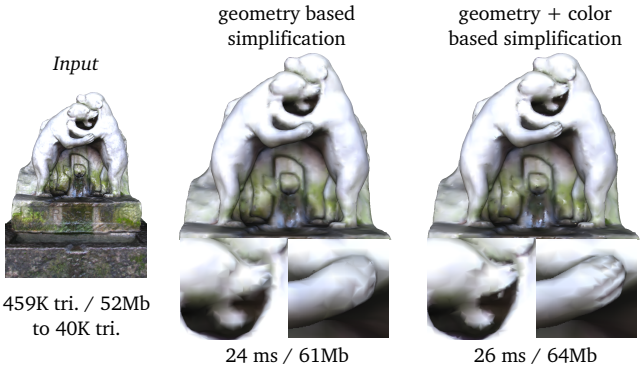

Figure 8: ***Influence of the color****, used to drive the simplification (on the right) or not (on the left), on the visual aspect, timings and amount of GPU memory used by the algorithm.*

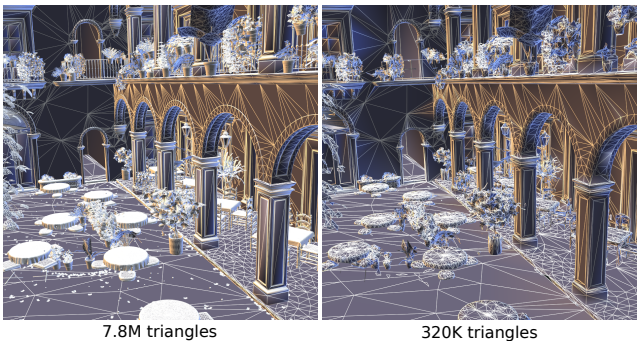

Figure 9: ***Simplification of a 7.8M triangles model with strongly varying vertex density*** *in 54ms using our approach.*

5 Conclusions

We have introduced a high performance adaptive geometry simplification algorithm which can process objects made of millions of polygons in real time and on commodity hardware. We achieve such a performance level by introducing *Morton integrals*, which are cumulative sums of samples attributes or error measures performed along their Morton enumeration. This intermediate object enables the parallel construction of a hierarchical approximation structure and its error-driven parallel traversal to extract a cut of nodes tailoring the simplified geometry. As a result, we obtain adaptive simplified meshes on-the-fly, with better quality than state-

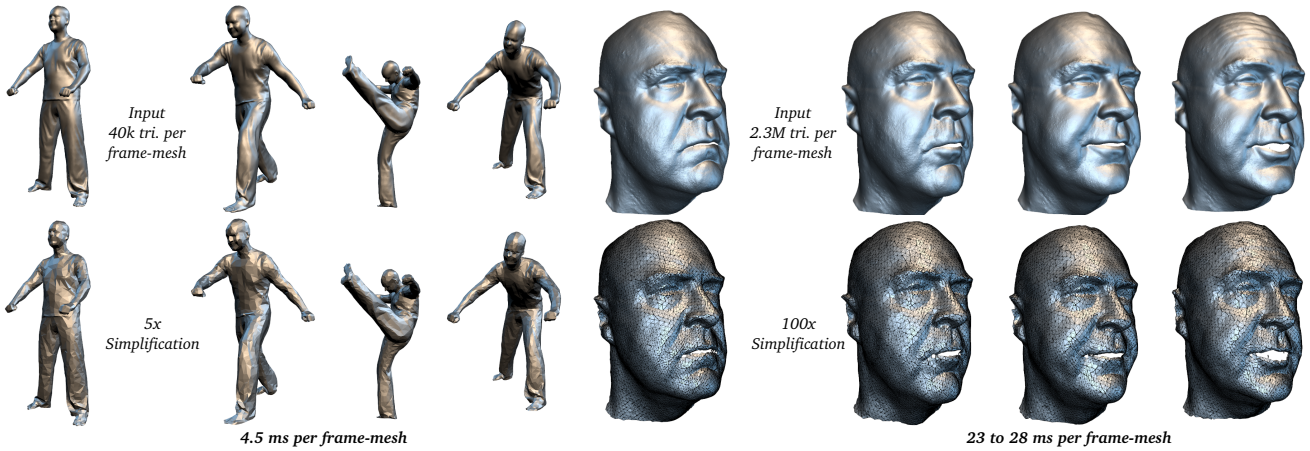

Figure 10: ***Adaptive simplification*** *of performance capture data.* ***Left:*** *500 frames are processed independently at about 200 Hz.* ***Right:*** *the three orders of magnitude downsampling is performed in real time on this dense face model.*

of-the-art high performance methods. We also showed that our method can be extended to account for surface attributes and meshless input. Our approach completes the high performance GPU geometry processing pipeline, which now features *live and adaptive* filtering, refinement and simplification. Beyond alternative metrics and additional surface attributes, we believe that the concept of Morton integration can be useful to other kinds of applications, providing a parallel scalable support for various flavors of multiresolution geometry processing and analysis methods.

Acknowledgments. Animated meshes are courtesy DRZ and MPI. This work has been partially supported by the European Commission under contracts FP7-323567 HARVEST4D and FP7-287723 REVERIE, and by the ANR iSpace&Time project.

References

ADAMS, A., GELFAND, N., DOLSON, J., AND LEVOY, M. 2009. Gaussian kd-trees for fast high-dimensional filtering. *Trans. Graph. 28*, 3, 21:1–21:12.

BEELER, T., HAHN, F., BRADLEY, D., BICKEL, B., BEARDSLEY, P., GOTSMAN, C., SUMNER, R. W., AND GROSS, M. 2011. High-quality passive facial performance capture using anchor frames. *Trans. Graph. 30*, 75:1–75:10.

BOUBEKEUR, T., AND ALEXA, M. 2009. Mesh simplification by stochastic sampling and topological clustering. *Computer & Graphics (Proc. Shape Modeling International) 33*, 3, 241–249.

CIGNONI, P., ROCCHINI, C., AND SCOPIGNO, R. 1998. Metro: measuring error on simplified surfaces. *Computer Graphics Forum 17*, 2, 167–174.

COHEN-STEINER, D., ALLIEZ, P., AND DESBRUN, M. 2004. Variational shape approximation. *Trans. Graph. 23*, 3, 905–914.

CROW, F. C. 1984. Summed-area tables for texture mapping. In *SIGGRAPH*, 207–212.

DE AGUIAR, E., STOLL, C., THEOBALT, C., AHMED, N., AND SEIDEL, H. Performance capture from sparse multi-view video. *Trans. Graph. 27*, 3, Art. 98.

DECORO, C., AND TATARCHUK, N. 2007. Real-time mesh simplification using the GPU. In *ACM I3D*, 161–166.

GARANZHA, K., PANTALEONI, J., AND MCALLISTER, D. 2011. Simpler and faster HLBVH with work queues. In *HPG*, 59–64.

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

GRUND, N., DERZAPF, E., AND GUTHE, M. 2011. Instant level-of-detail. In *Vision, Modeling and Visualization*, 293–299.

HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., AND STUETZLE, W. 1993. Mesh optimization. In *SIGGRAPH*, 19–26.

KARRAS, T. 2012. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *High Performance Graphics*, 33–37.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum 28*, 2 (Apr.), 375–384.

LINDSTROM, P. 2000. Out-of-core simplification of large polygonal models. *SIGGRAPH*, 259–262.

LINDSTROM, P. 2003. Out-of-core construction and visualization of multiresolution surfaces. In *I3D*, 93–102.

NVIDIA, 2011. Thrust. https://developer.nvidia.com/Thrust.

PANTALEONI, J., AND LUEBKE, D. 2010. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *High Performance Graphics*, 87–95.

PIXAR, 2013. Opensubdiv.

ROSSIGNAC, J., AND BORREL, P. 1993. Multi-resolution 3d approximations for rendering complex scenes. In *Modeling in Computer Graphics*, 455–465.

SCHAEFER, S., AND WARREN, J. 2003. Adaptive vertex clustering using octrees. In *Geom. Design & Computing*, 491–500.

SHAFFER, E., AND GARLAND, M. 2001. Efficient adaptive simplification of massive meshes. In *Visualization*, 127–551.

SHAFFER, E., AND GARLAND, M. 2005. A multiresolution representation for massive meshes. *TVCG 11*, 2, 139–148.

VIOLA, P. A., AND JONES, M. J. 2001. Robust real-time face detection. In *ICCV*, 747.

ZHOU, K., GONG, M., HUANG, X., AND GUO, B. 2010. Data-Parallel Octrees for Surface Reconstruction. *TVCG 17*, 5, 669–681.

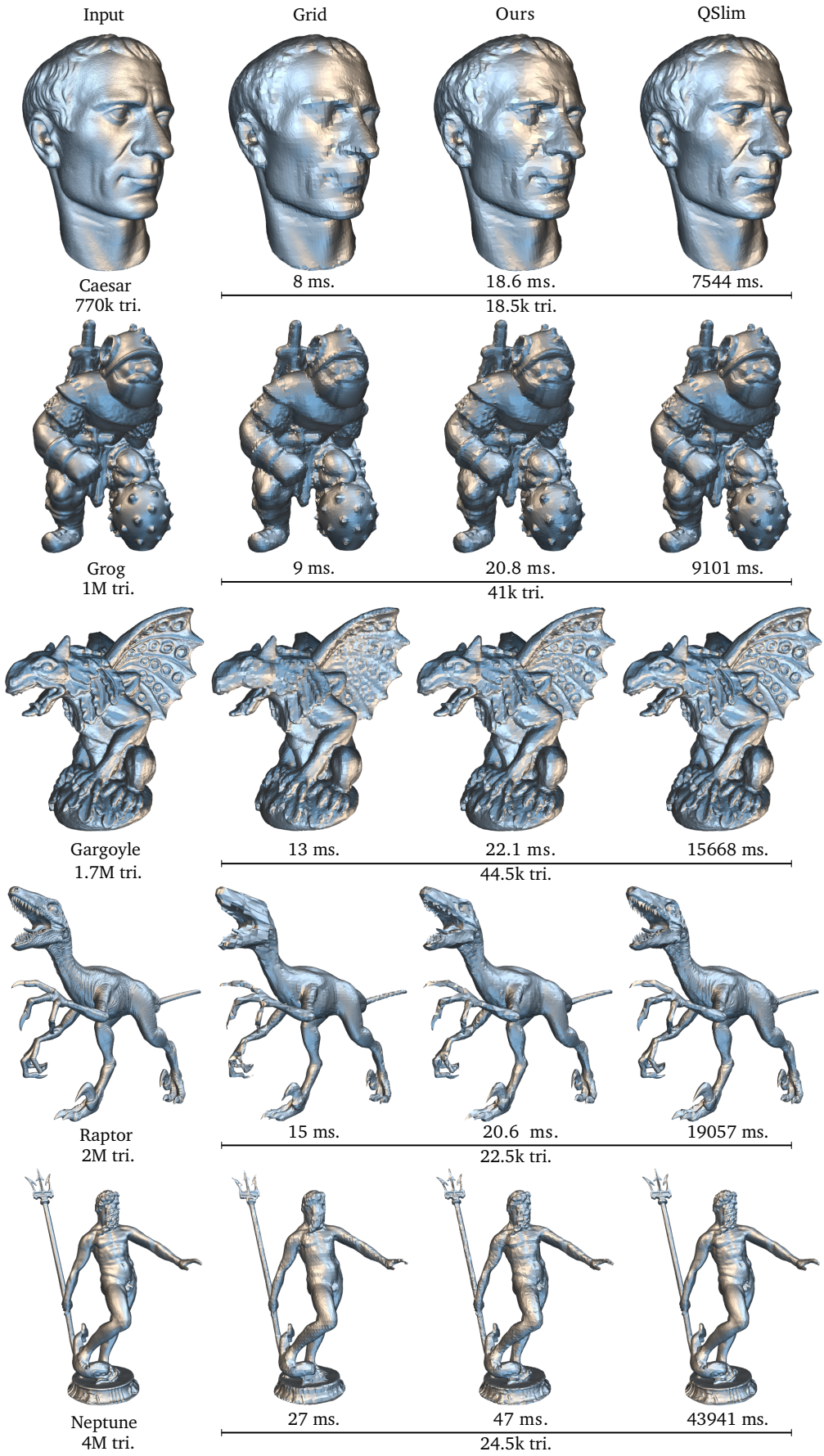

Figure 11: *Examples.*