Voronoi fracturing

Iris Kotsinas and Viktor Tholén

19 December 2021

Abstract

This project explores how the destruction of 2D objects, such as walls and glass planes, can be simulated realistically. This is done with 2D Voronoi diagrams with the use of Fortune's algorithm in *C++/OpenGL*. The generated fragments are extruded into 3D and simulated with *Bullet Physics* as rigid bodies. Different patterns are also implemented for the distribution of points. The resulting simulation provides a semi-realistic destruction of objects in real-time, although improvements can be made in several areas.

Contents

1 Introduction

Simulations of destruction of objects are widely used in the visual effect and gaming industry. When simulating the breaking of an object it is important to find a method of producing pieces of appropriate dimensions. For this, 2D Voronoi diagrams can be utilized in order to fracture an object appropriately. Voronoi diagrams have numerous applications across various fields, and they help understand the proximity and distance of features. In this project such diagram has been used to simulate fracturing of an object. The project was created as a part of the course TSBK03 Advanced Computer Games at Linköping University.

For simulations of destruction of objects, results can either be stored and read later through offline rendering, or simulated real-time. For offline rendering, each frame of the animation may be processed at a slower rate than what is displayed at. It is therefore not a method well-suited for computer games. In games, the latency cannot be too large as there is a minimum requirement of frames per second for a good user experience and better player performance [1]. This project focuses on real-time simulation, and it should be computed fast to achieve a desired frame rate. The destruction of objects is a well researched subject, however, due to the real-time limitations the technique of how to achieve it can be unique for each project.

1.1 Aim

The aim of this project is to simulate Voronoi fracturing with the use of a 2D Voronoi diagram. The aim is also to simulate the fracture with applied physics, in this case rigid body dynamics. This report further investigates how Voronoi diagrams can be used and altered to simulate different types of breaking behaviors.

2 Background

This chapter describes the algorithms and techniques used in the implementation. A brief overview of used methods for fracturing an object is presented, as well as the data structure used for polygon meshes. The topic of rigid body dynamics is also presented.

2.1 Voronoi diagram

A Voronoi diagram is a way to divide a plane into smaller pieces from a set of points. Each point creates a region, also known as a Voronoi cell. Given an arbitrary number of points p_i in 2D, the region of the point p is defined by the area where the distance to p is less or equal to its distance to other points in the diagram [2]. This creates a pattern which is procedurally generated depending on the points' position and quantity.

2.1.1 Fortune's algorithm

Although a Voronoi diagram can be quite easily produced in a texture pixel by pixel, a more challenging problem is to do it geometrically. The Fortune's algorithm [3], by Steven Fortune is as of now the fastest way to produce a Voronoi pattern due to its $O(nlog(n))$ time complexity. Hence, it is one of the most popular choices for generating Voronoi diagrams.

The algorithm uses a sweep-line approach where each region grows until it is surrounded by other regions. This way, we only need to take the regions closest to the sweep-line into account. The regions are grown using arcs that grows in the direction of the sweep [3]. When the sweep-line crosses a point, it creates an arc at that position. If a parabola intersects another arc, it will create the final line in the middle of the points which the parabolas belong to [3]. This is clearly seen in Figure 1.

This is an event-based approach where the sweep-line is not dependent on the spatial coordinates, but instead uses a queue system [4]. The queue maintains the events that the sweep-line will come across. This queue can be filled beforehand and is one of the reason for the high performance of the algorithm. This also requires the arcs to be ordered by the horizontal coordinate since this will avoid any additional lookup time. A very simplified pseudo code of the algorithm is displayed below [4].

```
1: for Each input region do
2:     Fill the event queue with region events
3: end for
4: while event queue is not empty do
5:     if Region event then
6:         Add region to the sweep-line.
7:     else
8:         Remove squeezed cell from sweep-line.
9:     end if
10: end while
11: Cleanup any remaining intermediate state.
```

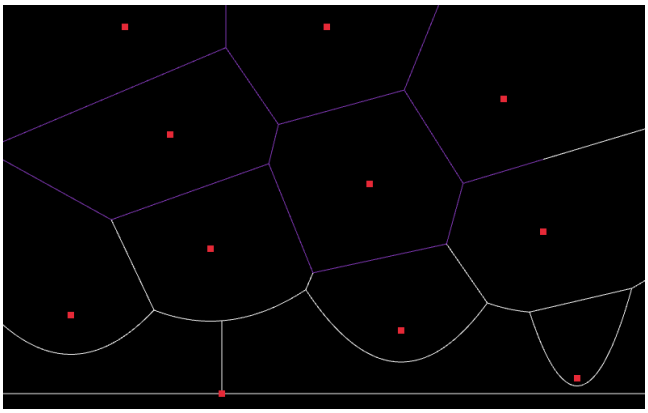

Figure 1: A visual demonstration of the Fortune's Algorithm where the white line in the bottom is the sweep-line [4]. The sweep is done from the top to bottom in this example.

2.2 Half-edge data structure

There are multiple ways of representing polygon meshes. Basic mesh data structures can be used if only basic operations need to be performed, such as *polygon soup*. It is a list of vertices together with a list of polygons referencing the vertices [5]. Polygon soup representation does however not facilitate traversal of polygon meshes, and is therefore in most cases not sufficient for some geometry processing tasks. This is where the half-edge data structure can be used [5].

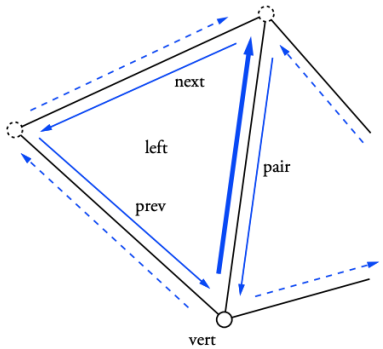

Figure 2: The half-edge data structure as seen from the bold half-edge.

The half-edge data structure, also known as doubly connected edge list (DCEL), is used to represent discrete surfaces as polygon meshes [5]. In a half-edge data structure, edges of the mesh are explicitly stored by representing each edge with a pair of directed half-edge twins, where each

of the two half-edges twins point in opposite directions. A half-edge stores a reference to its twin, in other words the pair, and the previous and next half-edges along the same face. In a vertex the position and reference to a half-edge that originates from that vertex is stored. A face stores a half-edge which belongs to that face [5]. The structure can be studied in Figure 2.

2.3 Rigid body dynamics

In physics simulations within computer graphics, the aim is to reproduce physical phenomena with the use of a computer. In general, these simulations apply numerical methods to existing theories to obtain results that are as close as possible to what we observe in the real world. Rigid body dynamics can be applied in computer graphics in order to give objects on screen a realistic behavior. In this project, rigid body dynamics have been implemented and applied to the objects.

In physics, a rigid body is a solid body which does not deform or change shape. During the course of motions of the body, the distance between the particles, of which the body consists of, remains unchanged [6]. The motion of rigid bodies can be calculated with Isaac Newton's Three Laws of Motion [7]:

1. **Inertia**: A body at rest stays in its state of rest, and a body in motion remains in constant motion along a straight line unless it is affected by an external force.
2. **Force, Mass, and Acceleration**: The acceleration of a body is directly proportional to the force exerted on it and is in the same direction as the force. This is given by the formula $F = ma$.
3. **Action and Reaction**: "For every action there is an equal and opposite reaction." In other words, if a body exerts a force on another, the second body exerts a force of the same magnitude and opposite direction on the first.

With these three laws implemented, a physics engine can be created which successfully can reproduce a realistic dynamic behavior. As previously mentioned a rigid body is a solid which cannot deform. Solids that can't deform does not exist in the real world, but it is a useful model of physics to use in games as deformations can be neglected.

In this project the physics engine *Bullet Physics* is used [8]. Therefore, in the following sections the basics of rigid body dynamics will be explained. These sections follow and is heavily based on the model of Baraff [9].

2.3.1 Position and rotation

A rigid body has a mass, velocity and acceleration, as well as volume and shape. This enables it to rotate. The shape of a rigid body is a fixed and unchanging space, and it is called *body space*. A rigid body rotates around its center of mass. It is defined in the initial state, with its center at the origin and the rotation angle set to zero. The position and rotation of the body at any instance of time will thereafter be an offset of the initial state [9].

For simplicity, $x(t)$ represents the position and $R(t)$ the rotation of the body at time t. If $R(t)$ specifies a rotation of the body around the center of mass, then a fixed vector r in body space will be rotated to the world space vector $R(t)r$ at time t. If p_0 is an arbitrary point on the rigid body, in body space, then the world-space location $p(t)$ of p_0 is the result of first rotating p_0 around the origin and then translating it (see Equation 1) [9].

$$p(t) = R(t)p_0 + x(t) \tag{1}$$

2.3.2 Linear velocity

It is of interest to define how the position and rotation changes over time. Therefore, the expressions $\dot{x}(t)$ and $\dot{R}(t)$ need to be calculated. From the position $x(t)$ of the center of the mass in world space, one can obtain the velocity $\dot{x}(t)$ via derivation (see Equation 2) [9].

$$v(t) = \dot{x}(t) \tag{2}$$

With the rotation of the body fixed, the body undergoes movement with translation. The velocity of the translation is given by $v(t)$ [9].

2.3.3 Angular velocity

In addition to the linear velocity of the translation, a rigid body can spin. If for example the position of the center of mass is fixed, and the points of the body still moves, the body spins around an axis that passes through the center of mass. The spin of the body is the angular velocity $\omega(t)$. The columns of $R(t)$ represents the directions of the transformed x, y and z axes at time t. Hence, the columns of $\dot{R}(t)$ describe the velocity of the transforms of x, y and z. The relation between the rotation $R(t)$ and the angular velocity $\omega(t)$ can be seen in Equation 3 [9].

$$\dot{R}(t) = \omega(t) \cdot R(t) \tag{3}$$

2.3.4 Mass of a body

One can assume that a rigid body consists of a large number of particles, indexed from 1 to N, where the mass of the ith particle is m_i. The mass M of a rigid body can be defined as Equation 4.

$$M = \sum_{i=1}^{N} m_i \tag{4}$$

The center of mass is the average of the particle positions of the body weighted by their mass. If the density of the body is uniform, the center of mass is the same as the geometric center of the body shape, also known as the *centroid*.

$$\frac{\sum m_i \boldsymbol{r}_i}{M} \tag{5}$$

The center of mass of a body in world space is defined as Equation 5, where M is the mass, N is the particles it consists of, each with mass m_i and location $\boldsymbol{r}_i$ inside the body. Equation 5 indicates that the center of mass is the average of the particle position weighted by their mass [9].

2.3.5 Force and torque

To gain angular velocity, the body needs to receive a rotational force known as torque. This can be applied with the use of Equation 6 where the torque is represented by τ, α is the angular acceleration and I is the moment of inertia, described in Section 2.3.7 [9].

$$\tau = I\alpha \tag{6}$$

2.3.6 Linear and angular momentum

The linear momentum $P(t)$ of a rigid body is the sum of the mass and velocity of each particle in the body (see Equation 7) [9].

$$P(t) = \sum m_i \dot{r}_i(t) \tag{7}$$

We also have the relation between the linear momentum $P(t)$ and total external force $F(t)$ as $\dot{P}(t) = F(t)$.

The angular momentum of a rigid body is defined in Equation 8. Similarly to the linear momentum, we have a relation between the angular momentum $\dot{L}(t)$ and the torque $\tau(t)$ as $\dot{L}(t) = \tau(t)$ [9].

$$L(t) = I(t)\omega(t) \tag{8}$$

In Equation 8 $I(t)$ is the inertia tensor.

2.3.7 The inertia tensor

The inertia tensor $I(t)$ can be defined as the scaling factor between the angular velocity $\omega(t)$ and angular momentum $L(t)$. We have that, for a point mass which rotates around a specific axis at distance r, the moment of inertia is a scalar $J = m \cdot r^2$. When applied to a set of particles, these can be summed together. If it then is generalized to 3D, we get the matrix in Equation 9 [9].

$$I(t) = \sum_i \begin{pmatrix} m_i(r_{iy}^2 + r_{iz}^2) & -m_i r_{ix} r_{iy} & -m_i r_{ix} r_{iz} \\ -m_i r_{ix} r_{iy} & m_i(r_{ix}^2 + r_{iz}^2) & -m_i r_{iy} r_{iz} \\ -m_i r_{ix} r_{iz} & -m_i r_{iy} r_{iz} & m_i(r_{ix}^2 + r_{iy}^2) \end{pmatrix} \tag{9}$$

The body space specific inertia tensor for the body in its non-rotated position can be defined as Equation 10, where $\mathbf{1}$ is an identity matrix. Since I_{body} is specified in body space, it is constant over the simulation. Therefore, I_{body} can be calculated beforehand. $I(t)$ can then be calculated from I_{body} and the rotation matrix $R(t)$ (see Equation 11) [9].

$$I_{body} = \sum m_i((r_{0_i}^T r_{0_i})\mathbf{1} - r_{0_i} r_{0_i}^T) \tag{10}$$

$$I(t) = R(t) I_{body} R(t)^T \tag{11}$$

2.3.8 Motion of a rigid body

The state vector $\boldsymbol{X}(t)$ for a rigid body can now be defined with the information from the previous sections (see Equation 12) [9].

$$\boldsymbol{X}(t) = \begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix} \tag{12}$$

$$\frac{d}{dt}\boldsymbol{X}(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ \omega(t) * R(t) \\ F(t) \\ \tau(t) \end{pmatrix} \tag{13}$$

The state of a rigid body is its position and rotation, and its linear and angular momentum. The mass M and body space inertia tensor I_{body} are constants, which should be known beforehand. We then get the derivative of the state vector, which can be studied in Equation 13 [9].

3 Implementation

The Voronoi fracturing was implemented in C++ and OpenGL on Linux and MacOS. A basic program structure was first implemented with a simple scene depicting a plane in 2D. From this, a more advanced scene was created. The following sections describe the technical implementations.

3.1 Voronoi diagram generation

Implementing the Fortune's sweep algorithm is a demanding and time consuming task. Since existing libraries already exist that do this very efficiently, it is a good alternative to consider using these. We used a library called *jc_voronoi* which is an implementation of the Fortune's sweep algorithm in C++. It is according to the authors, the fastest library for generating Voronoi diagrams although it requires a bit more memory allocations than other alternatives. Similar libraries are *fastjet*, *voronoi++* and *boost*. When scaling the algorithm to high number of points, performance is of high value compared to memory, since memory is seldom a limitation in modern computers.

The library works by entering the 2D-positions in space as well as setting the boundaries of the diagram. We then get back all regions and half-edges of the diagram. Since the library uses the half-edge data structure, it is easy to traverse the site and access each edge's neighbor. From the data we get back, we can triangulate a region by going through each half-edge *next* pointer and make a triangle with the edge's vertices and the center point of the region.

3.2 Extrusion

In this project we focus on 2D Voronoi diagrams. However, Voronoi fracturing requires 3D objects to simulate the destruction of flat objects, such as a wall or a glass plane. Hence, a method for extruding the regions of the diagrams needs to be used. Extrusion is a technique which can be applied to objects with fixed cross-sectional profiles. It is widely used in 3D-software such as Blender and Maya to easily create 3D objects from 2D shapes.

We propose a rudimentary approach to the problem. All edges of the region are first copied to a separate list and offset by a depth. Then we add each triangle of that list and flip the normals to point in the opposite direction. This represents the back side of the region. At this point, the geometry for the back and front of the region is finished. The last part is to triangulate the sides of the region. This is done by looping through the front and back edge-lists and pairing their indices to create two triangles for each side.

3.3 Point distribution

Voronoi diagrams are dependent on the points to create the regions of the diagram. This heavily affects what the final result looks like. In this

project, three different methods were used to sample the points and as a result create unique looking patterns.

One approach to sample the points is to generate random points in 2D within the boundaries. This creates a uniform pattern which can be seen in Figure 3.

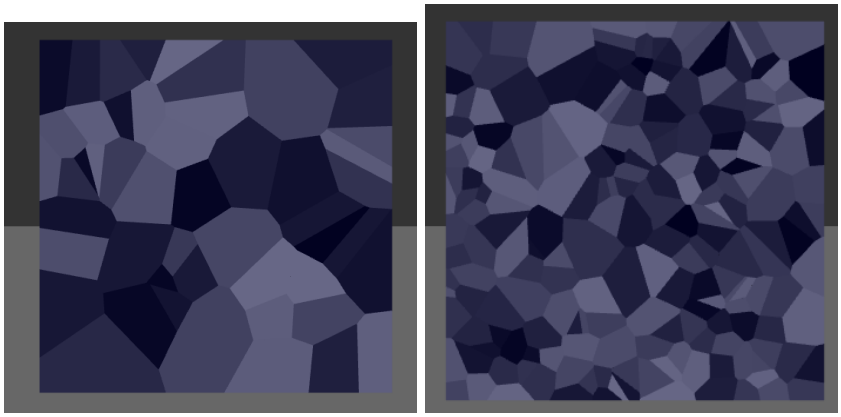

Figure 3: Uniform pattern with 50 points (left) and 200 points (right).

Another approach is to sample the points along a line. As can be seen in Figure 4, this pattern resembles a crack where smaller pieces are formed closer to the line and bigger pieces will form further away.

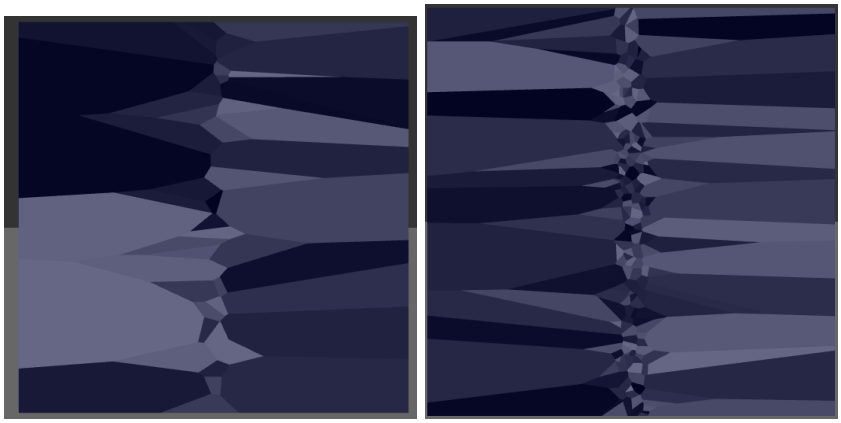

Figure 4: Crack pattern with 50 points (left) and 200 points (right).

The last method that was implemented was based on one point p within the boundaries. Every point were then sampled around p, resembling the pattern of a bullet destroying the object. This pattern is displayed in Figure 5.

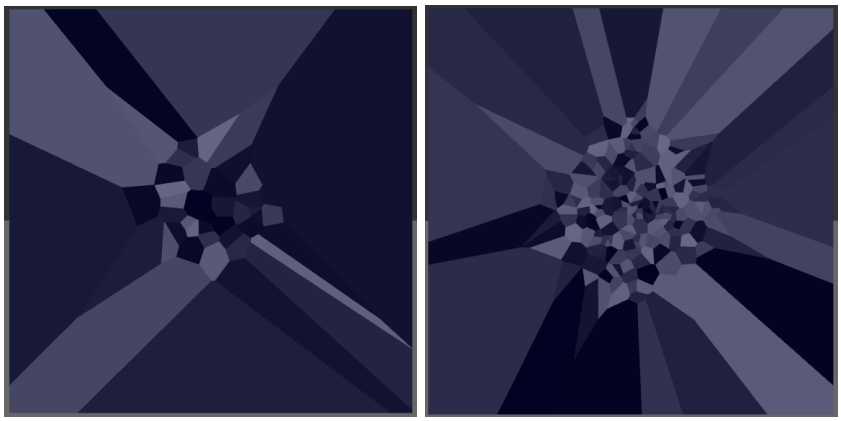

Figure 5: Hole pattern with 50 points (left) and 200 points (right).

3.4 Bullet Physics

The physics engine is the software component that performs the physics simulation. In this project we used *Bullet Physics* [8]. Bullet Physics is a powerful open source physics engine with collision detection, rigid body and soft body dynamics written in C++. The engines is mainly designed for use in visual effects and games. This project specifically uses the rigid body dynamics implementation in Bullet Physics. As described in Section 2.3, a rigid body is a solid body which does not deform or change shape. The physics behind the implementation can be read more about in Section 2.3.

The objects created with the Voronoi diagram are simulated as rigid bodies. It receives a specification of the bodies that are going to be simulated and the simulation can be stepped. Each step moves the simulation forward by a few fractions of a second, and the results can be displayed on screen afterwards. Forces are applied, such as gravity and an impulse force. The impulse force allows the objects to not only be affected by the gravity and fall down, but it also simulates the fracturing moving in a specific horizontal direction.

4 Results

The result of this project is an application built with C++ and OpenGL that runs in real-time on Mac and Linux. Functionality for changing patterns, adjusting number of fragments and changing colors exists. Screenshots of a simulation with the different point based patterns can be seen in Figure 6, 7 and 8.

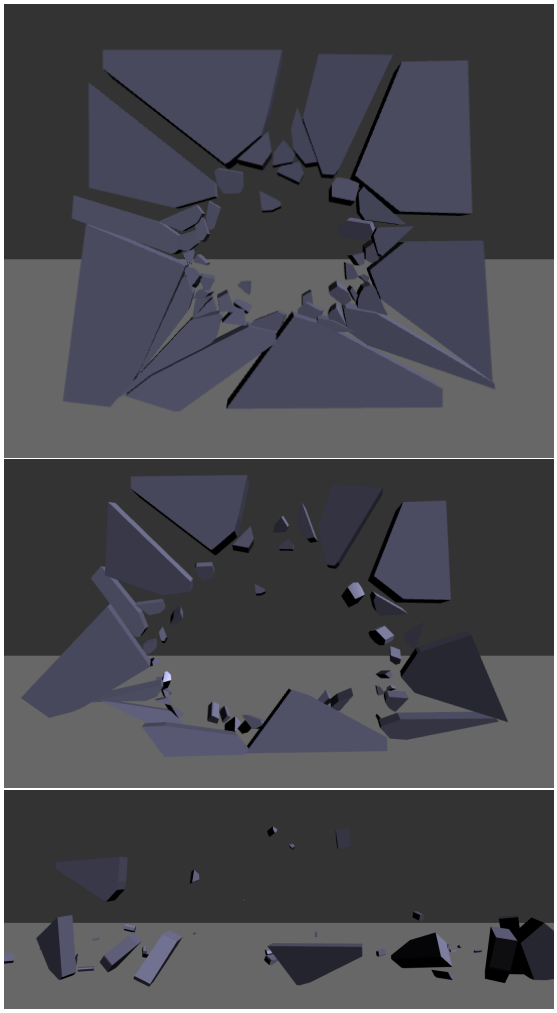

Figure 6: Three images from stepping through the simulation with the hole pattern. Force is applied from the center point at start.

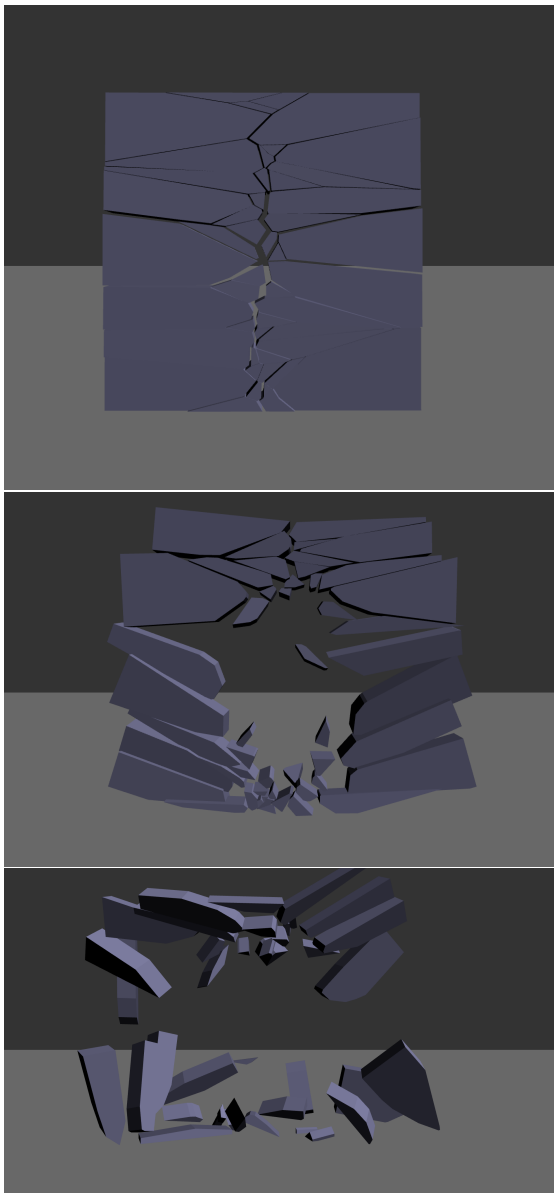

Figure 7: Three images from stepping through the simulation with the crack pattern. Force is applied from the center point at start.

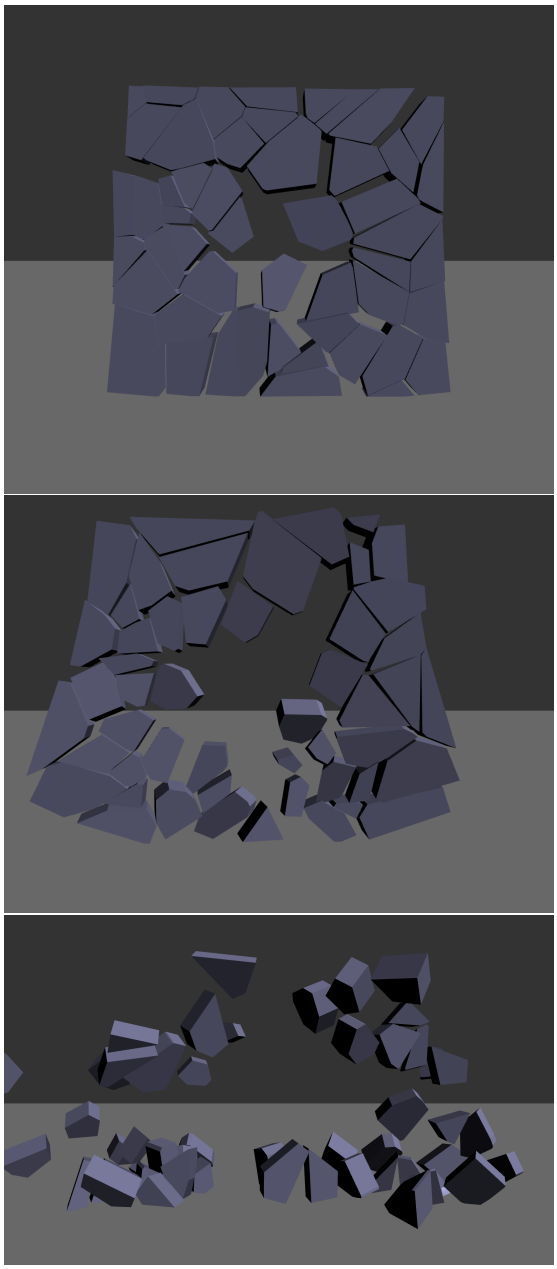

Figure 8: Three images from stepping through the simulation with the uniform pattern. Force is applied from the center point at start.

5 Discussion

The Voronoi fractures in the left Figures 4 and 5 with 50 points are balanced in size and create generally believable fracture fragments. The simulations with 200 points seen on the right in Figure 4 and 5 can be perceived as cluttered and less realistic due to the thickness of the object. Therefore, the use of less points can be more optimal since it allows for better performance as well as more visually realistic results.

Adding another dimension to the fracturing would heavily increase the number of use-cases but also the complexity and scope of the project. As of now, the implementation only supports 2D objects such as walls and glass planes. In order to support fracturing of 3D meshes, another method for constructing Voronoi diagrams would be required since the Fortune's algorithm does not support 3D as of today.

As previously mentioned, this project uses the rigid body implementation in the physics engine Bullet Physics. The engine handles the collisions between the fractured objects and simulates their behavior. In this project's specific physics implementation the collisions behaves strangely at times, and the objects get caught in each other. A solution for this could be to add additional separation to the objects. Due to a shortage of time this behavior could not be fixed, and it is therefore a future improvement.

The project was developed on both MacOS and Linux simultaneously, and therefore operating system specific problems were encountered. The main problems with developing on multiple platforms were that different libraries had to be linked. Additionally, the *makefiles* differed in order to work with the different libraries too, which was difficult at times when new libraries were added to the project.

As the amount of generated points in the Voronoi diagram is increased, the performance of the simulation can decrease and slow down. Therefore, due to the performance requirements set on the computer when generating a Voronoi fracture, the point generation cannot be too high. For real-time simulations this has to be taken in account, since the latency cannot be too large as there is a minimum requirement of frames per second for a good user experience and better player performance.

6 Conclusions

A simulation of Voronoi fracturing using 2D Voronoi diagrams was successfully achieved. Fortune's sweep algorithm was implemented with the library *jc_voronoi* in a efficient way. The physics engine Bullet Physics was used to apply rigid body dynamics to the simulation. Finally, different point distributions were used to create several patterns.

The physics collisions sometimes encounter issues where the objects get stuck in each other. This could have been improved by separating or scaling the fragments before creating the physics colliders. There is also room for improvement in the simulation controls where a GUI could improve the visual feedback for the user.

The use of Bullet Physics deepened our understanding for rigid body

dynamics, and to further develop our knowledge we researched more about the underlying theory behind the engine implementation.

Working within a C++/OpenGL environment has been a learning experience. This also entails the platform specific makefiles that were used to compile the project. We learned a lot by examining the complex libraries that were used in this project while adapting the code to those. Writing an object oriented rendering codebase with OpenGL was also new to us, which created some challenges.

It was interesting to learn more about Voronoi fracturing in general, and this project gave us a basic understanding of the concept. Overall, the project resulted in a semi-realistic real-time simulation which resembles the destruction of objects.

References

[1] K.T. Claypool and M. Claypool. On frame rate and player performance in first person shooter games. *Multimedia systems*, 2007.

[2] David Austin. Voronoi Diagrams and a Day at the Beach. `http://www.ams.org/publicoutreach/feature-column/fcarc-voronoi`, 2006. Accessed: 2021-12-19.

[3] Steven Fortune. A Sweepline Algorithm for Voronoi Diagrams. *Algorithmica*, 1986.

[4] Jacques Heunis. Fortunes Algorithm: An intuitive explanation. `https://jacquesheunis.com/post/fortunes-algorithm/`, 2018. Accessed: 2021-12-14.

[5] Jerry Yin and Jeffrey Goh. Half-Edge Data Structures. `https://jerryyin.info/geometry-processing-algorithms/half-edge/`. Accessed 2021-11-15.

[6] Britannica. Rigid bodies. `https://www.britannica.com/science/mechanics/Rigid-bodies`. Accessed 2021-11-16.

[7] NASA. Newton's Laws of Motion. `https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/newtons-laws-of-motion/`. Accessed 2021-11-16.

[8] Erwin Coumans. Bullet Physics. `https://github.com/bulletphysics/`. Accessed 2021-11-15.

[9] A. Witkin and D. Baraff. Physically Based Modeling. *SIGGRAPH course notes*, 2001.