Datalogisk Institut
KØBENHAVNS UNIVERSITET

A Simple Plane Patcher Algorithm

Kenny Erleben & Knud Henriksen

Technical Report no. 06/09
ISSN: 0107-8283

Dept. of Computer Science
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

A Simple Plane Patcher Algorithm

Kenny Erleben*and Knud Henriksen†

Department of Computer Science, University of Copenhagen, Denmark

Technical Report DIKU-TR-06/09

Abstract

The daily work with mesh data structures can be a painful experience. Topological inconsistency and digital mockup is part of daily life and often unwanted in both visualization, collision detection and animation.

In this paper we focus on the particular problem of patching (also termed capping) an open boundary of a mesh after it has been cut by a plane. The paper describes an algorithm for planar patching and outlines an implementation. The novel contribution is a divide and conquer algorithm working on a spatial hierarchical data structure of the cutting boundaries of the mesh.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Boundary representations; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Constructive solid geometry (CSG);

Keywords: Mesh Plane Clipping, Mesh Patching

1 Introduction

In computer graphics and animation we often cut a closed twofold mesh without self-collisions by a plane. For instance, this is required when building octrees for collision detection [I.J.Palmer and R.L.Grimsdale 1995; Dingliana and O'Sullivan 2000; O'Sullivan and Dingliana 1999; Tzafestas and Coiffet 1996; Zachmann 1995], adaptive grids for level set methods [Frisken et al. 2000], and convex decompositions [Coutinho 2001; O'Rourke 1998].

The actual plane clipping [Eberly n. d.] is not a very big problem although numerical precision can be cumbersome in cases of elongated polygons. The problem with the plane clipping is that it turns a closed twofold into an open twofold, which is unpleasant when doing simulation. In simulation we like objects to be closed, in order to resemble the fact that all real-world objects are volumetric when one look at them at an appropriate scale.

After a clipping operation has taken place we wish to post-process the clipped pieces with a "patching" algorithm that turns the open two-folds into closed two-folds without self-collisions.

This paper describes and shows results of a simple method we have implemented for patching a twofold mesh that have undergone a clipping operation by a plane.

*kenny@diku.dk

†kaiip@diku.dk

From a constructive solid geometry (CSG) point of view the problem, we have outlined, is easily dealt with. However, for such a simple problem we will rather not have to implement something as complex as arbitrary CSG operations on B-reps [James D. Foley and Hughes 1996]. Furthermore, using CSG operations is likely to be computational intractable compared to the simple algorithm we describe in this paper. Just imagine the large number of operations involved in creating a small octree (5-10 levels). We will rather have a simple and inexpensive alternative that deals with the specific problem at hand.

In [Krishnan et al. 1995] a system for boolean combinations of B-reps is described, although intended for patch-surfaces it could easily be applied to polygons as well. From [Krishnan et al. 1995] it is seen that general CSG operations on B-reps include a lot of data structures and book-keeping which is not really necessary for plane clipping and patching. We refer the reader to [Mantyla 1988; Hoffmann 1989; Naylor 1992; Requicha and Rossignac 1992; Requicha and Voelcker 1985] for more information on CSG with polyhedral B-reps.

There exists other methods for doing CSG. Some are based on rendering [Stewart et al. 2002], they do not produce a B-rep as a final result and is therefore not usable for us.

Finally, other approaches use voxel [Bærentzen 2001] or implicit representations [Bærentzen 2001; Museth et al. 2002]. Methods like these would require us to convert between representations, which can be very costly and therefore not attractive.

In conclusion, existing methods in CSG working on polyhedra is not optimized for our specific problem, other existing methods would either require us to convert between representations which is far to costly or will not produce a B-rep at all.

In Section 2 we will introduce our terminology and definitions, afterwards in Section 3 we will explain how the open cutting boundaries are detected. In Section 4 we will present a solution of a simple case, and in Section 5 we will introduce a spatial data structure for the cutting boundaries. In Section 6 we show how the data structure is used in an incremental divide and conquer method. Finally, we show some results in Section 7 and conclude our work in Section 8.

2 Terminology and Definitions

By a plane patching algorithm we mean an algorithm capable of capping a mesh volume after it has been cut by a plane. In this paper we only consider plane clipping, meaning the cutting boundaries are all planar.

Throughout this paper we will assume that the original mesh (before the clipping) is a twofold mesh. If a mesh is twofold then it means that any path traveled around any given vertex is isomorphic with a 2D circle. This means one can walk around on the neighboring faces of the vertex from a starting point and back without ever crossing ones own path. Assuming the mesh is a twofold implies that the mesh is closed, meaning that no face can be found having an edge without a neighboring face. Furthermore, it is assumed that there are no self-collisions of the mesh surface.

```
class Edge
  Vertex origin
  Vertex destination
  Edge twin
  Edge next
  Edge prev
  Face face

class Vertex
  Vector3 coord
  list<Edge> edges;

class Face
  list<Edge> boundary

class Mesh
  list<Vertex> vertices
  list<Edge> edges
  list<Face> faces
```

Figure 1: Mesh data structure pseudo code.

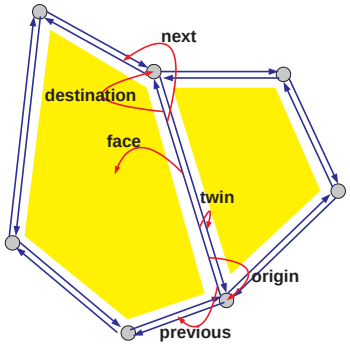

Figure 2: A simple 2D mesh. The red arrows indicate the member references of the edge.

We use a mesh data structure similar to the half edge data structure in [Mantyla 1988], also known as a double connected edge list. The data structure is roughly as shown in Figure 1. Observe that an edge is in fact represented by two edges going in opposite directions. The relationship between an edge and its "neighbors" is illustrated in Figure 2. Edge boundaries of faces are assumed to be given in counter clockwise (CCW) order, and edges that have a twin with a face null pointer are called open edges and are part of the open boundaries, i.e. they lie on the cutting plane. Observe that the mesh is not capable of representing holes in faces, further we require faces to be convex, but not necessarily triangles.

From our choice of data structure and the assumptions described above, it is evident that we have three important properties: The open boundary is connected, all open boundaries are planar and there is no collision between the open boundaries. We refer to these properties as A, B and C, they are explained in more detail below.

Property A: If we encounter an open edge then this edge is connected to exactly two other open edges, one at each end point. If we pick a direction and walk along these open edges then we will get a closed path. We refer to such a path as a ring.

Property B: All rings are coplanar. In fact the plane they lie in is the cutting plane. Projecting the rings onto the cutting plane will allow us to work in 2D instead of 3D.

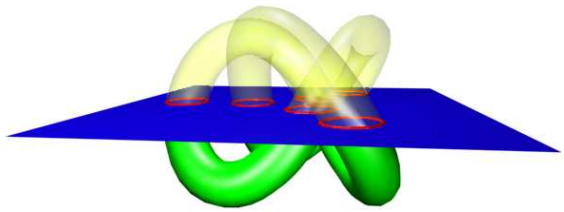

Figure 3: A blue cut plane through a knot mesh, dividing it into two parts, a yellow and a green. Rings are shown as red intersection curves with the plane.

```
void init(Mesh  mesh)
  for each edge in mesh.edges
    edge.seen = false
  next edge
  for each edge in mesh.edges
    if edge.face is NULL and edge.seen is false then
      Ring ring = boundaryWalk(edge)
      ... do something with ring...
    end if
  next edge
End init
```

Figure 4: The initialization method.

Property C: A ring might be concave but never self-colliding. and no two given rings intersect with each other. Therefore, we can determine whether a ring lies inside or outside of another ring simply by testing if a single projected vertex of the ring lies inside or outside the projection of the other ring.

Figure 3 illustrates some of the concepts.

3 Determining Open Boundaries

Initially our algorithm must determine all the rings of the mesh. We can exploit property A for doing this. First we search for an open edge not previously seen as illustrated in Figure 4.

Having found such an open edge it must be part of an unseen boundary. We therefore walk around the other open edges of the boundary until we reach the initial open edge. This is shown in Figure 5). The algorithm works as long as rings do not touch. If rings touch then we could sort the incoming edges in CCW order around the normal of outgoing vertex of the touching boundary vertices. The proper edge to continue along while walking the boundary would be the last edge in the ordered CCW sequence. However, the current implementation ignores the case of touching rings, since this case can not occur if the original mesh was a true "volume".

```
Ring * BoundaryWalk(Edge seed)
  Ring ring
  seed.seen = true
  Edge loop  = seed
  Vertex cur = seed.destination;
  do
    Ring.add(cur);
    bool found = false
    for each edge e in cur.edges do
      if e.twin.face is NULL then
        loop = e.twin
        cur  = e.twin.destination
        found = true
        break
      end if
    next e
    if not found then
      error...
      return NULL
    end if
    loop.seen = true
  while loop!=seed
  return ring
End BoundaryWalk
```

Figure 5: The boundary walk algorithm.

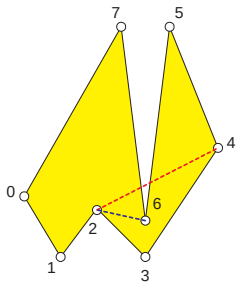

Figure 6: Diagonal example. The red diagonal is illegal, it crosses the lines from 5 to 6 and from 6 to 7. The blue diagonal is valid.

```
void RecursiveDecomp(Ring ring,list<Ring> pieces)
  if ring.vertices.size() == 3 then
    pieces.add(ring)
    return
  end if
  rightTurn = false
  for each vertex, B, in ring.vertices do
    let A be previous vertex of B
    let C be next vertex of B
    if A,B and C makes right turn then
      rightTurn = true
      break
    end if
  next B
  if not rightTurn then
    pieces.add(ring)
    return
  end if
  let A be next vertex of B
  do
    let A be next vertex of A
  while A<>B and line(A,B) inside ring
  if A is B then
    ... error ...
    return
  end if
  Ring piece1;
  for vertex V=B to A do
    piece1.add(V)
  Ring piece2;
  for vertex V=A to B do
    piece2.add(V)
  RecursiveDecomp(piece1,pieces)
  RecursiveDecomp(piece2,pieces)
End RecursiveDecomp
```

Figure 7: The recursive ring decomposition algorithm.

4 Patching a single Ring

Assume we only have a single ring. In this case it would be a simple matter to carry out the patching. All we have to do is to decompose the ring into convex pieces, project back the convex pieces of the ring from the 2D cutting plane to the 3D space, and then insert faces into the mesh corresponding to the convex pieces.

Instead of doing a convex decomposition a first thought might be to just triangulate the ring, because triangles are convex, so this will not violate our requirements to our mesh data structure. However, in general the ring could have any shape, which makes it a bit difficult to apply a simple brute force approach like ear-clipping [O'Rourke 1998] for triangulating the ring. We could of course resort to something like a constrained Delaunay triangulation [Shewchuk 1996; Shewchuk 2002] to deal with our problem, but this is far from being a simple and fast approach as we want.

Instead, we use a simple recursive convex decomposition. The method works by searching for a non-intersecting "diagonal". That is, a line between two vertices of the ring, such that the line lies inside the ring, but does not cross the ring. Figure 6 shows examples of valid and invalid diagonals. When a valid diagonal is found we simply split the ring into two rings along this diagonal and process the resulting two rings recursively. Eventually we end up with a list of convex rings.

Since faces are given in CCW order the vertices of a ring will also be in CCW order. Therefore, if all vertices make a left turn around the ring, then the ring is convex.

We use this fact to find a diagonal. First, we search for a vertex making a right turn, this vertex will be one end point of the diagonal. Then we search for another vertex of the ring such that the line segment between the right turn and the vertex lies inside the ring. If such a vertex is found it is used as the other endpoint of the diagonal, and if no such vertex is found an error is reported.

The algorithm is illustrated with pseudo code in Figure 7. Unfortunately, it is only a limited number of meshes, which will result in a single ring when they are cut by a plane. In general there could be many rings, some of them might even be nested within each other.

The algorithm is similar to Hertel and Mehlhorn [O'Rourke 1998]. The main difference lies in how we find diagonals, our method uses an ear-clipping [O'Rourke 1998] strategy whereas Hertel and Mehlhorn requires a triangulation. Our worst case time complexity is $O(nlgn)$ whereas Hertel and Mehlhorn is linear.

```
class Ring
  list<Vertex> vertices
  list<Ring> children

class Hierarchy
  list<Ring> outermost
```

Figure 8: Ring hierarchy data structure pseudo code.

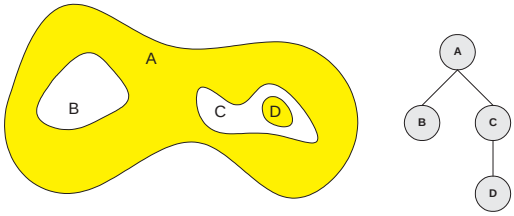

Figure 9: The rings of a cut plane, yellow color shows regions that should be patched. On right side the corresponding hierarchy is drawn.

5 Hierarchical Ordering

In this section we will describe a method for constructing a hierarchical data structure of rings. In the next section it will be evident how this hierarchical data structure can be used in a divide and conquer manner to reduce the problem of handling multiple possibly nested rings as the simple case described in the previous section.

From the properties A,B and C we can build a hierarchical data structure giving us the required spatial information about the rings in constant time. The hierarchy is constructed such that a ring which is not enclosed by any other rings is placed at the root of the hierarchy. If a ring encloses a set of rings then these rings are the immediate children of that ring. Rings enclosed by a child are not immediate children, but grand-children. At the leaves of the hierarchy we will find rings that do not enclose other rings. Figure 8 shows the hierarchal data structure in pseudo code. In Figure 9 an example hierarchy is shown. There is a single outermost ring, *A*, enclosing two other rings, *B* and *C*, describing "holes" in *A*. Inside *C* another ring, *D*, is placed. Observe that *D* is a descendent of *A* and not its immediate child. Another property of the hierarchy is that all rings at odd depths in the hierarchy describe holes in the ring of their parent. Rings at even depths describe the outer boundaries of an area that should be patched.

We build the data structure in an incremental way as illustrated in Figure 10. We search for new rings, one at the time, by looking for an unvisited open edge and when such an edge is encountered we do a boundary walk, tracing the edges of the new ring. During the trace we mark all the edges of the new rings as being visited so we will not trace the same ring more than once see Section 3 for details.

After having found a new ring, we recursively traverse the hierarchical data structure, testing the already inserted rings against the new ring in order to determine where the new ring should be placed in the hierarchy. We use the term candidate ring, when we refer to a ring that is already inserted into the hierarchy.

We keep a set of the outermost rings encountered so far. After having found a new ring, we test if it lies inside any of the other outermost candidate rings. If it does not then we can add it to the set of outermost rings. However, if the ring lies inside a candidate ring then we will test the ring against the children of the candidate ring recursively. The recursion ends when a descendent candidate

```
void init(Mesh  mesh)
  for each edge in mesh.edges
    edge.seen = false
  next edge
  for each edge in mesh.edges
    if edge.face is NULL and edge.seen is false then
      Ring ring = boundaryWalk(edge)
      PlaceInHierarchy(outermost,ring,NULL)
    end if
  next edge
End init
```

Figure 10: The complete initialization method.

ring is found, where the ring does not lie inside any of its children. In this case we add the ring as a child to that descendent candidate ring.

It might happen that the new ring does not lie inside any other candidate ring, but encloses it instead. In this case we must remove the candidate ring from its parent, and add the new ring as a child to the parent ring. Finally, the candidate ring must be added as a child to the new ring. The pseudo code is shown in Figure 11. A nice thing about the data structure is that we can update it in constant time when we do a "split-and-merge"- operation on two rings (explained in the next section).

6 Split and Merge Operation

If we know a ring that does not enclose any other rings then we can simply patch it with the algorithm outlined in Section 4. If there are other rings nested inside the ring then we are in trouble, because these nested rings describe "holes" in the ring "face", a complex topology which our mesh data structure can not deal with.

Our approach to the problem is to reduce the complex problem of handling the nested rings to the simple case of an empty ring.

We accomplish this by applying a split-and-merge operation for each nested ring. The split-and-merge operation is inspired by the algorithm we used for convex decomposition of a ring.

For each nested ring we search for a "diagonal"-line from a vertex on the nested ring to a vertex on the outer ring. The line is chosen such that it does not intersect the boundary of the outer ring nor the boundaries of any of the nested rings. If we find such a line we call it a split-line.

The split-line is used to cut the outer ring and inner ring topologically into open polylines starting and ending at the vertices of the split-line end points.

This is the splitting operation, next we merge the open outer ring and the open inner ring together to form a single closed ring. This is called the merge-operation. In Figure 12 we have illustrated the effect of the first split-and-merge operation carried out on the example from Figure 9. Observe that even though the resulting merged ring is geometrically touching along the split-line it is "topologically" separated. We have sort of tunneled our way from the outside of *A* to the inside of *B*.

In Figure 13 we have illustrated what happens during the second split-and-merge operation. Observe that the "tunneling" means that now the ring *D* has been promoted to the same level as the ring *ABC*. We have now reduced the complex nesting problem to two simple cases.

The pseudo code for the incremental divide and conquer approach we have explained is shown in Figure 14 and Figure 15.

```
void PlaceInHierarchy(list<Ring> level,
                      Ring ring,Ring parent)
  if level is empty then
    level.add(ring)
    return
  end if
  for each candidate ring in level do
    if ring is inside candidate then
      PlaceInHierarchy(
         candidate.children,ring,candidate
         )
      return
    end if
  next candidate
  bool enclosing  = false;
  for each candidate ring in level do
    if candidate is inside ring then
      ring.children.add(candidate)
      if parent<>NULL then
        parent.children.remove(candidate)
      else
        outermost.remove(candidate)
      end if
      enclosing = true
      break
    end if
  next candidate
  if enclosing then
    if parent<>NULL then
      parent.children.add(ring)
    else
      outermost.add(ring)
    end if
    return
  end if
  level.add(ring)
End PlaceInHiearchy
```

Figure 11: The place in hierarchy method.

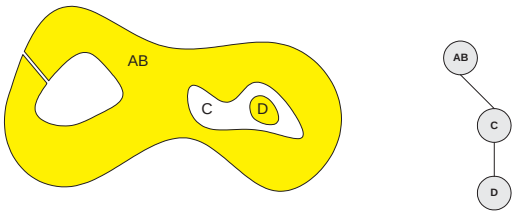

Figure 12: The rings of a cut plane after the first split-and-merge operation.

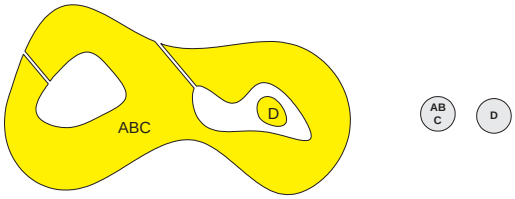

Figure 13: The rings of a cut plane after the second split-and-merge operation.

```
void run(Mesh  mesh)
  init(mesh)
  while not outermost is empty do
    Ring ring = outermost.front
    divideAndConquer(ring,mesh)
  end while
End run
```

Figure 14: The driver method of the algorithm.

```
void DivideAndConquer(Ring ring,Mesh mesh)
  if ring is empty and outermost then
    list<Ring> pieces
    RecursiveDecomp(ring,pieces)
    for each piece in pieces do
      mesh.createFace(piece.vertices)
    next piece
    outermost.remove(ring)
  else
    outermost.remove(ring)
    while not ring.children is empty do
      Ring inner = ring.children.first
      ring.children.remove(inner)
      outermost.insert(inner.children)
      inner.children.clear()
      ring = splitAndMerge(ring,inner)
    end while
    outermost.add(ring)
    divideAndConquer(ring,mesh)
  end if
End DivideAndConquer
```

Figure 15: The divide and conquer method.

We will now explain the details of finding a split-line, and performing the split-and-merge operation.

First, we search for the leftmost projected vertex, B, of the leftmost inner ring. Second we search for a projected vertex, A, of the outer ring lying to the left of B, such that the line, $\overline{AB}$, does not intersect the outer ring. Observe that the line would never intersect any of the inner rings, because we are always using the leftmost vertex of the leftmost inner ring.

If a line $\overline{AB}$ exist (we will give an existence proof later in this section) it will be a valid split-line, and we can therefore cut up the rings at A and B, and construct a new ring as follows: Add vertex A as the first vertex to the new ring, follow the vertices of the outer ring until vertex A is reached again, each time a vertex is encountered it is added to the new ring. By now vertex A should have been added twice to the new ring! Now the same procedure is repeated for the inner ring, starting at vertex B. Notice that also B will be added twice. The search for a valid split-line is dependent on a proper left-to-right ordering of the inner rings. This is easily accomplished by keeping the children of a ring as an ascending sorted list based on the x-coordinate of the projected leftmost vertex of the ring. In degenerate cases a descending sorting on the projected y-coordinates can be used.

If there is no vertex on the outer ring to the left of the leftmost vertex of the inner ring then the inner ring must touch the outer ring at a vertical edge. However due to our assumptions in Section 2 the inner ring can never touch the outer ring, so there must be a vertex on the outer ring that is to left of the leftmost vertex of the inner

```
Ring SplitAndMerge(Ring outer,Ring inner)
  let B be leftmost vertex of inner
  bool foundSplit = false
  for each vertex A in outer do
    if  A.x < B.x then
      if not line(A,B) intersect outer and inner then
        foundSplit = true
        break
      end if
    end if
  next A
  if not foundSplit then
    ... errror ...
    return
  end if
  Ring merged
  for vertex V=A to A do
    merged.add(V)
  next A
  for vertex V=B to B do
    merged.add(V)
  next B
  merged.children.add(outer.children)
  outer.children.clear()
  return merged
End SplitAndMerge
```

Figure 16: The Split and Merge Operation.

ring. This is illustrated in Figure 17. The split-line will always

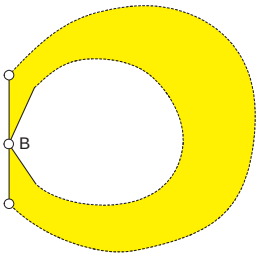

Figure 17: No vertex *A* to the left of *B* implies touching ring as shown. This is illegal. Hence there exists at least one vertex *A* to the left of *B*.

point to the left of the leftmost inner ring. Therefore it can not intersect the inner ring or any of the siblings of the inner ring, since these always lie on the right side of the leftmost inner ring.

Finally, there is the question if there exists a vertex on the outer ring such that the split-line does not intersect the outer ring. If there is only one vertex to the left of the leftmost vertex of the inner ring, then it is trivial true that the split-line does not intersect the outer ring. If there is more than one vertex and we have picked a vertex such that the split-line intersects the outer ring, then there must be a vertex lying to the right of the chosen vertex on the outer ring. This vertex will eliminate two intersections with the outer ring, and since we only have finitely many vertices on the outer ring it implies the existence of a vertex on the outer ring, which will create a valid split-line. The proof of split-line existence is illustrated in Figure 18.

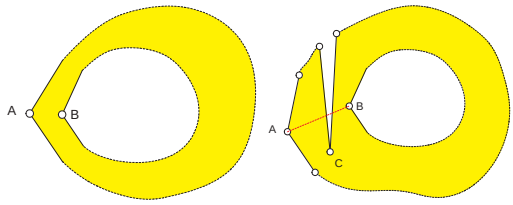

Figure 18: The left picture illustrates the trivial case of exactly one vertex *A* to the left of *B*. The right picture illustrates the case of an invalid split-line. The line *A* to *B* intersect the outer ring, however there must be a vertex *C* to the right of *A*, eliminating the two intersections.

7 Results

In Figure 19 we have shown how the algorithm have patched several meshes clipped by a plane. For visualization purpose we only visualize one half of the clipped mesh. The knot case shows that we can deal with multiple outermost rings and quite large rings (rings consist of little more than 64 vertices). The torus case shows that a nested ring does not pose any problem, the box with holes case demonstrates that nested rings in several layers can be handled. Finally the last case illustrates that multiple nested rings is also handled.

Mesh statistics and time measurements of the plane clipping and the plane patching algorithms are listed in Table 1. All measurements were done on a Dell Inspiron 8100, 933 MHz, 256 MB, W2K, algorithms were implemented in MSVC7.1.

We also tried the plane patcher algorithm in a convex decomposition based on half angle cutting of reflex edges, some decomposition results are shown in Figure 20. In our experience the plane patcher algorithm works well. However, we have seen the algorithm fail due to numerical roundoff and inaccuracies from the plane clipping algorithm.

8 Conclusion

In this paper we have described a simple patching algorithm for planar clipping operations. We have outlined an implementation in pseudo code and given results of running the algorithm on several test-cases, which we believe clearly show that the algorithm solves the problem we have stated.

The algorithm has shown to be fairly easy to implement, most of our difficulties have been concerned with numerical inaccuracies in the 2D polygonal intersection testing.

In our opinion the algorithm clearly fulfill our wish for a simple and fast alternative to more advanced solutions to the planar patching problem.

We believe that spherical parameterization [Praun and Hoppe 2003] of open boundaries of a clipped twofold mesh, followed by a 2D constrained Delaunay Triangulation [Shewchuk 1996; Shewchuk 2002], would provide one with a strong tool for both planar patching and more general patching. Such a tool would be valuable in helping cleaning up digital mockup as a pre-processing step as well as patching gaps from various clipping operations.

These problems might at first hand seem trivial and unimportant. However, our experience tells us otherwise. Far too often we have to deal with repairing meshes obtained from segmentation of medical images, or simply generated from 3D modeling applications such as 3D Max and the like. This is very tedious and time-consuming, and really not something we want to do as researchers.

A future goal is to implement the algorithm based on constrained Delaunay Triangulation as described above and compare it in terms

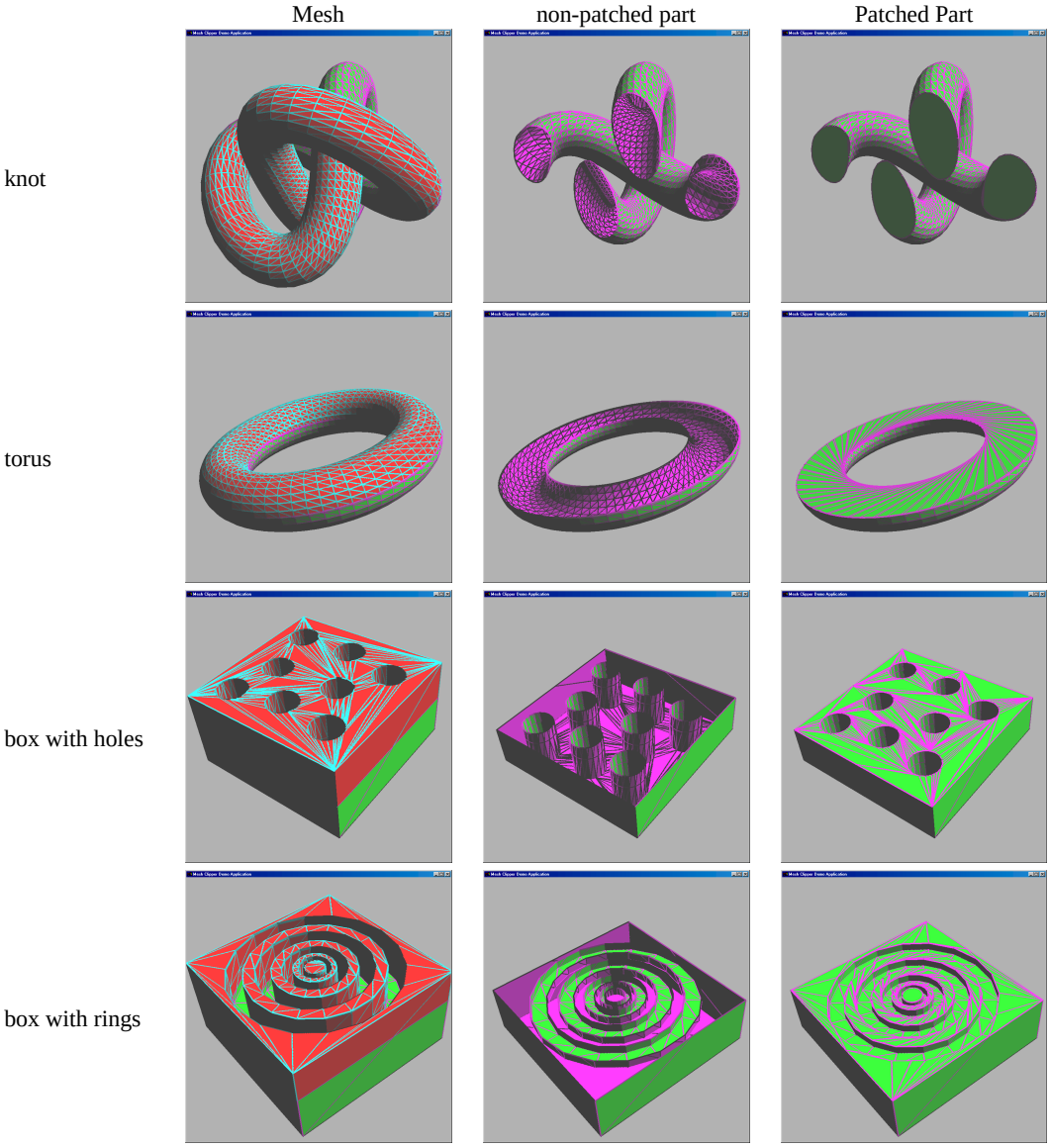

Figure 19: Screen-dumps of meshes before plane clip, after plane clip and after plane patching.

	#V	#E	#F	Clip (secs)	Patch (secs)	#R
knot	2880	8640	5760	0.551	0.03	8
torus	1536	4608	3072	0.22	0.17	4
box with holes	685	2097	1398	0.09	0.08	18
box with rings	585	1749	1166	0.09	0.051	18

Table 1: Statistics of meshes and timings of clipping and patching. #V: Number of Vertices, #E: Number of Edges, #F: Number of Faces, and #R: Number of Rings.

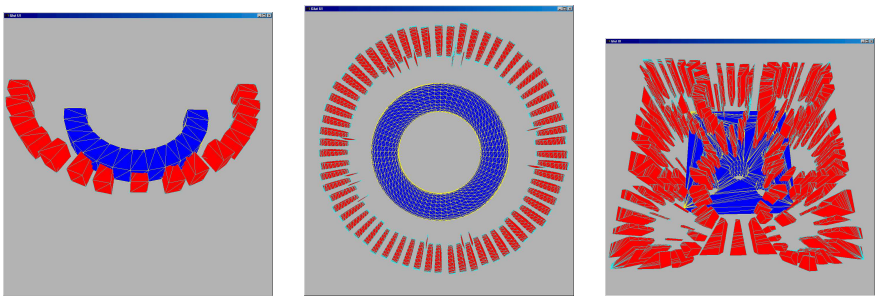

Figure 20: Screen-dumps of Convex Decompositions. The blue objects show the original meshes before decomposition, the red pieces shows the decomposition with pieces exploded outwards for visualization purpose.

of performance, versatility and user-ability to the simple algorithm presented in this paper.

References

BÆRENTZEN, J. A. 2001. *Manipulation of Volumetric Solids, with application to sculpting*. PhD thesis, IMM, Technical University of Denmark. BMP 08-0011-311.

COUTINHO, M. G. 2001. *Dynamic Simulations of Multibody Systems*. Springer-Verlag.

DINGLIANA, J., AND O'SULLIVAN, C. 2000. Graceful degradation of collision handling in physically based animation. *Computer Graphics Forum 19*, 3.

EBERLY, D. Clipping a mesh against a plane. http://www.magic-software.com.

FRISKEN, S. F., PERRY, R. N., ROCKWOOD, A. P., AND JONES, T. R. 2000. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Siggraph 2000, Computer Graphics Proceedings*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., 249–254.

HOFFMANN, C. M. 1989. *Geometric and Solid Modeling*. Morgan Kaufmann.

I.J.PALMER, AND R.L.GRIMSDALE. 1995. Collision detection for animation using sphere-trees. *Computer Graphics Forum 14*, 2, 105–116.

JAMES D. FOLEY, ANDRIES VAN DAM, S. K. F., AND HUGHES, J. F. 1996. *Computer Graphics: Principles and Pratice*, 2nd ed. in c ed. Addison-Wesley.

KRISHNAN, S., NARKHEDE, A., AND MANOCHA, D., 1995. Boole: A system to compute boolean combinations of sculptured solids. online paper. http://www.cs.unc.edu/ geom/CSG/boole.html.

MANTYLA, M. 1988. *An Introduction to Solid Modeling*. Computer Science Press.

MUSETH, K., BREEN, D. E., WHITAKER, R. T., AND BARR, A. H. 2002. Level set surface editing operators. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, 330–338.

NAYLOR, B. 1992. Interactive solid geometry via partitioning trees. In *Procedings of Graphics Interface*, 11–18.

O'ROURKE, J. 1998. *Computational Geometry in C*, 2nd ed. ed. Cambridge University Press. http://cs.smith.edu/ orourke/.

O'SULLIVAN, C., AND DINGLIANA, J. 1999. Real-time collision detection and response using sphere-trees. *15th Spring Conference on Computer Graphics*, 83–92.

PRAUN, E., AND HOPPE, H. 2003. Spherical parametrization and remeshing. *ACM Transactions on Graphics (TOG) 22*, 3, 340–349.

REQUICHA, A. A. G., AND ROSSIGNAC, J. R. 1992. Solid modelling and beyond. *IEEE Computer Graphics and Applications* (September), 31–44.

REQUICHA, A. A. G., AND VOELCKER, H. B. 1985. Boolean operations in solid modeling: Boundary evaluation and merging algorithms. *Procedings of the IEEE 73*, 1.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, M. C. Lin and D. Manocha, Eds., vol. 1148 of *Lecture Notes in Computer Science*. Springer-Verlag, May, 203–222. From the First ACM Workshop on Applied Computational Geometry.

SHEWCHUK, J. R. 2002. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications 22*, 1-3 (May), 21–74.

STEWART, N., LEACH, G., AND JOHN, S. 2002. Linear-time CSG rendering of intersected convex objects. *The 10-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision '2002 - WSCG 2002 II* (Feb), 437–444.

TZAFESTAS, C., AND COIFFET, P. 1996. Real-time collision detection using spherical octrees : Vr application. *IEEE Int. Work. on Robot and Human Communication*.

ZACHMANN, G. 1995. The boxtree: Exact and fast collision detection of arbitrary polyhedra sive. *First Workshop on Simulation and Interaction in Virtual Environments, University of Iowa*.