Real Time Voronoi Fracturing of Polygon Meshes

TSBK03 - Techniques for Advanced Computer Games

Daniel Camarda Rasmus Haapaoja Fredrik Johnson

Thursday 7th January, 2016

Abstract

Destruction of 3D assets, both in computer games and the visual effects industry is a non-trivial problem. Computer games often use different sets of the same mesh which are swapped upon destruction, and hence the results tend to be unconvincing. In this paper, we present an implementation of real time destruction of polygon meshes using 3D voronoi patterns, which allows for more interesting and realistic looking destruction of 3D environments. The implementation allows for interactive placement of voronoi centroids as well as the use of predefined patterns upon fracturing.

1 Introduction

Destruction of buildings and other 3D assets are a common feature in almost every modern computer game which adds another dimension of realism to the gaming experience. However, the game industry often uses predefined fractured assets, in order to achieve real-time performance. This puts an immense amount of workload on the artists since they need to provide a hierarchy of fracture levels in order to allow for repeated destruction. Pre-fracturing might destroy the illusion of the object actually being destroyed since there is no way to control the position and magnitude of the fracturing during run-time. Instead, the destroyed geometry is often covered by a smoke effect or something similar, when one asset is replaced with another.

In order to solve the problem of requiring several levels of pre-defined fractured assets and to lower the workload of the artists, there are at least two methods that show great promise. One of them is to represent each object to be fractured as a signed distance volume, hence making collision detection and the fracturing process fairly simple. A collision is detected by evaluation of the signed distance functions of the two objects that might collide. Where a positive value would correspond to the outside of the object, a negative value would correspond to the inside of the object and a value of zero is considered as the object surface. A volume representation also allows for simple creation of more interesting fracture patterns. By applying difference operators on impact the object might split into one concave and one convex part, instead of a completely flat surface which is the case with the approach that is being covered in this paper. While the advantages listed here seems very promising, the volume approach also suffers from several drawbacks. The major ones being that an object mesh needs to be converted to a volume representation before the proposed operations can be applied, then converted back into a polygon mesh. This makes it difficult to keep track of material properties and might also cause changes to the original surface, since the conversion to a volume and back again often lacks in precision.

The method of choice for our implementation is using 3D voronoi patterns to define splitting planes which allows for fracturing of polygon meshes in a somewhat straightforward way. The method does not suffer from the same precision issues as the volumetric approach while still performing well enough for a real-time implementation.

2 Method

The algorithm used within this implementation is largely based off of [4]. The project is written in C++, OpenGL and Bullet which were elected for use due to their popularity within the computer graphics field and their familiarity among the project group. Bullet Physics was chosen over a physics engine written in-house due to the time constraints and its compatibility with C++ and OpenGL. Bullet also has a relatively simple learning curve associated with it. The data structure which was chosen is a half-edge mesh due to its familiarity and benefits over other data structures which will be explained further. These concepts and resources will be expounded upon within the subsections to follow.

2.1 Mesh Data Structures

When working with 3D computer graphics, objects that are drawn on screen are most commonly represented by triangular polygons, known as faces. Where each face consists of three vertices, i.e. points in 3D space. A data structure for this type of representation, known as the independent face list, is often represented as:

Listing 1: Independent face list.

```
struct Face{
  Vertex v1, v2, v3;
};
struct Mesh{
  Face faces[F];
};
```

In an independent face list, every face stores its corresponding vertices and `F` denotes the number of faces. This is, however, a crude way of storing the required data, since many vertices will be stored multiple times and thus occupy unnecessary memory.

In order to get rid of this vertex redundancy, each face can store pointers to the corresponding vertices. This structure is known as the indexed face list and is represented as:

Listing 2: Indexed face set.

```
struct Face{
  Vertex *v1, *v2, *v3;
};
struct Mesh{
  Vertex verts[V];
  Face faces[F];
};
```

`V` and `F` represent the number of vertices and faces, respectively. This structure is considered the lower limit for data structures with random access to individual triangles, and often allows the memory consumption to be almost halved compared to the independent face list.

The two mentioned data structures perform well when it comes to linear traversal through triangles, when rendering for example. However, if access to neighboring faces of a vertex is required, it becomes more complicated. For the independent and indexed face lists, this operation requires a search through the entire face list to find the ones containing the current vertex. This corresponds to $\mathcal{O}(\texttt{F})$ operations, which is linear time. Consider that for every vertex in the mesh instead, it suddenly becomes $\mathcal{O}(\texttt{VF})$, which is quadratic in time complexity. Since the implementation requires fast neighborhood access for computations of differentials and triangulation, the more refined half-edge data structure has been implemented.

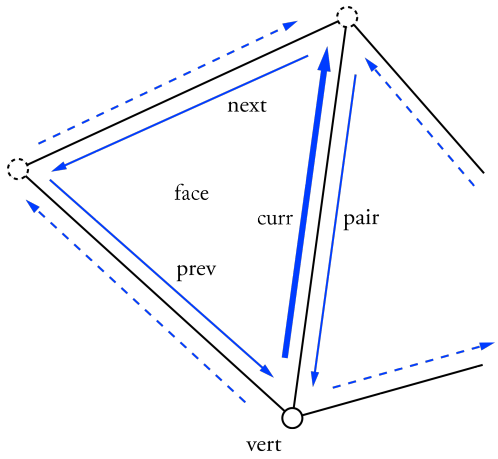


Figure 1: The half-edge data structure as seen from the bold half-edge. The filled lines correspond to explicit information while dashed lines correspond to implicit information which can be accessed through the pairs.

2.1.1 Half-Edge Data Structure

The mesh structures considered previously store information about vertices and faces. In order to gain more efficient access to the neighborhood of each vertex, information of edges might also be added to the mesh. The most common way of doing this, is by using the half-edge data structure [3]. An edge is defined as the line between two vertices, and thus by splitting this edge along its length, two half-edges are obtained, which point in opposite directions. The half-edge marked as `curr` (bold blue), in figure 1, only stores explicit information of the face marked as `face`. Information of the neighboring right face is accessed through the half-edge marked as `pair`. The entire neighborhood can be accessed via calls to `next->pair` and `prev->pair`.

Listing 3 shows the complete half-edge structure, where each `Vertex` is assigned a half-edge pointer `edge`. Note that there are several possible half-edges connected to a vertex, however it does not matter which one is chosen. Given a vertex, it is possible to access information about the neighborhood through this half-edge. Similarly, each `Face` is also assigned a

half-edge pointer `edge`, which can be any of the inner half-edges of the face (solid blue in figure 1).

Listing 3: Half-edge data structure.

```
struct Face;
struct Vertex;

struct Halfedge{
  Vertex* vert;
  Halfedge* next;
  Halfedge* prev;
  Halfedge* pair;
  Face* left;
};

struct Vertex{
  float x,y,z;
  Halfedge* edge;
};

struct Face{
  Halfedge* edge;
};

struct Mesh{
  Vertex verts[V];
  Face faces[F];
  Halfedge edges[3F];
};
```

Furthermore, this data structure is limited to manifold meshes, which is not sufficient enough for the implementation since the traversal of borders of a non-closed mesh is needed. In order to represent non-manifold meshes with borders, storage of empty faces is required and they are flagged as `BORDER`. It is also required to point the border-edges in a slightly different way to be able to traverse over the border. This is done by performing two merges, first around the vertex denoted as `vert` in figure 2 and finally around the vertex of the `next` half-edge. In the first case, start by repeated calls of `prev->pair` until a face flagged as `BORDER` or `UNINITIALIZED`[1] is encountered. Then the `prev` pointer of this half-edge is connected to the `pair` pointer of `curr`. Then the `next` pointer of `curr->pair` is connected to the encountered half-edge. The second merge is performed in the same manner. Start by traversing around the vertex accessed from `curr->next` by repeated calls of `next->pair`, until a face flagged as `BORDER` or `UNINITIALIZED` is encountered. Then the `next` pointer of this half-edge is connected to `curr->pair`, and the `prev` pointer of `curr->pair` to the encountered half-edge. The boundary edges are now connected in a way that allows for traversal around the borders of an open mesh.

[1] An edge flagged as `UNINITIALIZED` denotes a face that has not yet been added to the data structure.

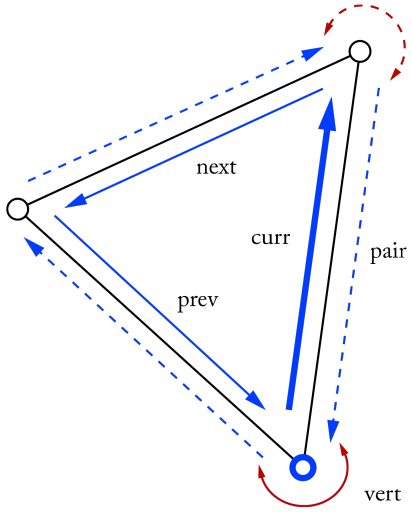


Figure 2: Merging of boundary edge (`pair`) to `curr`. Dashed edges are border-edges or not initialized.

Normals

The half-edge structure allows for fast computation of geometric differentials since simple access of neighbourhood data is supported. The simplest of the geometric differentials is the normal vector. Given counterclockwise orientation of vertices in a face, when viewed from the outside of the manifold, a face normal can be constructed as the cross product of $(\mathbf{v}_2 - \mathbf{v}_1)$ and $(\mathbf{v}_3 - \mathbf{v}_1)$. This yields a vector with the desired properties; perpendicular to the plane spanned by the three vertices of the face and pointing outwards.

$$\mathbf{n} = (\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1) \tag{1}$$

This allows for flat shading of the surface, which is enough for this implementation since every single triangle should be distinguishable for debugging purposes.

Computation of face normals can be performed efficiently even with a more crude data structure. However, if a smoother surface is desired and computation of vertex normals are required, then the half-edge

structure serves its purpose well. Since a vertex normal is computed as the normalized sum of adjacent face normals, efficient access of neighborhood data is crucial. The normal for vertex $\mathbf{v}_i$ is computed as:

$$\mathbf{n}_{v_i} = \widehat{\sum_{j \in N_1(i)}^{n} \mathbf{n}_{f_j}} \tag{2}$$

Where $N_1(i)$ is the 1-ring neighborhood (all faces sharing vertex $\mathbf{v}_i$) and $\mathbf{n}_{f_j}$ is the j:th face normal of the 1-ring neighborhood.

Volume

A slightly more complicated differential to compute is the volume of a manifold mesh. This differential was implemented in order to determine the new mass of a fractured object, as well as for debugging purposes. Since topological information is stored in our data structure, Gauss' theorem can be applied to relate the volume and surface integrals:

$$\int_S \mathbf{F} \cdot \mathbf{n} \, dA = \int_V \nabla \cdot \mathbf{F} \, d\tau \tag{3}$$

The theorem states that the surface integral of a vector field $\mathbf{F}$ times the unit normal $\mathbf{n}$ equals the volume integral of the divergence of the vector field. If $\mathbf{F}$ is assumed to have a constant divergence, $\nabla \cdot \mathbf{F} = c$, and the vector field is chosen as, $\mathbf{F} = (x, y, z)$, the divergence can be computed as:

$$\begin{aligned} \nabla \cdot \bar{\mathbf{F}} &= \\ &= \nabla \cdot (x, y, z) \\ &= \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right) \cdot (x, y, z) \\ &= \frac{\partial x}{\partial x} + \frac{\partial y}{\partial y} + \frac{\partial z}{\partial z} \\ &= 3 \end{aligned} \tag{4}$$

This means that the related surface integral will compute $3V$, since $c = 3$. Finally the integral can be approximated by a Riemann sum over each face:

$$V = \frac{1}{3} \sum_{i \in S} \frac{(\mathbf{v}_1 + \mathbf{v}_2 + \mathbf{v}_3)_{f_i}}{3} \cdot \mathbf{n}(f_i) A(f_i) \tag{5}$$

where $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$ are the vertices of face i, $\mathbf{n}(f_i)$ and $A(f_i)$ corresponds to the normal and area of the i:th face, respectively. The area is computed as half the magnitude of the cross product of any two edges in the face:

$$A(f_i) = \frac{1}{2} \left| (\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1) \right| \tag{6}$$

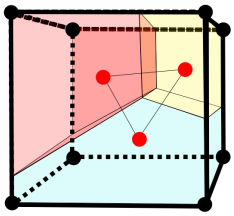


Figure 3: A voronoi diagram in three dimensions, which consists of several convex shapes

2.2 Voronoi diagram

The voronoi diagram is a crucial part of the implementation which decides how a certain mesh should be split up. It is derived from a set of centroids placed in 3D space, each point in the space belongs to the centroid which it has the shortest euclidean distance to, equation 7, where x, y, z specifies a point in 3D space.

$$f(x, y, z) = \sqrt{x^2 + y^2 + z^2} \tag{7}$$

In three dimensions, the computed voronoi diagram consists of several convex shapes, figure 3. The implementation allows a user to interactively place voronoi nodes where it is suited. This improves the freedom of choice an artist has while working with meshes. When the centroids have been placed, the voronoi diagram is calculated and applied to the mesh. The boundaries of the diagram are limited by the other nodes in the space and the bounding box of the mesh.

In figure 4, a basic voronoi diagram which consists of three centroids has been applied to a Stanford bunny. It is possible to increase the amount of nodes until the diagram fulfills the needs for a specific mesh.

2.3 Mesh Clipping

When fracturing a polygon mesh, one of the major challenges is how to divide the original mesh into

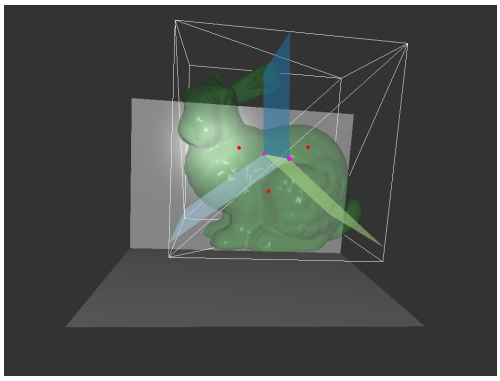


Figure 4: Basic voronoi Diagram applied to the Stanford bunny using three nodes. The red spheres represent a centroid in the diagram

several fractured pieces. By using the volumetric approach this step can be performed in a very efficient and trivial manner. Simply apply the difference operator and then triangulate the volume using e.g. the marching cubes algorithm presented by Lorensen and Harvey [2].

The mesh based approach used in our implementation is, however, slightly more complicated. An outline of the clipping process is presented in algorithm 1.

Algorithm 1 Mesh clipping

```
create new half-edge mesh
for all planes ∈ centroid do
  get point ∈ plane
  compute plane normal
  for all faces ∈ mesh do
    compute face normal
    if face intersects plane then
      clip face
      if num verts > 3 then
        triangulate
      end if
    end if
  end for
  create polygon from border vertices
  triangulate
  for all new faces do
    add face → half-edge mesh
  end for
end for
```

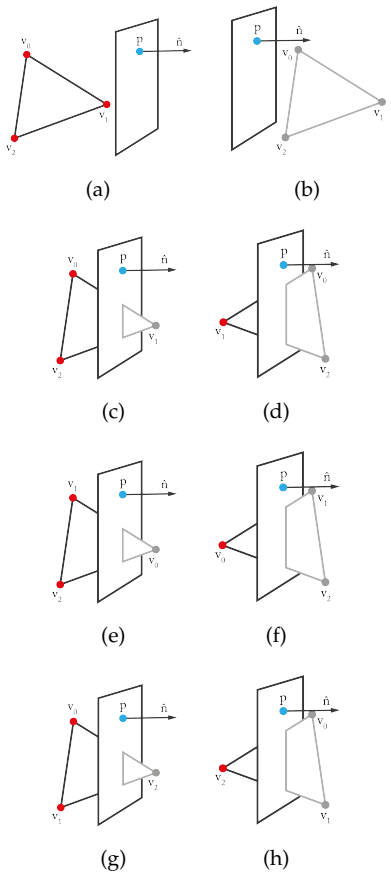


Figure 5: Main figure caption

Since each splitting plane stores two voronoi centroids from which it was created, this step starts by determining if the current plane contains the voronoi centroid that is being evaluated. This control is performed in order to determine which planes the mesh should actually be clipped against. The normal of the plane is then determined, with a direction pointing away from the current voronoi centroid, along with the point located between the centroids that is stored in the plane. The necessary information required for clipping a triangle against a plane, i.e. the plane normal and a point on the plane, has now been obtained. Each face is then tested for intersections against the current plane. When performing this test, eight different cases need to be considered, as illustrated in figure 5. In order to evaluate the resulting case of a triangle, consider the equation of a plane:

$$Ax + By + Cz + D = 0 \tag{8}$$

where (A, B, C) is the normal. D is determined by substituting the known point $\mathbf{P}$ on the plane into the plane equation:

$$D = -(A\mathbf{P}_x + B\mathbf{P}_y + C\mathbf{P}_z) \tag{9}$$

For a vertex $\mathbf{v} = (v_x, v_y, v_z)$ the following expression:

$$side(\mathbf{v}) = A\mathbf{v}_x + B\mathbf{v}_y + C\mathbf{v}_z + D \tag{10}$$

allows which side of the plane the vertex lies on to be determined. A positive value corresponds to the same side as the normal, a negative value corresponds to the other side and a value of zero is on the plane. By determining which vertices lie on which side of the plane, which edges intersect the splitting plane become known. In order to compute the intersection point on an edge intersecting the plane, the vertices creating this edge are, in this case, denoted as $\mathbf{v}_0$ and $\mathbf{v}_1$. The new vertex is then computed as:

$$\mathbf{v}_{new} = \mathbf{v}_0 + u(\mathbf{v}_1 - \mathbf{v}_0) \tag{11}$$

where u lies between 0 and 1. This expression is then substituted into the equation of the plane:

$$\begin{aligned} A(\mathbf{v}_{0_x} + u(\mathbf{v}_{1_x} - \mathbf{v}_{0_x})) \quad &+ \quad B(\mathbf{v}_{0_y} + u(\mathbf{v}_{1_y} - \mathbf{v}_{0_y})) + \\ C(\mathbf{v}_{0_z} + u(\mathbf{v}_{1_z} - \mathbf{v}_{0_z})) \quad &+ \quad D = 0 \end{aligned} \tag{12}$$

and solve for u:

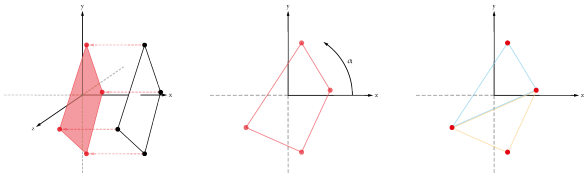
$$u = -\frac{side(\mathbf{v_0})}{side(\mathbf{v}_1) - side(\mathbf{v}_0)} \tag{13}$$

Substituting this back into equation 11 results in the intersection point where a new vertex should be added.

When a polygon is clipped, it might consist of three or four vertices. In the case of three vertices, the polygon is simply added to the list of new faces. However, in the case of four vertices, the new polygon needs to be triangulated into two separate faces. In order to perform this operation in a robust manner, the vertices of the newly created polygon are sorted counterclockwise. This is done by first projecting the polygon onto the plane which has the least amount of variation among the vertices in the polygon. E.g. if all vertices share the same x-value, the polygon is projected onto the plane spanned by the y and z-axis, as illustrated in figure 6(a). The angle of each vertex $\mathbf{v}_i$ is then computed as:

$$\alpha = arctan(\mathbf{v}_{i_y}, \mathbf{v}_{i_z}) \tag{14}$$

Note that the y and z-components of vertex $\mathbf{v}_i$ are used due to the choice of projection plane. If another projection plane is chosen, these components will also change. The vertices are then sorted according to figure 6(b) and triangulated according to figure 6(c).



(a) (b) (c)

Figure 6: Triangulation process of a polygon consisting of 4 vertices. (a) Projection step, (b) vertices sorted by angle, (c) triangulation step.

Once all faces from the original mesh have been clipped, the result is a a shell that is open along the splitting plane. In order to close this hole, the same sorting method presented above could be used. However, this would only be sufficient enough for convex polygons (figure 7(a)) since a polygon with some concavity could produce an overlap (figure 7(b)) which the previous method cannot handle. This is where the half-edge structure gets to shine. Since neighborhood information is stored in memory, any vertex on the border can be chosen as the starting point, in this case the one marked as blue in figure 7(c). By using the half-edge connected to this vertex, repeated calls of `prev->pair` can be used to find the half-edge with a face flagged as `BORDER`. This means that the half-edge marked as bold blue in figure 7(c) has been found. The border can now be traversed in order by repeatedly using the `next` half-edge and fetch its corresponding vertex. By using this method the vertices are ensured to be along the border and are always traversed in the correct order without the need to consider if the polygon is convex or concave.

Once the method to traverse the border is both efficient and stable, the polygons need to be triangulated in a robust way without any overlap. For this purpose, the ear clipping method is used which was proposed by ElGindy et al. [1]. Considering figure 8(a), vertex $\mathbf{v}_0$ is chosen as the starting point and is tested for the possibility to create a valid triangle consisting of $\mathbf{v}_0$, $\mathbf{v}_1$ and $\mathbf{v}_2$. In order to determine this, two criteria have to be fulfilled. Start by ensuring there is no other

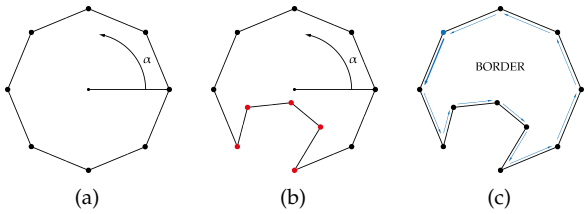


Figure 7: Ordering and traversal of border created from clipping. (a) Convex polygon sorted by angle, (b) concave sorted by angle, red vertices denotes vertices sorted in incorrect order, (c) border traversal with half-edges.

vertex located inside of the potential triangle. Then, verify that the angle β between the vectors $\mathbf{v}_1\mathbf{v}_0$ and $\mathbf{v}_1\mathbf{v}_2$ is smaller than π. If both of these criteria are fulfilled, a triangle can be created. Then simply remove the vertex $\mathbf{v}_1$ and repeat the process until only one triangle remains. The entire process is visualized in figure 8.

Once all polygons of the fractured part of the mesh have been triangulated, a new half-edge mesh is created and all new faces are added. The clipping process for a mesh containing one splitting plane is visualized in figure 9. The entire process is performed for all voronoi centroids applied upon fracturing. Thus creating one new half-edge mesh for each voronoi centroid.

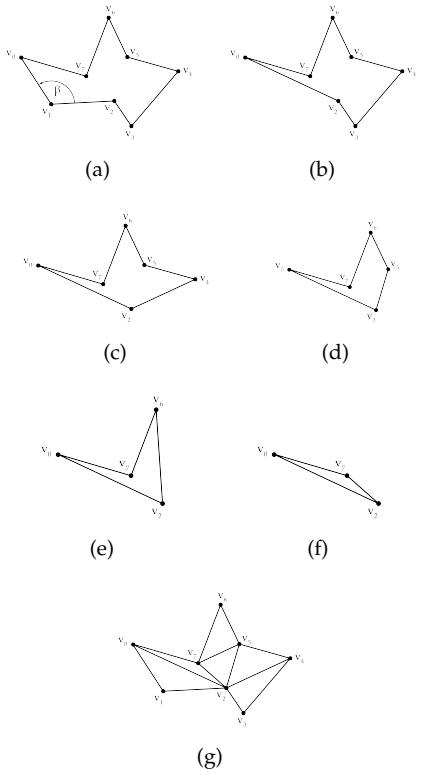


Figure 8: Ear clipping triangulation scheme. (a) control of angle β between edges $\mathbf{v}_1\mathbf{v}_0$ and $\mathbf{v}_1\mathbf{v}_2$, (b)-(f) repeated removal of valid vertices, (g) triangulated polygon.

2.4 Physics

Realistic physics is important regarding destruction of objects, especially collisions between objects. Writing a physics engine from scratch is not a trivial task and therefore an already thoroughly tested and established physics engine, Bullet Physics [5] was chosen for the implementation. Bullet allows for the creation of arbitrary convex collision shapes based on a mesh. This form of collision shape is needed since the shapes created by the voronoi pattern are complex convex shapes.

The half-edge mesh data structure provides more benefits when creating the physical representation of a mesh, the ease of calculating the volume of an object proves to be valuable since different sized objects should have different masses. The density is then calculated by using a modified version of the usual formula which assumes that the density (equation 15) is constant throughout the entire object. The mass for a newly created convex object can then be correctly set relative to its volume.

$$Density = \frac{Mass}{Volume} \tag{15}$$

The graphical world has no information on how its counter part in the physical world behaves. However, Bullet allows for the extraction of an objects world transform which can then be applied to the graphical counterpart of an object, figure 10, where the red lines represent the physics world.

3 Results

The three dimensional meshes are loaded in to the scene and the user sets out a limited number of voronoi points. Three points tends to give the most

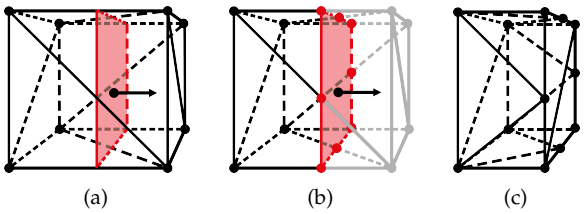


Figure 9: Main figure caption

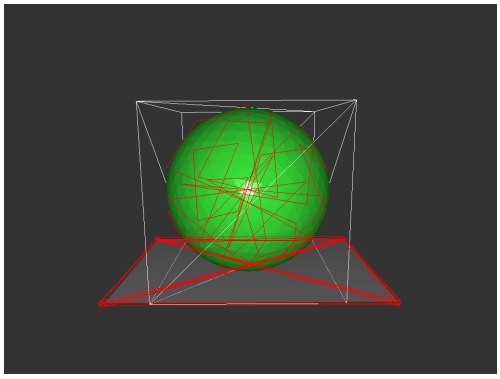


Figure 10: The red lines represent the physics world which the mesh tries to follow. Note: the physics debug drawer does not take into account the order of the vertices and is only used to see the boundaries of the convex hull

stable results but greater numbers have been successful. Those points are then used to calculate the convex shapes used within the mesh clipping step. If, for example, the convex shape includes more than one part of the original mesh (for example two legs below the knee), the clipping algorithm will incorrectly create a single new mesh containing these legs, figure 12. Erroneous triangles are then bridged between the two objects. Island detection, which will be discussed within the future work section, is a way of solving this.

The object starts to fall apart due to the effects of the physics on it, which at this stage is only a force of gravity acting upon the dynamic objects within the scene. This can be seen in figure 11. The collision detection and response are plausible but some difficulties arose when calculating the center of mass in some instances which results in abnormal rotations.

These new meshes are both watertight and influenced by physics. They are are not, however, able to be fractured further which was intended and is clearly an area of future work. As for the real time

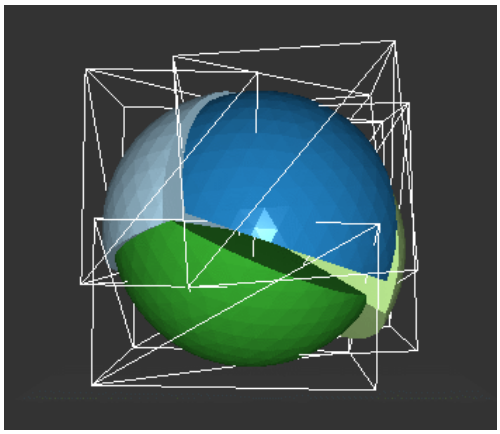


Figure 11: An icosphere falling apart

portion of the algorithm, the above stated substeps are carried out without any visible delay during rendering. The frames per second readout also confirms no noticeable change and thus the program can be considered real time.

4 Discussion & Future Work

We have presented a method for dynamic destruction of complex convex shapes in real time, where the main idea is to fracture a mesh along a voronoi diagram.

As of now, the implementation is limited to only destruction of an original mesh. A mesh that has been destroyed into several pieces cannot be destroyed again. However, since it is already possible to destroy a mesh with a predefined voronoi diagram, this feature should be easy to implement. It is a matter of having a voronoi diagram and applying scale, rotation and translation transformations to it so that it matches up with the boundaries of the newly created pieces.

The interactivity to place centroids for the voronoi diagram is a great tool for an artist that wants to control how an object should be destroyed. When working with multiple objects in a scene that need to be destroyed, however, this might not be a suitable approach. By automating the process of assigning a voronoi diagram to an object to one that is dependent on the mesh which then lets an artist modify it as required, is a very interesting topic. This would allow for a larger scene to be created quickly while

still allowing objects to be destroyed in a particular way.

A way to solve the problem apparent in figure 12, is to apply an island detection algorithm proposed by Müller et al. [4].

Define property
: Two pieces are physically connected if one piece has at least one face that (partially) overlaps a face of the other piece and the two faces lie in the same plane with opposite normals.

Create list
: A list which has entries for all faces of all the convexes of the compound is created.

List of links
: Each entry contains a link to the convex and the absolute value $|d|$ given by the plane equation $n \times x + d = 0$ of the face.

Sort
: Sort by the value of $|d|$, this will result in a list where entries with identical values (to a numerical epsilon), $|d|$, can be identified with a single pass through the list.

Find matching pair
: For each matching pair, check if the defined property is satisfied.

Seperate
: Seperate and create new compounds based on the result of matching pairs.

Replacing Bullet with a physics engine that we have written on our own would be preferable. Bullet is great to get a head start for a physics simulation with its "Hello World" tutorials, however when the project is more advanced and total control is needed, it is not optimal. As with any extensive library it has great features which are thoroughly tested but often comes at the expense of freedom and modifiability. Both freedom and access to core functions regarding physics transformations are required to transform the graphical representation correct.

An additional area of future work would include an optimization of the triangulation step during the mesh clipping. At the moment, the triangulation step often creates "thin" triangles which are undesirable from a graphics perspective. "Thin", or "skinny" triangles as they are sometimes referred to, can create problems within the performance of the simulation as well as numerical instabilities. As a general rule, when creating a triangulated surface, its best that the triangles are as close to equilateral as possible.

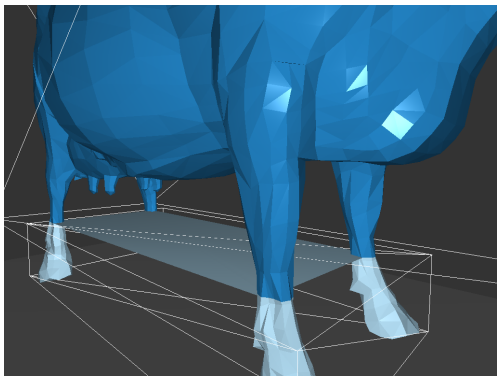


Figure 12: The cow has been split below the knees which results a single new mesh

References

[1] Hossam ElGindy, Hazel Everett, and Godfried Toussaint. Slicing an ear using prune-and-search. *Pattern Recogn. Lett.*, 14(9):719–722, September 1993.

[2] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA, 1987. ACM.

[3] Martti Mäntylä. *An introduction to solid modeling*. Computer Science Press, Inc., New York, NY, USA, 1987.

[4] Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Trans. Graph.*, 32(4):115:1–115:10, July 2013.

[5] Bullet Physics. Real-time physics simulation, 2015. Online; accessed 4-January-2016.