Volume xx (200y), Number z, pp. 1–9

Progressive Hulls for Intersection Applications

Nikos Platis and Theoharis Theoharis

Department of Informatics & Telecommunications, University of Athens,
Panepistemiopolis, 157 84 Ilissia, Greece
{nplatis|theotheo}@di.uoa.gr

Abstract
Progressive meshes are an established tool for triangle mesh simplification. By suitably adapting the simplification process, progressive hulls can be generated which enclose the original mesh in gradually simpler, nested meshes. We couple progressive hulls with a selective refinement framework and use them in applications involving intersection queries on the mesh. We demonstrate that selectively refinable progressive hulls considerably speed up intersection queries by efficiently locating intersection points on the mesh. Concerning the progressive hull construction, we propose a new formula for assigning edge collapse priorities that significantly accelerates the simplification process, and enhance the existing algorithm with several conditions aimed at producing higher quality hulls. Using progressive hulls has the added advantage that they can be used instead of the enclosed object when a lower resolution of display can be tolerated, thus speeding up the rendering process.

Keywords: surface simplification, progressive mesh, bounding volume, hull, intersection test, ray tracing, collision detection.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation, I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling, I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism.

1. Introduction

Hulls or bounding volumes of 3D models are used frequently in Computer Graphics, especially in applications involving intersection queries such as ray tracing and collision detection. Hierarchical bounding volumes, in particular, are essential tools for accelerating such queries by successively restricting the areas of potential intersection on the model.

Progressive hulls are hierarchical hulls of triangle meshes, constructed by suitable adaptation of progressive meshes, a mesh simplification technique based on edge collapses.

In this work, we couple progressive hulls with a selective refinement framework and present an algorithm that exploits this structure to considerably accelerate intersection queries on a given mesh. Regarding the generation of progressive hulls, we propose a new, efficient formula for assigning priorities to edge collapses, which significantly speeds up the simplification process; moreover, we enhance the original construction with several conditions, in order to produce higher quality hulls that enclose the original mesh tightly. Owing to their good fit, the selectively refinable progressive hulls can be reused, without additional overhead, in place of the original models when the display resolution is sufficiently small, thus further accelerating a graphics application.

2. Related Work

Progressive meshes Hoppe[1] introduced a generic framework for the simplification of manifold triangle meshes by iterative *edge collapses*. The edge collapse operation contracts an edge to a single vertex (Figure 1). Given an initial detailed mesh $\hat{M}$, successive edge collapses produce a sequence of meshes with decreasing numbers of triangles, up to a coarse *base* mesh M^0:

$$(\hat{M} = M^n) \xrightarrow{\text{ecol}_{n-1}} \ldots \xrightarrow{\text{ecol}_1} M^1 \xrightarrow{\text{ecol}_0} M^0.$$

The inverse of the edge collapse operation is termed a *vertex split*. Performing vertex splits on the base mesh, in

submitted to COMPUTER GRAPHICS *Forum* (12/2002).

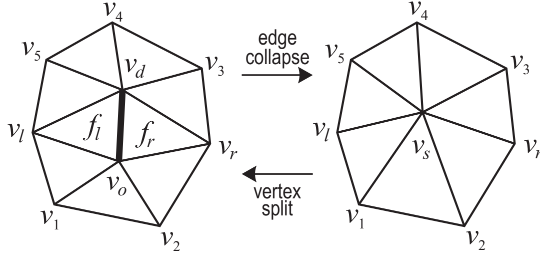

Figure 1: *Edge collapse / vertex split operations*

reverse order to the corresponding edge collapses, recovers the original mesh exactly:

$$M^0 \overset{\text{vsplit}_0}{\longrightarrow} M^1 \overset{\text{vsplit}_1}{\longrightarrow} \dots \overset{\text{vsplit}_{n-1}}{\longrightarrow} (M^n = \hat{M}).$$

A *progressive mesh* is a representation of $\hat{M}$ consisting of the base mesh M_0 and the sequence $\{\text{vsplit}_i,\ i = 0, 1, \dots, n-1\}$ of vertex splits that produce the original mesh.

The progressive mesh construction can be adapted for many purposes. The two issues that affect the resulting progressive mesh are the order in which the edges are collapsed and, for each collapse, the position of the new vertex[1, 2, 3, 4, 5, 6, 7].

Progressive meshes can also be extended to support *selective refinement*[8, 9, 10], the ability to perform an arbitrary vertex split without first performing all the splits preceding it in the progressive mesh representation. In this way, detail may be added only on specific parts of the mesh, as required by the application.

Progressive hulls Recently Sander et al.[11] presented an application of progressive meshes for the generation of hulls of closed manifold meshes.

Their reasoning is straightforward: the edge collapse operation affects the mesh only locally, on the triangles at the neighborhood of the collapsed edge; thus if, for each edge collapse, the generated vertex is placed on the "outside" of all these triangles, then the resulting mesh will completely enclose the original mesh, forming an outer hull for it. The hulls produced from the original mesh by this constrained simplification are appropriately called *progressive hulls*.

Hierarchical bounding volumes The first hierarchical structure used in the context of interference detection is probably the Dobkin-Kirkpatrick hierarchy[12]. It concerns only convex polyhedra, and determines the intersection of two polyhedra by updating their *separation distance* while traversing the hierarchy. This work is primarily of theoretical interest, as no practical results are presented.

Techniques utilizing hierarchical bounding volumes enclose the original model in nested sets of "boxes" of various shapes; their choice is a compromise between void space and simplicity of the bounding volume with the aim of maximizing the speed of intersection tests. Spheres and AABBs have been used for their simplicity, but hierarchies of k-DOPs[13] (polyhedra whose faces may only have predefined orientations) and OBBs[14] (arbitrarily oriented rectangular boxes) have proven most successful. These two methods are focused on collision detection between two (possibly moving) objects and are able to process arbitrary *polygon soups*. Both construct binary trees of nested volumes with a top-down approach: they start from the outer bounding volume that comprises all triangles of the object, and subdivide it in finer volumes enclosing fewer triangles, up to individual elements of the original object. During queries, the tree structure helps to quickly restrict the areas of potential intersection.

3. Progressive Hulls for Intersection Applications

3.1. Progressive hull construction

Terminology and basic formulae Given a vertex v, its *star* is the set of triangles that share this vertex. The number of triangles in the star of a vertex is called its *valence*. The *link* of a vertex v is a polygonal line made up from the boundary of its star, if all edges incident on v are removed. Similarly, the star (resp. the link) of an edge is the union of the stars (resp. the links) of its two endpoints. In Figure 1, the polygonal line $[v_l, v_1, \dots, v_r, \dots, v_5]$ is the link of the edge $v_o v_d$ (on the left) and of the vertex v_s (on the right).

Also let T_i be a face on a triangle mesh, with vertices $P_{i0}(x_{i0}, y_{i0}, z_{i0})$, $P_{i1}(x_{i1}, y_{i1}, z_{i1})$ and $P_{i2}(x_{i2}, y_{i2}, z_{i2})$ ordered counter-clockwise. The equation of its supporting plane is

$$\Pi_i(x,y,z) = \begin{vmatrix} x & y & z & 1 \\ x_{i0} & y_{i0} & z_{i0} & 1 \\ x_{i1} & y_{i1} & z_{i1} & 1 \\ x_{i2} & y_{i2} & z_{i2} & 1 \end{vmatrix} = 0$$

which can be written more concisely as

$$\Pi_i(x,y,z) = a_i x + b_i y + c_i z + d_i = 0. \tag{1}$$

An arbitrary point $\bar{P}(\bar{x}, \bar{y}, \bar{z})$ is on the "outside" of face T_i iff $\Pi_i(\bar{x}, \bar{y}, \bar{z}) > 0$. Additionally, the tetrahedron formed by $\bar{P}$ and the three points P_{i0}, P_{i1}, P_{i2} has volume $\frac{1}{6}\Pi_i(\bar{x}, \bar{y}, \bar{z})$.

3.1.1. Algorithm outline

Our algorithm for constructing progressive hulls follows the greedy scheme of most other mesh simplification algorithms based on iterative edge collapses:

1. For each edge of the mesh calculate a collapse *priority*, and sort the edges according to their priorities in a priority queue. The priority of each edge collapse is a measure of its impact on the mesh.
2. While there are more legal collapses in the queue and the simplification target (in our case, a specific number of faces) has not been reached,

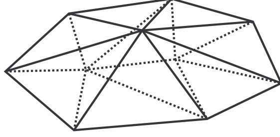

Figure 2: *The volume enclosed between the star of an edge and the star of the vertex resulting from its collapse.*

a. Remove the edge collapse with highest priority from the queue and apply it to the mesh.
b. Recalculate the priorities of all edge collapses affected by the changes on the mesh and update their position in the priority queue.

In contrast to k-DOPs and OBBTrees, our hierarchy is constructed bottom-up, starting from the original model and successively simplifying it to the base hull. Moreover, it does not use bounding volumes of some predetermined shape, but instead it is a hierarchy of hulls of decreasing complexity, each enclosing the original mesh completely.

3.1.2. Computation of new vertex position

We begin by showing how the position of the vertex resulting from an edge collapse is determined. Sander et al.[11] outlined this procedure in their paper.

Suppose that the star of an edge being collapsed contains triangles T_i, $i = 0, 1, \ldots, n$. We require that the resulting vertex $P_s(x_s, y_s, z_s)$ be on the outside of all faces T_i; thus, using the above notation, it must satisfy

$$\Pi_i(x_s, y_s, z_s) > 0 \quad \text{for all } \; i = 0, 1, \ldots, n. \qquad (2)$$

All these relations are linear on x_s, y_s and z_s and so they naturally form the constraints of a linear programming problem by which the three unknowns (the position of the new vertex) may be determined.

Seeking an appropriate objective function, we observe that the sum

$$\sum_{i=0}^{n} \Pi_i(x_s, y_s, z_s) \qquad (3)$$

is (six times) the total volume enclosed between the star of the collapsed edge and the star of the new vertex (Figure 2). Thus, by positioning the vertex so as to minimize this volume, the resulting mesh will intuitively be fairly close to the original one.

3.1.3. Calculation of edge collapse priorities

Candidate edge collapses are assigned a priority that determines the order in which they are applied to the mesh. We have investigated two different strategies for calculating these priorities.

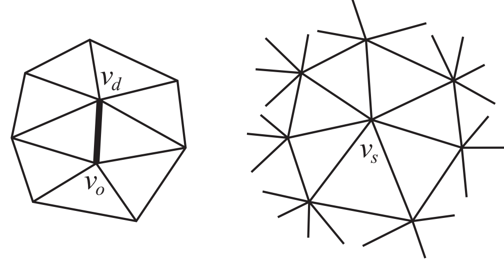

Figure 3: *After collapsing the edge $v_o v_d$, the priorities of all edges appearing on the right must be recomputed since they depend on the affected faces of the star of v_s.*

- The first strategy is the one used in [11]. The position of the vertex that would result from this collapse is calculated as described above; the enclosed volume given by equation (3) is used as the priority of the collapse, so that collapses with lower enclosed volumes are performed first.
  Unfortunately, owing to step (2.b.) of the simplification algorithm of §3.1.1, several computed priorities are thrown away in the course of the algorithm, being updated as a result of nearby collapses (see Figure 3). Given that the calculations performed are rather expensive, as they involve solving a linear programming problem for each candidate edge collapse, a large amount of computation is wasted with adverse effects on performance.
- The second strategy is our suggestion for a simpler formula to determine a priority for each edge collapse. We average the points on the link of the edge to get a "center" $P_c(x_c, y_c, z_c)$ of its star, and sum the unsigned volumes of the tetrahedra formed by P_{i0}, P_{i1}, P_{i2} and P_c for all faces T_i of the star. The resulting total volume is used as the priority of the edge collapse.
  This formula for assigning edge collapse priorities gave good results in practice, while dramatically reducing simplification times (see Section 4.1). Its success can be justified by noticing that the volume used as the priority of the edge collapse is the total volume spanned by the star of the edge, with P_c to "close" it. This volume will be small if the star is relatively planar or if its faces are relatively small; collapsing such edges will likely have little impact on the mesh and thus they should be given high priority. It should be evident that, using this strategy for assigning edge collapse priorities, the costly calculation of the resulting vertex position is performed only when the collapse is about to be applied to the mesh.

3.1.4. Enhancements to the simplification algorithm

Conditions on candidate edge collapses When an edge is considered for collapse, we must ensure that the mesh will remain manifold after performing it. The link condition of [15] is checked and violating edges are rejected immediately.

Candidate edges are also rejected if their valence is over

some user-specified value n_{max}. Vertices of high valence are not desired in triangular meshes since they render poorly; additionally in our algorithm they would complicate subsequent calculations by leading to linear programming problems with many constraints, which are inefficient to solve.

Conditions on the resulting vertex The position of the vertex generated from an edge collapse is determined by the optimization procedure of §3.1.2. Unfortunately, this procedure does not incorporate any measure of the quality of the simplified mesh; thus the position of the new vertex may create triangles of very low compactness (very thin and elongated ones), which are not desirable since they cause rendering artefacts, or triangles whose normal direction changes significantly after the edge collapse, which could lead to creases or self-intersections on the mesh. Incorporating relevant measures in the minimization problems would result in quadratic problems, which we avoided for reasons of efficiency. Instead, we introduce two additional tests after computing the position of the new vertex but before accepting the collapse.

The first test ensures that the compactness of the triangles on the star of the collapsed edge will not decline considerably after the collapse. We calculate the minimum compactness c_e of the triangles at the star of the collapsed edge $v_o v_d$ as well as the minimum compactness c_v of the triangles at the star of the new vertex v_s, and allow the collapse only if $c_v/c_e > r$ where r is a user-specified tolerance value.

The second test rejects collapses that lead to significant changes in the orientation of normals of the affected triangles. For each triangle at the star of the new vertex, we calculate the deviation of its normal before and after the collapse; if it is greater than a user-specified maximum deviation θ then we reject the collapse. The angle θ must be in the range $[0, \pi/2]$ in order to avoid creases on the mesh.

These two tests (as well as the valence test above) were previously proposed by Guéziec[3] in the context of his simplification method based on edge collapses, but, to our knowledge, they were not applied to progressive hulls.

Priority queue Most other authors have used a priority queue, typically implemented as a binary heap, to sort candidate edge collapses. We experimented with a red-black tree instead, which performed faster. All the basic operations of both these data structures[16] are $O(\log n)$. However, the most frequent operations in the simplification algorithm are updates and deletions: in any iteration, two edges are removed from the queue and $O(n_{max}^2)$ edges are updated. The better internal sorting of the red-black tree contributes to faster running times.

3.2. Intersection Tests with Progressive Hulls

Our initial motivation for the work of this paper was the use of progressive hulls for the efficient determination of intersections between a triangle mesh and a geometric primitive (ray, triangle, line segment). In this section we describe in detail the relevant algorithm we developed.

3.2.1. Progressive Hull Intersection Algorithm

Our algorithm selectively refines the progressive hull of the mesh in areas where intersections with the given primitive are detected, until the original mesh is reached or no more intersections are found. Selective refinement is realized through suitable vertex splits, and, owing to their localized effect on the mesh, the algorithm continuously restricts the areas of the mesh where intersections may occur, up to individual faces on the original mesh M^n.

Specifically, our algorithm maintains a list of *candidate faces*, which will be tested for intersection with the given primitive. This list is initially populated with all the faces of the base mesh M^0.

Each face is removed from the list and tested for intersection with the primitive. The following three cases must be distinguished:

1. The intersection test is negative. In this case the face is simply disregarded.
2. The test is positive and the face belongs to the original mesh; this is true if none of its vertices can be further split in the progressive hull representation. In this case the primitive intersects with the original mesh and the intersection point(s) can be recorded.
3. The test is positive but the face belongs to some intermediate level of detail in the progressive hull, so it is possible to split some of its vertices. In this case for every such vertex we perform the corresponding split (see next paragraph), refining the hull in the vicinity of the face. We also record all the faces affected by the splits. These faces essentially belong to levels of finer detail in the progressive hull; they must be (re)tested for intersection against the primitive, and are thus appended to the list of candidate faces (unless they are already in the list).

The algorithm terminates when the list of candidate faces becomes empty. The intersections between the original mesh and the primitive, if any, will have been found in the second case above, thanks to the refinement of the progressive hull performed in the third case (Figure 4).

The above algorithm may fail to detect some "inner" intersections between the primitive and the mesh, if the mesh folds and the primitive lies completely below the hull in the vicinity of the intersection points (Figure 5). Our experiments have shown that such intersections occur rarely and most "inner" intersections are detected correctly. In practice, most intersection applications are only interested in the "outer" (extreme) intersections only, which are always detected by our algorithm.

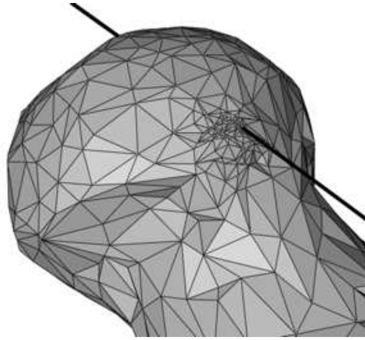

Figure 4: *A progressive hull of the balljoint model intersected by a line segment. The 2,000 face base hull has been locally refined around the intersection point.*

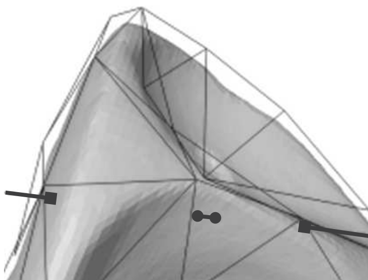

Figure 5: *The "inner" intersections of the line segment and the screwdriver model (dotted) are not detected by our algorithm. The line segment passes completely below the hull in this area.*

Performing selective refinement on a progressive mesh Selective refinement of a progressive mesh assumes the ability to perform vertex splits in arbitrary order. Various ways to accomplish this have been devised, each imposing different dependencies for vertex splits[8, 9, 10]. For our application, the link (and thus the star) of the vertex that will be split must be identical to what it would be if all the preceding splits were performed in order. Referring back to Figure 1, in order to split v_s, all vertices v_l, v_r, $v_1, \ldots, v_5$ must exist on the current mesh. Some of these vertices would be created by splits earlier in the progressive mesh, and these splits must be performed (recursively) prior to splitting v_s.

This scheme is similar to the one of Xia and Varshney[9], and imposes dependencies between vertex splits as mentioned in their work. The selective refinement scheme of Hoppe[8] is less constrained, requiring only the neighboring faces of f_l and f_r to exist on the mesh, and thus more efficient; unfortunately, it cannot be used in the context of intersection queries, since the star of selectively split vertices may not be recovered exactly and consequently the enclosing property of the progressive hull hierarchy may not be retained.

Due to the vertex split dependencies, our hull hierarchy is more involved than the simple binary trees of k-DOPs and OBBTrees, which incurs higher cost to update it during its traversal; however it is also finer, allowing more localized updates on the mesh.

3.2.2. Incremental updates of the hull

Intersection tests against a mesh are typically performed in batches, for example with sets of rays or with the triangles of another mesh. Therefore, coherence can be potentially exploited in order to accelerate the procedure.

To this end, we experimented with incremental updates of the hull. For each test, we used the partially refined hull that resulted from the previous test, applied the intersection algorithm as above, and additionally coarsened the parts of the hull that were no longer near the areas of intersection. However, our tests showed that for our particular application this technique actually *deteriorated* performance. This can be justified by noticing that to coarsen the mesh selectively, we need to verify that candidate edge collapses are legal and do not affect intersecting faces. This requires several validity checks[8, 9] and makes this method inefficient in practice. On the contrary, for any intersection test only small parts of the mesh are refined and relatively few vertex splits are performed; thus it is very efficient to invert them and restore the base hull. This is the solution we used for our tests.

A variation of this scheme is used in [17]: selective coarsening is also avoided, but the partially refined hull is used for several consecutive queries before reverting to the base hull.

4. Results

We tested our approach using progressive hulls for intersection tests with six models of varying numbers of triangles and geometric complexity: a rocker arm model (20,088 faces), a screwdriver model (54,300 faces), a dinosaur model (84,288 faces), a horse model (96,966 faces), an Igea head model (134,342 faces) and a balljoint model (274,120 faces). We constructed their progressive hulls at various levels of detail and report our results in Section 4.1. We then performed intersection tests, representative of real-world applications, using the constructed hulls, and present the outcome of our tests in Section 4.2. All tests were performed on 800 MHz Pentium III PCs with 384 MB of RAM.

4.1. Progressive Hulls

Our simplification algorithm requires three user-defined parameters; we carried out most of our tests using the following values: we imposed a maximum valence of 12 (most vertices of triangle meshes have initial valence of 6 to 8), we employed a tolerance of 0.8 for compactness deterioration, so as to allow some decline in triangle shape, and we used a moderately low maximum normal deviation of $\pi/4$.

This choice of rather strict parameters generated very good progressive hulls, which, for resolutions of up to 2,000 faces were visually very close to the original models (see Figures 7, 9–12); however it also terminated the simplification process relatively early, at around 20–70 faces in most cases. We also experimented with looser parameters but the differences were small: models with a lot of detail, such as the dinosaur, gave somewhat better results, whereas smooth models such as Igea gave marginally worse results. In any case, such small differences in the progressive hulls have no essential effect during the intersection queries, as will be evidenced by the results in the next section.

Table 1 reports times needed to construct progressive hulls of 2,000 faces for the six models, using the two strategies of §3.1.3 for assigning edge collapse priorities. The new strategy proposed in this paper reduces simplification times significantly.

Table 1: *Simplification times to construct progressive hulls of 2,000 faces.*

Model	#faces	1st strategy	2nd strategy
Rocker Arm	20,088	54 s	8 s
Screwdriver	54,300	161 s	26 s
Dinosaur	84,288	245 s	37 s
Horse	96,966	298 s	45 s
Igea	134,342	478 s	69 s
Balljoint	274,120	1,006 s	158 s

Table 2 presents the percentage of void space between the original models and the generated base hulls of 2,000 and 200 faces. This is an appropriate measure of the fit of these hulls to the original models. More concrete measures used to assess the quality of simplified meshes, such as distance from the model, would probably be misleading in this context since the construction of progressive hulls imposes severe restrictions on the position of new vertices.

The void space was computed by subtracting the volume of the original model from the volume of the hull. Volumes were calculated by summing $\frac{1}{6}\Pi_i(0,0,0)$ for all triangles T_i of the mesh, using double precision. Guéziec[3] suggests more accurate methods for this calculation, which were not applied for simplicity.

For 2,000 faces the hulls enclose the original models very tightly, confirming the visual impression of Figures 7 and 9–12. In all cases, as expected, the first strategy for assigning collapse priorities produces better results than the second one (Figures 6 and 7), but the deterioration is acceptable given the large decrease in simplification times and the small impact on intersection test times, as will be shown next.

Table 2: *Percentages of void space for hulls of 2,000 and 200 faces.*

Model	Hull size	1st strat.	2nd strat.
Rocker Arm	2000	2.87%	5.72%
	200	28.90%	62.91%
Screwdriver	2000	2.21%	3.53%
	200	20.70%	38.99%
Dinosaur	2000	15.08%	21.70%
	200	164.84%	165.85%
Horse	2000	4.11%	6.86%
	200	65.06%	65.60%
Igea	2000	2.04%	3.02%
	200	12.06%	17.14%
Balljoint	2000	3.46%	5.32%
	200	19.60%	31.16%

Figure 6: *Screwdriver:Initial model (54,300 faces), hulls of 200 faces generated with the 1st and 2nd strategy respectively.*

4.2. Intersection Tests

To test our algorithm for intersection queries we chose to implement intersection tests between line segments and the constructed progressive hulls. This situation is representative of applications that may benefit from our algorithm and

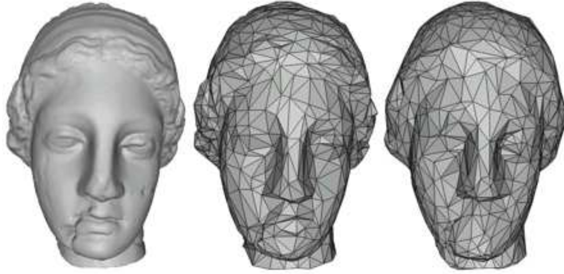

Figure 7: *Igea:Initial model (134,342 faces), hulls of 2,000 faces generated with the 1st and 2nd strategy respectively.*

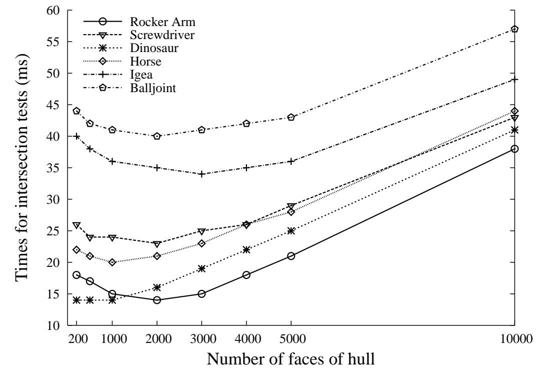

Figure 8: *Times for intersection tests with progressive hulls of various numbers of faces.*

demonstrates our concept in a straightforward manner. For each of the six models we created 10,000 random line segments with ends on the sides of a box twice as large as their minmax box and tested for intersections against the model.

Table 3 presents in detail our results for the horse model. In all cases the minmax box of the base hull (or the model) was used to quickly discard non-intersecting line segments.

From the data in this table we can deduce, first and foremost, that the use of progressive hulls can dramatically decrease the workload of intersection applications. Secondly, we observe that the increase in void space for the hulls obtained with the second strategy for assigning edge collapse priorities does not cause significant increase in the time required for the intersection tests; this is a good indication of the usefulness of our new strategy.

Intersection tests with progressive hulls for the other models follow a similar pattern. Figure 8 presents intersection times as a function of base hull size, using progressive hulls generated with the second strategy. The height of each curve depends on the number of line segments that intersected each model, cf. Table 4. In general, hulls of 500–3000 faces performed best, with small variation; they represent a good trade-off between quality of fit, number of intersection tests and number of vertex splits performed. It should be noted that the performance of progressive hulls cannot be determined only based on these factors; it depends on the number of faces and the shape of the original model, and is certainly affected by the implementation efficiency of intersection tests and of updates to the mesh for vertex splits/edge collapses.

Comparison with other bounding volume hierarchies Comparisons between techniques that accelerate intersection queries is inherently difficult, since the type of tests can strongly affect the performance of each method. To have some measure for the performance of our algorithm, we repeated the above tests using *k*-DOPs, which are generally accepted as one of the most efficient bounding volume hierarchies[13].

It should be noted that the original code of *k*-DOPs implements intersection queries between two triangle meshes. We adapted it to our test situation by changing the final triangle–triangle intersection test with the same line segment–triangle intersection test that we use in our algorithm[18] with a performance gain of 30%–40% compared to the unmodified code. Moreover, we tried the whole family of 6-, 14-, 18- and 26-DOPs, and 26-DOPs were faster in all cases, at the expense of considerably higher preparation times. Table 4 summarizes our results.

5. Conclusions and Further Work

An efficient framework for intersection queries against triangle meshes using progressive hulls was presented. Through selective refinement of the hull, our algorithm quickly localizes areas of potential intersection and can determine the intersection points in less time than alternative methods. In addition, we have streamlined the progressive hull construction algorithm with a new efficient strategy for assigning priorities to edge collapses and with several enhancements for higher quality results.

The progressive hull generation procedure could be further improved and accelerated. Concerning our implementation, we have used the Simplex method to solve the linear programming problems, which is adequately efficient given the small size of the problems. Seidel[19] has proposed a fast randomized algorithm for such optimization problems, which could be used instead. On the other hand, the encouraging results of our new strategy for calculating collapse priorities indicate that simpler strategies should be investigated; even the optimization procedure for computing the resulting vertex position could be replaced by a looser (but more efficient) computation.

Our algorithm for intersection queries against progressive hulls also has potential for improvement. We are currently investigating the possibility to fully detect all “inner” inter-

Table 3: *Intersection tests for progressive hulls of the horse model. The times are total for 10,000 random line segments. The numbers of intersection tests and performed splits are averaged for the 10,000 line segments. 1,119 line segments intersected the model.*

Base hull	1st strategy			2nd strategy		
(#faces)	time	#tests	#splits	time	#tests	#splits
200	22.11 s	759	226	23.11 s	798	238
500	21.15 s	812	203	22.38 s	868	217
1,000	**20.77 s**	939	179	**21.79 s**	986	192
2,000	21.41 s	1,209	145	22.59 s	1,254	155
3,000	23.93 s	1,522	124	24.44 s	1,550	131
4,000	26.07 s	1,841	108	26.98 s	1,871	114
5,000	28.79 s	2,176	96	29.46 s	2,200	100
10,000	44.12 s	3,904	61	44.29 s	3,907	63
Horse (96,966)	362.63 s	35,751				

Table 4: *Intersection times for k-DOPs and progressive hulls (PH).*

Model	#inters. lines	6-DOPs	14-DOPs	18-DOPs	26-DOPs	PH
Rocker Arm	2,044	45 s	24 s	21 s	18 s	14 s
Screwdriver	1,641	120 s	69 s	50 s	45 s	24 s
Dinosaur	955	190 s	73 s	68 s	49 s	14 s
Horse	1,119	216 s	79 s	76 s	52 s	21 s
Igea	1,808	298 s	107 s	68 s	45 s	35 s
Balljoint	1,545	707 s	258 s	204 s	137 s	41 s

section points when the surface folds; we feel that it will be necessary to use the enclosed volume, as opposed to the hull itself, to make this possible. We are also considering the use of our algorithm in intersection queries between two meshes, and any possible enhancements in this context. Exploitation of spatial and temporal coherence for successive intersection queries should be further considered, and techniques similar to the ones presented in [17] could be applied.

Acknowledgements

The authors would like to thank Vassilis Zissimopoulos for helpful discussions on optimization, André Guéziec for providing the Technical Report of [3], and James Klosowski for the code of *k*-DOPs. The authors would also like to thank the anonymous reviewers for their constructive comments and suggestions on the original version of this paper.

The horse model is from the Large Geometric Models Archive at Georgia Institute of Technology (`http://www.cc.gatech.edu/projects/large_models/`), all other models were obtained from Cyberware Inc. (`http://www.cyberware.com/`).

References

1. H. Hoppe, “Progressive Meshes”, in *SIGGRAPH 96 Proceedings*, pp. 99–108.
2. M. Garland and P. S. Heckbert, “Surface Simplification Using Quadric Error Metrics”, in *SIGGRAPH 97 Proceedings*, pp. 209–216.
3. A. Guéziec, “Locally Toleranced Surface Simplification”, *IEEE Visualization and Computer Graphics*, **5**(2), pp. 168–189 (1999). Extended version as IBM TR-97-8700.
4. L. Kobbelt, S. Campagna, and H.-P. Seidel, “A General Framework for Mesh Decimation”, in *Proceedings of Graphics Interface ’98*, pp. 43–50.
5. P. Lindstrom and G. Turk, “Fast and Memory Efficient Polygonal Simplification”, in *Proceedings of IEEE Visualization ’98*, pp. 279–286.
6. M. Garland, “Multiresolution Modeling: Survey & Future Opportunities”, *State of the Art Report, Eurographics ’99*, (1999).
7. E. Puppo and R. Scopigno, “Simplification, LOD and Multiresolution — Principles and Applications”, *Eurographics ’97 Tutorial Notes*, (1997).

8. H. Hoppe, “View-Dependent Refinement of Progressive Meshes”, in *SIGGRAPH 97 Proceedings*, pp. 189–198.

9. J. C. Xia and A. Varshney, “Dynamic View-Dependent Simplification for Polygonal Models”, in *Proceedings of IEEE Visualization ’96*, pp. 327–334.

10. L. D. Floriani, P. Magillo, and E. Puppo, “Building and Traversing a Surface at Variable Resolution”, in *Proceedings of IEEE Visualization ’97*, pp. 103–110.

11. P. V. Sander, X. Gu, S. J. Gortler, H. Hoppe, and J. Snyder, “Silhouette Clipping”, in *SIGGRAPH 2000 Proceedings*, pp. 327–334.

12. D. P. Dobkin and D. G. Kirkpatrick, “Determining the Separation of Preprocessed Polyhedra — A Unified Approach”, in *Automata, Languages and Programming (Coventry 1990)*, no. 443 in Lecture Notes in Computer Science, pp. 400–413, Springer-Verlag, (1990).

13. J. T. Klosowski, M. Held, J. S. Mitchell, H. Sowizral, and K. Zikan, “Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs”, *IEEE Visualization and Computer Graphics*, **4**(1), pp. 21–36 (1998).

14. S. Gottschalk, M. C. Lin, and D. Manocha, “OBBTree: A Hierarchical Structure for Rapid Interference Detection”, in *SIGGRAPH 96 Proceedings*, pp. 171–180.

15. T. K. Dey, H. Edelsbrunner, S. Guha, and D. V. Nekhayev, “Topology Preserving Edge Contraction”, *Publications de l’Institut Mathématique (Beograd)*, **66**(80), pp. 23–45 (1999).

16. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, first ed., (1990).

17. A. Guéziec, ““Meshsweeper”: Dynamic Point-to-Polygonal-Mesh Distance and Applications”, *IEEE Visualization and Computer Graphics*, **7**(1), pp. 47–61 (1999).

18. T. Möller and B. Trumbore, “Fast, Minimum Storage Ray-Triangle Intersection”, *Journal of Graphics Tools*, **2**(1), pp. 21–28 (1997).

19. R. Seidel, “Small-dimensional linear programming and convex hulls made easy”, *Discrete Computational Geometry*, **6**, pp. 423–434 (1991).

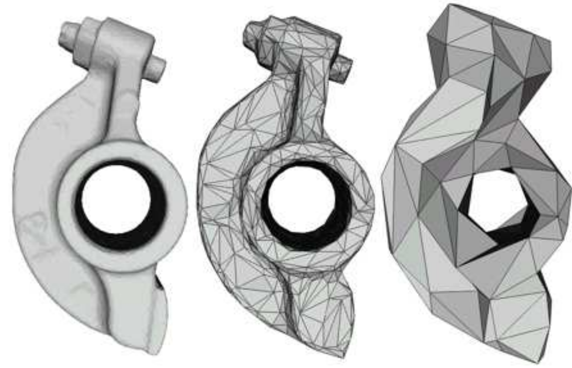

Figure 9: *RockerArm:Initial model (20,088 faces), hulls of 2,000 and 200 faces generated with the 2nd strategy.*

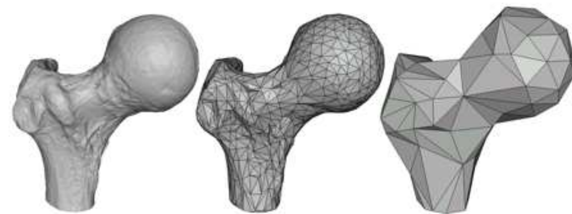

Figure 10: *Balljoint:Initial model (274,120 faces), hulls of 2,000 and 200 faces generated with the 2nd strategy.*

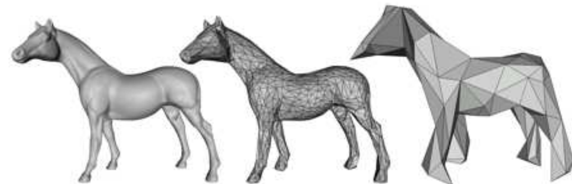

Figure 11: *Horse:Initial model (96,966 faces), hulls of 2,000 and 200 faces generated with the 2nd strategy.*

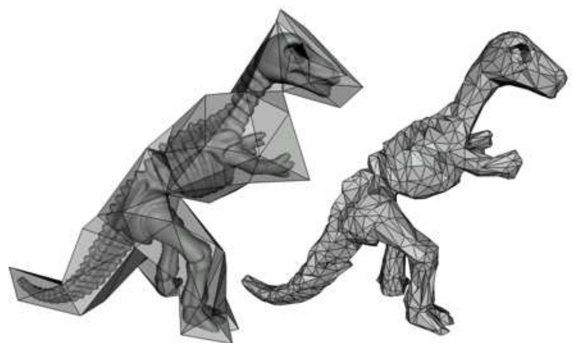

Figure 12: *Dinosaur:Initial model (84,288 faces) in its 200-face hull, hull of 2,000 faces generated with the 2nd strategy.*