Real-Time Mesh Fracturing and Destruction using Voronoi Diagrams

Alice Anselmi || aanselmi@kth.se
Stefano Scolari || sscolari@kth.se[a]

[a]*KTH Royal Institute of Technology, Stockholm, Sweden*

Abstract

In recent years, the incorporation of destructibility in video games has captured the attention of both game developers and players. This feature not only enhances the immersive experience but also opens up new possibilities for gameplay mechanics and visual effects.
The objective of this project was to explore the computation of 3D Voronoi diagrams of meshes and their accompanying Delaunay tetrahedralization as a means of achieving realistic mesh destruction and fracturing in video games. The Voronoi diagram is a geometric representation that divides the space around each mesh vertex into regions, assigning each region to its closest vertex. By utilizing this spatial partitioning technique, we aimed to simulate the behavior of fractured objects in a visually realistic manner. We were able to implement a working approach that involved computing the Delaunay tetrahedralization followed by the Voronoi diagram of the mesh. This makes it possible to achieve a more realistic and visually appealing destruction effect.
However, it is important to note that while our approach yielded promising results in terms of visual impact and dynamic gameplay, it did not preserve the exact shape of the initial mesh. The process of mesh destruction inherently introduces alterations to the original geometry, as the initial tetrahedrization method of the mesh we implementented was not able to exactly match the meshes' initial shape.
In conclusion, the integration of 3D Voronoi diagrams and Delaunay tetrahedralization as a computational framework for mesh destruction and fracturing in video games has demonstrated its potential in enhancing realism and interactivity. Despite the inability to preserve the exact shape of the initial mesh, our approach offers a compelling solution for creating visually good and dynamic destructibility effects. Further research and refinement of these techniques will undoubtedly contribute to the continued advancement of interactive virtual worlds, bringing players closer to the boundaries of reality and imagination.

Keywords: Voronoi Diagrams, Destructibility, Computer Graphics, Geometry Processing

1. Introduction

Destructibility in games refers to the ability of the environment or objects within it to be damaged or destroyed. This feature has become increasingly common in modern video games, and it can serve several important purposes.
Firstly, destructibility can enhance realism and immersion by allowing players to interact with the game world more dynamically. Instead of being limited to pre-scripted interactions, players can break through walls, demolish buildings, and alter the terrain in real time, creating a more immersive experience.
Secondly, destructibility can be part of the gameplay itself, adding a tactical element to the gameplay by allowing players to use destruction strategically to gain an advantage over their opponents. Good examples of this are for example the Levolution mechanic in the *Battlefield* series, by *EA DICE*, where whole sections of the game environment get totally destroyed and altered, leading to new and interesting ways of playing the same map. *Control*, by *Remedy Entertainment*, is another more recent example where destruction is an integral part of the game.
Finally, destructibility can provide players with a satisfying sense of power and impact. Seeing the environment crumble and break apart in response to their actions can be a visceral and rewarding experience.

2. Delaunay tetrahedralization

We need to initially talk about Delaunay Tetrahedralization because a Voronoi Diagram (VD) and a Delaunay Tetrahedralization (DT) of the same structure are actually dual. This means that the knowledge of one implies the knowledge of the other one.
In general, it is easier to handle and manage triangles instead of arbitrary polygons, meaning that we first compute the DT of a structure and then simply extract the VD from the DT.
The Delaunay criterion states that for a tetrahedralization to be Delaunay, the circumsphere of each tetrahedron in the mesh must not contain any other input points. In other words, the circumsphere of each tetrahedron should be empty or "enclose" only the four vertices of that tetrahedron.
The construction of a DT starts with its initialization phase: the *big tetrahedron construction*.

2.1. Constructing a DT

There are three main paradigms for computing a Delaunay triangulation: divide-and-conquer, sweep plane, and incremental insertion. In two dimensions, all three paradigms yield optimal algorithms with a time complexity of $O(nlogn)$. However, in three dimensions, the divide-and-conquer algorithm has a worst-case time complexity of $O(n^3)$, although its implementation speed is comparable to incremental algorithms. Sweep algorithms exist for constructing the constrained Delaunay triangulation, but they are suboptimal in three dimensions. In R^3, only incremental insertion algorithms have a worst-case optimal time complexity of $O(n^2)$. These algorithms update the triangulation locally by inserting one point at a time, while other paradigms require rebuilding the entire triangulation. Incremental insertion algorithms are essential for building dynamic spatial models. Two algorithms commonly used for point insertion in R^3 are the Bowyer-Watson algorithm and a flip-based algorithm. The Bowyer-Watson algorithm deletes tetrahedra conflicting with the new point and fills the resulting hole, while the flip-based algorithm is discussed further in the paper as an alternative method.
In our case, the flip-based algorithm was implemented.

2.2. Initialisation

To work, the algorithm assumes that the set S of points is initially entirely contained in a big tetrahedron. This way, every point from S we want to insert, we know it will exist within a tetrahedron.

2.3. Flips

As previously mentioned, the algorithm implemented is *flip-based*. A bistellar flip is a topological local operation that modifies the configuration of some adjacent tetrahedra. Two main possible configurations exist:

1. all 5 points lie on the convex hull boundary. In this case there are two different ways to tetrahedralize such polyedron, with two or three tetrahedrons 1(a).
2. one point e lies inside the convex hull boundary, in this case the tetrahedron that contains the point is flipped into 4 tetrahedrons incident to the point e 1(b).

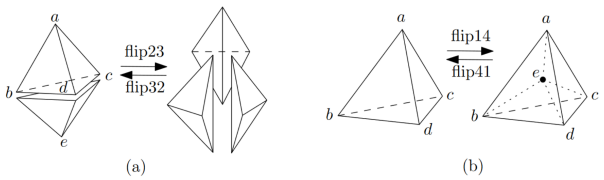

Figure 1: Bistellar flips cases: (a) case 1, (b) case 2.

Four different flips in R^3 can be described: *Flip23*, *Flip32*, *Flip14* and *Flip41* (where the numbers represent the number of tetrahedra before and after the flip).
We also needed to implement a flip44 to handle some degenerate cases.

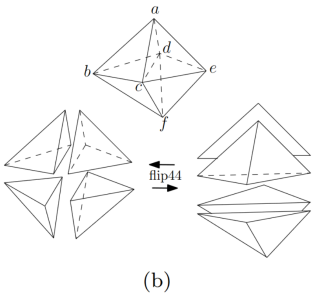

Figure 2: Degenerate case flip44.

3. Voronoi Diagrams

A VD is a geometric structure that divides space into regions (*Voronoi polygons)* based on the proximity to a set of input points. In the case of a 3D mesh, a Voronoi diagram can be constructed to partition the space around the vertices of a mesh. Each region is a *Voronoi Cell* and consists in a convex hull with an arbitrary number of vertices, edges and facets. The Voronoi Diagram of a mesh can be computed in many ways - even without going through DT first. However, in the 3-dimensional case the implementation can be problematic and bring to degeneracies. Thus, the easiest way is going through DT first.
The duality between the DT and the VD, given that our mesh is formed by a set S of point in $\mathbb{R}^3$, consists in the following:

- **Delauney Tetrahedron to Voronoi Vertex**: a Voronoi Vertex can be easily computed by finding the center of the sphere that passes through the vertices of the dual Delauney Tetrahedron. However, as we will mention later on, we found that using the barycenter instead of the circumcenter leads to better looking results.
- **Delauney Facet to Voronoi Edge**: every Delauney Tetrahedron has 4 triangular facets. Let's suppose that two tetrahedra, which have an equivalent Voronoi Vertex each, are sharing a facet. Then, a Voronoi Edge can be computed by connecting the two Voronoi Vertices that are dual to the two tetrahedra.
- **Delauney Edge to Voronoi Facet**: similarly to the previous point, let's assume that we have some Delauney Tetrahedra sharing a common edge. Then, a Voronoi Facet is formed by all the Voronoi Vertices that are dual to those tetrahedra that share an edge.
- **Delauney Vertex to Voronoi Cell**: a Delauney Vertex can be shared by multiple Delauney Tetrahedra which have a dual Voronoi Vertex each. Such Voronoi Vertices form the Voronoi Cell dual to the Delauney Vertex taken into account.

Starting from the DT, the corresponding VD can be obtain by first computing all the dual Voronoi Vertices. Then, following the last equivalence in the list, the Voronoi Vertices forming each cell can be found. The convex hull of the cell can then be evaluated through one of the many existing algorithms - we

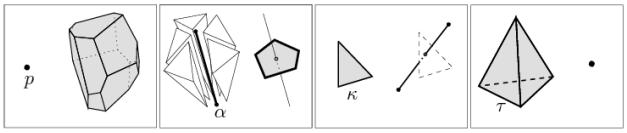

Figure 3: Duality between 3D Delauney Tetrahedralization and Voronoi Diagrams.

picked the *Quickhull* algorithm (5), which uses a divide and conquer approach.

4. 3D Incremental Flip-Based Algorithm overview

4.1. InsertOnePoint

`InsertOnePoint` is the algorithm used to restore a Delaunay state once a point p is inserted.

Algorithm 1 InsertOnePoint(T, p)

```
τ ← Walk (to obtain tetrahedron containing p)
insert p in τ with a flip14
push 4 new tetrahedra on stack
while stack is non-empty do
    τ ← {p, a, b, c}                          ▷ Pop from stack
    τ_a ← get adjacent tetrahedron of τ having abc as a facet
    if d is inside circumsphere of τ then
        Flip(τ, τ_a)
    end if
end while
```

4.2. Walk

In order to find out in which tetrahedron a point p is contained, we call `Walk`. This algorithm can be performed in several ways depending on the desired performances and complexity. *Walking in a triangulation*(4) gives an overview of some implementations of the algorithm; among them, we chose to apply the Remembering Stochastic Walk, which starts from a random point and makes its way through the group of tetrahedra according to an *orientation* test.

In the algorithm's pseudocode, T is the group of tetrahedra and p is the point of which we need to find the location. `Orient` is a function that, given a point p and a tetrahedron t with one of its facet f, tells whether p lies above, below or on the same plane as f compared to the center of t. We avoid writing the pseudocode for this, as it's a geometrical computation that can be performed in several ways.

Algorithm 2 Remembering Stochastic Walk(T, p)

```
t ← random tetrahedron from T
previous ← t
end ← false
while not end do
    for facet in t do
        isPointOnOtherSideOfFacet ← ORIENT(t, f, p)
        if (p not neighbour of previous through f) and
            (isPointOnOtherSideOfFacet) then
            previous ← t
            t ← neighbour of t through f
            continue          ▷ go back to start of while loop
        end if
    end for
    end ← true
end while
return t
```

4.3. Flip

The `Flip` algorithm is called if, once the Flip14 was performed and the point p inserted, the Delaunay Tetrahedralization is not Delaunay anymore. Once p in inserted and the flip14 is performed. This point p will have 4 incident tetrahedrons. For each of these tetrahedrons, if the flip produced a non-Delaunay state, we then need to check each one of these 4 new tetrahedrons with their neighbours sharing the only facet not already shared amongst the 4 new tetrahedrons obtained from the flip14.

Algorithm 3 Flip(τ, τ_a)

```
if case #1 then
    FLIP23(τ, τ_a)
    PUSH(tetrahedra pabd, pbcd, and pacd) on stack
else if case #2 and T_p has tetrahedron pdab then
    FLIP32(τ, τ_a, pdab)
    PUSH(pacd and pbcd) on stack
else if case #3 and τ and τ_a are in config44 with τ_b and τ_c
then
    FLIP44(τ, τ_a, τ_b, τ_c)
    PUSH(the 4 tetrahedra created) on stack
else if case #4 then
    FLIP23(τ, τ_a)
    PUSH(tetrahedra pabd, pbcd, and pacd) on stack
end if
```

Looking at point p, and remembering that *pabc* is one of the four tetrahedrons resulting from the *flip14*, and that *abcd* is the neighbouring tetrahedron (not part of the four new tetrahedron from the *flip14*) the four different cases that can be seen in 4 are the following:

- *Case* #1: Looking from p, only one face of *abcd* is visible, making the union of *pabc* and *abcd* a convex polygon. If this is the case, a flip23 is performed.
- *Case* #2: Looking from p, two faces of *abcd* are visible, making the union of *pabc* and *abcd* a non-convex polygon.

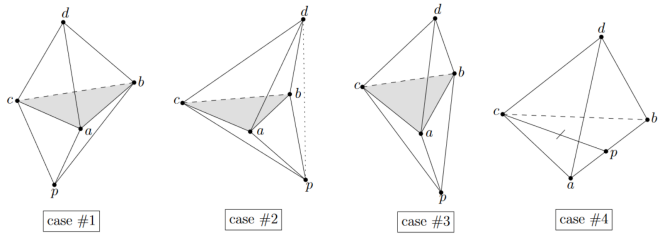

Figure 4: Flip algorithm 4 possible cases.

If the tetrahedron *abpd* already exists amongst the other tetrahedrons, then a flip32 is performed, otherwise no flip is performed.

- *Case* #3: This is the case if p and d are coplanar with any other of the vertex pairs forming the edges of the shared facet *abc*. If the four tetrahedrons are in *config44* 2 then a *flip44* is performed, nothing otherwise.
- *Case* #4: This is the case if p is both coplanar with two faces of the tetrahedron, meaning it is on one of the three edges of the shared facet. In this case, instead of performing a flip12 for such degenerate case, by performing first a flip32 followed by a flip23 we get the wanted result.

5. Project structure, usage instructions and results

The final result shows a 3D mesh (which is arbitrary, in our case we chose the default *monkey* mesh by *Blender*(1)) falling to the ground and tearing into pieces. Due to time constraints we didn't implement any lighting/shadowing system, therefore the textures don't look as good. We used the OpenGL graphics API to achieve such a result, along with many other libraries like the *Bullet Physics SDK* (2) for the physics engine, *Assimp*(3) for asset loading and an open-source implementation of the QuickHull algorithm.

We created a data structure for the Delauney tetrahedra and the Voronoi meshes respectively, so that we could store useful data to perform the algorithm. The code performs the following steps:

1. Mesh loading and allocation of corresponding rigidbody.
2. Computation of the *big tetrahedron*.
3. Iteration of `InsertOnePoint` for an arbitrary number of times.
4. Simulation can now start.
5. Conversion of Delauney Tetrahedra to Voronoi Meshes (in Real-Time when the collision between the mesh and the ground takes place).

Two parameters in our project can be tweaked according to the desired result and computational time: the number of points added with `InsertOnePoint` and the maximum volume of the generated Voronoi meshes. As a matter of fact, in order to achieve a realistic result a high number of points (somewhere around 1000) needs to be added, which requires a great amount of computational time. With less points ($\sim$ 100), bigger fragments are created which sometimes are too big to look realistic. For this reason, we filtered the resulting meshes based on their volume to obtain a more good-looking result.

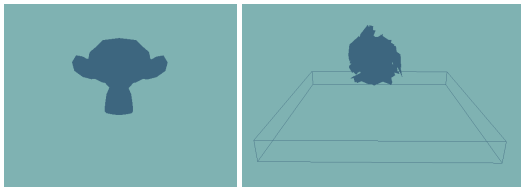

Figure 5: Initial *monkey* mesh and generated fragments at the moment of collision

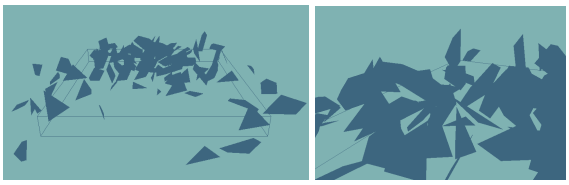

Figure 6: Fragments after collision

When running the project, the number of added points is displayed on *Command Line*. As of today, we know about a bug that could interrupt the execution of the insertion: if this situation occurs, build and run the project again. For the final version, we are adding 200 points, data that can be found in *main.cpp*. Once the mesh is loaded, the user can move the camera with `WASD` and accelerate with `Shift`. To add gravity to the simulation, the user needs to keep `X` pressed: if they release the key, the simulation will pause so that the user can look at the result more in detail.

6. Possible improvements

As mentioned in **Section 5**, due to time costraints we couldn't implement any lighting which would give the loaded mesh a more pleasant look, as well as display more clearly the rupture. Furthermore, when converting the mesh to a set of tetrahedrons, shape is not fully preserved. To account for this, a more fine-grained tetrahedralization can be used or a mesh-cutting algorithm implemented. This kind of algorithms would be useful to *re-shape* the mesh to it's original form, using its convex hull to guide the cutting.
Moreover, although we optimized the original algorithm (especially as far as `StochasticWalk` and the DT-VD conversion are concerned), it can take a few minutes to compute the tetrahedralization if the number of inserted points is in the thousands.
Many features can be implemented to expand the functionality of this project. For example, fracturing only sub-parts of a mesh or changing the fracturing based on the desired material.
To improve the project in the context of perceptual user studies, several enhancements can be implemented to upgrade the visual experience and user interaction. Here are some potential improvements informed by perceptual user studies. For instance,

conducting user studies to evaluate the impact of lighting models and materials on the perception of fractures and overall aesthetics would be beneficial. By implementing lighting techniques such as ambient, diffuse, and specular lighting, we could create more visually pleasing and realistic scenes. User studies could also focus on investigating the perception of fractures and rupture effects on the mesh, or on the effectiveness of reconstruction algorithms in preserving the original shape of the mesh after fracturing. By evaluating different mesh-cutting algorithms and tetrahedralization techniques, users could provide feedback on the accuracy and visual quality of the reconstructed mesh. This feedback could be used to refine and improve these algorithms.

7. Conclusion

Throughout the project, we faced a set of challenging problems, such as optimizing the performance of the Voronoi fracturing algorithm and ensuring seamless integration with OpenGL rendering and the various libraries used.
In conclusion, this project provided us with valuable insights and learning experiences: we explored OpenGL and what revolves around it, as well as discovering the potential of Voronoi Diagrams as a technique for generating realistic fractures of 3D meshes. We also greatly improved our knowledge, confidence and understanding of C++.

References

[1] Blender.
Accessible at: `https://www.blender.org/`.

[2] Bullet real-time physics simulation.
Accessible at: `https://github.com/bulletphysics/bullet3`.

[3] Open asset import library.
Accessible at: `http://assimp.org/`.

[4] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. In *Proceedings of the seventeenth annual symposium on Computational geometry*, pages 106–114, 2001.

[5] Wikipedia contributors. Quickhull — Wikipedia, the free encyclopedia.
Accessible at: `https://en.wikipedia.org/w/index.php?title=Quickhull&oldid=1151612817`, 2023.