Real-time Mesh Destruction System for a Video Game

Anton Grönberg

Högskoleingenjör, Datorspelsutveckling
2017

Luleå tekniska universitet
Institutionen för system- och rymdteknik

LULEÅ UNIVERSITY OF TECHNOLOGY,
SKELLEFTEÅ

BACHELOR'S THESIS

Real-time Mesh Destruction System for a Video Game

Author:
Anton GRÖNBERG

Supervisor:
Patrik HOLMLUND

June 18, 2017

Department of Computer Science, Electrical and Space Engineering

Abstract

Destructive environments in video games are a feature that can give a game more depth and realism by being able to change the landscape or raze a building. This report talks about the research and implementation of a dynamic destruction system for the video game Scrap Mechanic. The end results are a system that could split convex 3D meshes to smaller pieces in almost real-time. It was a somewhat stable implementation that needs some future work before it can be used in the game. Therefore, in this report, things that could be improved with the current implementation and how to use it are discussed.

Sammanfattning

Destruktiva miljöer i tv-spel är en funktion som kan ge ett spel mer djup och realism genom att kunna ändra lanskaped eller förstöra byggnader. I den här rapporten undersöks och implementeras ett dynamiskt förstörelse system för spelet Scrap Mechanic. Slutresultatet är ett system som kan dela upp konvexa 3D meshar till mindre bitar nära realtid. Det blev en någorlunda stabil implementation som kommer behöva lite framtida arbete innan den kan användas i spelet. Därför diskuteras även saker som skulle kunna förbättras med den nuvarande implementationen samt hur den kan användas.

Acknowledgements

I want to thank everyone that helped me during this project. A big thanks to my supervisor Patrik Holmlund who have read and given valuable feedback on all my drafts. Also, a big thanks to Axolot Games for allowing me to do my thesis work there. And last but not least, a thank you to Luleå University of Technology Campus Skellefteå for all support and help throughout my education.

Abbreviations and Terms

- **C++:** Object-oriented low-level programming language
- **DirectX:** Multimedia API for game development, created by Microsoft
- **Game Engine:** A software framework designed for creation and development of video games
- **Mesh:** 3D Mesh, a collection of vertices, edges and faces that defines the shape of a polyhedral object in 3D computer graphics
- **UV:** Texture coordinates for a mesh
- **Pointers:** C++ pointers that points to a place in the computers memory.
- **Debug mode:** Program configuration with unoptimized complied code that contains a lot of data which is used by developers. Runs much slower than Release mode.
- **Release mode:** Program configuration with optimized compiled code. Runs much faster then Debug mode.
- **Steam:** A online video game distribution service

Contents

Chapter 1

Introduction

Video games have over the last 30 years expanded from arcade halls and cabinets into a multi-million business [19] and where you can find games on almost every device [6]. From portable devices such as smartphones and hand-held consoles to powerful computers and home consoles, games can be seen everywhere and most people are playing them even if some don't classify themselves as gamers [12].

Games come in all shapes and sizes, they can have different genres, styles, budgets [9] and be created by an indie studio of just a few people or by a huge AAA [23] company with a few hundred employees but still, hold the same amount of joy and craftsmanship. Games are created to give the players an experience and to make people escape everyday life [20] for a while or to just kill a little bit of time. Since the player is in full control they also create their own experiences [15] with what the creators made for them. Games are also the only type of entertainment that actively are trying to keep their consumers from leaving. This is what makes games very unique compared to movies and books where the story is already written [15].

The possibility of creating your own adventure and experiences are a big part of what the creative open world game genre do best [4]. Where the creators give the tools to create something grand and spectacular to the players and let them roam freely. Games like Minecraft [16] and Scrap Mechanic [10] are examples of creative open world games, the players are given blocks and tools and are free to build and create whatever their mind can come up with. Huge houses with fully automatic features like automatic doors, elevators, huge harvest machines and escalators. Almost only your imagination sets the boundaries and this is the type of game Axolot Games [10] want to create. Scrap Mechanic is a game that heavily relies on physics, all the objects in the game have physics and different mechanical gadgets like ball bearings, engines and wheels to be able to build different mechanical inventions. These gadgets can be connected to each other seamlessly without the need of wires which makes it possible to create, for example, a house that can transform into a car and be driven. And since it relies so heavily on physics, Axolot want everything to feel like you actually are making an impact when you do something, they want resource gathering to feel like you are harvesting the resources. When you are cutting a tree it should fall off pieces from it and leave a hole in it and not still be a whole tree.

This report is based on a project done for Axolot Games with the goal to do a real-time fracturing system using user-defined Voronoi diagrams and a half-edge data structure to be able to break wood and stone into pieces that will be their own separate physics objects that can be moved and have an impact on the world around them.

1.1 Goal and Purpose

The goal of this project was to:

- Develop a system that can use a half-edge mesh
- Use a Voronoi pattern to fracture a half-edge mesh
- Convert the half-edge mesh to a renderable mesh
- Run the algorithms in real time

This will allow players to hit a mesh at any point and it will shatter in a more realistic way than a pre-fractured mesh would. By having a dynamic fracture system the artist only need to create a few fracture patterns instead of having to create pre-fractured models for each mesh that would be used for destruction. Also if the meshes would be changed the artist would have to redo the fracturing.

The purpose of this system is to make resource gathering in the game feel better and to give a more fun and realistic way of gathering, for example, stones and trees.

1.2 Limitations

The main limitation of this project were the limited time available. Due to the time constraints, there was a lot of things that couldn't be properly researched and implemented.

The original goal was to have the system be able to cut the meshes in real-time when the user interacted with a mesh. But this was a difficult task to do since there is a lot of things that really need to be considered and well designed. This was not completed in its entirety but a more simplified version was created that works just as it was planned though it is not fast enough for real-time usage.

1.3 Axolot Games

Axolot Games is a indie game studio located in Stockholm where they currently employ 9 people. It was founded in the start of 2016 after they released their game Scrap Mechanic as Early Access [27] on the game distribution platform Steam [26]. Scrap Mechanic gained a lot of popularity [28], mostly through the video platform YouTube where players would upload videos of them playing the game.

1.4 Background

Destruction in video games existed as early as 1979 when it was implemented in the Atari [2] game Asteroids [17], where you would shoot an asteroid and it would split into smaller pieces, but it wasn't a common practice. As computer graphics have progressed and computers have become more powerful, destruction effects have become more common in modern computer games. Effects such as exploding buildings, shattering glass and breaking objects add significantly to the immersion and feeling of the game, especially when added in combination with particles. Most modern games, to achieve real-time performance, pre-fracture their game assets so when the models are hit they switch them for the pre-fractured versions behind a smokescreen or explosion so the players won't really notice anything.

Some of the big challenges with creating a destruction system has a lot to do with the way computer graphics work. One problem is that 3D meshes are hollow shells instead of volumes which create the problem that, if you cut a mesh, it will just leave a piece of a shell. Therefore you need to add triangles to the mesh to fill the gap that is created. Another problem has to do with the way data are stored on the graphics card. A mesh can be stored in a vertex buffer object, VBO, and changing it dynamically is a expensive and time consuming task which can slow down the system.

The Voronoi diagram is a special kind of decomposition of a metric space and is named after the Russian-Ukranian mathematician Georgy Voronoy but it was considered as early as 1644 [13]. The diagram has a very natural pattern that can be found in different parts of nature [1]. It is used in a lot of different areas like metallurgy, astronomy and computer science. When used in computer science the Voronoi diagrams can easily be used as a dual graph and be converted back and forth between Delaunay triangulation and a Voronoi diagram using the same set of points [13]. Voronoi diagrams are useful for a destruction system because the cells of a diagram are convex pieces which are recommended when used for collision detection.

1.5 Related Work

Real-time destruction for a 3D environment has always been a bit tricky to implement so that it runs smooth and actually delivers somewhat realistic demolition since it can take up a lot of time and money. That's why most 3D games that have destruction are AAA titles [23].

The two most known AAA series that includes a good destruction system are the Red Faction series [11] created by Volition Inc and the Battlefield [8] series created by DICE. Volition Inc's destruction system was called GeoMod, short for Geometry Modification, and allowed the players to completely change the environment around them. Players could dig holes, go through any wall and demolish an entire building. DICE, on the other hand, use their own system simply called "Destruction". It is more restricted than the GeoMod system because they want a more controlled destructive environment so that the gameplay runs smooth and won't create bugs or graphical artefacts for players.

In 2013 Matthias Müller et al. [18] published a paper regarding destruction, and the new way they found to be able to destroy an entire scene into small pieces. They were able to destroy an object multiple times and they contributed with a new volumetric approximate convex decomposition algorithm to faster calculate the convex hulls needed to clip the mesh into smaller pieces.

Last year, 2016, Daniel Camarda et al.[5] wrote a paper about their method of real-time fracturing. They used the well known Half-edge data structure to easily walk through the meshes and stitch the holes together and used a 3D Voronoi diagram to cut the mesh into smaller pieces.

1.6 Method

The work process during this project was composed of a few phases, as can be seen in figure 1.1. The first phase was to learn about the subject and to create a design for the implementation. This involved reading papers, like the ones mentioned in section 1.5, and researching already existing code libraries to help with the toughest and most time-consuming parts of the implementation. When that was done a test environment, based on Axolot's game engine, was set up and the two external code libraries Voro++ [21] and OpenMesh [7] were implemented, these are explained in section 2.1.1 and 2.1.2 After that, a phase of testing and studying the external libraries began, to learn what the libraries could do and if they would be helpful to the final implementation.

When that was done it was time to start the implementation of the actual destruction system. It utilises a Half-edge mesh algorithm to easily split and rebuild the mesh and a 3D Voronoi diagram to create the mathematical planes that will be used to split the mesh. The destruction system converts a renderable mesh into a Half-edge mesh and runs a cutting algorithm on it to split it into smaller pieces. With each new vertex that is added per piece the UV coordinates are also interpolated to each vertex to try and keep texture intact over the mesh. Each of the pieces is then simplified with an edge collapse algorithm to decrease the number of unnecessary extra triangles that is created by the cutting algorithm. Then each piece is run through an ear clipping algorithm that will fill the open hole, created from the cutting algorithm, with triangles so that each convex piece will be water tight and not have any missing triangles or holes left in it. Then each piece is added to a list of meshes that is then converted back into renderable meshes.

When the implementation was done, the finalisation started. During the finalisation, a large number of tests was done and the code was optimised using the results of the tests to try and make destruction system run in real time.

1.6.1 Social, Ethical, and Environmental Considerations

This project is implemented on its own Github branch of Axolots game engine, this will limit the risk of corrupting or changing any of the core code for the

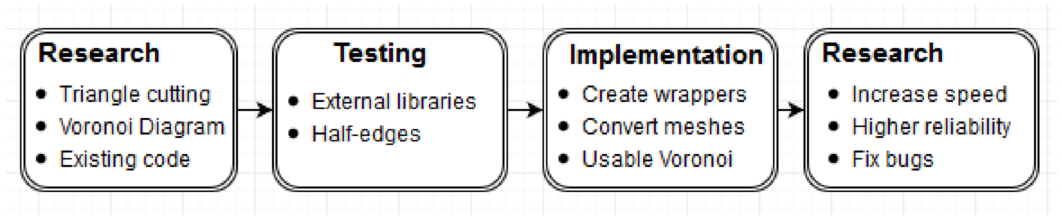

FIGURE 1.1: Phases during implementation

engine while doing tests and research. The implementation is designed to reduce the workload for the artists and to give a more realistic result in the destruction of objects.

From an environmental perspective, since the system is created for a game that is released on a digital platform it lessens the need to make physical copies. But because the game right now is quite a performance heavy consumers might need to upgrade their computers to be able to run the game. It can have a small environmental impact but usually, once a person has upgraded their computer they do not need to do it again for a few years.

The largest social and ethical impacts would be the same as with many other games. If the user gets addicted to the game it may have an impact on the person's personal life, like lack of sleep or nutrition. But that is nothing that really can be avoided except by having a warning in the game that notifies the user if they have been playing for too long.

Chapter 2

Design and Implementation

The general design of the implementation was to create a destruction system that would work coherently with Axolot's own game engine. It needed to be a moderately isolated system that would work with the engine's render system and to be written in the programming language C++, the same as the engine. The use of a Voronoi diagram was Axolot's wish right from the start. This was because Voronoi diagrams have a very natural pattern and it was going to be used on natural objects like trees and stones. The future goal of the implementation is to have it run in real time so it will seemlessly work when the game is running.

2.1 The Ground Pillars

When working with 3D meshes, especially when one is modifying meshes, there is a lot of things that can go wrong, for example, holes or faulty triangles. Therefore one needs to have a stable base to work on. To have good and well-known data structures and algorithms is key for a successful mesh editing system. Because of this the half-edge data structure [14] was used because it is well tested and easy to use and has significant benefits when you need to split or fix holes in a mesh.

2.1.1 Half-Edge Mesh

The doubly connected edge list, also known as a half-edge data structure, is a data structure that allows for easy traversal over a mesh, is reasonably compact and its only requirement is that the mesh is manifold [25]. The data structure is called half-edge because instead of storing the edge of a mesh, it stores a half-edge. The edge are split into two directed half-edges that make up a pair where each half-edge points in opposite directions. This allows for both clockwise and counter-clockwise orientation around the border of a mesh face. A visualisation of the data structure can be seen in figure 2.1.

To implement a completely developed half-edge library can be very time-consuming. Therefore the external library OpenMesh was used since it contains a well implemented half-edge data structure and a lot of helping functions to make the creation and modification of a half-edge mesh much simpler.

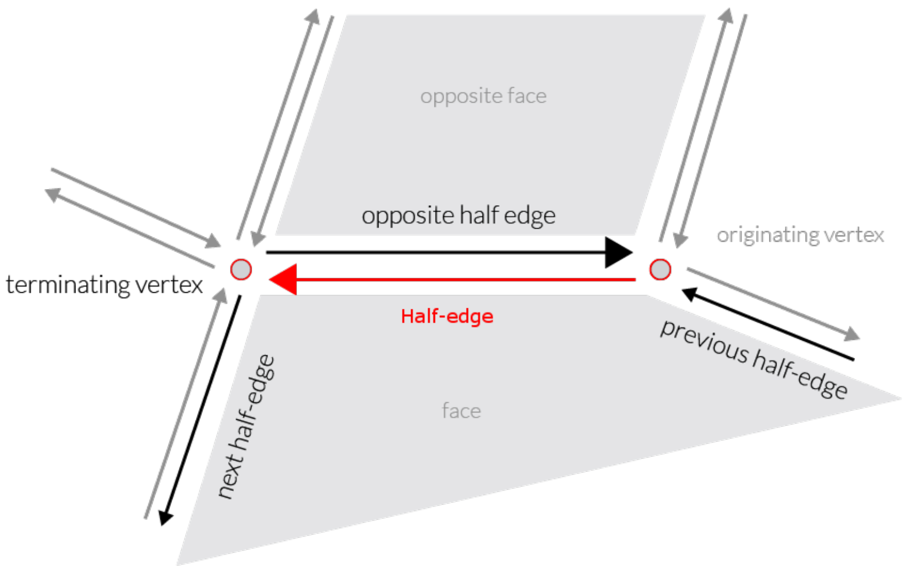

FIGURE 2.1: The half-edge data structure

When implementing the half-edge data structure, each half-edge is made up of five pointers:

- **vert**, a pointer to a vertex from which the half-edge originates from
- **pair**, a pointer to a oppositely oriented adjacent half-edge
- **face**, a pointer to a face that the half-edge borders
- **next**, a pointer to the next half-edge around the face
- **prev**, a pointer to the previous half-edge around the face

Because the information that is stored contains a reference to its neighbor, one can easily traverse the mesh by using the pair, next and prev pointers.

As can be seen in listing 2.1, this data structure is a reasonably compact one. Especially when compared to, for example, the winged-edge data structure [22] that stores a lot more information per edge

2.1.2 Voronoi Diagram

The Voronoi diagram, also called Voronoi tessellation, is a diagram where you have a finite set of points $\{p_1, ..., p_n\}$, called sites, where every site is the center point of a Voronoi cell. Each cell in a Voronoi diagram is always a convex shape where every line segment falls on the perpendicular bisector of two neighbouring sites and each vertex of the line segments are the largest possible empty circle that passes through 3 sites. An example of a 2D Voronoi can be seen in figure 2.2.

```
struct HE_edge
{
    HE_vert* vert;
    HE_edge* pair;
    HE_face* face;
    HE_edge* next;
    HE_edge* prev;
};

struct HE_vert
{
    float x;            //
    float y;            // Can also be a vec3 or a float[3]
    float z;            //

    HE_edge* edge;
};

struct HE_face
{
    HE_edge* edge;
};
```

LISTING 2.1: Half-edge code structures

When constructing a line segment, a half-plane intersection method can be used, which is repeated for every site to build the entire diagram. This method takes $\mathcal{O}(n^2 \log n)$ time to create a diagram which is not viable for real-time applications. Fortunately, in 1986 Steven Fortune discovered an algorithm for Voronoi tessellations called Fortune's algorithm [3]. This algorithm is running in $\mathcal{O}(n \log n)$ time and is described as a sweep line algorithm for Voronoi tessellations.

The algorithm uses a straight line, called the sweep line, that sweeps every point and then creates a distorted sweep line, called the beach line, that follows the straight sweep line and is shaped as parabolic curves where each point on the curve has an equidistance to a site and the straight sweep line. The beach line creates the edges as they move towards the straight line. When the circumcircle of a beach line lies within the straight line and its center is equidistant to three points the center is a Voronoi vertex.

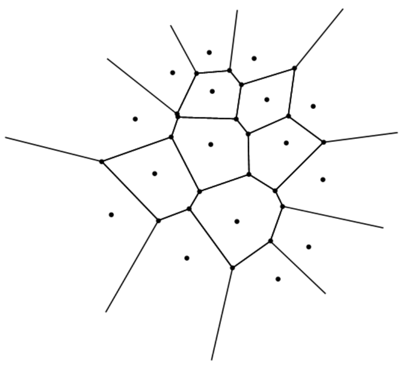

FIGURE 2.2: A 2D Voronoi diagram

The Voro++ library is a free, open source software that focuses on 3D Voronoi tessellation. It is written in object-oriented C++ and uses a direct method of calculating each cell individually, which provides a high degree of flexibility since each cell can be individually tailored to the needs of the user. Voro++ can easily be used to create a region shaped as a cube, but can also utilise many other convex shapes, that is then filled with arbitrary points which are used to create the 3D Voronoi tessellation. An example of a 3D Voronoi tessellation that can be created using the Voro++ library can be seen in figure 2.3.

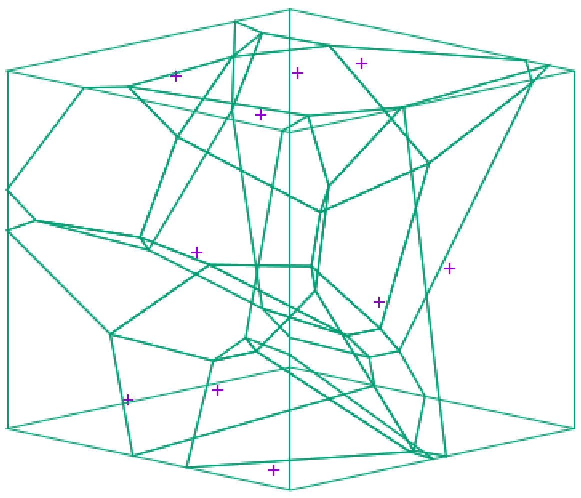

FIGURE 2.3: A 3D Voronoi diagram

2.2 Implementing the System

The implementation was written inside an isolated test environment that used Axolot's own game engine as a base. First, an implementation of the Voro++ library was made to test the performance and if it was usable for this kind of system. A class called **Voronoi** was implemented to wrap the Voro++ functions so the data from each Voronoi cell would be easy to retrieve. An initialization function was created to handle the creation of the Voronoi diagram and to save all the necessary information that would be used to create splitting planes. These splitting planes are made up of a single point in 3D space that tells where the planes position is in a 3D world and a normal that tells in which direction the plane is facing. They are used for the mesh cutting part of the system where each vertex in each triangle is checked on which side of the plane they are. If they are above the plane, the vertex is removed and the edges that intersect the plane gets new vertices to connect to and new triangles is created.

```
List outputList = subjectPolygon;
for (Edge clipEdge in clipPolygon) do
   List inputList = outputList;
   outputList.clear();
   Point S = inputList.last;
   for (Point E in inputList) do
      if (E inside clipEdge) then
         if (S not inside clipEdge) then
            outputList.add(ComputeIntersection(S,E,clipEdge));
         end if
         outputList.add(E);
      else if (S inside clipEdge) then
         outputList.add(ComputeIntersection(S,E,clipEdge));
      end if
      S = E;
   done
done
```

LISTING 2.2: The unmodified Sutherland–Hodgman algorithm

The half-edge class, was then created and the OpenMesh library was added. This class contained all the mesh handling. When creating the half-edge mesh, since the engine uses their own render pipeline, a conversion is needed between the renderable objects and the half-edge objects. Therefore a function was created to take the position, UV and normal data from the engines renderable objects and create a half-edge mesh.

All the triangle slicing and mesh modifications were made in the same function. The function does a set-up step where it creates all the necessary lists that are going to be used during the cutting. Then it runs a modified version of the Sutherland-Hodgman algorithm [24], that can be seen in listing 2.2, to find all the vertices that need to be cut by doing a plane intersection against the splitting plane. Since this implementation only uses triangles, the algorithm iterated each triangle instead of each edge. The UV's were also handled in this algorithm to interpolate a new UV coordinate if a vertex is removed.

After each cut step a simplification step is started. This step walks through the boundary that is created from the cutting and collapses vertices that lie in a straight line between two other vertices. This is to lessen the number of triangles created per cut step and to ease the triangulation of the ear clipping algorithm. After the simplification step a list of the boundary vertices is created which is passed to a ear clipping function. A visualisation of the ear clipping algorithm can be seen in figure 2.4. The ear clipping algorithm loops through the list of vertices and creates triangles using three following vertices and checks the edge between the first and third vertex. If they don't collide with any of the other edges in the list a triangle is created and the "ear", the second vertice, of the triangle is removed. This is repeated until the list only contains three vertices, then a triangle is created from the vertices that are left and the algorithm ends.

After the ear clipping is done a new mesh is created, from all the triangles created from the cutting and the ear clipping, that is pushed into a list of meshes which is returned at the end of the function. This list is then used to convert each mesh back to a renderable object.

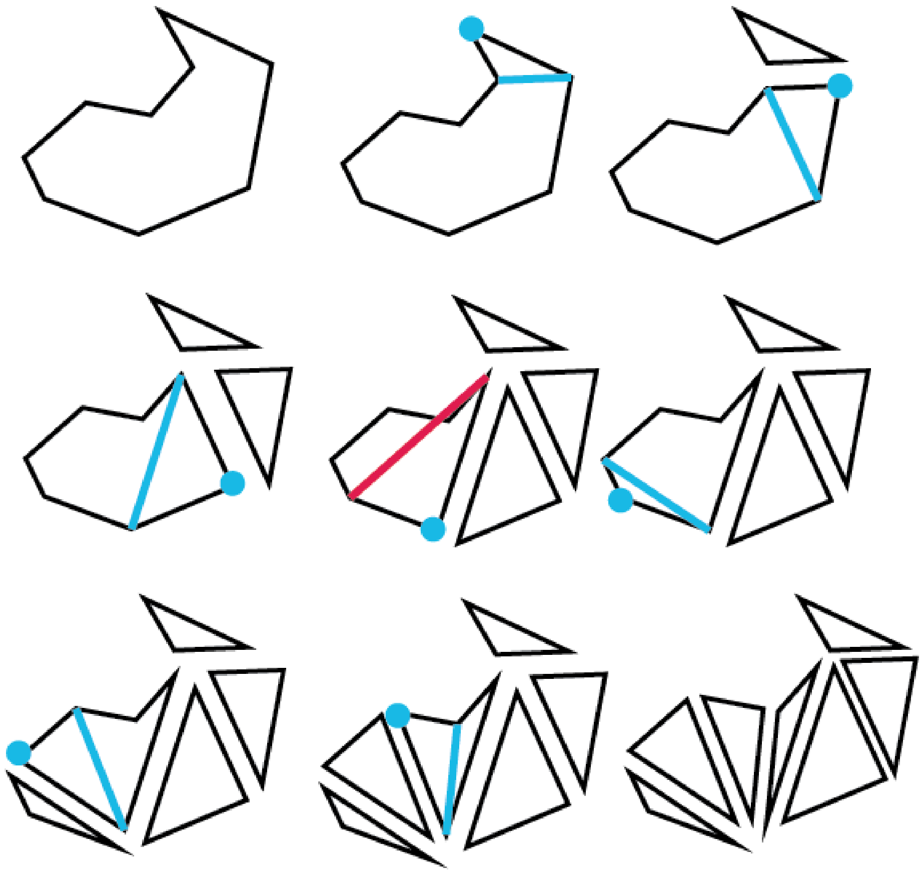

FIGURE 2.4: Example of a ear clipping algorithm

Chapter 3

Results

The goal of the project was to implement a destruction system that was based on a 3D Voronoi diagram that could run in real-time. This was going to be used in their game Scrap Mechanics upcoming game mode, Survival. A player would use a tool on a tree or a stone, the destruction system was supposed to activate and dynamically cut the tree or rock into smaller pieces based on a randomly generated 3D Voronoi diagram. At the end of the implementation stage, however, this was not exactly the result that it ended up with.

During the implementation, two meshes, seen in figure 3.1, were used to test if everything worked as intended. The cube was the mesh that was used the most due to its simplicity and low triangle count. Also, its easier to know how a cut would look like on a cube than a tree.

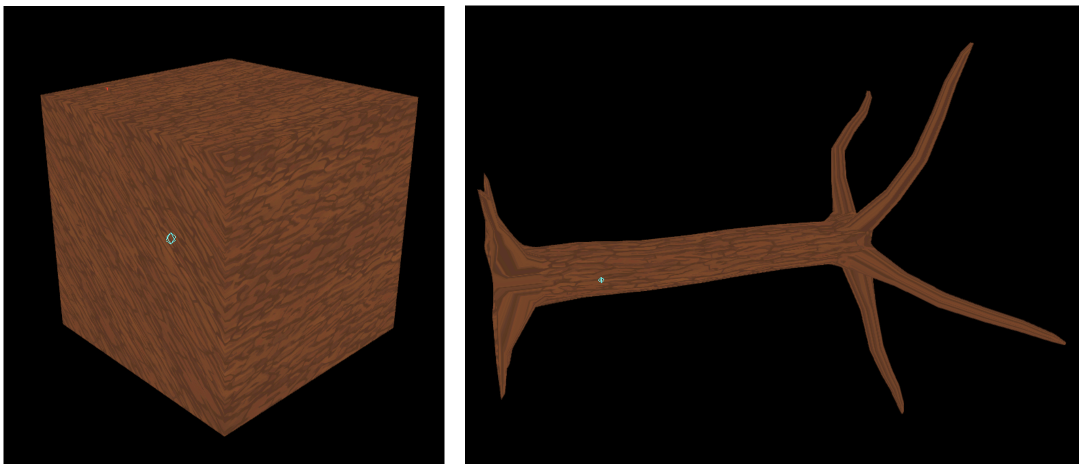

FIGURE 3.1: The two test meshes, a cube and a tree

3.1 Iterations

On the first working iteration, very simple UV coordinates were assigned to the new vertices and gave it a very uneven and broken UV mapping with lots of stretched textures and artefacts, seen in 3.2. The implementation could not, however, cut the cube into ten convex pieces but could only cut it into one convex piece since the program would crash with more than one cell, it also had a few huge bugs that could appear and break the program. At this point a performance test was made, the code could cut one cell in fifty-three milliseconds when run in debug mode so it performed much better if done in release mode.

After the first iteration, the code was looked over to remove and clean up some excessive code and improve the crude design. With each step and iteration, the code improved in both speed and code structure. The UV mapping improved because of a mistake made in the code, that also could end up crashing the program.

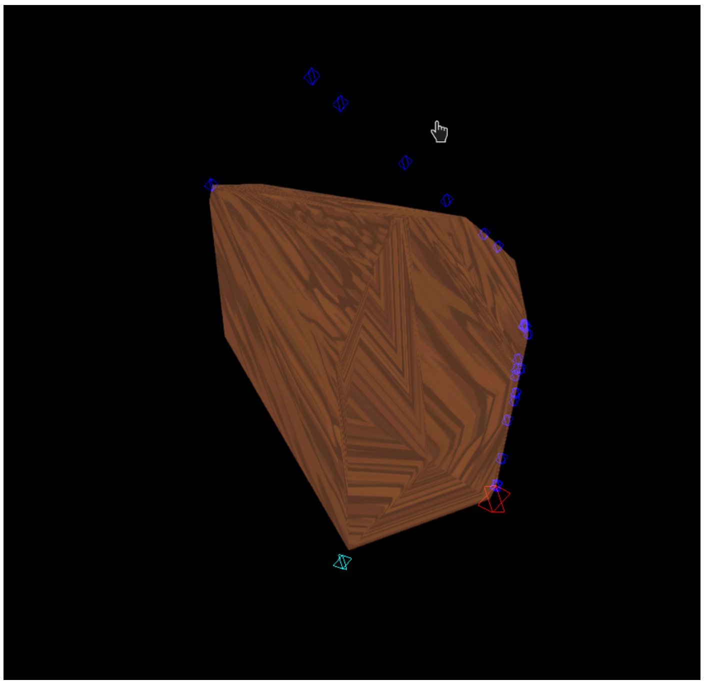

FIGURE 3.2: One cell from the first version of the implementation

3.2 End Results

In the final version, the code could split a cube into ten separate meshes in around eighteen milliseconds on average and a tree around seventy-four milliseconds on average, in release mode, but the timings vary a bit because of the randomly generated Voronoi diagram. This result could not be used in a real-time application, it would leave an impact on the frames and give a big frame drop, but it can be used in an application where the system is used to pre-fracture the meshes. Judging by the results of the tests, the Sutherland-Hodgman cutting algorithm is, most of the time, the slowest of the algorithms. In some cases, the edge collapsing algorithm can take at least 4 times as long as any of the other algorithms during a run. In the end, the implementation usually create good automatic UV mapped convex pieces that can be used within the game as long as it is pre-fractured. Figures of the end results can be seen in figure 3.3 and a table of the results can be seen in table 3.1

Mesh	**First Iteration (Average)**	**Final Iteration (Average)**
Cube (12 triangles)	52.9295 ms	3.7192 ms
Tree (1024 triangles)		10.6529 ms

TABLE 3.1: Performance of one cut cell, left debug mode, right release mode

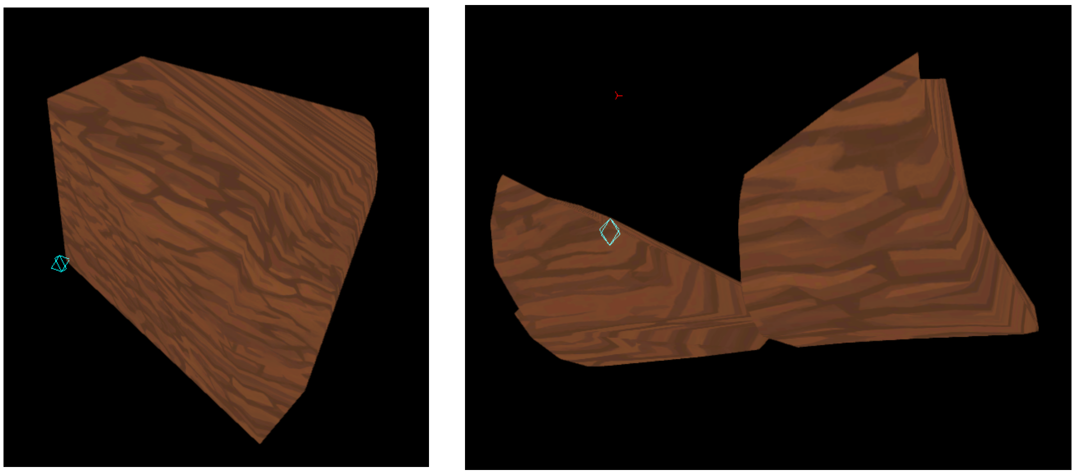

FIGURE 3.3: Final cells, One cube cell (Left), Three tree cells (Right)

Chapter 4

Discussion

The task of creating a working, effective destruction system that would be run in real-time was known to not be an easy task, since creating this kind of system has been researched for years by researchers and big game studios. It was not until somewhat recently a method was discovered that could dynamically destroy a 3D environment in real-time, even though it still doesn't run fast enough to be used extensively in a 3D game with modern graphics at the goal framerate of 60 frames per second or more.

The half-edge data structure is a reliable data structure but has its speed limitations, so it could be exchanged into either a faster data structure or some custom implementation that works for just the goals that are needed. Then some of the other algorithms like the ear clipping and Sutherland-Hodgman algorithms may be exchanged to faster algorithms to increase the performance.

As mentioned in section 3.2 the different algorithms had very varied execution times. For the most part, the Sutherland-Hodgman algorithm was the slowest algorithm, this is mostly because it gets slower the more triangles it needs to check. Also in some tests, the edge collapsing took the majority of the execution time, this could happen if a lot of vertices are aligned and can be collapsed. The fastest algorithm were the ear clipping algorithm since the algorithm is very effective in filling the holes. It is faster mostly because it doesn't need to handle a large amount of vertices. Table 4.1 shows a few time samples on the different algorithms.

There were a few problems that came up along the way and most of them are because of the shortage of time. Because of the constraints, there wasn't much time to research on how to implement the system, what data structures and algorithms to use and to find external libraries that would work with the design.

When researching different libraries to use, one library seemed to be able to do everything that needed to be done. CGAL, short for Computational Geometry Algorithms Library, was first chosen because it contains a massive amount of algorithms that looked like it would fit perfectly with the tasks that needed to be done. It could both handle Voronoi diagrams and Half-edges. But it is not a library that is easy to work with nor is it easy to learn. But after some time, CGAL were switched to OpenMesh and Voro++ since both libraries were much easier to learn and work with.

Once the implementation started everything went smooth and all the algorithms just needed to be implemented and put together to create two

Sutherland-Hodgman	Edge Collapsing	Ear Clipping
2.037050	0.060413	0.004153
1.660602	0.049841	0.003776
1.631906	0.086466	0.008307
0.167646	0.026808	0.006796
1.352118	9.267731	0.021522
0.818974	0.025298	0.003776

TABLE 4.1: Time samples of the different algorithms, in milliseconds, using the tree mesh

reasonably isolated classes that could do all the necessary work without modifying the game engines original files.

Everything was implemented on a fairly high-end PC and still performs worse than what was desired, this means that it won't be able to be used during the gameplay.

4.1 Future work

Some future work for this implementation is to implement a tool for the system so that a user can generate pre-fractured meshes. This also gives the possibility to increase the flexibility of the system because an artist could create more planned pieces and to expose the internal options to be able to change the generation on the fly.

Some algorithms used in the current version, could be exchanged into better and more effective algorithms that weren't implemented now due to the time constraints. Also, the code could be looked over to find out if there are any data structures that could either be added or exchanged to increase the speed of the code.

Chapter 5

Conclusion

In conclusion, this project did not reach its goals of being real-time but it created a good base to work from. The implementation could cut a mesh based on the cells of a randomly generated Voronoi diagram and interpolate the UV coordinates in a somewhat simple way that gives a working UV mapping. So, in the end, the implementation worked as intended though with one difference, it was not in real-time. However, since everything else was implemented it is in a good state for further work from Axolot.

At the end of the project, profiling and testing were made in an attempt to optimise the code and try to get it closer to real-time. Also, all the code was documented to try and decrease the time Axolot's programmers have to learn how all the code works.

This implementation requires some future work and development until it can be used in the game. The most important area of future work is to look over the code and optimise it further.

Bibliography

[1] Future Concepts in Architecture. *Voronoi Diagrams:Nature and Architecture*. URL: `https://neoarchbeta.wordpress.com/2011/05/07/voronoi-diagramsnature-and-architecture/`.

[2] Atari. *Atari History*. URL: `https://www.atari.com/history/1972-1984-0`.

[3] Binay Bhattacharya. *Fortune's Algorithm*. URL: `http://www.cs.sfu.ca/~binay/813.2011/Fortune.pdf`.

[4] Steve Breslin. *The History and Theory of Sandbox Gameplay*. URL: `http://www.gamasutra.com/view/feature/132470/the_history_and_theory_of_sandbox_.php?print=1`.

[5] Daniel Camarda, Rasmus Haapaoja, and Fredrik Johnson. "Real Time Voronoi Fracturing of Polygon Meshes". In: *Real Time Voronoi Fracturing of Polygon Meshes* (Jan. 2016), pp. 1–10.

[6] Riad Chikhani. "The History Of Gaming: An Evolving Community". In: (Oct. 2015). URL: `https://techcrunch.com/2015/10/31/the-history-of-gaming-an-evolving-community/`.

[7] RWTH Aachen Computer Graphics Group. *OpenMesh*. URL: `https://www.openmesh.org/`.

[8] DICE. *Battlefield*. URL: `https://www.battlefield.com/`.

[9] The Economist. *Why video games are so expensive to develop*. URL: `http://www.economist.com/blogs/economist-explains/2014/09/economist-explains-15`.

[10] Axolot Games. *Scrap Mechanic*. URL: `http://www.scrapmechanic.com/`.

[11] Volition Inc. *Red Faction*. URL: `http://www.dsvolition.com/games/red-faction/`.

[12] Jesper Juul. *A Casual Revolution: Reinventing Video Games and Their Players*. URL: `https://www.jesperjuul.net/casualrevolution/casual_revolution_chapter1.pdf`.

[13] Wolfram Mathworld. *Voronoi Diagram*. URL: `http://mathworld.wolfram.com/VoronoiDiagram.html`.

[14] Max McGuire. *Half-Edge Data structure*. URL: `http://www.flipcode.com/archives/The_Half-Edge_Data_Structure.shtml`.

[15] Chris Melissinos. *Video Games Are One of the Most Important Art Forms in History*. URL: `http://time.com/collection-post/4038820/chris-melissinos-are-video-games-art/`.

[16] Microsoft. *Minecraft*. URL: `https://minecraft.net/en-us/`.

[17] The International Arcade Museum. *Asteroids*. URL: `https://www.arcade-museum.com/game_detail.php?game_id=6939`.

[18] Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. "Real Time Dynamic Fracture with Volumetric Approximate Convex Decompositions". In: (2013).

[19] Graham Nelson. *The History Of Video Games, By The Numbers*. URL: `http://www.huffingtonpost.com/entry/the-history-of-the-biggest-video-games-ever-by-the-numbers_us_559edf12e4b096729155d0b9`.

[20] Greg Perreault. *Why Do We Love Video Games?* URL: `http://www.huffingtonpost.com/greg-perreault/why-do-we-love-video-game_b_4740425.html`.

[21] Chris Rycroft. *Voro++*. URL: `http://math.lbl.gov/voro++/`.

[22] Hanan Samet. *Winged-edge data structure*. URL: `https://www.cs.umd.edu/class/fall2011/cmsc420-0101/lecnotes/we.pdf`.

[23] Warren Schultz. *AAA Game*. URL: `https://www.thoughtco.com/what-is-aaa-game-1393920`.

[24] Ivan Sutherland and Gary W. Hodgman. *Reentrant Polygon Clipping*. URL: `http://www.cs.gettysburg.edu/~ilinkin/courses/Fall-2014/cs373/handouts/papers/sh-rpc-74.pdf`.

[25] Michigan Technological University. *Mesh Basics*. URL: `https://www.cs.mtu.edu/~shene/COURSES/cs3621/SLIDES/Mesh.pdf`.

[26] Valve. *Steam*. URL: `http://store.steampowered.com/about/`.

[27] Valve. *Steam Early Access*. URL: `http://store.steampowered.com/earlyaccessfaq/`.

[28] Emmy Zettergren. *Svenska Axolot Games spel Scrap Mechanic gör succé*. URL: `http://feber.se/spel/art/342446/svenska_axolot_games_spel_scra/1`.