Voronoi fracturing -
Project in High Performance Computer Graphics, EDAN35

Tom Hansson, dat13tha Daniel Cheveyo, dat13dbj

December 13, 2017

Abstract

The problem that was going to be solved in this rapport is about shattering a shape into smaller pieces and letting physics interact with them. This was done by using Voronoi diagrams and using Bullet Physics API. The resulting Voronoi cells were evenly distributed and worked as intended.

1 Introduction

The goal of this project was to make objects break into smaller pieces or shards. To fracture an object on contact will take a lot of effort and time to optimize. Another easier approach was decided, by precalculating a random fracture of an object. To see how these shards interact with physics, the library Bullet Physics was used with its shaders, because the focus was to get a good fracture and shatter not actually an incredible shading.

In this rapport the next section will have a brief theory of what has been done and a subsection with the pseudo code version of the used algorithm. After that screenshots are presented of how it looks and evaluate the performance. Then at the end discuss the results and suggest improvements and further optimizations.

2 Algorithms and methods

Like mentioned above, this section will be about how the theory underneath the code works. The core of the application is to understand Voronoi diagrams, so this will be presented first. Later on there will be a short subsection about the library and physics engine we used, Bullet Physics, and lastly a subsection with explained pseudo code of the fraction algorithm.

2.1 Voronoi diagrams

Voronoi diagrams is also called Dirichlet tessellations. The definition of this is, from a set of site points, $s_1, s_2, ..., s_n$, three of those site points, call them A, B and C, in either 2D or 3D space are connected as a triangle or a so called Voronoi cell, V_i. A, B and C are chosen so that no other site point, D, is closer within a circumference of a circle which circumscribes A, B and C, as illustrated in Figure 1.

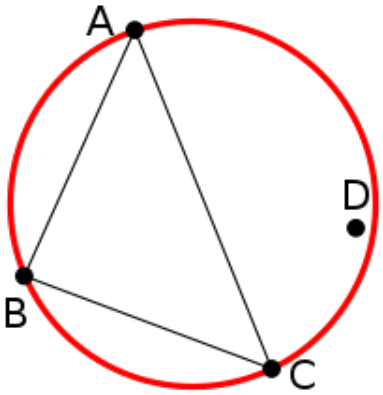

Figure 1: An up close Dirichlet tessellation with site points (black dots), and their circumference (red circle). Point D should be outside the circumference for the triangle to be valid.

Points in the 2D or 3D space, $p_1, p_2, ..., p_m$, that form a mesh, M, are then assigned to a cell V_i comparing with a distance function, $D(sitePoint, point)$, which site point is the closest. This gives a definition of a Voronoi cell, as described in [2]:

$$V_i = \{p | D(s_i, p) \leq D(s_j, p), i \neq j, p \in M\} \quad (1)$$

The distance function are usually a linear function as Euclidean distance, where the difference in the x- and y-coordinates are squared, added and then square rooted.

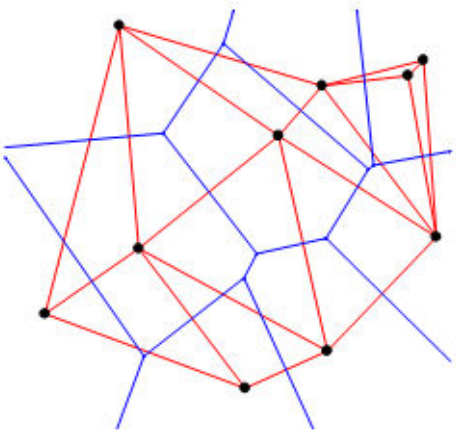

Figure 2: A Voronoi diagram with site points (black), Voronoi edges, results from Dirichlet tessellations (blue) and Delaunay triangulation (red). Taken from [3]

2.2 Bullet physics engine

Now a set of Voronoi cells are generated and are now going to be integrated into the Bullet physics engine. This workspace is created by Erwin Coumans and Yunfei Bai. It is according to their own guide, available at their website [4], a Python module for physics simulation for robotics, games, visual effects and machine learning. It uses a build-in OpenGL GPU visualizer, running OpenGL 3.

With this API, we have work with rigid bodies and collision shapes. Creating them for each shape we want to render and apply physics to.

2.3 Fracture Algorithm

In this section a pseudo code version of the code implemented is going to be presented and explained. The pseudo code for the Voronoi fracturing is as follows and is inspired from a post at the Bullet Physics forum [1]:

```
1.  Create a boundary box for a shape
2.  Generate a number of random site points
3.  for each site point
4.      clear plane equations
5.      sort every other site point in a list
6.      for each point in the sorted list
7.          Add plane equation between sorted...
            ...point and site point
8.          check which vertices are inside the...
            ...plane equations
9.          Delete the planes that are outside
10.         check if the shortest distance of...
            ...  the vertices is shorter than the...
            ...normal length, then break from loop
11.     generate edges from these vertices
12.     generate faces counter-clockwise
13.     create rigid body and collision shape
```

The created boundary box at row 1 is in this case a cube. When randomly translated points inside the cube is generated at row 2, the cube's rotation was also considered.

At row 4 the plane equations for a site point are reseted to the planes of the boundary box. This was calculated with the rotation matrix from the box in x, y, z and also with corresponding inverse rotation.

At row 5 all other points are sorted in a list according to distance to this current site point. This has to be sorted to optimize the code.

At row 7 add to the plane equations for this site point to take account to the sorted point and set this plane's normal length to $(sortedpoint - sitepoint)/2$.

At row 8 vertices, plane intersections of 3 planes, are tested against every plane equation combination to see if they are inside all of them at the same time, by calculating the cross product between pairs of planes, get the intersection and test if this possible vertex can be projected into a last plane, using the dot product. This is certainly a brute-force method way to do it, so this could be done in a prettier way.

At row 9 update the plane equations by deleting planes that are outside the vertices.

At row 10 the shortest length of the vertex list is compared with the normal vertex length, multiplied by 2. If the normal length is greater, then the other points are considered to be too far away to take account to. Then the sorted list loop is broken and all other points for a site point are considered.

At row 11 edges are calculated from the vertices for the shard mesh/vonoroi cell, using the Bullet Physic's btConvexHullComputer class, that basically generates a convex hull for the mesh by expanding over the given vertices. Vertices are also translated relative to center of mass of the volume that the faces generate, to get a more synchronized representation. In the same process row 12 are computed from the extracted edges from the convex hull.

The last step, row 13, the rigid body and the collision shape of the mesh are set with the right, rotation, translation, scaling and density, using Bullet Physic's btRigidBody respectively btCollisionShape.

3 Results

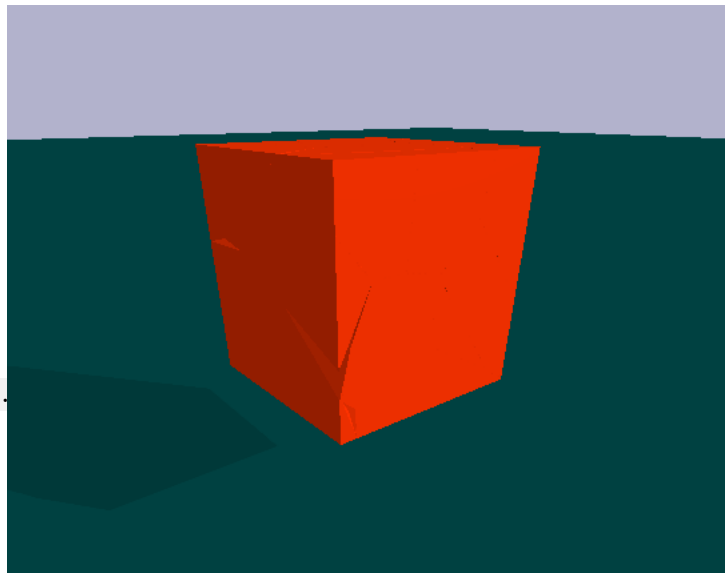

Figure 3: Generated intact cube with 50 Voronoi cells.

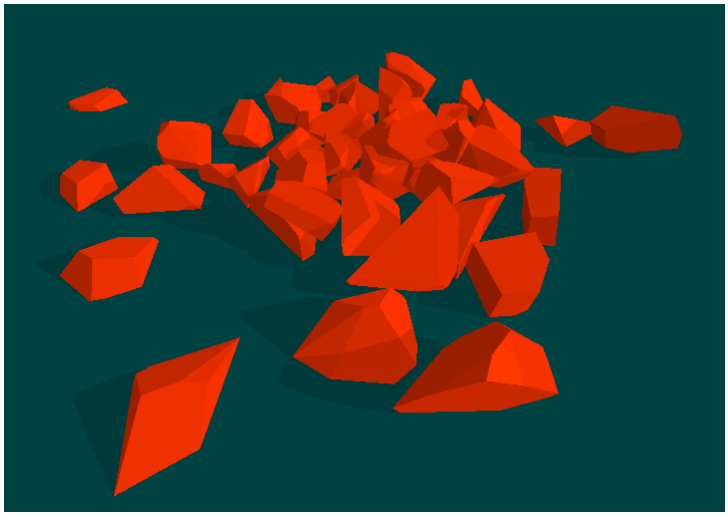

Figure 4: Shattered version of Figure 3.

Above there are two images from the executed code for generating Voronoi cells and shatter them. The table below is a performance test of how the amount of cells affects generation time.

number of Voronoi cells	Generation time [s]
10	0.007
20	0.034
50	0.237
100	0.528
200	1.468
500	4.147
1000	10.614
2000	26.429

Table 1: Showing correspondence between chosen number of Voronoi cells and the execution time for generating them in seconds.

4 Discussion

The result was what we expected. The generated shards are evenly distributed as intended and the physics' impact on the shards are satisfying.

Like mentioned in the subsection above, when explaining row 7. Testing vertices in this way is not optimized and could be improved further.

Since we focused a lot on just fracturing an object into shards, we laid very little time on making a good shader for the application for it, to look even more stunning. With more time, this could have been improved.

A problem we encountered is that some kind of artifact appears when converting a shape to Voronoi cells. While the combined cells closely represents the original shape it is not perfect. There appear some cracks and crevices along the shape's surface, as can be seen in Figure 3. The flaw probably exist in the Voronoi cell generation code, but we couldn't actually pin-point it with the given time. Though, We found out that by increasing the number of Voronoi cells, the cracks became smaller.

Our implementation only fractures cuboid shapes, but expanding the algorithm to handle any type of convex polygon should only require a way of providing plane equations and randomizing site points inside the convex hull.

References

[1] Real-time voronoi fracture and shatter for Bullet Physics, Real-Time Physics Simulation Forum, December 2011, URL: http://bulletphysics.org/Bullet/phpBB3/viewtopic.php?f=17&t=7707
Taken 2017-12-09

[2] Qin Yipeng, Yu Hongchuan and Zhang Jianjun: Fast and Memory-Efficient Voronoi Diagram Construction on Triangle Meshes, Computer Graphics Forum, August 2017, Vol.36 Issue 5, p93-104

[3] Michael S. Rosenberg and Corey Devin Anderson: Pattern Analysis, Spatial Statistics and Geographic Exegesis, Center for Evolutionary Functional Genomics, School of Life Sciences, Arizona State University, URL: http://www.passagesoftware.net/webhelp/Delaunay_Dirichlet_Tessellation.htm
Taken 2017-12-09

[4] Erwin Coumans and Yunfei Bai: pybullet, a Python module for physics simulation for games, robotics and machine learning, 2017, URL: http://pybullet.org/
Taken 2017-12-09