# Rapid Simplification of Multi-Attribute Meshes

Andrew Willmott[*]
Maxis/Electronic Arts

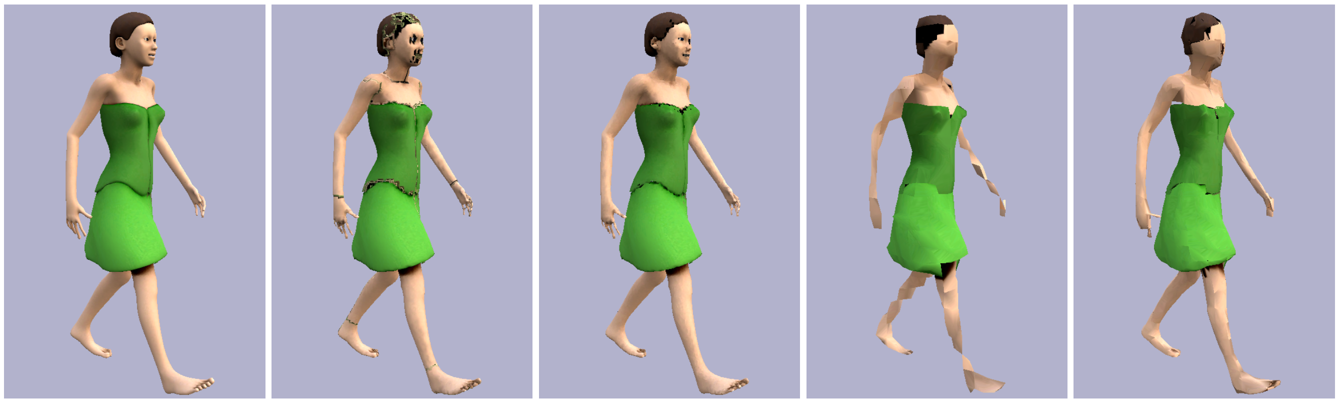

**Figure 1:** *From left: 160,000-triangle textured model; simplification by vertex clustering to 30,000 triangles in 40ms, causing artefacts at UV seams; artefacts eliminated by our algorithm in an additional 7ms; heavy simplification causing blockiness and dropouts in 13ms; improved by our extended algorithm in 3ms.*

## Abstract

We present a rapid simplification algorithm for meshes with multiple vertex attributes, targeted at rendering acceleration for real-time applications. Such meshes potentially feature normals, tangents, one or more texture coordinate sets, and animation information, such as blend weights and indices. Simplification algorithms in the literature typically focus on position-based meshes only, though extensions to handle surface attributes have been explored for those techniques based on iterative edge contraction. We show how to achieve the same goal for the faster class of algorithms based on vertex clustering, despite the comparative lack of connectivity information available. In particular, we show how to handle attribute discontinuities, preserve thin features, and avoid animation-unfriendly contractions, all issues which prevent the base algorithm from being used in a production situation.

Our application area is the generation of multiple levels of detail for player-created meshes at runtime, while the main game process continues to run. As such the robustness of the simplification algorithm employed is key; ours has been run successfully on many millions of such models, with no preprocessing required. The algorithm is of application anywhere rapid mesh simplification of standard textured and animated models is desired.

## 1 Introduction

Mesh simplification is generally seen as an offline process. Performance is important, particularly for large meshes, but it is of secondary concern to the quality of the results. However, for applications where meshes are generated by some user-controlled process, or procedurally, it is highly desirable to perform simplification at runtime, in such a manner that the application remains interactive. If we wish to use such meshes in the place of traditional artist-authored content in an interactive context, with the same performance guarantees, we must generate a number of levels of detail (LODs) for each source mesh, in a reasonable timeframe. This is the key problem this paper tries to address.

In such a situation we can generally only dedicate a small fraction of the runtime to mesh processing. For example, if we have a situation where we must prepare twenty meshes for display, each requiring three LODs, and we reserve as much as 10% of the CPU for this purpose, an algorithm that takes ten seconds to simplify a single mesh will take over an hour and a half before all the meshes are ready for display. This is clearly unacceptable, but unfortunately ten seconds is optimistic for most existing simplification algorithms operating on meshes of 100,000 triangles or more, and those that can do substantially better are unable to deal with the more general classes of mesh we must support, namely textured, animated, and with potentially other per-vertex attributes that control appearance, such as lighting coefficients.

In addition to the question of throughput, simplification for rendering performance on modern GPUs is a subtly different problem from optimising appearance for a given triangle count. Rendering speed is dependent on a number of factors, including vertex and fragment throughput, and triangle setup cost. While vertex throughput is sensitive to mesh size, increasingly render costs are much more dominated by fragment shading, which has to be addressed by the use of shader LODs, rather than mesh LODs. Triangle setup costs, on the other hand, have become more of an issue; once triangles cover less than ten to twenty pixels, rasteriser efficiency drops rapidly, as the fixed setup cost is no longer hidden by fragment processing. Thus, ensuring that the screen space triangle density of a mesh remains above that threshold is also of importance. A secondary reason to consider triangle density is that aliasing becomes

[*]e-mail: awillmott@maxis.com

an issue in this situation, because texture filtering is no longer effective.

A final consideration is input robustness. Even artist-authored meshes typically have issues with non-manifold edges, and often require manual cleaning before simplification tools provided by content creation packages can process them. When meshes are instead created as the result of a user process, guaranteeing the mesh is well-formed is an even harder problem. Thus we desire that our algorithm can handle any mesh input.

## 2 Related Work and Discussion

**Algorithm Classes**. When choosing a suitable approach for real-time simplification, there are three major classes of algorithms to consider:

*Edge Contraction.* This has been the most popular class of simplification algorithms in recent history, owing to the way it casts simplification as a sequence of simple invertible "edge collapse" operations that preserve mesh structure [Hoppe 1996]. This both reduces implementation complexity (the half-edge data structure suffices [Muller and Preparata 1978]) and ensures any one operation has a strictly local effect, meaning successive operations compose well. Examples include Hoppe [1997], Ronfard and Rossignac [1996], Gueziec [1995], and Garland and Heckbert [1997]. The algorithms mostly vary in the metric employed to choose the next edge to contract. To do so they keep some form of history about the progressively simplified surface to evaluate current error. Using cumulative quadric error matrices is a particularly efficient and hence popular way of doing so, although it has been shown that re-evaluating the metric on the current version of the mesh can lead to better quality results, at the cost of performance [Lindstrom and

边折叠缺点：需要输入网格流形以保证面边连接关系用于边坍塌。需要预处理网格清理，无法并行处理。

One drawback to these algorithms is that they require a manifold input mesh, in order to efficiently represent the face/edge adjacency information needed to perform edge collapses. This can require running a mesh cleaning algorithm as a pre-processing step. Aggregates of manifold meshes can be handled more easily, by adding contraction edges between nearby vertices [Garland and Heckbert 1997]. Another issue is that they can be hard to parallelise, given the inherently serial nature of the iterative process. However the advantage of this process is that a single simplification run can produce multiple static LODs, or, by logging collapses in order, a data structure capable of dynamic LOD generation.

*Vertex Clustering.* Rossignac and Borrel [1993] cluster vertices in the input mesh according to a surrounding grid, and then discard all degenerate triangles that result. The representative replacement vertex for those vertices within a cluster may be chosen by selecting the highest weight vertex according to some importance metric, by averaging, or by some higher-order approximation, such as error quadrics. Low and Tan [1997] improve the basic method by using *floating cells* rather than a fixed grid to define clusters. These are constructed by aggregating vertices in order of their importance, which leads to a more adaptive clustering, at the expense of some performance. Luebke and Erikson [1997] alternatively form a hierarchy of clusters. This both improves simplification quality, and can be used to perform view-dependent simplification, by tracking an active cut through the cluster tree.

Apart from its implementation simplicity, clustering is appealingly robust; there are no constraints on the input mesh in terms of being a manifold mesh. However, it is difficult to control the exact number of triangles in the output mesh, and the algorithm does a poor job of preserving detail in the original mesh; it cannot trade off simplification in flat areas for detail in curved areas. There is only one

顶点聚类优点：实现简单，聚类鲁棒，无需输入网格流形。具有相对一致的存储器访问模式，对于大型网格可执行out-of-core简化。缺点：难以控制输出网格三角形的精确数量，难以保细节、保特征等。

由于算法线性经过源顶点、源三角形，具有相对一致的存储器访问模式以及对应的内存层次。对于大型网格可采用out-of-core简化方式，当网格简化到适应in-core，再采用保细节的简化算法作为后处理。

global simplification control (the grid resolution), and its insensitivity to surface connectivity means it can't maintain topology if that is desirable.

Because the algorithm is based on a linear pass through the source vertices, and then a linear pass through the source triangles, it has a relatively coherent memory access pattern, and interacts well with memory hierarchies. Lindstrom [2000] takes advantage of this to efficiently perform out-of-core simplification of extremely large meshes. (Other simplification algorithms quickly deteriorate due to thrashing once core memory is exceeded.) Optionally once the mesh has been reduced to a size that fits in-core, a more strongly detail-preserving simplification algorithm can be applied as a post-process.

*Resynthesis.* These algorithms retriangulate part of or even all of the mesh, potentially substituting new vertices. Schroeder et al. [1992] iteratively select unimportant vertices for removal, and then retriangulate the resulting hole, in an approach termed *vertex decimation*. Kalvin and Taylor [1996] instead iteratively cluster faces in the mesh that can be well approximated by a plane. These these clusters are then replaced with the corresponding "superface", a triangulation of the edges formed with other clusters. Hoppe et al. [1993] cast simplification as a general optimisation problem, generating new vertices to fit the original mesh, and triangulating appropriately. Because these algorithms typically rely on somewhat involved manifold mesh operations, they can be complicated to implement and slow to execute.

重组。重新三角化部分或所有网格，替换新节点。

**Mesh Attributes**. There has been comparatively less work on the simplification of multi-attribute meshes. The edge collapse operation supports meshes with multiple potentially discontinuous attributes natively if the attributes are stored at face corners [Hoppe 1996]. Further research has thus concentrated on finding a suitable extended cost metric that takes account of these attributes [Garland and Heckbert 1998] [Hoppe 1999]. With some care, the vertex decimation algorithm can also be adapted to handle additional attributes [Bajaj and Schikore 1996]. However, algorithms that are more aggressive and remove many faces and vertices in a single step, such as vertex clustering or many of the more involved resynthesis methods, have difficulties preserving vertex attribute values in the output mesh.

In an alternative approach, Cohen et al. [1998] resample attributes into texture maps, constrain edge collapses so texture chart boundaries aren't crossed, and weight the cost metric to avoid texture distortion. Sander et al. [2001] take this a step further and automatically generate texture charts suitable across all simplification levels of the progressive mesh model. We don't have this option because our charts are pre-generated, and re-sampling maps would add too much overhead.

**Real-time Simplification** DeCoro and Tatarchuk [2007] propose a real-time simplification algorithm based on vertex clustering. By using a direct grid, the GPU can take responsibility for several parts of the algorithm, including quadric generation, though the CPU cost still dominates. They achieve speeds of 200ms for a 70,000 triangle model. The algorithm only handles position-based meshes however, making it unusable for our purposes.

That aside, vertex clustering seems promising as a starting point. The same coherent memory access patterns that make clustering suitable for out-of-core simplification also apply when we move up in the memory hierarchy and consider multi-level data caches or core-local memory. Moreover, the overhead of the more complicated data structures required by edge contraction or resynthesis algorithms is avoided, which in turn reduces memory pressure. We observe that one of the biggest drawbacks of the algorithm, inability to preserve fine detail, is not such an issue when simplifying for

顶点聚类方法的优点

该方法不能保证细节的缺点，并不是用于渲染问题的缺点。

需要解决的问题是如何避免局部三角形密度过高，输出比较均匀的网格

rendering purposes on modern hardware. As discussed previously, it is desirable to avoid high local screen-space triangle densities, and regulating output vertex spacing by means of a uniform grid works well for this. Finally, the algorithm is extremely robust to input. In the next section, we address the issue of extending vertex clustering to properly handle vertex attributes.

## 3 Simplification Algorithm

### 3.1 Vertex Clustering

We consider a multi-attribute mesh as defined in Table 1. It contains at least an array of unique positions, and optionally additional unique attribute arrays such as normals or texture coordinates. These arrays are supplemented with corresponding arrays of per-vertex array indices, which can be used to indicate element sharing between vertices. For instance if vertices $i$ and $j$ have matching $e_p[i]$ and $e_p[j]$, they are considered to have the same position. They may have differing $e_{a_n}[i]$ and $e_{a_n}[j]$ however, indicating the presence of an attribute discontinuity at the vertex. Finally there is a set of primitive vertex indices that indicate inter-vertex connectivity, and by the same token, vertex sharing. For a triangle list with $N_f$ triangles, this is an array of $N_f$ index triples.

顶点处属性不连续

| | |
|---|---|
| $p[N_p]$ | Positions array |
| $a_n[N_{a_n}]$ | $n$th attribute array |
| $e_p[N_v]$ | Per-vertex position indices array |
| $e_{a_n}[N_v]$ | $n$th per-vertex attribute indices array |
| $e_v[3N_f]$ | Per-triangle vertex indices array |

**Table 1:** *Basic mesh definition*

The base vertex clustering algorithm works by clustering nearby vertices according to some metric, and then discarding any triangles that are now degenerate. An outline of the algorithm is:

```
QuantiseVertices:
    foreach (i in Nv)
        Generate cell label
        Record replacement index ep[i]
        Accumulate p into representative point p_label

RemoveDegenerateTriangles:
    foreach (i in Nf)
        if (p[ep[ev[3i]]] = p[ep[ev[3i + 1]]] = p[ep[ev[3i + 2]]])
            Discard triangle

Compact:
    Share all vertices with identical element references
    Remove all unindexed data
```

A detail is that the label lookup required for the index of the replacement point can be handled either by an explicit per-grid-cell lookup table, or by using a hash map from label to index. The latter has the advantage that storage is of the order of the number of output points rather than $O(g^3)$, where $g$ is the grid resolution, and that explicit grid bounds do not need to be calculated. This leads to it often being termed the *virtual grid* approach. It also lends itself to the extended label use we describe in Section 4.

Finally, the `GenerateNewVertices` routine may be run in parallel to the remainder of the algorithm, and RemoveDegenerateTriangles is easily divided into multiple work units with a single post-merge step.

### 3.2 Vertex Attributes

The most obvious way to extend this algorithm to support multiple attributes is for `GenerateNewVertices` to also generate representative attributes for each output vertex, either by synthesis or by selecting a particular input vertex to be the representative. The drawback to this approach is that any vertex attribute discontinuities in the input mesh will be removed, as all output vertices must have a single associated set of attributes. For vertex normal attributes, the result is that any sharp edges are removed, as can be seen in Figure 2. Such edges are either smoothed, if normals are properly averaged, or result in unrepresentative lighting if a particular input vertex is used, as shown. Even if the result is acceptable at a distance, there is obvious visual popping when LODs change, as the eye is quite sensitive to such lighting discontinuities.

A more subtle issue arises with texture coordinates, particularly when used with the standard approach whereby multiple texture charts are combined into a single page. Two vertices from different charts may be collapsed into one vertex, leading to triangles in the output mesh that span more than one chart. For these triangles the uv mapping is ill-defined, and disturbing visual artefacts appear close to chart boundaries, as in Figure 3. Similar issues occur for non-convex charts.

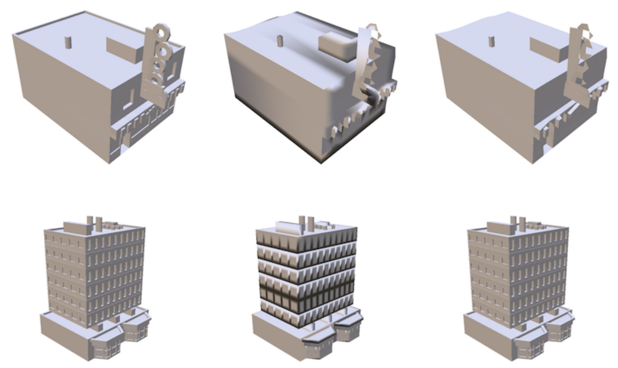

**Figure 2:** *Left: original model (normals only). Middle: simple vertex clustering removes all normal discontinuities, leading to edge smoothing and normals inconsistent with the local surface. Right: our algorithm preserves these discontinuities.*

One can also take the same approach as the progressive mesh algorithm, and associate attributes with face corners. However, in the case of vertex clustering, although this successfully retains discontinuities, it also leads to all output corners at a vertex having different attribute indices, which effectively forces all vertices in the output mesh to be unshared, an untenable result.

### 3.3 Cell Boundary Edges

The difficulty with attribute preservation under the vertex clustering model is that we operate per-vertex during the collapse step, so we do not have local information about the discontinuities we may want to preserve. Fixing up the attributes as a post process also seems difficult, because at first glance this requires knowledge of the set of triangles collapsed into a given representative point, as a discontinuity may potentially pass through any of the edges in this set. Tracking this set and searching it for appropriate discontinuity edges would result in a dependency on input mesh size, and lose the algorithm's main speed advantage, namely the ability to discard large numbers of triangles via linear passes through first the vertices and then triangles.

We can simplify matters by only considering discontinuities that pass completely through a cluster cell, as in Figure 4a. This is rea-

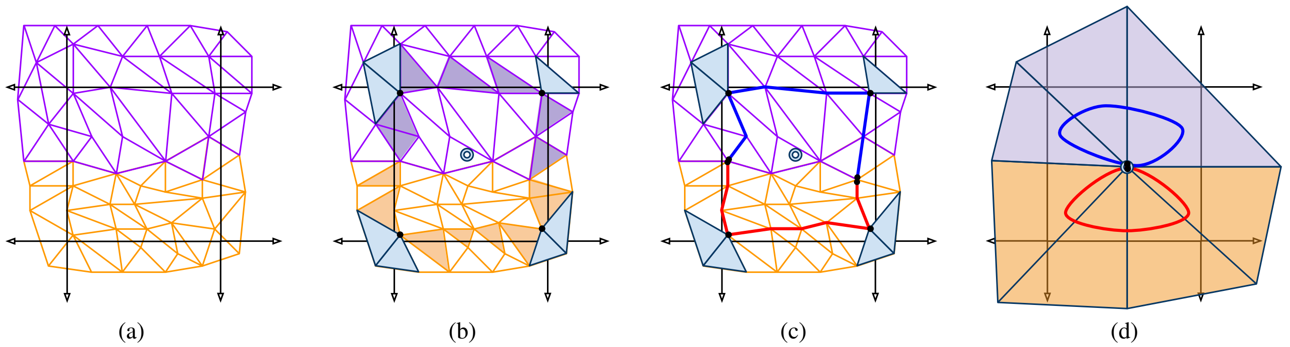

**Figure 4:** *Original mesh with a colour attribute discontinuity (a), output and singly-degenerate triangles, and cluster position (b), boundary edges identified by these triangles used to link output triangle corners (c), the final mesh with the attribute discontinuity preserved (d).*

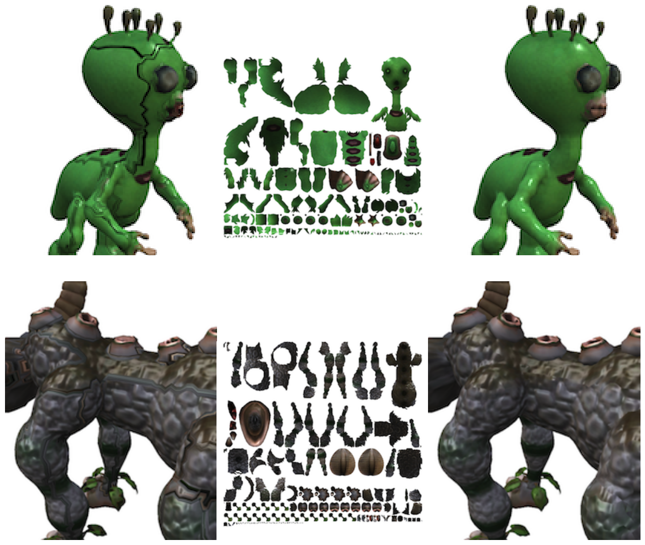

**Figure 3:** *Left: vanilla vertex clustering leads to visible seams at chart boundaries. Middle: the underlying texture chart. Right: our algorithm.*

sonable as any discontinuity contained completely within the cell, or only partially entering the cell, can be regarded as small enough to be elided. If we consider the triangles that survive in the output mesh, as in 4b, the problem reduces to detecting any continuous set of discontinuity edges that pass between their pre-collapse vertices within the cell under consideration, as these vertices correspond to corners surrounding the new vertex. In this particular example we wish to detect that colour attributes are discontinuous between the two top and two bottom vertices, and share attributes correspondingly.

An equivalent way of stating this problem is that for any two such vertices, they can be regarded as sharing an attribute if, for a given set of pre-collapse edges connecting them, there is no internal discontinuity. Re-examining 4b, we can see that one potential set of such edges corresponds to the *cell boundary edges*, namely those edges that make up the external boundary of the triangles that will be removed after vertex clustering. Any discontinuity that passes completely through the cell must cross these boundary edges at some point. Furthermore, we see that such edges can be identified from singly degenerate triangles in the mesh, which are partially shaded in the diagram. These are the set of triangles that are degenerate in one edge only; this edge corresponds to a boundary edge in the pre-collapse mesh. Once these edges have been identified, as in 4c, the triangle vertices can be classified accordingly, and used to determine which corners should share a single attribute, as in 4d.

There are two advantages to this approach. The first is that the set of boundary edges in a cell is considerably smaller than the set of collapsed triangles, and gets smaller as the cell size increases relative to the mesh. (For an evenly tessellated flat mesh passing completely through the cell, the boundary edges increase as $O(w)$, and the triangles as $O(w^2)$.) The second is that the singly degenerate triangles can be trivially identified as part of the existing `RemoveDegenerateTriangles` routine. It remains to find an efficient way to use those boundary edges to classify the surviving triangle vertices, given that they are identified in no particular order.

### 3.4 Sharing Attributes

We wish to assign the same attribute label to those surviving triangle vertices that are connected by cell boundary edges. One way to do so is to build a list of these edges as they are discovered, aggregating those that share end vertices into chains, and then, in a second pass, iterate over each of these chains in turn, and assign all referenced vertices the same label. However, we can do better. We use the Union-Find algorithm [Galler and Fisher 1964] (see Manber [1989] for discussion) to successively classify boundary vertices as belonging to the same set, by treating each successive boundary edge as a union operation between the two vertices it joins. Then, in a second pass, for each surviving triangle vertex, we look up the corresponding set, and use it as the attribute label. As the Union-Find algorithm has amortised cost of effectively $O(1)$ for both operations, this is substantially more efficient, particularly as in the second pass we iterate over output triangle vertices, rather than the larger number of boundary edges.

The standard Union-Find algorithm requires both *union by rank* and *path compression* to gain its performance guarantees. In practice, our algorithm is heavily data dependent, our *find*s are heavily outnumbered by *union*s, and the size of the sets is limited by the maximum boundary edge count in a cell. As a result we see a performance improvement if we omit path compression, which reduces the number of memory writes, and a slight further improvement if we perform the step of compressing only the input vertices for both operations. Secondly, we graft the shorter subtree onto the highest seen node in the larger subtree as soon as we find its root, rather than continuing to the root of the larger subtree, which simplifies the code and saves memory accesses.

```
rv0 = dv0 = ea_n[ev[e0]]
rv1 = dv1 = ea_n[ev[e1]]
level = 0

while (setLinks[rv0]) >= 0)
    rv0 = setLinks[rv0]
    level++

while (setLinks[rv1] >= 0)
    rv1 = setLinks[rv1]
    level--

if (rv0 != rv1)
    if (level < 0)
        setLinks[rv0] = rv1
        setLinks[dv0] = rv1
    else
        setLinks[rv1] = rv0
        setLinks[dv1] = rv0
```

```
foreach (iv in 3 Nf)
    i  = ev[iv]
    dv = ea_n[i]
    rv = dv

    while (setLinks[rv] >= 0)
        rv = setLinks[rv]

    if (setLinks[rv] == -1)
        setLinks[rv] = -2 - dv;

    if (dv != rv)
        ea_n[i] = -2 - next
        setLinks[dv] = next
```

The resulting algorithm is shown above. It requires the allocation of a single additional scratch array, `setLinks`, of size $N_{a_n}$, with all values initialised to $-1$.

The union operation shown on the left is performed for any singly degenerate triangle detected during the main triangle removal loop, and the replacement attribute index lookup on the right is run immediately after triangle removal has finished, on the remaining triangles in the mesh. As shown, the replacement algorithm uses a simple encoding trick to choose the attribute index of the first vertex visited in each set as the representative attribute, but this can be replaced with code to generate new attribute indices, if it is desired to generate more accurate representative attributes later.

## 4 Quality Improvements

Although preserving attribute discontinuities goes a substantial way towards allowing vertex clustering to be used in a production situation, there are several other areas in which we found it necessary to extend the algorithm to address quality issues.

### 4.1 Shape Preservation

An issue with vertex clustering in particular is that any feature smaller than the cell size in at least one dimension will be removed. This can lead to thin but elongated features, such as limbs or branches, being removed well before we would wish. To avoid this, we can extend the cell label generated for each vertex prior to clustering according to the corresponding vertex normal. We classify this normal according to its sign along each major axis, leading to eight possible directional tags, which we append to the cell label before index lookup. (Other normal clustering schemes are possible, but this is the fastest to calculate, and gives good results.) By doing so we prevent surface regions that are in opposing directions from being collapsed together, at the expense of some additional faces being generated. This has the effect of preserving curved surfaces within a cell, as seen on the right side of Figure 5. The impact to label generation in `QuantiseVertices` is not major: the vertex normal must be read in addition to vertex position, but the sign bits are trivially extracted. Also, although more faces are generated for heavily curved surfaces, a flat surface passing through a cell is relatively unaffected, as its vertex normals will mostly fall into the same quadrant.

### 4.2 Bone Preservation

An issue with applying any simplification technique to an animated mesh is inappropriate simplification of features that are nearby in the base pose, but are animated independently, and may be far apart in other poses. Commonly this leads to webbing between adjacent features that are collapsed together, as seen in Figure 6.

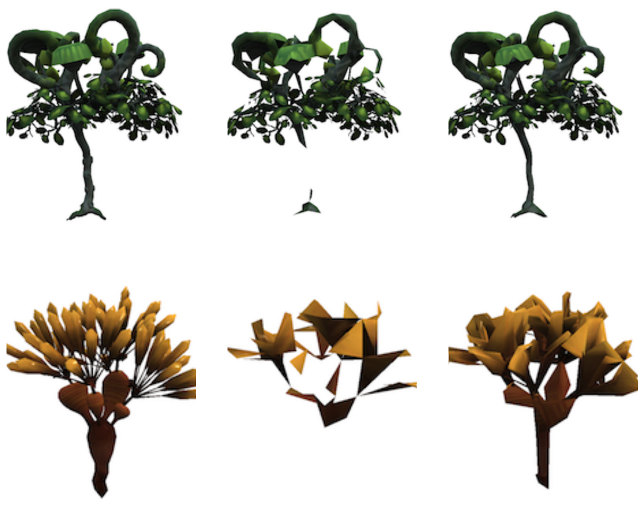
**Figure 5:** *From left to right: the original model; long, thin features disappear under simplification; after applying shape preservation.*

We can avoid this issue by finding the bones[1] influencing each vertex, and appending their indices to the extended cell label in turn. This has the effect of preventing any triangle that spans two different bones from being removed, and thus avoids collapsing together any two parts of the mesh that move independently. We find it suffices to only use the bone with greatest influence, which is easily found in a standard animated mesh set up. Typically a set of at most four bone indices and three bone weights are stored per vertex[2], in sorted order, to make it easy to process a subset of weights for shader LOD purposes. Hence the major bone index for a vertex is simply $a_{weights}[i][0]$.

Although we do wind up generating additional triangles over the base algorithm, not all of which are necessary to avoid webbing, because the bones represent functional parts of the mesh, they can still help with general visual and animation detail preservation. Also, the approach composes well with skeleton simplification approaches. If we have available a reduced skeleton for more distant LODs, in order to reduce the work the animation system is doing, the vertex collapse constraints can be similarly relaxed due to the reduced set of bone indices.

### 4.3 Simplification Control

The previously described extensions trade face count for quality. Thus we find it useful to be able to selectively control which are enabled in certain scenarios. For instance at the farthest LOD, animation may be disabled altogether, removing the need for bone preservation. Shape preservation may not be needed in all scenarios either, and it would also be desirable to have more fine-grained control over the simplification amount over different areas of the mesh.

We can achieve these goals by modifying the cell size used to determine the base cell label, and which extensions to the base cell label are applied, according to *bone groups*. We classify bones according to functional group (e.g., body, limbs, detail, head), and use

[1]In this context a bone is an animated transform matrix from the underlying mesh skeleton.

[2]The last weight is implicit as all weights sum to one.

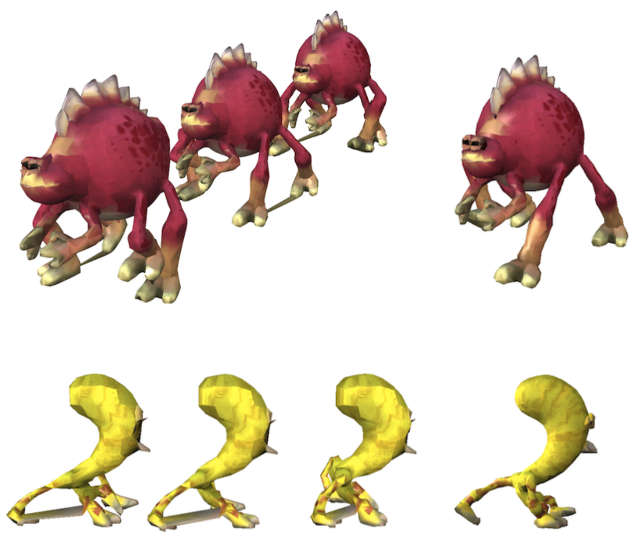

**Figure 6:** *Left: undesirable vertex collapse leads to webbing between limbs under animation. Right: after bone preservation is applied.*

each vertex's major bone index to quickly look up the corresponding group. This can be used to further look up the corresponding control factors from a small table, and thus generate the appropriate vertex label. (Having multiple cell sizes in the same virtual grid is not problematic as long as vertices with different sizes are never collapsed together, which can be achieved by further extending the label with its functional group.) As an example of how this might be used, we could set vertices in detail regions to be much more aggressively simplified than others via a larger cell size, and apply bone and shape preservation only to limbs. Although this approach does prevent vertices in different functional groups from ever being merged, it is usually worth it for the additional control.

## 5 Results

We compare our algorithm to the implementation of the progressive mesh algorithm present in the DirectX graphics API [Hoppe 1997] [Gray 2003]. This is an industrial-strength implementation capable of handling generic meshes, and intended for use in interactive applications, and as such represents the best comparison we could find. For our `GenerateNewVertices()` implementation we use position averaging for each cluster, as this gives us better results than selecting the 'best' input position, and is considerably faster than using error quadrics, as we avoid having to handle degenerate quadrics or best-fit points that lie outside the original cluster cell. For attributes we semi-randomly pick a representative from the surviving corners in the cluster, which provides acceptable results for no additional overhead.

The time taken to simplify our 160,000 triangle "green lady" model is shown in Figure 7, against both face count and vertex count of the resulting mesh. The progressive mesh implementation is weighted by a factor of 100 to bring it into range with the other results. An obvious feature of the graphs is the semi-linear performance dependence of the vertex clustering variants the output mesh size; the smaller the output mesh, the shorter the time taken. This is a desirable trait for our LOD generation purposes, as for slower platforms, we can generate more aggressively simplified meshes in a shorter amount of time. In contrast, the progressive mesh implementation must do progressively more work to produce smaller LODs.

Comparing our algorithms to basic vertex clustering reveals more subtle results. Per-triangle the latter is consistently faster as expected, but adding shape and bone preservation appear to increase the speed of our attribute-handling algorithm. Looking at the per-vertex results, and observing that data points shown use the same simplification factor (i.e., grid cell size) for each algorithm, clarifies the reason. For a given factor, as we preserve more features in the output mesh, the output vertex and triangle counts are affected. Preserving attribute discontinuities results in more output vertices, while the face counts remain the same. It adds additional cost to `RemoveDegenerateTriangles`, but other routines are largely unaffected, leading to the effective vertex throughput appearing to increase for smaller output meshes. For shape and bone preservation, face count also increases for a given simplification factor, and the only additional cost is fixed for a given input mesh. This increases their apparent throughput in both graphs.

Table 2 shows timing details for our algorithms for two specific simplification factors. Notable is that the overhead of attribute discontinuity preservation goes up with smaller cell sizes, as the ratio of boundary edges to input edges drops[3], and that shape and bone preservation primarily affect the running time of `QuantiseVertices`, though the larger number of output triangles also impacts `RemoveDegenerateTriangles`. Figure 8 shows a visual comparison of our results to the progressive mesh algorithm.

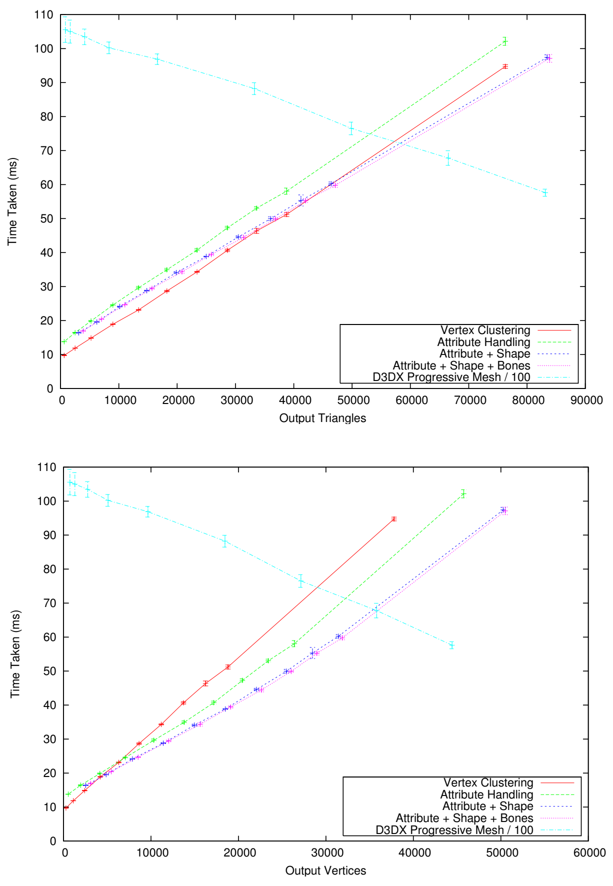

**Figure 7:** *Performance results for the "Green Lady" model.*

[3]The time taken in QV is also slightly reduced, as generation of replacement attribute indices can be skipped.

| Algorithm | QV | RDT | Total | QV | RDT | Total |
|---|---|---|---|---|---|---|
| | $w = 1/10$ | | | $w = 1/50$ | | |
| VC | 5.18 | 1.81 | 9.7 | 17.5 | 2.28 | 23.1 |
| Attributes | 5.04 | 2.46 | 13.8 | 17.3 | 4.83 | 29.6 |
| Attr/Shape | 6.23 | 3.61 | 16.4 | 20.8 | 5.79 | 34.1 |
| Attr/Shape/Bone | 6.79 | 3.74 | 16.9 | 21.4 | 5.69 | 34.3 |

**Table 2:** *Performance impact for two different cell sizes w on the QuantiseVertices (QV) and RemoveDegenerateTriangles (RDT) routines. All times in milliseconds.*

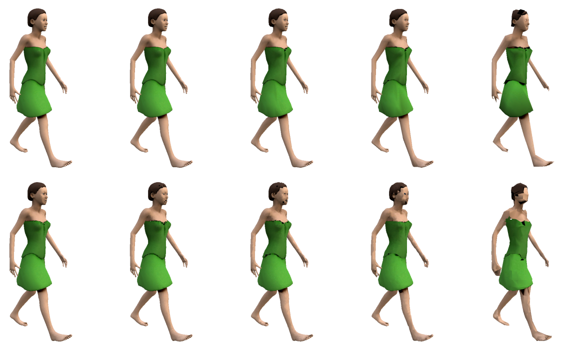

**Figure 8:** *Top: D3DX progressive mesh algorithm results, from 70% to 97.5% triangle reduction. Bottom: our vertex-clustering-based algorithm at similar face counts, running from 100 (left) to 660 times faster (right).*

## 6 Conclusions and Future Work

We have presented a simplification algorithm based on vertex clustering that retains its speed and suitability for modern CPU and memory architectures, but fixes the drawbacks that prevent it from being used in a production situation, by handling attribute discontinuities, addressing feature dropout, avoiding animation artefacts, and allowing finer-grain simplification control. The algorithm doesn't match the ability of edge-contraction approaches to preserve fine features, but is well suited to generating LODs to accelerate rendering performance on current GPUs, where this can be if anything detrimental. For our particular application, where we are dealing with input meshes of 50,000 to 100,000 triangles, we see simplification times of 10ms to 40ms. For the example situation presented in Section 1, this allows us to complete LOD generation in around fifteen seconds.

The algorithm parallelises well in a multiple LOD situation with many meshes to process; in our implementation we distribute work across as many cores as are available. Parts of the algorithm are particularly well suited to a GPU compute architecture [Nickolls et al. 2008], namely representative position and attribute calculation, and face and vertex compaction, which primarily rely on scan, sort, and stream compaction operations. In future we would like to find a way to additionally formulate the attribute set generation in a data parallel way, so the entire process can be run in a GPU-based environment.

## Acknowledgements

Thanks to Ocean Quigley for art support, the Core Engine team for discussions, and the anonymous reviewers for their comments. Thanks also to all my colleagues at the Maxis studio, past and present, for providing such a stimulating working environment.

## References

BAJAJ, C. L., AND SCHIKORE, D. R. 1996. Error-bounded reduction of triangle meshes with multivariate data. In *Storage and Retrieval for Image and Video Databases*, vol. 2656, 34–45.

COHEN, J. D., OLANO, M., AND MANOCHA, D. 1998. Appearance-perserving simplification. In *SIGGRAPH*, 115–122.

DECORO, C., AND TATARCHUK, N. 2007. Real-time mesh simplification using the GPU. In *Symposium on Interactive 3D Graphics (I3D)*, vol. 2007, 6.

GALLER, B. A., AND FISHER, M. J. 1964. An improved equivalence algorithm. *Commun. ACM 7* (May), 301–303.

GARLAND, M., AND HECKBERT, P. S. 1997. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '97, 209–216.

GARLAND, M., AND HECKBERT, P. S. 1998. Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the conference on Visualization '98*, IEEE Computer Society Press, Los Alamitos, CA, USA, VIS '98, 263–269.

GRAY, K. 2003. The Microsoft DirectX 9 Programmable Graphics Pipeline.

GUEZIEC, A. 1995. Surface simplification with variable tolerance. In *Proceedings of Medical Robotics and Computer Assisted Surgery (MRCAS 95)*, 132–139.

HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J. J., AND STUETZLE, W. 1993. Mesh optimization. In *Annual Conference on Computer Graphics*, 19–26.

HOPPE, H. 1996. Progressive Meshes. In *SIGGRAPH96*, ACM Press/ACM SIGGRAPH, New York, H. Rushmeier, Ed., CG-PACS, 99–108.

HOPPE, H. 1997. View-dependent refinement of progressive meshes. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '97, 189–198.

HOPPE, H. 1999. New quadric metric for simplifying meshes with appearance attributes. In *Proceedings of the 10th IEEE Visualization 1999 Conference (VIS '99)*, IEEE Computer Society, Washington, DC, USA, Visualisation '99.

KALVIN, A. D., AND TAYLOR, R. H. 1996. Superfaces: Polygonal mesh simplification with bounded error. *IEEE Comput. Graph. Appl. 16* (May), 64–77.

LINDSTROM, P., AND TURK, G. 1998. Fast and memory efficient polygonal simplification. In *Proceedings of the conference on Visualization '98*, IEEE Computer Society Press, Los Alamitos, CA, USA, VIS '98, 279–286.

LINDSTROM, P. 2000. Out-of-core simplification of large polygonal models. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '00, 259–262.

LOW, K.-L., AND TAN, T.-S. 1997. Model simplification using vertex-clustering. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, I3D '97, 75–ff.

LUEBKE, D., AND ERIKSON, C. 1997. View-dependent simplification of arbitrary polygonal environments. In *Annual Conference on Computer Graphics*, 199–208.

MANBER, U. 1989. *Introduction to algorithms: a creative approach*.

MULLER, D. E., AND PREPARATA, F. P. 1978. Finding the intersection of two convex polyhedra. *Theoretical Computer Science 7*, 217–236.

NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. 2008. Scalable parallel programming with CUDA. *ACM Queue 6*, 40–53.

RONFARD, R., AND ROSSIGNAC, J. 1996. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum 15*, 67–76.

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

SANDER, P. V., HOPPE, H., SNYDER, J., AND GORTLER, S. J. 2001. Texture mapping progressive meshes. Association for Computing Machinery, Los Angeles, CA, 409–416.

SCHROEDER, W. J., ZARGE, J. A., AND LORENSEN, W. E. 1992. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proc.) 26*, 2 (July), 65–70.