UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Marko Täht

Real-time Cave Destruction Using 3D Voronoi

Master's Thesis (30 ECTS)

Supervisor: Jaanus Jaggo, MSc

Tartu 2018

Real-time Cave Destruction Using 3D Voronoi

Abstract:

Environment modification in video games are often done by using simple methods like voxels or pre-calculated destruction. The aim of this thesis is to study different ways of making it more realistic by generating the environment destruction in real time using Voronoi diagrams. This approach represents the world as a 3D Voronoi diagram where the cave is represented as a region where some of the Voronoi cells have been removed. The goal of this thesis is to find the suitable algorithms for such cave generation, compare them and implement a proof of concept simulation in Unity game engine. In this simulation the user can modify the cave by cutting out more pieces, thus expanding the Voronoi diagram in real-time. To cut off pieces of already fixed geometry different approaches for geometry manipulation are also compared.

Keywords:

Voronoi, geometry manipulation, real-time

CERCS: P170 Computer science, numerical analysis, systems, control

Reaalajas koopa lõhkumine kasutades 3D Voronoid

Lühikokkuvõte:

Arvutimängudes kasutatakse keskkonna muutmiseks enamasti lihtsaid meetodeid, nagu maailma kujutamist vokslitena, või eelkalkuleeritud hävitamist. Käesolevas töös uuritakse, kuidas muuta seda reaalsemaks, kasutades Voronoi diagramme. Selles lähenemises kujutatakse kogu maailma ühe 3D Voronoi diagrammina, millesse lisatud koopad on saadud Voronoi rakkude eemaldamise teel. Töö eesmärgiks on leida sobivad algoritmid sellise koopa genereerimiseks, võrrelda nende sobivust ja luua prototüüprakendus Unity mängumootoris, millega testida, kas selline lähenemine on mõistlik. Selles simulatsioonis saab kasutaja mõjutada koobast, lõigates sealt tükke välja ning seeläbi suurendades Voronoi diagrammi reaalajas. Töös uuritakse ka erinevaid lähenemisi juba olemasolevast geomeetriast tükkide välja lõikamiseks ja vaadeldakse erinevaid algoritme geomeetria manipuleerimiseks.

Võtmesõnad:

Voronoi, geomeetria manipulatsioon, reaalaeg

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

Voronoi diagram is very common in nature. For example, the way earth cracks when it dries or in the pattern of a turtle shell are both visually similar to Voronoi diagrams. Rock cracks in this manner, because the energy is distributed between the harder and softer parts of the rock. Softer parts will crack and harder parts will make up the core of the broken off piece. This is also why Voronoi distribution is often used for modeling destruction of objects in computer graphics [1].

In many games, the environment destruction is used as a central part of the gameplay mechanics. However, the modern games usually rely on pre-calculating the destruction by calculating the broken up pieces using Voronoi distribution beforehand. Such technique is also setting a limit to the design of the game. For example, in the game Battlefield: Bad Company 2 [2] there are a lot of destructible buildings. Whenever a wall is blown up the same looking hole is always created. This means that the destruction will eventually become repetitive and the player is not allowed to manipulate the entire environment.

In other games like Minecraft [3], the player is given more freedom by building the entire world out of voxels. While this approach allows the player to change the entire world, removing the voxels do not look realistic compared to how objects break in the real world.

1.1 The goal of the thesis

The aim of this thesis is to study different ways of making cave destruction more realistic by using Voronoi diagrams. This approach represents the world as a 3D Voronoi diagram where the cave is represented as a region where some of the Voronoi cells have been removed. The user can then remove more Voronoi cells to enlargen the cave. For example, the user might mine the cave with a pickaxe, and smaller Voronoi cells could fall out of the cave wall where the pickaxe hit it. With the actions of the user, Voronoi diagram becomes larger, and adding new points to the diagram gets slower. Hence, the diagram generation needs to be fast and must allow dynamic modifications to the Voronoi cells with a minimum performance cost.

The goal of this thesis is to find the suitable algorithms for such cave generation, compare them, and implement a proof of concept simulation in the Unity game engine. Based on this implementation, the feasibility of such approach can be evaluated.

1.2 Layout of the thesis

The second chapter gives an overview about what is a Voronoi diagram, how is it calculated, where it is used, where it can be found in nature and a short description of its dual graph, Delaunay triangulation. Also, different Voronoi generation algorithms are compared to each other, namely: Brute Force algorithm, Fortune's algorithm and Bowyer-Watson algorithm. The third chapter describes how the Voronoi diagram is represented in the simulation. The fourth chapter proposes two approaches to modify the environment in simulation and compares them to each other.

2 Voronoi diagram

Voronoi diagrams are named after Georgy Feodosevich Voronoy, who defined and studied the general n-dimensional diagrams in 1908 [4].

Voronoi diagram is the partitioning of space into regions, defined by the points that are given beforehand. These points are called sites or seeds. Within a region, all the points are closer to the site of the region than to any other region's site. In Figure 1 and Figure 2, the black spots are the seed points and the colored areas are the Voronoi cells of the corresponding seed.

Most commonly, Voronoi diagrams are using Euclidian distance between points (Figure 1), but other distance metrics, like Manhattan distance, are also possible (Figure 2). With Euclidian distance, the edges of the region are the perpendicular bisectors of the adjacent sites. In this thesis, Euclidian distance metric is used to construct Voronoi diagrams.

More formally and generally, Voronoi region R_k defined by point P_k is a set of all points in space whose distance to P_k is not larger than their distance to P_j, where j is any index different from k. This corresponds to the following equation 1:

$$R_k = \{x \in X | d(x, P_k) \leq d(x, P_j) \; for \; all \; j \neq k\} \qquad 1$$

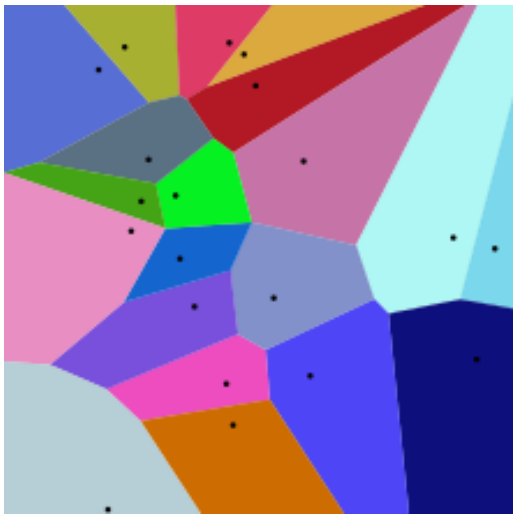

Figure 1. 20 point Voronoi diagram using Euclidian distance [5].

Figure 2. Voronoi diagram using Manhattan distance [6].

Voronoi diagrams have many uses in different disciplines. In biology, Voronoi diagram is used to model different biological structures like cells and bone microarchitecture. In machine learning, Voronoi diagrams are used for 1-NN classifications [7]. Victoria, a state in Australia, uses Voronoi diagrams for government schools to admit eligible students to nearest primary school or high school to where they live [8].

2.1 Voronoi diagrams in nature

Voronoi diagrams appear in nature in many different forms. For example, it can be seen in the patterns on the giraffes (Figure 3), in the way cells are arranged in the leaves (Figure 4) or in the way the dried earth has been cracked (Figure 5). While these are not perfect Voronoi diagrams, in nature nothing is perfect. There are irregularities on the edges and even some extra dead-end lines, but the pattern of Voronoi diagrams is still clearly visible. Being

so widespread in nature makes it a very good choice to model natural breakage of rocks within a cave.

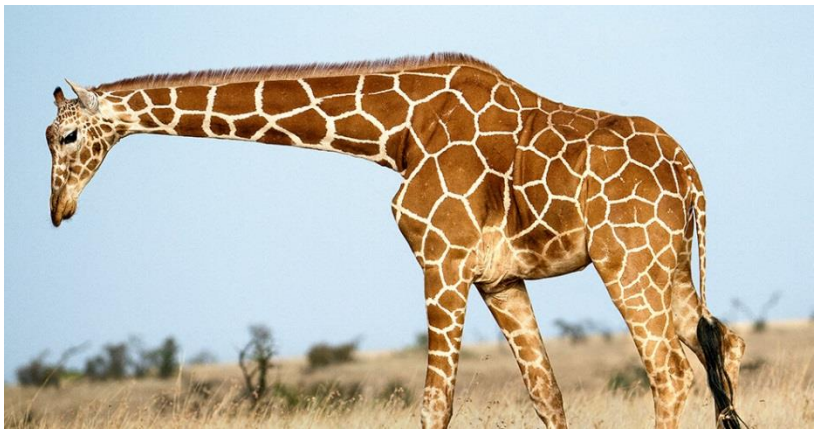

Figure 3. Pattern on a giraffe [9].

Figure 4. Cells on a leaf [10].

Figure 5. Dried cracked earth [11].

2.2 Delaunay triangulation

Delaunay triangulation was created by Boris Delaunay in 1934 [12]. Delaunay triangulation is a triangulation of a plane using a set of discrete points, so that no point is inside the circumcircle of any triangle, as seen in Figure 6. This triangulation maximizes the minimum angle of all the triangles and tends to avoid triangles, whose area is much smaller than the area of its circumcircle, also known as sliver triangles. This can also be used in higher dimensions, for example in 3D tetrahedrons and circumspheres are used instead of triangles and circumcircles.

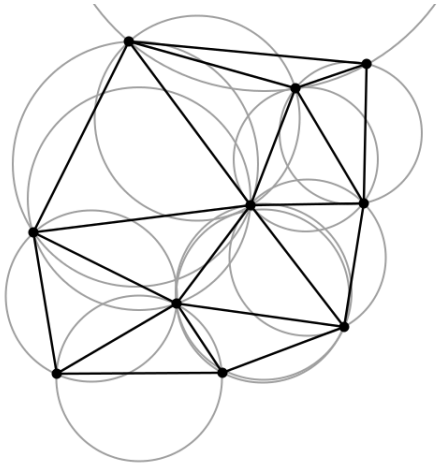

Figure 6. Delaunay triangulation on a plane. Circles are representing the circumcircles of the triangles [13].

In the case of using Euclidian distance, Delaunay triangulation is a dual graph of Voronoi diagram. In dual graph all the regions of the original graph become vertices and all the vertices of the original graph become regions of the dual graph. Regions are the area surrounded by edges or the area outside the graph. Connected vertices in the original graph are neighbouring regions in dual graph. Graph and its dual graph is illustrated in Figure 7, where the red dots and black lines represent the original graph and blue lines and blue dots represent the dual graph. The blue point outside the graph represents the area that surrounds the original graph.

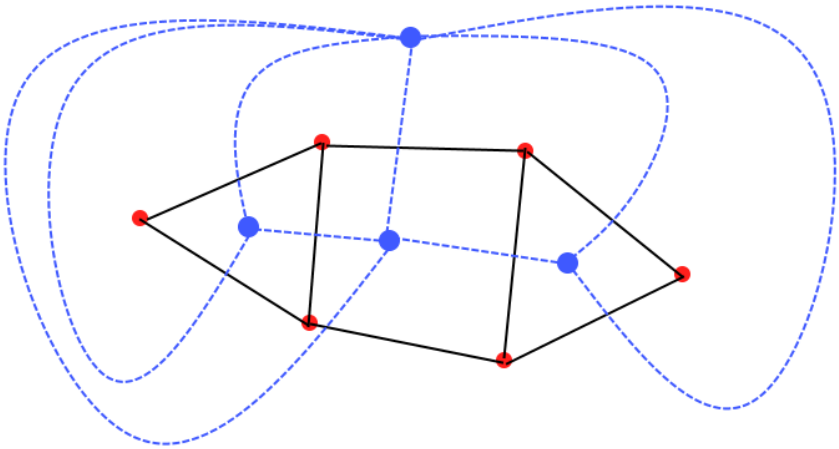

Figure 7. Graph (red vertices and black edges) and its dual graph (blue vertices and blue edges).

This gives an additional way to calculate Voronoi diagrams by first finding the corresponding Delaunay triangulation. Voronoi diagram is formed by connecting neighbouring triangle circumcenters in Delaunay. Delaunay triangulation can be gotten from Voronoi diagram by connecting seed points of Voronoi diagram.

2.2.1 Delaunay triangulation to Voronoi diagram conversion

Once the Delaunay triangulation is calculated. It has to be transformed to Voronoi diagram. Since the Delaunay triangulation and Voronoi diagram are dual graphs, this process is not difficult and consists of five steps:

1. Choose a vertex from triangulation.
2. Find all triangles connected to the vertex.
3. Get the circumcenters of the triangles.
4. Join the circumcenters together clockwise to form a polygon.
5. Repeat 1-4 until all the vertices are processed.

In Figure 8 the algorithm corresponding to these five steps is given.

```
DT = Delaunay triangulation
VC = [] // Voronoi cell will be stored here
For each vertex V in DT:
        Triangles = find all triangles connected to V
        Circumcenters = []
        For each triangle T in Triangles:
                Circumcenters.add(T.circumcenter)
        Sort_clockwise(circumcenters)
        VC.add(circumcenters)
```

Figure 8 Pseudocode for Delaunay to Voronoi conversion.

For every vertex in triangulation, all the connected triangles are found. Circumcenters of the triangles are stored in a list and sorted clockwise around their center of mass. In dynamic solutions, this can be optimized to only work with the vertices that were added and that were affected by the addition of new vertices.

To draw the Voronoi cells, just use the list of circumcenters that was found for every cell. Edge is defined by vertices at index i and i+1. The only exception is the final edge, that is defined by the first and last vertex in the list. Figure 9 shows how the Delaunay triangulation and corresponding Voronoi diagram look on the same set of points. Voronoi diagram is represented with the white lines and Delaunay triangulation with red lines.

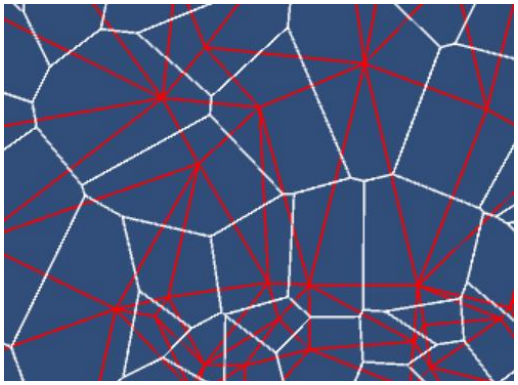

Figure 9. Delaunay triangulation (red lines) and the corresponding Voronoi diagram (white lines).

2.3 Calculating Voronoi diagram

The first step in doing real-time destruction calculation using Voronoi diagrams is to find a suitable algorithm for it. The following chapter will explain three well-known algorithms to calculate the Voronoi diagram. Each of the algorithms will be analyzed to see if they are suitable for real-time calculation. Most important property is the ability to add new vertices to the diagram without having to recalculate the entire graph.

For the ease of explanations, the following algorithms are explained in the context of 2D but their properties are compared in 3D because the final result of the thesis will be in 3D.

2.3.1 Brute force algorithm

Brute force approach [14] is the easiest to implement. It is good to use in the early stages of development when the performance is not the main goal and getting things up and running is more important.

The brute force algorithm (Figure 10) consists of following steps:

1. Pick a seed point from the list of points. Let's call it point A.
2. Calculate the distance from A to every other point.
3. Pick the point closest to A. Let's call it point B.
4. Find the perpendicular bisector (Appendix II) between points A and B.
5. Remove all the points on the other side of the bisector that are farther away from the bisector than B.
6. Remove B.
7. Repeat steps 3-6 until no more points remain.
8. Find the intersection points of the bisectors.
9. Connect the intersection points clockwise or counterclockwise to create a polygon.

This process is repeated with every seed point and a list of polygons, representing Voronoi cells, is returned.

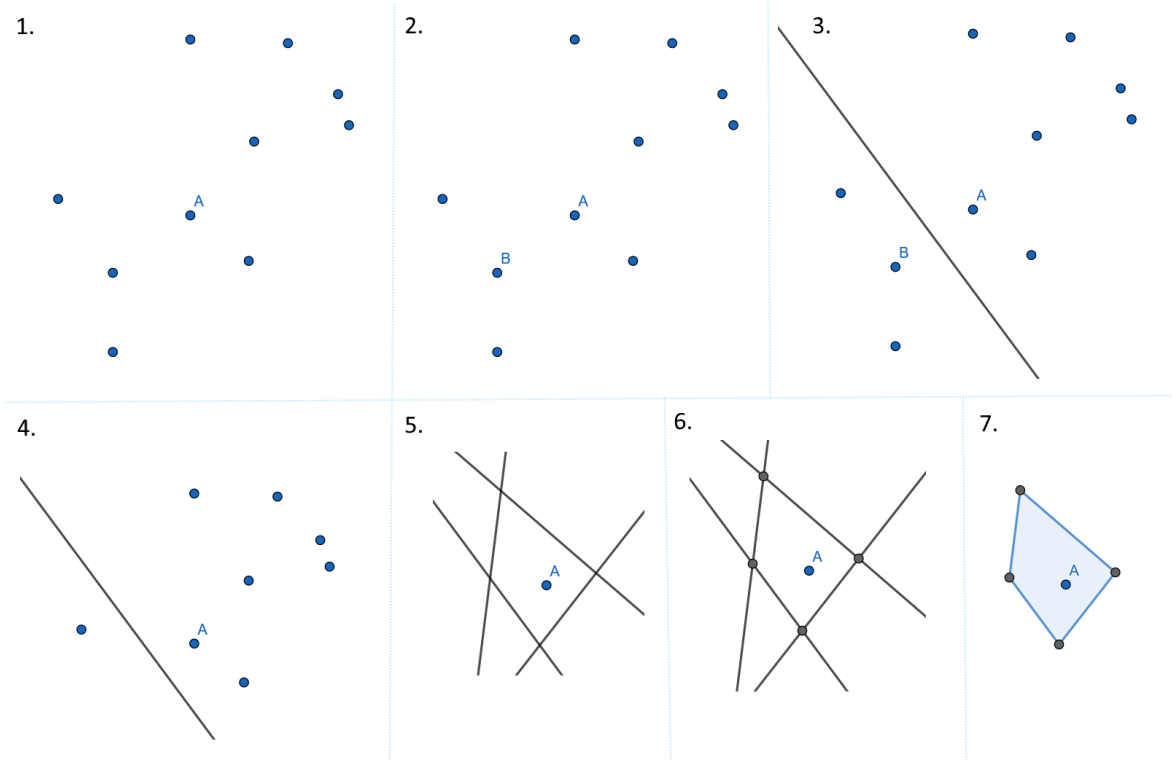

Figure 10. Brute force algorithm for calculating Voronoi cell. 1. Choose a point to calculate cell. 2. Find the closest point. 3. Calculate perpendicular bisector. 4. Remove unnecessary points. 5. Repeat with all closest points. 6. Calculate intersections of bisectors. 7. Order the points clockwise to get the Voronoi cell.

Steps 2-7 find the lines that represent the edges of the Voronoi cell. All the points on the other side of the bisector will be removed since they are further away from the closest point and cannot share an edge with the chosen point.

Steps 8-9 calculate the exact edges of the cell and join them into a polygon to do that intersection points of the lines will be calculated and the lines clipped to edges. Most bisectors will intersect with the other bisectors forming a closed loop. Only exceptions are for the points on the edge of the diagram. These edges will be clipped by the boundary of the area.

While brute force is easy to implement, it has a poor performance in later stages when the number of vertices can reach thousands. A Voronoi cell can potentially have N-1 sides where N is the total number of seed points. This means that for calculating every cell, the time complexity is $O(N^2)$. Thus, the overall time complexity for calculating the entire diagram is $O(N^3)$. Furthermore, adding new points when the diagram is already generated is not easy, since all the vertices affected by the new insertion need to be recalculated. The brute force algorithm sometimes removes points that would contribute to the Voronoi cell. This is not a big issue in destruction modelling, but the result is not a real Voronoi diagram. However, it can be easily extended to higher dimensions, which makes it good for early testing and implementation.

2.3.2 Fortune's algorithm

Since the brute force algorithm is too slow for a large number of points, a faster algorithm is needed. A big improvement is to use Fortune's algorithm. Fortune's algorithm is an algorithm for generating Voronoi diagram in 2D. It was published by Steven Fortune in 1968 in his paper "A sweepline algorithm for Voronoi diagrams" [15]. Fortune's algorithm is a

sweepline algorithm and it has $O(n \log n)$ time and $O(n)$ of space complexity, for generating a Voronoi diagram on a plane. The following chapter explains how the Fortune's algorithm works and is based on „Fortune's algorithm and implementation" [16] and "Construction of Voronoi diagrams using Fortune's method: A look on an Implementation" [17].

Fortune's algorithm uses a conceptual sweepline to solve problems in Euclidean space. The sweepline moves across the plane and stops whenever a point of interest is reached. Operations are restricted to objects that intersect or are near to the sweepline. The sweepline is processing all the point one-by-one and once it is finished the Voronoi diagram is formed. Additionally, the algorithm uses a beach line, shown in Figure 11, that consists of parabolas that are defined by the input points above the sweepline and sweepline itself. The Voronoi diagram above the beach line is known regardless of the points that are below the sweep line.

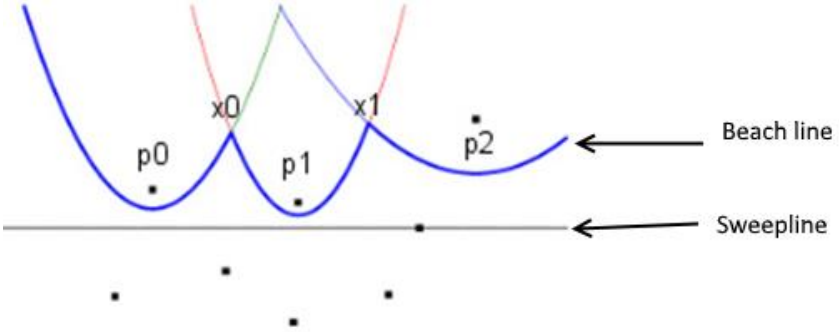

Figure 11. Horizontal black line is the sweepline and the blue curved line is the beachline [16].

As the algorithm progresses, two kinds of events happen: site events and circle events. Site event happens when the y coordinate of the input point equals to the y coordinate of the sweep line. When this happens, the point is added to the Voronoi diagram and its parabola is inserted into the beach line (Figure 13). All of the site events are known beforehand because every input point is a site event waiting to happen. Circle event happens when two parabolas that were separated by another parabola meet on the beach line. Figure 12 shows how the intersection point of moving beach lines define the Voronoi edges. When the third parabola is eaten up by its neighbours the circle event happens. Whenever a circle event happens, two Voronoi diagram edges intersect. The intersection point becomes a vertex in the Voronoi diagram and a new edge starts from the intersection point.

To make sure that all the events happen in order, priority queue is used as a list for potential future events. All the events in the queue are ordered by their y coordinate. Additionally, a binary tree is used to keep track of the beach line. Leaves of the tree represent the parabolas currently in the beach line and inner nodes of the tree represent the intersection edge between the parabolas. The intersection edge between the parabolas will be made into an edge in the Voronoi diagram when a circle event has happened at both ends When the algorithm has finished, all the unfinished edges are clipped by the edges of the area to be partitioned thus becoming finished edges too.

Fortune's algorithm is fast since every event is processed only once and there are N events giving the time complexity of $O(N)$. Each event is either insertion of new point to the diagram or removal of a parabola from the beach line. Because the data is stored in the priority queue and binary tree it takes $O(\log N)$ time to process a single event. This gives Fortune's algorithm the total time complexity of $O(N \log N)$.

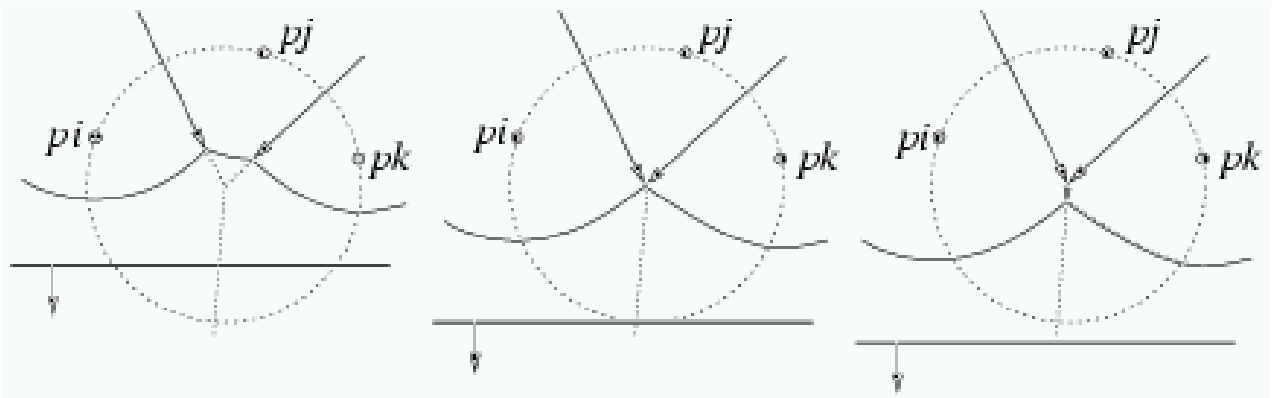

Figure 12. On the left, we can see the middle parabola is about to be swallowed. Center image shows the circle event. On the right image, the middle parabola has been removed from the beach line and a new edge is starting from the location of the circle event [18].

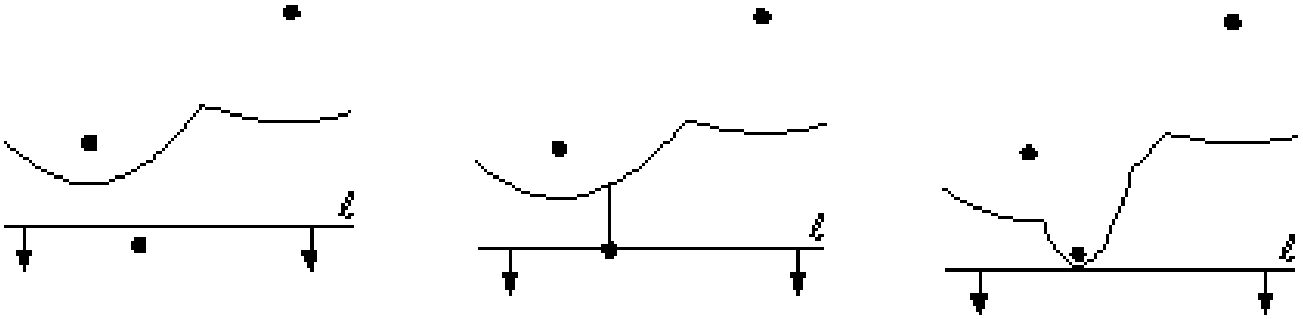

Figure 13. On the left sweepline is about to reach a new point. In the center, sweepline has reached a new point and its parabola is added to the beachline. On the right new point is processed and its parabola is part of the beachline [17].

2.3.3 Bowyer-Watson algorithm

The Fortune's algorithm is a lot faster, but the lack of ability to add new vertices later on makes it a bad choice for real-time destruction calculation. With a bit of sacrifice in calculation time, this ability can be gotten from the Bowyer-Watson algorithm.

Bowyer-Watson algorithm, also known as Bowyer algorithm or Watson algorithm, was devised independently of each other by Adrian Bowyer (1981) [20] and David Watson (1981) [19]. This algorithm will be used to calculate the Delaunay triangulation. If needed, the Voronoi diagram can be easily derived from that.

The Bowyer-Watson algorithm works as follows:

1. Create a triangle that contains all the points to be processed.
2. Add a point to the triangulation.
3. Find all the triangles whose circumcircle contains the new point.
4. Remove these triangles.
5. Connect the vertices of the remaining hole to the new point.
6. Repeat 2-5 until all points are processed.
7. Remove all triangles that are connected to the original triangle.

Point insertion steps are shown on the images in Appendix I.

The Bowyer-Watson algorithm needs a pre-existing Delaunay triangulation to start on. The simplest valid Delaunay triangulation is a triangle. This is why the large triangle is added in step 1. This triangle is called the super-triangle. Once the algorithm finishes, this can be removed.

Steps 2-5 re-triangulate the part of the diagram where the point was added to make it a valid Delaunay triangulation again. The condition is that no point can be inside any triangle's circumcircle.

Step 7 removes the triangles that share a vertex with the super-triangle. This cleans up the graph and only leaves the triangulation of the points. This step can be ignored if the initial triangle is not in the way of calculation or it is known that new vertices will be added later. If the new points are added to the existing triangulation the super-triangle will not be needed, but it is required when adding points outside the already triangulated area.

In all scenarios, the algorithm takes $O(N^2)$ time where N is the number of points, but with different optimizations, time complexity can be improved. By using the connectivity of triangles, the time complexity of $O(N \log N)$ can be achieved. This means, that instead of checking all the triangles, a walk from a triangle towards the new point is performed [21]. Additional time efficiency can be achieved by pre-calculating the circumcircles. For real-time applications, additional data structures like quadtree could be used to further improve the search time.

In conclusion, this algorithm is easy to understand and since it is an incremental algorithm, it has the ability to dynamically change whenever a new point is added without the need to recalculate the entire diagram every time. Extending Bowyer-Watson to higher dimensions does not require big changes. In 3D, triangles are replaced with tetrahedrons, circumcircles with circumspheres, and edges with faces, nothing else changes. And with multiple optimization techniques, the same time complexity can be reached that the Fortune's algorithm has.

2.4 Optimizing and benchmarking Bowyer-Watson algorithm

From previously described Voronoi Generation algorithms the most suitable for this thesis is the Bowyer-Watson algorithm. Therefore, this algorithm was selected for implementation and it was optimised for real-time use. The classical implementation of this algorithm is very slow even on a small number of points, but multiple optimizations can be made to make it run faster. The following chapter will benchmark the different optimizations as well as the classical implementation.

For benchmarking the Bowyer-Watson algorithm, the calculations are done inside an 800x500x800 area, which is filled with control points. Normally, Voronoi cells on the edge of the area get clipped by the bounding area, but in the simulation, such boundary does not exist. To make sure that all the points of interest get finite Voronoi cells (Figure 14), the outer seed point are not made into Voronoi cells but only used to bound the other points. These bounding seed points will be called control points.

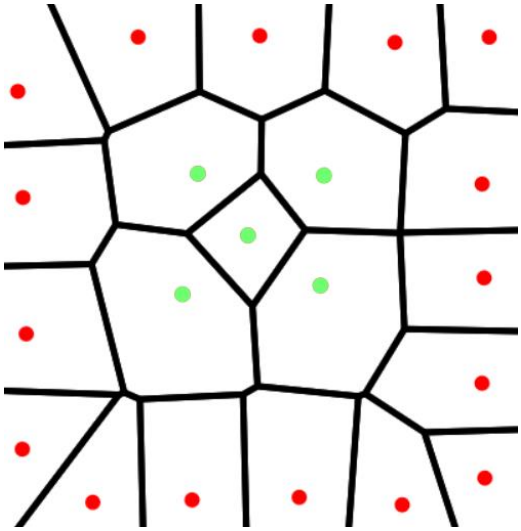

Figure 14. Voronoi diagram with control points. Red points are control points and green points are the points of interest.

First initial test is made on the un-optimized algorithm. Then, improvements on face validity check and bad triangle localization are made and benchmarked to see how much improvement they give.

In benchmarking diagrams, the number of points is chosen as a power of two from 16 to 1024 with the additional 120 control points. The point counts are written in the format of A+B, where A defines regular and B control points.

The base implementation of the Bowyer-Watson algorithm was executed on 16+120 points for a baseline. The algorithm took a long time so it was stopped after 4 hours. By the 4-hour mark, the algorithm had processed 100 points. First 60 points were processed within the first hour. Based on this, the last 36 points would have taken at least another 4 hours.

The following optimizations were made to improve the runtime of the algorithm:

1. Improved tetrahedron face validity check.
2. Improved bad tetrahedron search once the first bad tetrahedron is found.
3. Improved localization of the first bad tetrahedron.

2.4.1 Improved tetrahedron face validity check

The first optimization added was to improve the speed of finding faces between "bad tetrahedrons". Bad tetrahedrons are tetrahedrons where the circumsphere contains the new point added. Faces between two bad tetrahedrons are removed, the remaining are used to re-triangulate the area.

In the base implementation, each face in each bad tetrahedron is checked against all the faces of all the other bad tetrahedrons. This nested loop check has the time complexity of $O(N^2)$.

With increasing memory usage, the performance of tetrahedron face validity check can be improved. During the iteration over all the tetrahedrons, whenever a bad tetrahedron is found, it is added to a separate list and a Boolean value is given to it. The Boolean value indicates whether the tetrahedron is bad or not. This simplifies the check of face validity but requires all the faces to know the tetrahedrons on each side of it. Now, when checking the faces, we can ask from the face about the validity of the tetrahedron on each side of the face. If both have the Boolean value set to true, then this face will not be used.

Some loops are required to make sure that all the faces know the tetrahedrons on either side. However, these loops are smaller and only iterate over new tetrahedrons that are added to the triangulation during a single point addition. Thus the impact on the performance is not as significant.

Points	16+120	32+120	64+120	128+120	256+120	512+120	1024+120
Before	~8h	N/A	N/A	N/A	N/A	N/A	N/A
After	0.168s	0.064s	0.114s	0.131s	0.228s	0.556s	1.167s

Table 1. Benchmarking of the Bowyer-Watson algorithm before and after improving face validity check.

Table 1 has the benchmarking results after the optimization was implemented. It can be seen that the performance boost is significant. This gives a fast working algorithm that is already good enough for real-time generation on modification of a cave using Voronoi diagram and its underlying Delaunay triangulation.

2.4.2 Improved bad tetrahedron search once the first is found

The second optimization added was to improve the speed of finding bad tetrahedrons. Base implementation iterates through all the tetrahedrons to find the bad tetrahedrons. This can be improved using the modifications made in the first optimization. Each tetrahedron knows its neighbouring tetrahedrons and all the bad tetrahedrons are connected. After the first is found, iterations can be stopped and breadth-first search can be done on the neighbouring tetrahedrons to find all the bad tetrahedrons.

The algorithm starts by iterating through the list of all tetrahedrons. Once it finds the first bad tetrahedron, it stops iterating the list of all tetrahedrons and instead starts checking the neighbours of this bad tetrahedron. Every time its neighbour is another bad tetrahedron, the neighbours of neighbour are also checked. This narrows down the search area to a small portion of the diagram.

Number of points	16+120	32+120	64+120	128+120	256+120	512+120	1024+120
Before	0.168s	0.064s	0.114s	0.131s	0.228s	0.556s	1.167s
After	0.044s	0.053s	0.078s	0.119s	0.214s	0.439s	1.014s

Table 2. Benchmarking of the Bowyer-Watson algorithm before and after improving the bad tetrahedron search once the first bad tetrahedron is found.

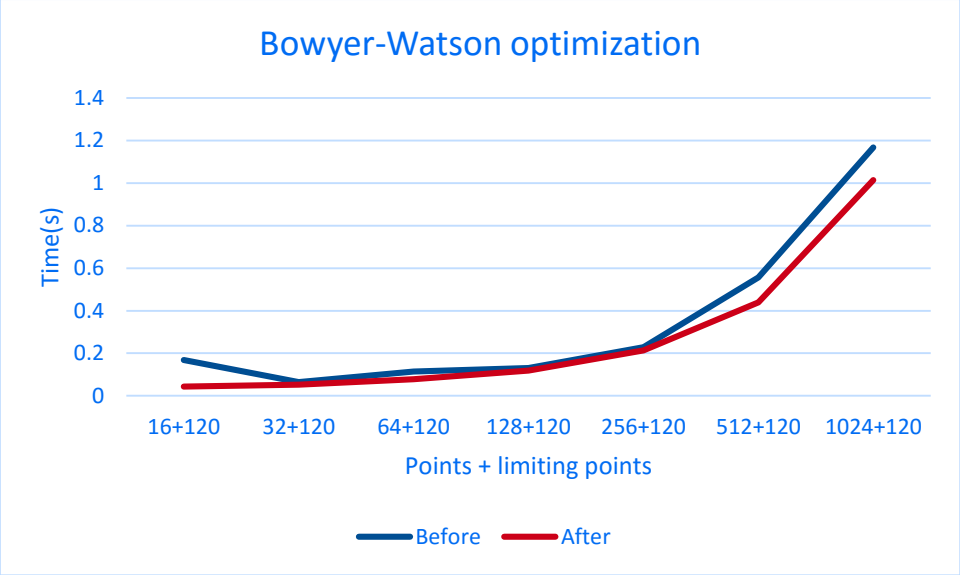

Figure 15. Graphical visualization of the benchmarking.

From Table 2 and Figure 15, it can be seen that a small improvement was gotten by improving the bad tetrahedron check after the first is found.

2.4.3 Improved localization of the first bad tetrahedron

Next step towards faster running time is to speed up the localization of the first bad tetrahedron using the walking algorithm. A modified version from [21] was implemented to achieve this. The algorithm takes a tetrahedron from the triangulation as a starting point. For all the faces of the tetrahedron, it is checked if the new inserted point is on the right of the face. Point is on the right of the face when the face is between the tetrahedron's center of mass and the point. This check can yield a result of multiple faces. A face whose plane is closest to the new point is selected. Tetrahedron on the other side of the face is selected for the calculation. This is repeated until the first tetrahedron whose circumsphere contains the point is found.

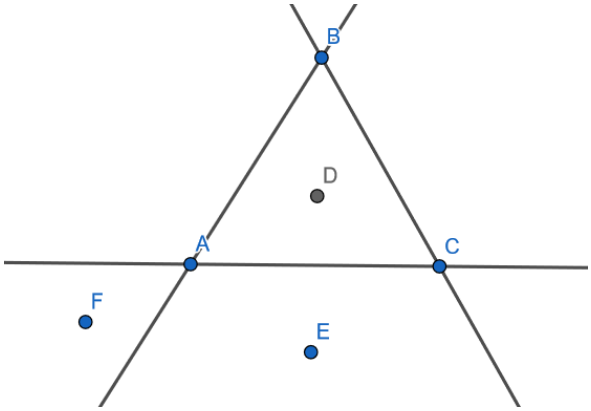

Figure 16. Points E and F are on the right of edge AC. F is also on the right of AB. Both are on the left of BC.

In Figure 16, triangle ABC has a center of mass D. All the edges are extended to infinity for calculating the distance from the point to the edge. Point E is on the right of edge AC but on the left of AB and BC. Hence, the way to move if point E is inserted is to step over edge AC. If point F is inserted, then point F is on the right of AC and AB, but on the left of BC.

This gives two edges to jump over, AB and AC, amongst these, edge closer to point F is chosen. Therefore, edge AB is jumped over.

Number of points	16+120	32+120	64+120	128+120	256+120	512+120	1024+120
Before	0.044s	0.053s	0.078s	0.119s	0.214s	0.439s	1.014s
After	0.05s	0.061s	0.083s	0.13s	0.235s	0.473s	1.063s

Table 3. Benchmarking of the Bowyer-Watson algorithm before and after implementing the walking algorithm for bad tetrahedron localization.

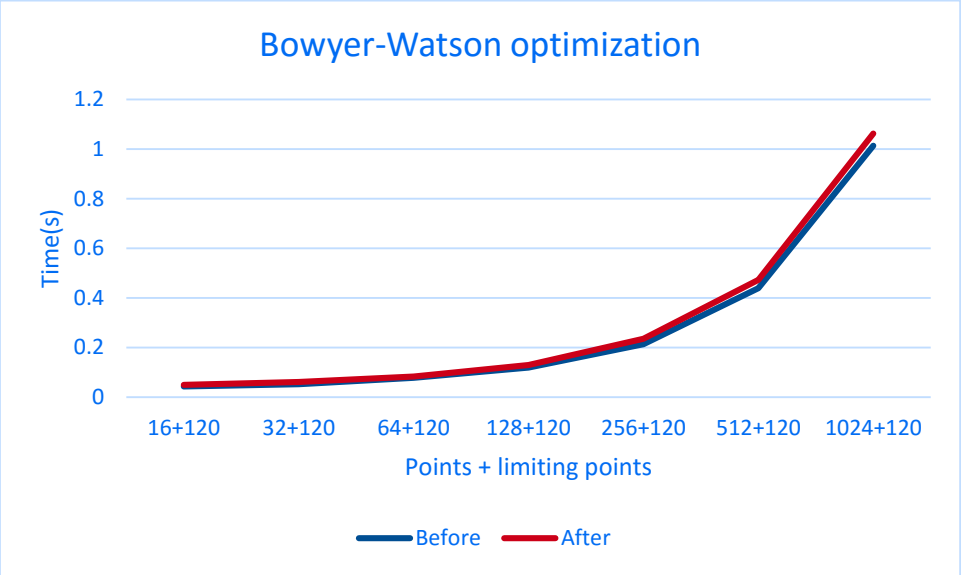

Figure 17. Comparison with and without the walking algorithm.

On Table 3 and Figure 17, it can be noticed that the walking algorithm actually degrades performance slightly. With some additional tests, the benefit of the walking algorithm started to come after 4000 points where it became faster than the original implementation.

3 Representing cave with Voronoi diagram

Before the simulation can be started, the starting cave where the player will be placed in has to be generated. For such a cave, a super-tetrahedron is created and randomly generated points are placed inside of it. For those points, the Delaunay triangulation is calculated using the Bowyer-Watson algorithm which is then converted to Voronoi cells where each initial point corresponds to a seed point of the Voronoi cell.

Figure 18 shows the three stages of cave generation in the 3D simulation. The first stage is the underlying Delaunay triangulation that is used to calculate the Voronoi diagram. On the image, there are 120 control points and 40 points for Voronoi cells. For clarity of images, the super-tetrahedron has been removed, but in the real simulation, it is not removed. The center image has the Voronoi cells of the 40 points and the right image has the 3D objects for the cells.

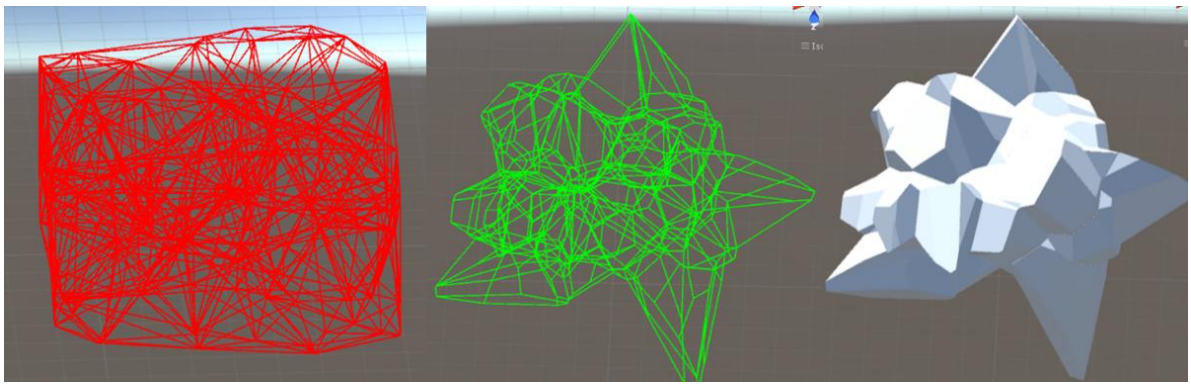

Figure 18. Tree stages of cave generation. Starting from left: Delaunay triangulation, Voronoi Diagram without control points, and Objects created for the Voronoi diagram.

Some of those cells are then marked to be empty and removed to create the cave area for the player. The neighbours of these cells will be converted to 3D geometry which will be rendered. Figure 19 shows the player view from inside the cave.

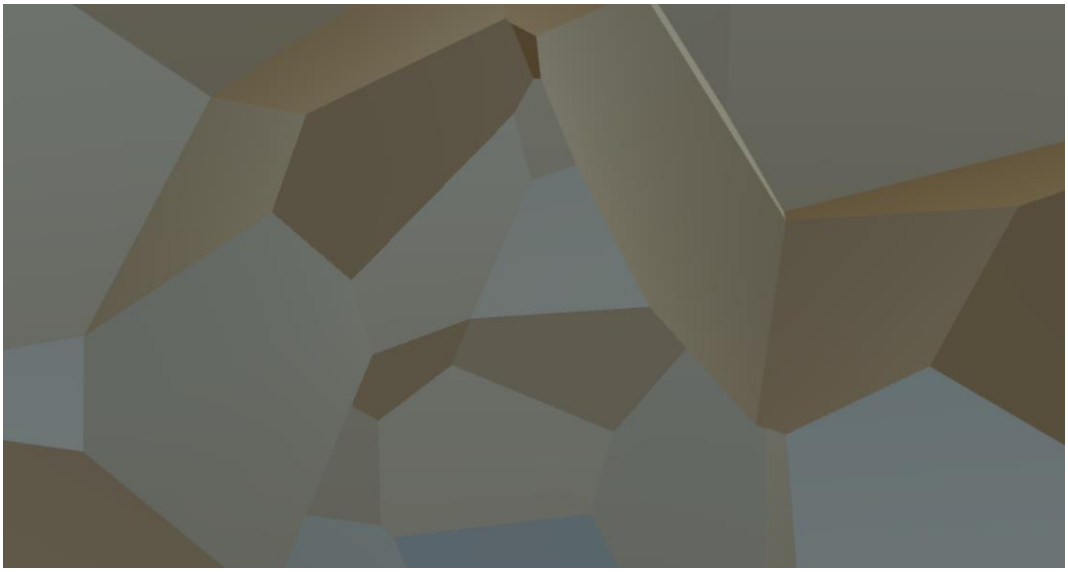

Figure 19. Player view from the inside of the cave.

Voronoi cells are represented as a list of faces. Each face has a set of points that make up that face. When needed, 3D mesh and objects are created to represent the Voronoi cells in the simulation. Once all the faces are processed and 3D objects are created, the Voronoi cell

can be rendered in the scene and the user can interact with it in the simulation. The cells are calculated for only the points that the user can see. All the other points are kept only as a Delaunay triangulation to save space and to make the recalculation of it easier when new points are added.

To turn mathematical Voronoi cells into 3D geometry, they have to be represented as a list of vertices and faces. The cells cannot be represented easily with a single mesh since the player will be inside the cells. While it is possible to create a mesh that only represents the inside of a cave, the problem comes with the Collision detection. The Colliders in game engines cannot have a hole inside of them. The solution is to partition the cave area into many colliders, each collider corresponding to one face of a Voronoi cell. This way, the faces can be rotated and flipped in any way needed and there would be no problems in collision detection.

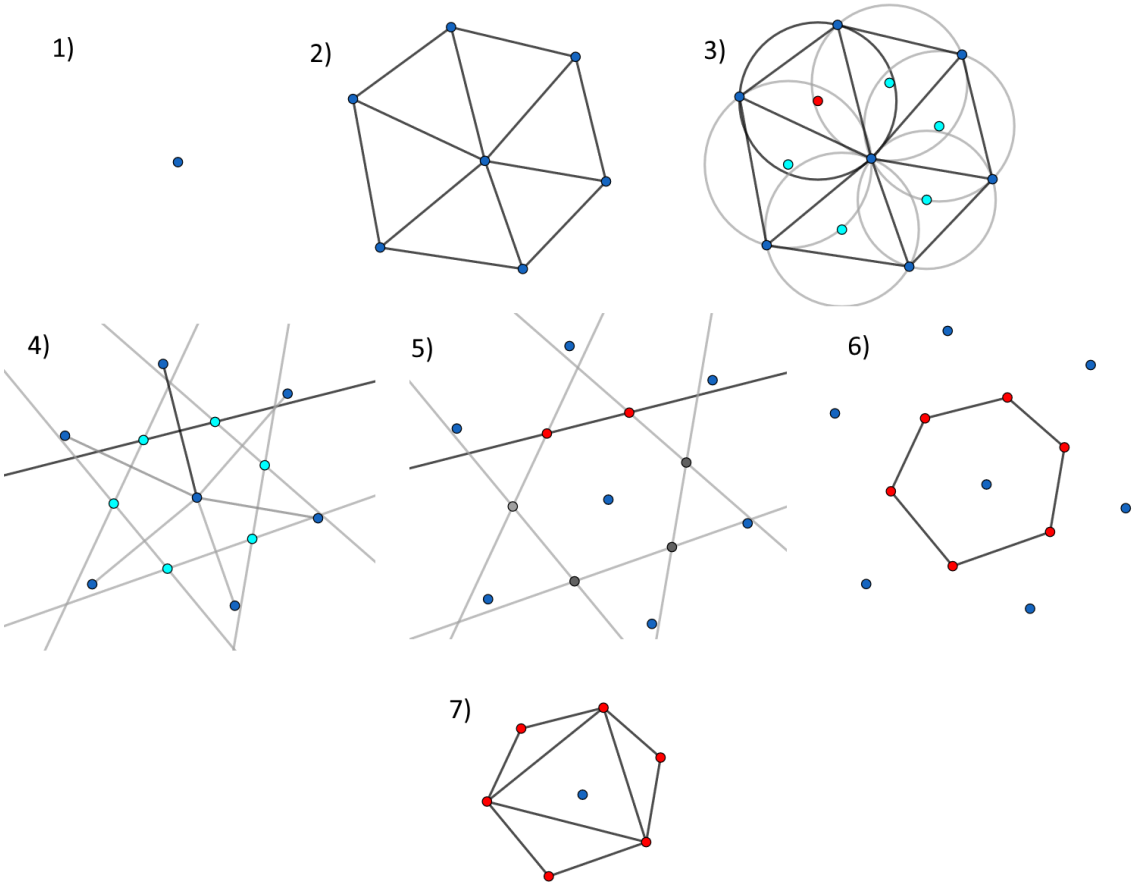

Figure 20. Creating a triangular mesh for Voronoi cell in 2D. 1. Select point for mesh creation 2. Find triangles connected to point 3. Find circumcenters for the triangles 4. Find the perpendicular bisectors of the connected edges 5. Find circumcenters on the bisector 6. Connect the circumcenters to form Voronoi cell 7. Triangulate the circumcenters.

To get the geometrical structure of the Voronoi cell (Figure 20 depicts it in 2D but 3D is similar) out from the list of tetrahedrons, following steps are taken:

1. Choose a seed point for which to create the geometrical structure.
2. Find all tetrahedrons connected to the vertex.
3. Get all the circumcenters of the tetrahedrons.
4. For every edge connected to the vertex, calculate a perpendicular bisecting plane.
5. For every plane find the set of circumcenters that are on the plane.

6. Using 2D Bowyer-Watson algorithm, the circumcenters on the plane can be triangulated.
7. The triangulation will be converted into the mesh for the face.
8. The mesh is used to create the 3D object.
9. Repeat steps 5-8 for every plane.

In steps 2-3, all the circumcenters that make up the cell for the point are found. These points will be used to create the faces for the cell.

All the faces of the cell lie on the perpendicular bisector planes between the vertices. In step 4, these planes are found by calculating the midpoint of the edge between vertices and setting the edge direction as the normal for the plane.

By step 5, all the points for the faces and the planes that define the faces have been found. The points need to be grouped into subsets. Each set will contain points for a single face. These groups are found by finding all the points that are on the corresponding plane. In step 6, the points are triangulated to find the triangular mesh for the face.

In step 7, the triangulation is converted from a list of triangles to a list of vertices and a list of triangle vertex indices (Figure 21). These are required for constructing a mesh to be used in step 8 to create the 3D object.

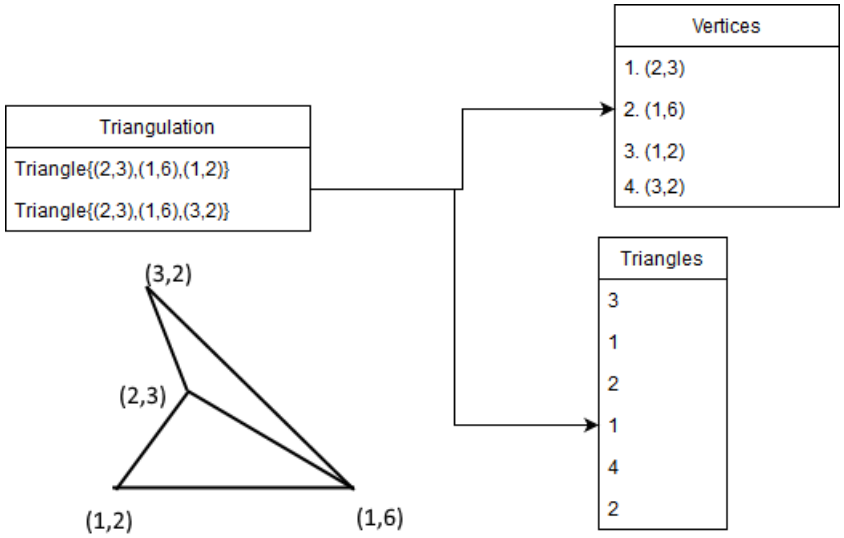

Figure 21. Converting a list of triangles to a list of vertices and a list of triangle vertex indices.

In Figure 21, Triangulation is a list of Triangle objects. These objects cannot be used to create meshes, so the conversion creates a list of all vertices and list of triangle vertex indices. The triangle vertex indices list, Triangles in the figure, has the indices of vertices from the Vertices list.

4 Dynamically expanding the cave

Generating and modifying the Voronoi diagram is only a part of what is required to make cave destruction. When the user removes Voronoi cells, the existing geometry of the cave should change accordingly. This thesis compares two different approaches for altering the Voronoi diagram on user input: pre-generated mining and geometry cutting.

Pre-generated mining adds one layer of non-empty Voronoi cells around the cave area. Whenever the user hits a surrounding cell it will get cut out. At that moment, new Voronoi cells will be generated behind cut out area that will allow the player to continue cutting. This is illustrated in Figure 22. This approach will be called pre-generated mining.

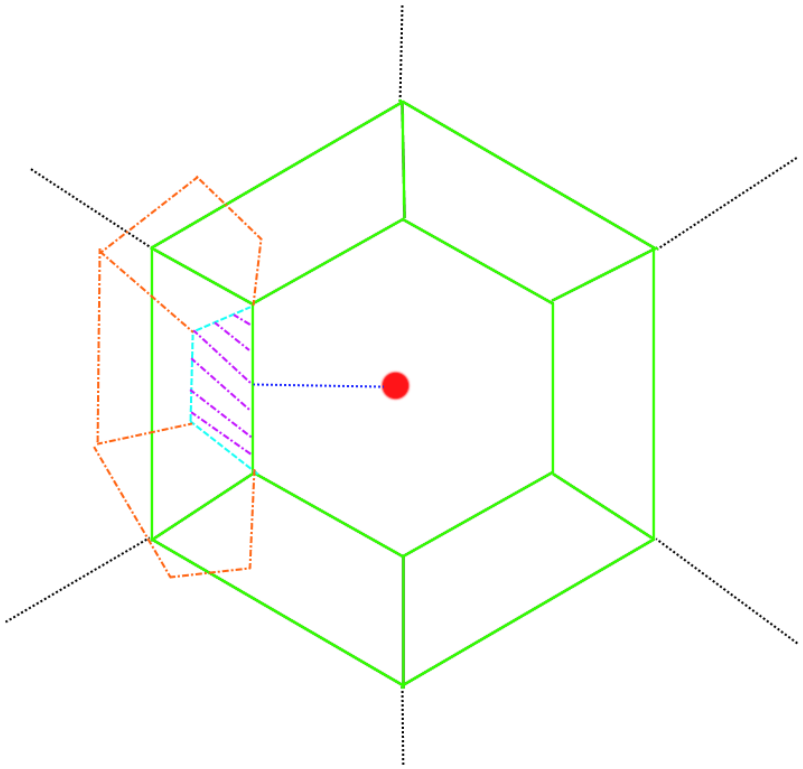

Figure 22. Red dot is the player. Blue dotted line represents the location of mining. Green lined regions represent the existing cells of the cave. Light blue dotted line is the piece that is removed. Orange dotted lines represent the new cells created behind the removable piece. This image is in 2D and from above.

Geometry cutting, on the other hand, adds new Voronoi cells on the input point and cuts them out of the existing geometry. Each hit creates a smaller Voronoi-cell-shaped hole in the wall. This approach is illustrated in Figure 23 where the red circle represents the player inside a cave marked with a green border. The blue line represents the raycast used to detect the mining location. The light blue area is the new Voronoi cell that is inserted by the mining action and the yellow striped area will be cutout from the environment to increase the area where the player can move around.

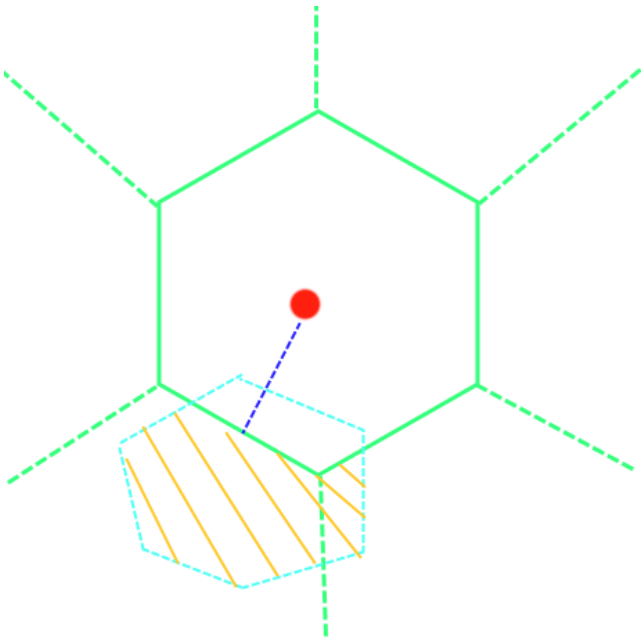

Figure 23. The red dot is the player. The blue dotted line represents the location of mining. The green lines and the dotted lines represent the existing cave and the diagram. The light blue dotted line is the cell that will be cut out and the striped area is the area to be removed. This image is in 2D and from above.

For such an approach, a method for clipping geometry with another geometry is needed which is described in chapter 4.2.

4.1 Pre-generated mining

The visible cave consists of parts resembling rocks that are actually individual Voronoi cells. When the user mines the surrounding, the rock that was hit will be removed. To make sure that there will be new Voronoi cells to cut out, new Voronoi seeds have to be added spherically around the seed point of the cut out cell (Figure 24). This keeps growing the diagram dynamically and the player can extend the cave away from the initially generated area. These points will be filtered so that when they are added to the Voronoi diagram, they can only affect the regions of the diagram that the user cannot see (Figure 25). Once filtered, they can be added to the diagram and the cell of the seed point can be removed (Figure 26).

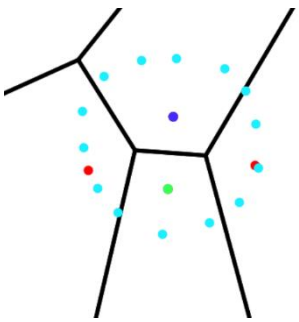

Figure 24. New light blue points generated around blue seed. Green seed's cell has been mined out. Red seed cells are the pre-existing cells of the environment.

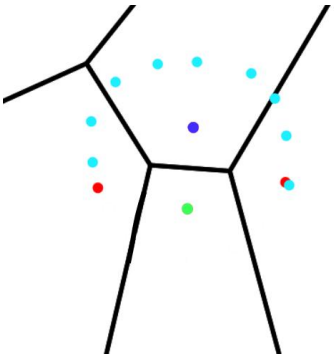

Figure 25. Points from Figure 24 filtered so that they will not affect the green mined out area.

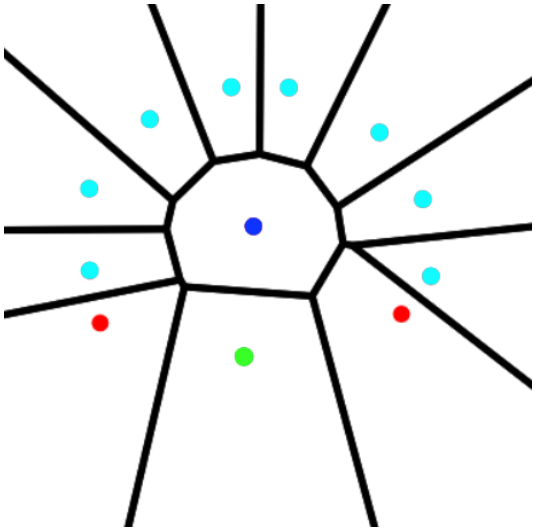

Figure 26. Points from Figure 25 inserted into the diagram, and the blue seed points cell can now be removed.

This approach creates an issue of how to filter the points that can be added to the diagram. The first idea is to measure the distance between the unchangeable cells and the cell where our new point resides. An unchangeable cell is a cell that has been mined out. If the new point is closer to the unchangeable cell than the seed of the residing cell, then the cell created by the new point will cut the unchangeable cell and it is not wanted. In this case, the new point is moved to the residing cell's seed and the issue is resolved. But this condition is not enough. New points can be added, that are further away than the removed sell's seed, but still cut the unchangeable cell. To fix that, the seeds of the neighbour cells of the removed cell have to be checked too.

The distance to the unchangeable cell is compared to the distances of the neighbouring seeds. If the new point is closer to the unchangeable cell than the neighbours, then the new point does fit the diagram. However, checking against all the neighbours is not the correct approach. Some neighbours are farther away from the new point than they are from the seed of the residing cell. These neighbours will not affect what the cell of the new point will look like and would cause only false positives. For example, there can be a point that will not create a cell that intersects with the unchangeable cell, but it is closer to the unchangeable cell than the neighbour. Without this restriction, during mining, the walls can change in unexpected ways creating new pieces out of nothing and blocking off previously opened ways.

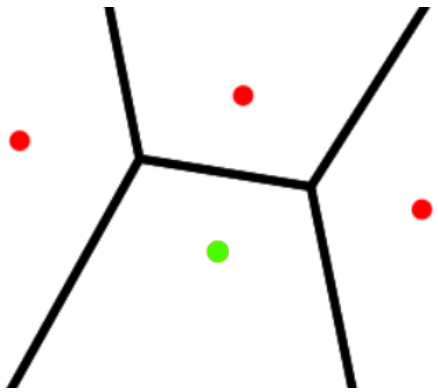

Figure 27. Before inserting a new point.

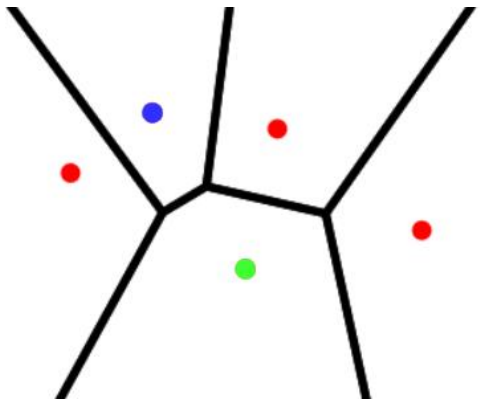

Figure 28 Bad addition of a new point.

Figure 27 has the diagram before inserting a point where the green point is the seed point of the unchangeable cell. In Figure 28 the blue point is the added point. It can be seen that the blue point is farther away from the green point than its two red neighbours, but it still forms an edge with the green point.

Nevertheless, even if all these conditions are fulfilled, we can still get wrong results. As it turns out, the best check can be made using the Delaunay triangulation. Each edge connecting two points in the triangulation represents an edge (face in 3D) between two seeds in the Voronoi diagram. New points that are added cannot make an edge with the seed of the unchangeable cell in the Delaunay's triangulation. To check it, a part of the Bowyer-Watson algorithm can be used. For each new point to be added, all the tetrahedrons whose circumcenter contains the new point will be found. If any triangle (tetrahedron) has a vertex as the unchangeable cell's seed, then the point is disregarded.

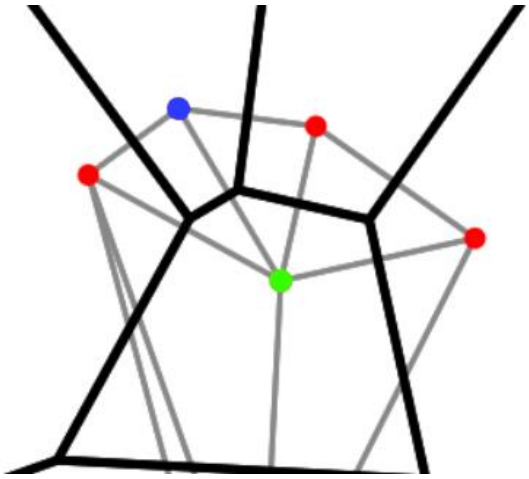

Figure 29. Delaunay triangulation of the bad addition.

Figure 29 shows the underlying Delaunay triangulation and there exists an edge between the blue and the green point. In the conversion to Voronoi diagram, this edge will become a Voronoi edge and this is unwanted.

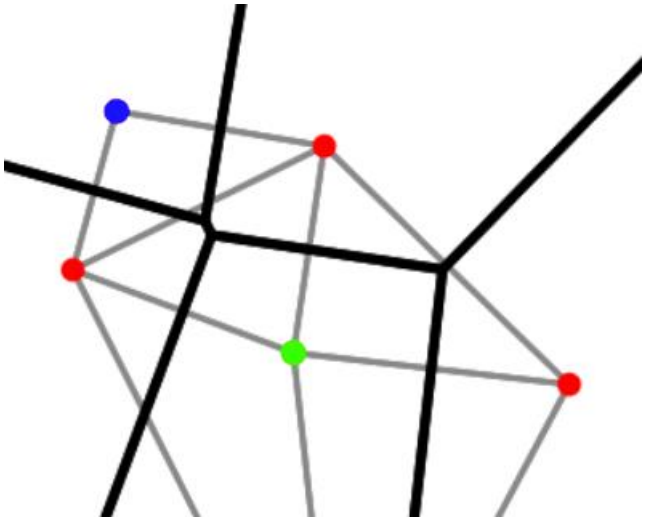

Figure 30. Good addition of the new point.

Figure 30 has the correct case of point insertion. Inserted blue point does not create a new edge with the green point. This is because the blue point is farther away than in Figure 29.

This means that the unchangeable cell will not be affected by the insertion and the algorithm can continue. With this addition, the unchangeable cell will always remain unchanged.

Since the cave is represented using faces as described in chapter 3 but pre-generated mining forms the cave out of rocks, the faces representing each rock can be merged together. This also speeds up the rendering [22]. Adding a convex collider to every rock also simplifies the collision calculations that the game engine needs to make.

4.2 Geometry cutting

In nature, when something breaks, pieces break off from the larger object leaving an empty area where the pieces used to be. This piece can be put back into the hole and it would fit almost perfectly. However, there might be some smaller pieces missing, so it could leave some cracks. To simulate the same behaviour in the simulation, pieces need to be cut off from previously existing geometry.

Cutting off parts from the cave wall can be looked at as adding empty spaces to the cave itself. The mesh representing the cave can be seen as a mesh representing the empty space but with inverted faces so we could see the cave walls from the inside instead of the outside of the cave. When mining, cut new pieces are joined together with the cave area. Following chapter looks at two ways to do geometry cutting.

4.2.1 Constructive solid geometry

The first way is through *constructive solid geometry* (CSG) [23]. Computer-aided design software uses CSG to do geometrical operations on objects. CSG is a technique where complex shapes are created from primitive shapes using Boolean operators.

A primitive shape can be represented as a procedure or a function. This would make CSG a technique in which simple procedures or functions are combined together to produce more complex procedures and functions.

Commonly, the set of simple objects is limited to cylinders, prisms, pyramids, spheres, and cones. The exact set depends on the implementation of the CSG. These primitives are combined by using a set of Boolean operations. Normally, the operations are union, difference and intersection (Figure 31, Figure 32 and Figure 33).

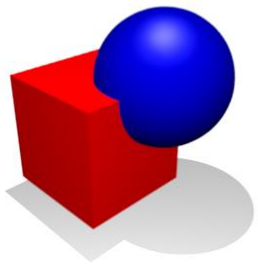

Figure 31. Union of cube and a sphere [24].

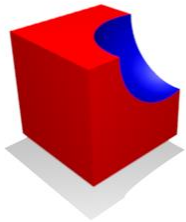

Figure 32. Difference. Subtracting sphere from cube [25].

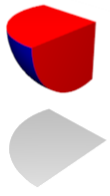

Figure 33. Intersection of cube and a sphere [26].

CSG would be a good solution for finding the union of two Voronoi cells, but the implementation would get difficult and it might not be able to work in real-time. Furthermore, Voronoi cells are not easily represented by a procedure or a formula. In Figure 34, it can be seen that no cell repeats and they are all unique. In 3D, the variety is even larger. Besides, our representation splits the shapes into a set of faces and CSG algorithms expect full 3D

objects without vertex duplicates. This makes our representation unsuitable for CSG and would force a conversion from our representation to the one CSG needs. After calculating the unions using CSG, it has to be converted back to our representation.

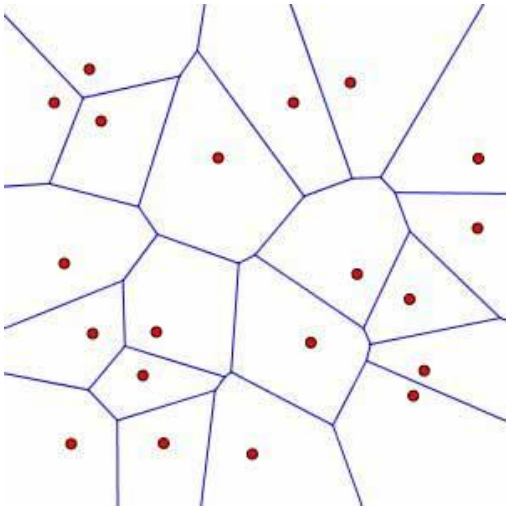

Figure 34. Different Voronoi cells in 2D.

4.2.2 Sutherland-Hodgman algorithm

The Sutherland-Hodgman [27] algorithm is an algorithm for clipping different shapes with a convex shape. Clipping is the process of cutting one geometrical shape with another, resulting in the removal of parts from the original shape that were outside of the clipping shape. This can be used to find parts of shapes within other shapes. This algorithm can also be reversed, then it will find the part that is outside of the shape instead. Figure 35 illustrates the process of clipping.

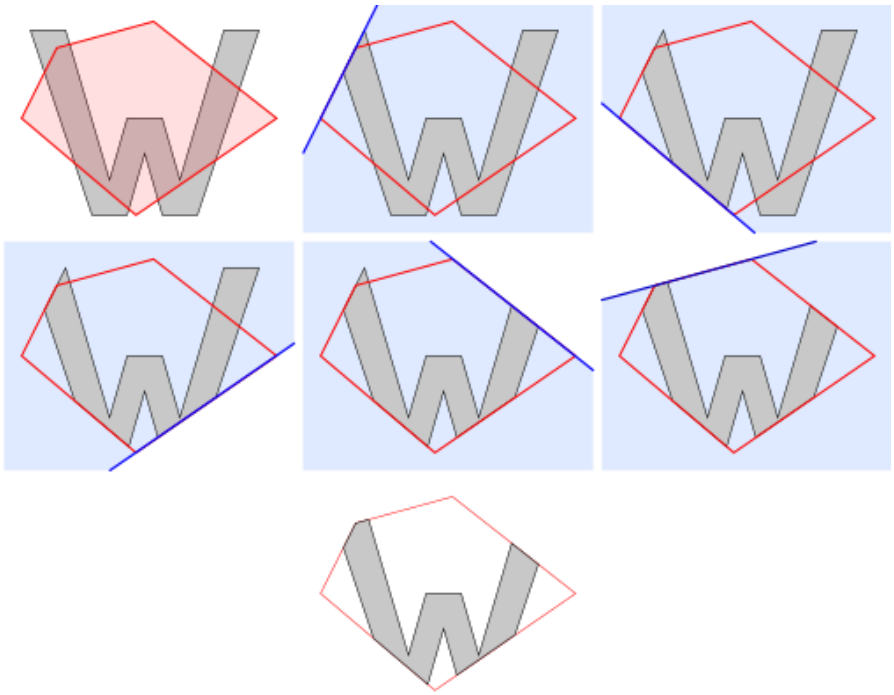

Figure 35. Clipping concave shape with a convex shape [28].

Before the algorithm can be used, vertices of the shapes need to be ordered. The order of the vertices defines how the shapes are constructed. Figure 36 demonstrates the ordering on a polygon in counter-clockwise order.

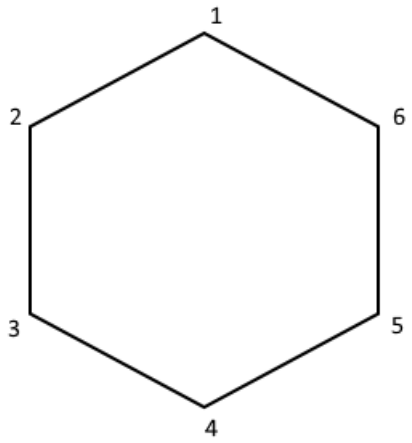

Figure 36. Ordering of vertices on a polygon.

The Sutherland-Hodgman algorithm consists of the following steps:

1. Start with the lists of ordered vertices of both shapes.
2. Take the edge of the clipping polygon and extend it to infinity.
3. Remove all the vertices that are on the other side of the edge.
4. If any edge of the subject polygon intersects with the extended edge, calculate the intersection point.
5. Add the vertices that are not removed and the intersection points of the edges to a separate list in the order of discovery.
6. Replace the subject polygon vertex list with the new list of vertices.
7. Repeat steps 2-6 on all the edges of the clipping polygon.

In Figure 35, the clipping process is illustrated. On the first image, there is a ‘W’ shape as the subject that will be clipped and a pentagon as the clipping polygon that will clip it. An edge of the pentagon is extended to infinity on both ends and everything on the other side of the edge is removed. This is repeated with all the edges of the pentagon. The final image has the result of the clipping process where the ‘W’ has been clipped with the pentagon.

Because Voronoi cells are always convex, the result of the clipping process between two Voronoi cells is also a convex polygon [29]. Using the remaining edges from the clipped polygon, the clipping polygon can also be clipped. When clipping a convex shape with another convex shape, the resulting polygon is the intersection of the two shapes. The intersection of two Voronoi cells can be used to join the Voronoi cells together by adding the points of intersection to both of the cells and then removing the points that are inside the other cell.

The Sutherland-Hodgman algorithm can be extended to 3D space by using the faces of the shapes instead of the edges. This makes the calculation more complex because plane-plane intersections are more difficult to calculate than line-line intersections (Appendix III).

4.3 Voronoi cell cutting

Voronoi cells in the simulation will be cut using the 3D version of the Sutherland-Hodgman algorithm described previously. Cutting and merging the Voronoi cells is a multistep process:

1. Finding the cutlines between the two cells.
2. Re-triangulating the faces using the vertices of the cutlines.
3. Removing triangles that are no longer wanted.

The first step finds the cutlines where the cutting needs to be done. The second step prepares the polyhedral for cutting and the final step does the cutting.

This can further be extended to be used when more than one cell is created at the same time. A single hit will create a set of cells. All of the cells will be cut with the previous existing cells and all the cell faces between the new cells are removed. This will result in a large more natural hole.

4.3.1 Finding the cutlines

Cutlines are used to cut pieces off the Voronoi cell. They are found using the Sutherland-Hodgman algorithm. The algorithm checks every face of the first cell against every face of the second cell. Areas, where the faces of different cells intersect, are called cutlines.

The cutline finding algorithm works as follows:

1. Take two Voronoi cells to calculate the intersection.
2. Pick a face on the first cell.
3. Pick a face on the second cell.
4. Calculate the intersection line between the planes (Appendix III) where the faces reside.
    a. If no intersection line exists, take the next face on the second cell and try step 4 again.
5. Calculate the intersection points between the face on the first cell and the intersection line of the planes.
    a. If no points exist, take the next face on the second cell and go to step 4.
6. Using the intersection points, create the intersection line on the face.
7. Calculate the intersection points between the face on the second cell and the intersection line of the planes.
    a. If no points exist, take the next face on the second cell and go to step 4.
8. Using the intersection points, create the intersection line on the face.
9. Find the overlapping line segments between the intersection line on both faces.
10. Take the next face on the second cell and go back to step 4.
    a. If all faces on the second cell have been checked, pick the next face on the first cell and go to step 3.

This algorithm will result in a list of cutlines along which the cells can be cut. A cutline exists between two faces, but finding the intersection line of two planes is a lot easier than finding the intersection line between two polygons in 3D space. This is why the planes of the faces are used to calculate the intersection line. An intersection line and a face lie on the same plane, giving the ability to simplify finding the intersection points to a 2D problem (Appendix III). If one face has no intersection points with the intersection line, then the two faces are not intersecting. Otherwise, the intersection points on a face are joined together to find the intersection line on the face. These intersection lines on the faces are then compared

to each other to find overlapping segments. These segments will be the cutlines. The result of the algorithm can be seen in Figure 37.

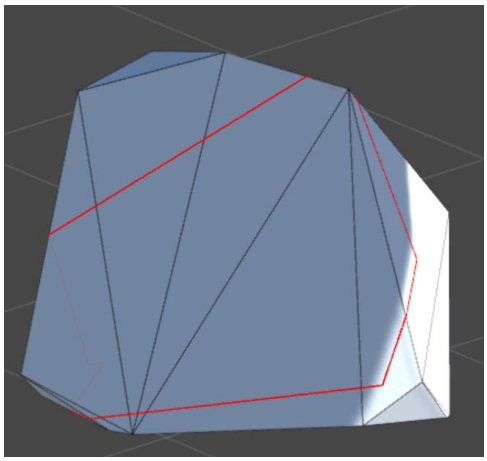

Figure 37. Cutlines marked red on a cell face.

4.3.2 Re-triangulating the face

Before the face can be re-triangulated, all the points of the new face need to be known. The points of the cutlines are added to the face, and the existing points of the face need to be filtered because there can be points that are within the cutlined area. Removing these points would make removing triangles faster, because adding new points to the triangulation would not have to take into account the point and triangles connected to it. Furthermore, removing these points removes some of the triangles which also speeds up the removal process.

Since the intersection of two convex shapes is always a convex shape, the Sutherland-Hodgman algorithm can be used to find all the points inside the cutlined area. Once the points are found, they are removed from the face and rest of the points will be triangulated using the 2D Bowyer-Watson algorithm.

Figure 38 shows the result of the filtering and triangulation. It can be seen that there are holes in the triangulation. This is the result of removing the vertices within the cutline area. However, this is not a problem, since the goal is to cut a hole along the cutline, and the missing triangles are within the cutlined area.

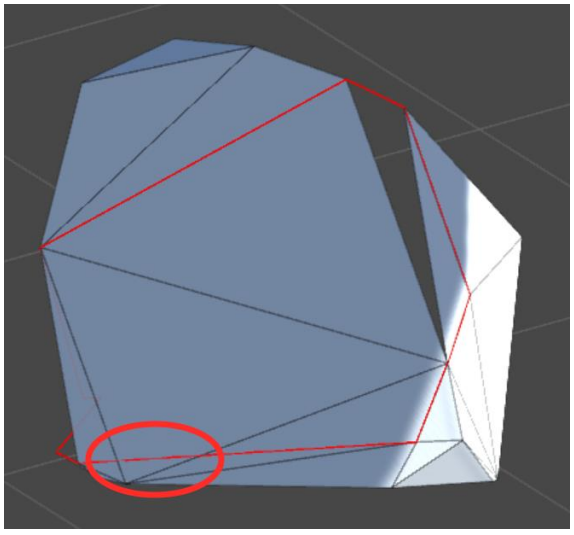

Figure 38. New triangulation after filtering and adding cutline points.

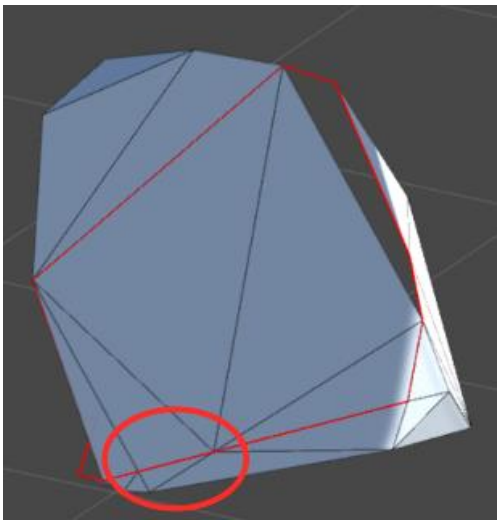

Figure 39. Triangle-Cutline intersection points added.

In Figure 38, it can also be seen that some triangles intersect with the cutlines. This will prevent the removal of triangles and the hole will not follow the cutlines. To fix the problem, intersection tests on the new triangles and cutlines are performed and the intersection points are added to the triangulation. Figure 39 shows the result after the intersection points are added into the triangulation.

4.3.3 Removing unwanted triangles

After the triangulation is completed and there are no triangles intersecting with cutlines, the unwanted triangles can be removed. Triangles are unwanted if they would be inside the other shape. To make it look like a union of the two shapes, there cannot be triangles in the area that is inside the joined shape. In the simulation, these triangles will be on the cave wall, and the hole would be created behind the triangles. If they were to remain, they would be covering the hole.

The simplest case for removing triangles is when these triangles form a convex shape like shown in Figure 40. These cases are easy to handle and there is not much work required to be done. In this situation, all the triangles whose three vertices lie on the cutline are removed. Unfortunately, these cases are not very common and most of the time more complex situations arise.

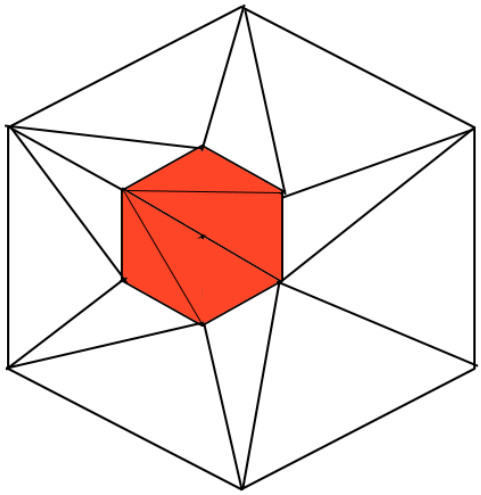

Figure 40. Red triangles to be removed form a convex shape.

Figure 41, Figure 42 and Figure 43 show some more common cases. These are more complex and a simple check, for instance, if points are on the cutline, a simple check like before will not work. A more general approach is needed.

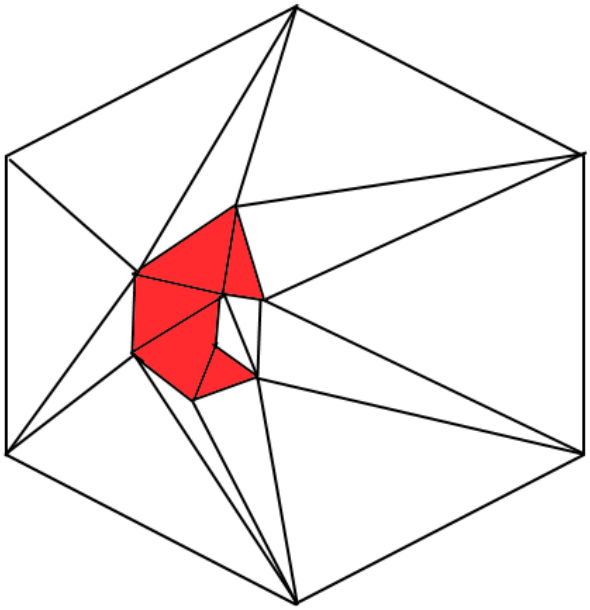

Figure 41. Concave area to be removed.

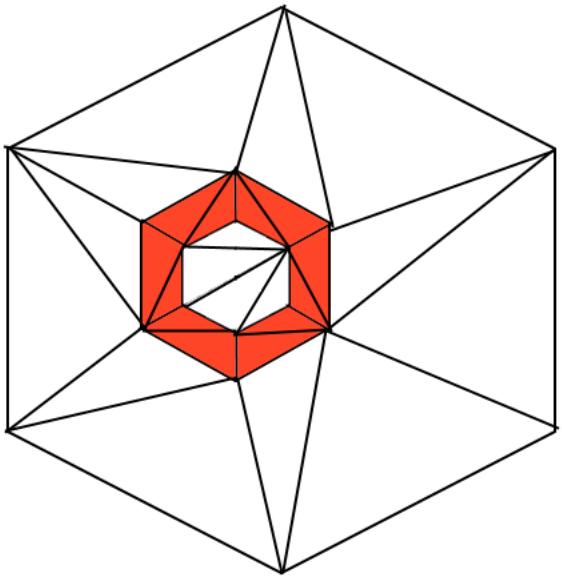

Figure 42. Area not to be removed surrounded by area to be removed.

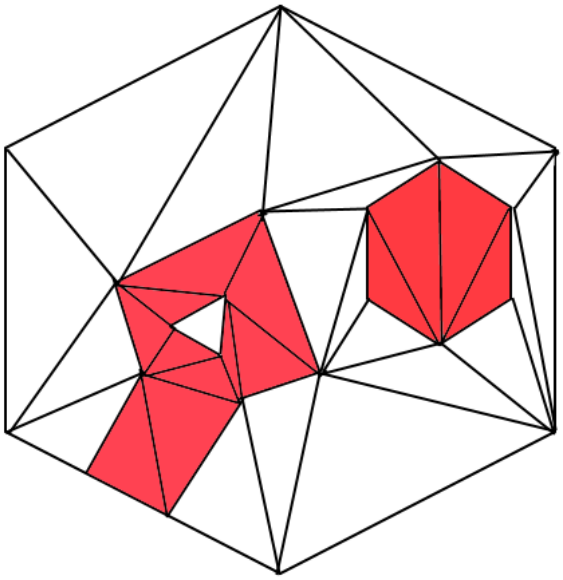

Figure 43. Multiple areas to be removed.

The more complex case can be simplified into a problem of coloring graph vertices in two colors. The graph will represent triangulation, where each triangle is represented by a vertex and vertices are connected by an edge if the corresponding triangles share an edge (Figure 44). To start coloring, a random vertex is chosen and a color is given to it. Next, the graph is traversed by moving along the edges to other vertices. Whenever we move over an edge that represents a cutline in the triangulation, we change the color.

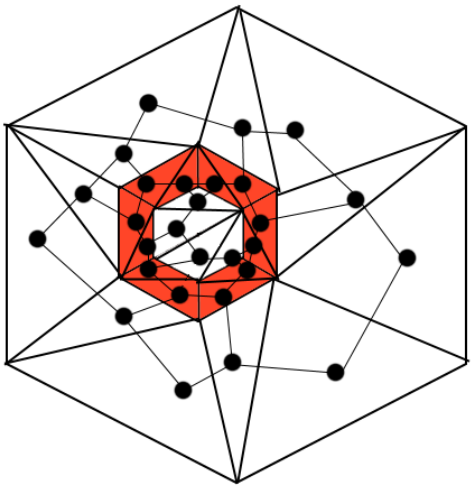

Figure 44. Triangulation and the corresponding graph.

Once the graph is colored, it must be determined what color represents the triangles that should be removed. This can be done by keeping track of the outermost edges of the shape. The color of the triangle connected to this edge will represent the color of the triangles that will remain. All the others will be removed. The end result will be a Voronoi cell that has a piece cut out from it (Figure 45).

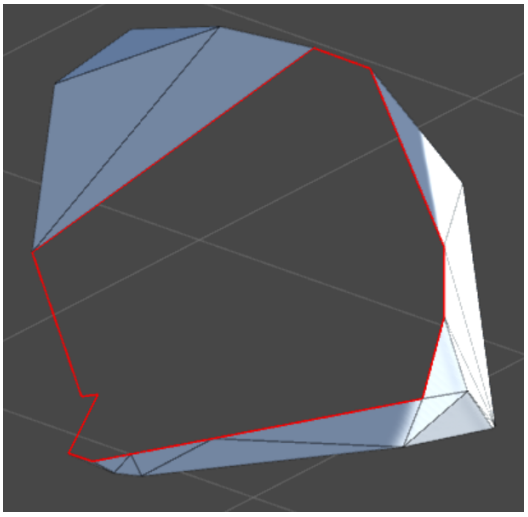

Figure 45. The result of triangle removal. The cell is see-through in the cut out hole because the rendering process does not render the backside of the triangles.

4.4 Comparison of pre-generated mining and geometry cutting

Pre-generated mining is fast and simple and if the initial Voronoi cells are generated small enough, it looks quite realistic. However, very small cells require more calculation time and by adding even more points during the runtime will make the simulation gradually slower. If the cells are left larger, then it works fast but it impacts how realistic the simulation looks.

Geometry cutting will give a more realistic look to the mining process since pieces are broken off from the existing cave wall instead of removing an entire cell. But this comes with an impact on performance. While pre-generated cutting can give good update speeds for quite a while, geometry cutting will be much slower. Since the objects are needed to cut, they need to be cut against other objects in the scene and if there are many objects, there will be many cutting calculations. Additionally, the calculation accuracy gets lower the further away the player goes from the 3D space origin. This is caused by the usage of float number types by the game engine. Float gives 7 significant digit accuracy. This means that at value 10 the accuracy is 5 decimal places but at 10 000 the accuracy is two decimal places. This restriction could be bypassed by using higher accuracy numbers for calculations and later converting to floats. But this only postpones the issue and does not solve it. Additionally, the calculation process needs to be written in an approximate form, not in the exact mathematical form, since computers cannot do mathematics in absolute accuracy.

In conclusion, both approaches are good depending on the resources available and the design choices in the game world. Pre-generated mining is faster and simpler but lacks the realism of geometry cutting. Geometry cutting is more realistic, but without some heavy optimisations and adding robustness to the floating point calculations, it will run slow and produce errors, like mining faces.

5 Conclusion

In this thesis, a proof of concept 3D cave destruction simulation was created using real-time calculated Voronoi diagrams to model the cave. As the Voronoi diagrams are often found in nature, it was an appropriate choice for modeling the rock geometry. Three algorithms were described for generating the Voronoi diagram for a set of points. These algorithms were the Brute Force algorithm, the Fortune's algorithm and the Bowyer-Watson algorithm. Amongst those algorithms, the most suitable for dynamic cave generation was the Bowyer-Watson algorithm, because it allows adding more points later without recalculating the entire diagram. This algorithm was also optimized further using the walking algorithm and improving the face validity check. After these optimizations, it could easily be used in real-time.

For the dynamic manipulation of the cave, two approaches were proposed. The first approach pre-calculates all the Voronoi cells that bound the cave area. Whenever the player cuts out some of these cells, the new cells will be generated behind it so there will always be more cells to mine. This algorithm worked well and was quite fast; however, because all the cells are pre-generated, they will not look very realistic. To solve that problem, another approach was proposed which is cutting out the new cells from the existing cave geometry. This approach allows creating new cells exactly to the point where the user presses and therefore will have a more realistic look. The thesis compares two geometry manipulation algorithms that allow the joining of new pieces to an existing region. Because the cave can also be represented as an empty area turned inside out, these techniques can be used to cut off parts of the geometry as well. Since the performance and accuracy became an issue with these methods, they were not suitable for real-time use.

5.1 Future work

The 3D simulation is still a proof of concept and more work is needed for it to be fully usable in games and larger simulations. The rock sizes currently in the simulation are quite large but would look a lot better if the smaller rocks would appear more frequently than larger ones.

Currently, the rocks in simulations are very rigid and polygonal. This makes them look unnatural as perfect polygonal shapes rarely appear in nature. Some irregularities and rounding added to the edges would make them look decent and more natural.

6 References

[1] Matthias, Müller. Realtime dynamic fracture with volumetric approximate convex decomposition http://matthias-mueller-fischer.ch/publications/fractureSG2013.pdf (18.05.2018)

[2] Battlefield Bad Company 2 https://www.battlefield.com/games/battlefield-bad-company-2 (12.05.2018)

[3] Minecraft Homepage https://minecraft.net/en-us/ (12.05.2018)

[4] Voronoi, Georges. "Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxième mémoire. Recherches sur les parallélloèdres primitifs.." *Journal für die reine und angewandte Mathematik* 134 (1908): 198-287. https://eudml.org/doc/149291 (07.01.2018)

[5] Image of Voronoi diagram using Euclidian distance https://upload.wikimedia.org/wikipedia/commons/5/54/Euclidean_Voronoi_diagram.svg (07.01.2018)

[6] Image of Voronoi diagram using Manhattan distance https://en.wikipedia.org/wiki/Voronoi_diagram#/media/File:Manhattan_Voronoi_Diagram.svg (07.01.2018)

[7] Classification problem: k-Nearest neighbor algorithm http://www.data-machine.com/nmtutorial/classificationproblemknearestneighboralgorithm.htm (07.01.2018)

[8] Victoria, Australia School districts via Voronoi diagram http://melbourneschool-zones.com/ (07.01.2018)

[9] Image of giraffe http://endextinction.org/sites/default/files/2017-08/giraffes1.jpg (01.05.2018)

[10] Image of a leaf. https://tomaszjaniak.files.wordpress.com/2011/04/169_lisc_l.jpg (01.05.2018)

[11] Image of dried an cracked earth https://i.pinimg.com/originals/d0/46/1c/d0461cc5961d7317fcca71aba58ce5dc.png (01.05.2018)

[12] *Delaunay, Boris (1934). "Sur la sphère vide". Bulletin de l'Académie des Sciences de l'URSS, Classe des sciences mathématiques et naturelles.* http://galiulin.narod.ru/delaunay_.pdf (07.01.2018)

[13] Image of Delaunay triangulation https://upload.wikimedia.org/wikipedia/commons/thumb/d/db/Delaunay_circumcircles_vectorial.svg/280px-Delaunay_circumcircles_vectorial.svg.png (07.01.2018)

[14] „Brute force method“ http://www.grasshopper3d.com/forum/topics/voronoi-brute-force (07.01.2018)

[15] Steven Fortune, „A Sweepline Algorithm for Voronoi Diagrams“(1987) http://www.wias-berlin.de/people/si/course/files/Fortune87-SweepLine-Voronoi.pdf (07.01.2018)

[16] Kutskir, Ivan. „Fortunes Algorithm and implementation“ http://blog.ivank.net/fortunes-algorithm-and-implementation.html (07.01.2018)

[17] Cuk, Roman. „Construction of Voronoi diagrams using Fortune's method: A look on an Implementation“ http://old.cescg.org/CESCG99/RCuk/index.html (07.01.2018)

[18] „Fortune’s Algorithm and Voronoi diagrams“ http://www.cs.wustl.edu/~pless/546/lectures/L14.html (01.04.2018)

[19] D. F. Watson; Computing the *n*-dimensional Delaunay tessellation with application to Voronoi polytopes, *The Computer Journal*, Volume 24, Issue 2, 1 January 1981, Pages 167–172, https://doi.org/10.1093/comjnl/24.2.167 (07.01.2018)

[20] A. Bowyer; Computing Dirichlet tessellations, *The Computer Journal*, Volume 24, Issue 2, 1 January 1981, Pages 162–166, https://doi.org/10.1093/comjnl/24.2.162 (07.01.2018)
[21] P.J.C Brown; C.T.Faigle. „A robust efficient algorithm for point location in triangulations“ http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-728.pdf (12.04.2018)
[22] Optimizing graphics performance https://docs.unity3d.com/Manual/Optimizing-GraphicsPerformance.html (21.05.2018)
[23] *Foley, James D. (1996), "12.7 Constructive Solid Geometry", Computer Graphics: Principles and Practice, Addison-Wesley Professional, pp. 557–558, ISBN 9780201848403* (20.05.2018)
[24] Union of two 3D objects. https://en.wikipedia.org/wiki/Constructive_solid_geometry#/media/File:Boolean_union.PNG (20.05.2018)
[25] The difference of two 3D objects. https://en.wikipedia.org/wiki/Constructive_solid_geometry#/media/File:Boolean_difference.PNG (20.05.2018)
[26] The intersection of two 3D objects. https://en.wikipedia.org/wiki/Constructive_solid_geometry#/media/File:Boolean_intersect.PNG (20.05.2018)
[27] Sutherland, Ivan; Hodgamnn, Gary. „Reentrant polygon clipping“ https://dl.acm.org/citation.cfm?id=360802 (07.01.2018)
[28] Image of Sutherland-Hodgman clipping algorithm https://upload.wikimedia.org/wikipedia/commons/5/53/Sutherland-Hodgman_clipping_sample.svg (07.01.2018)
[29] Sigur (https://math.stackexchange.com/users/31682/sigur), Is the area of intersection of convex polygons always convex?, URL (version: 2012-12-31): https://math.stackexchange.com/q/268255 (17.05.2018)
[30] Insertion of the first point in Bowyer-Watson https://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson_algorithm#/media/File:Bowyer-Watson_0.png (21.05.2018)
[31] Insertion of the second point in Bowyer-Watson https://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson_algorithm#/media/File:Bowyer-Watson_1.png (21.05.2018)
[32] Insertion of the third point in Bowyer-Watson https://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson_algorithm#/media/File:Bowyer-Watson_2.png (21.05.2018)
[33] Insertion of the fourth point in Bowyer-Watson https://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson_algorithm#/media/File:Bowyer-Watson_3.png (21.05.2018)
[34] Insertion of the fifth point in Bowyer-Watson https://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson_algorithm#/media/File:Bowyer-Watson_4.png (21.05.2018)
[35] Remove edges with vertices on super-triangle in Bowyer-Watson https://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson_algorithm#/media/File:Bowyer-Watson_6.png (21.05.2018)
[36] Line Plane intersection, Plane Plane intersection. http://geomalgorithms.com/a05-_intersect-1.html (07.01.2018)
[37] Image of the line-plane intersection. http://www.xbdev.net/maths_of_3d/collision_detection/line_with_plane/images/sketchA.gif (07.01.2018)

Appendix

I. Bowyer-Watson point insertion

The process of creating a Delaunay triangulation using five points is shown in the following figure.

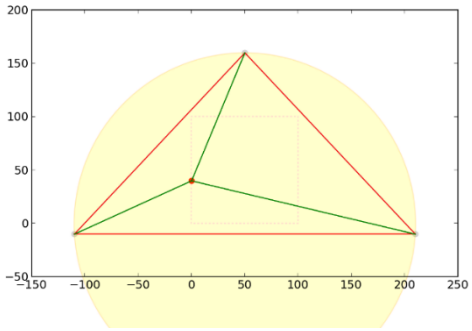

Figure 46. Insertion of first point [30].

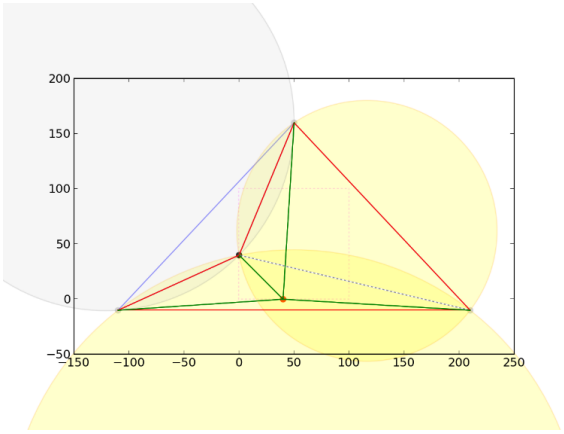

Figure 47. Insertion of second point [31].

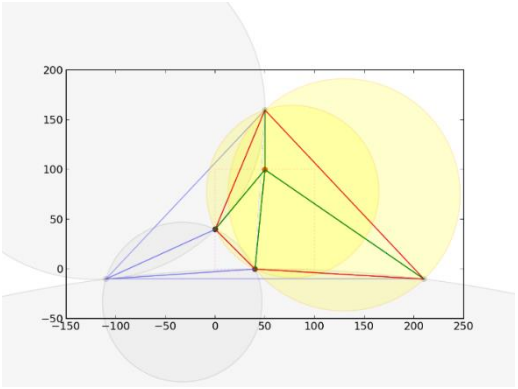

Figure 48. Insertion of third point [32].

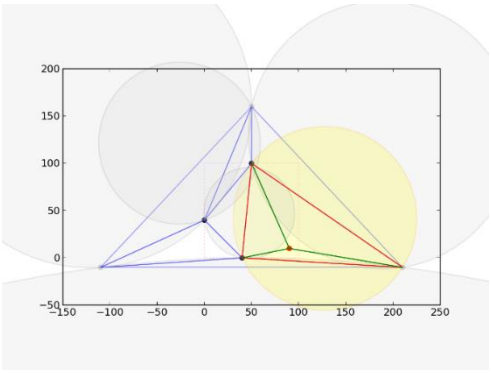

Figure 49. Insertion of 4th point [33].

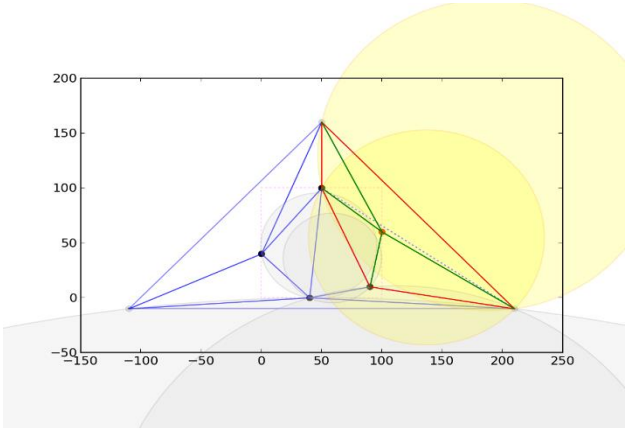

Figure 50. Insertion of 5th point [34].

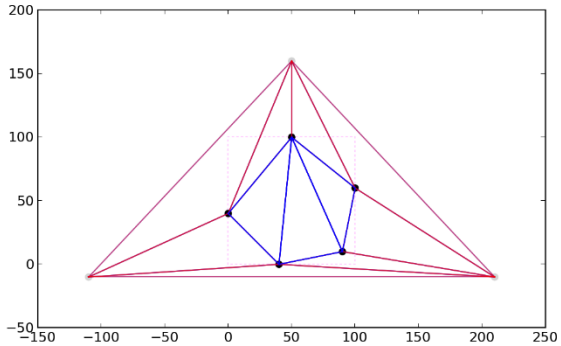

Figure 51. Remove the red edges [35].

II. Finding the perpendicular bisector of two points

Finding perpendicular bisector is a very important in brute force a Voronoi diagram calculation. The following calculations show how to find the perpendicular bisector of two points.

$$P_1 = (x_1, y_1) \quad P_2 = (x_2, y_2)$$
$$midpoint = \left[\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}\right]$$
$$slope = \frac{y_2 - y_1}{x_2 - x_1}$$

Because this is perpendicular to our points we need to take the negative reciprocal of the slope.

$$m = -\frac{1}{slope}$$
$$y = mx + b$$
$$b = y - mx$$
$$b = \frac{y_1 + y_2}{2} - m\frac{x_1 + x_2}{2}$$

From this we get the line formula for the perpendicular bisector of the two points is:

$$y = -\frac{1}{slope}x + \frac{y_1 + y_2}{2} - m\frac{x_1 + x_2}{2}$$

III. Mathematical calculations to find intersections

Plane-Line intersection

The following chapter is based on [36].

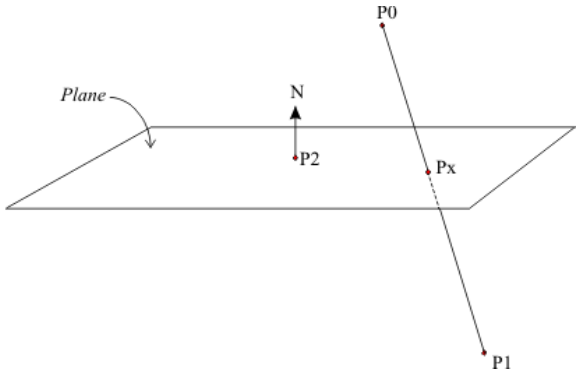

Figure 52 Plane-Line intersection [37].

Line L can be represented as $p(s) = P0 + s(P1 - P0) = P0 + su$. The plane can be represented by a point P2 on the plane and the normal vector n of the plane. $n \cdot u = 0$ if line and plane are parallel. This means either that they never meet or the line is on the plane. If $n \cdot (P0 - P2) = 0$ then the line is on the plane.

If a line and a plane are not parallel, there is a unique intersection point p(s_i). At the intersection point, vector $p(s) - P2 = w + su$, where w = P0 – P2, is perpendicular to n. This means that at the intersection point $n \cdot (w + su) = 0$. By solving this we get that $s_i = \frac{-n \cdot w}{n \cdot u}$ and the intersection point p(s_i) can be calculated.

Plane-Plane intersection

The following chapter is based on [36].

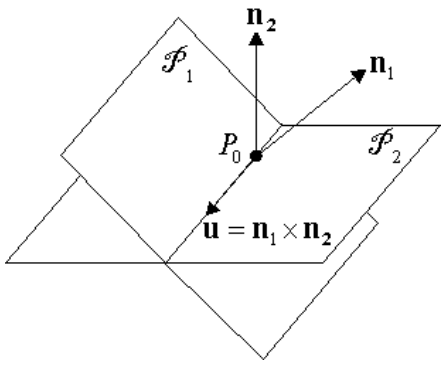

Figure 53 Plane-Plane intersection [36].

First, a check is needed to see if the planes are parallel. For that, a cross product between the normals of the planes, $n1 \times n2$, is calculated. If $n1 \times n2 = 0$, then the planes are parallel. Otherwise $n1 \times n2 = u$. u is the direction vector for the intersection line. Cross product between u and n1 gives a vector d perpendicular to u and is on the plane P1. A random point on P1 and vector d defines line I on P1. Using I and P2 intersection point P0 can be calculated. P0 and u define the intersection line between planes P1 and P2

Line-Line intersection

In 3D two lines rarely intersect, so the common approach is to calculate the smallest distance between the lines and find the points where the two lines are closest to each other. But this calculation is slow. Since in the simulation both of the lines are on the same plane, these lines can be rotated in such a way that all the points on the lines have roughly the same z coordinate. Now the intersection point can be calculated in 2D. Once the intersection point is found a reverse rotation can be applied to move it back into 3D space.

In 2D intersection point C between lines defined by point A and direction t and length s and point B and direction g and length l (Figure 54) can be found as follows:

$$A + s * t = B + l * g$$
$$(A + s * t) \times g = (B + l * g) \times g$$
$$A \times g + s * (t \times g) = B \times g$$
$$s * (t \times g) = (B - A) \times g$$
$$s = \frac{(B - A) \times g}{t \times g}$$
$$C = A + s * t$$

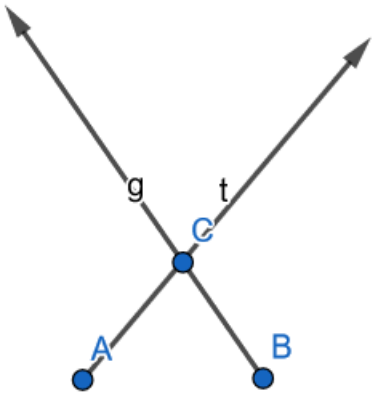

Figure 54. Intersection of two lines at point C.

Cross product in 2D is defined as $a \times b = a.x * b.y + a.y * b.x$. Using the s the intersection point can be found.

Once the intersection point is found a reverse rotation can be applied to move it back into 3D space.

IV. Using the simulation

The simulation is made in Unity. Keyboard and mouse are recommended to use.

To run the simulation, open the .exe file. In the opened window press start. Once the splash screen has finished the main menu (Figure 55) will appear. To play the game press start, to exit press exit. Controls give the list of controls used in the simulation.

Figure 55. Main menu

In-game movement is controlled with **WASD** or Arrow Keys. Move mouse to look around and left click uses the mining laser. Shift is used for crouching and space for jumping.

To mine with the mining laser, hold down the left mouse button. An indicator around the crosshair (Figure 56) will appear. When the indicator gets full the rock is mined. Mining process can be canceled at any time by releasing the left mouse button.

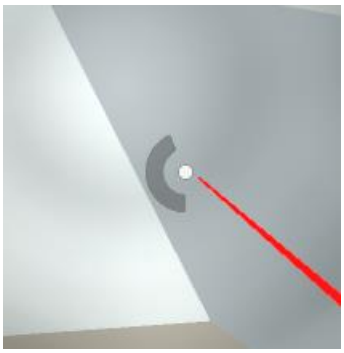

Figure 56. Mining indicator

Pressing the Esc key opens the in-game menu (Figure 57). There are four buttons on the menu. Continue resumes the current simulation, Start Game restarts the game, Controls shows the key bindings for the game and Exit closes the game.

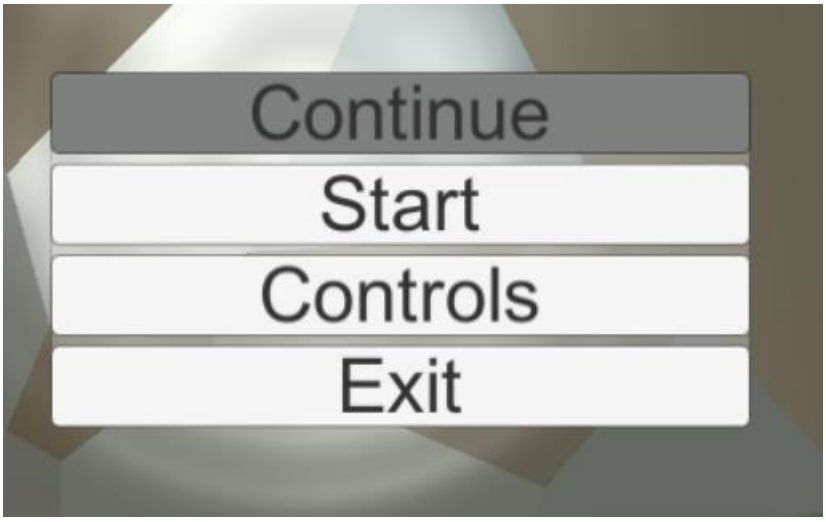

Figure 57. In-game menu

In case of a problem or curiosity, pressing **G** will enable god mode. In god mode move through walls and flying is enabled.

V. Simulation and Unity project

The Unity project and the simulation can be found at https://bitbucket.org/Marko_T/real-timecavedestructionusing3dvoronoi/src/master/ . Simulation is located in the build folder and is ready to use. The project can be opened in Unity to see the setup of the scene and source codes created with this thesis are inside the Assets/Scripts folder.

VI. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Marko Täht**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

    1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

    1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Real-time Cave Destruction Using 3D Voronoi,

supervised by Jaanus Jaggo,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **23.05.2018**