CS 348C Final Project Report

Robert Neff, Stanford University
rneff@stanford.edu
December 7, 2017

1 Background

A simple way to subdivide an object upon fracture is using a Voronoi diagram. A Voronoi diagram takes a set of n distinct points or sites in the provided plane and subdivides the plane into n cells, with one around each site. The edges that make up each cell are placed, such that given any point p in the plane, the distance between it and some site s_i and a different site p_j along the edge is the same. It follows that if these distances are not the same, p lies in the cell of the site with smaller distance from site to p.

This can be simply implemented by using a half plane intersection algorithm ($O(nlog(n))$ [6] to find the cell for each of the n sites. This yields an overall runtime of $O(n^2log(n))$ which can be costly to do at runtime if their is a large number of sites. As such, this project uses the methods detailed below to create Voronoi diagrams and procedural meshes more efficiently.

2 Method

2.1 Fortune Voronoi

This project procedurally generates three-dimensional fracture meshes from two-dimensional object bounds to provide semi-realistic fracture animation. This is achieved through two methods: Fortune's algorithm to create a Voronoi diagram from which to create a mesh and naive triangle fan division of the original mesh.

The implementation of Fortune's algorithm represents the bulk of the work completed in this assignment and so is also the primary method at use in creating a convincing fracture animation. It is used as it offers a more efficient method (specifically $O(nlogn)$ time) to generate Voronoi diagrams, whose cells are used to create the fractured mesh pieces, than standard Voronoi diagram construction method. It achieves this by following a horizontal sweep line vertically, from site to site, down the plane, for which to generate the diagram. This allows for additive construction of the diagram, maintaining the parts above the sweep line that cannot change and introducing changes as the line sweeps each new event. Each time a site is encountered a parabola is created about it with the site as its focus. If the parabolas of multiple sites intersect, they are clipped at the breakpoint creating the moving "beach line" that traces cells as it moves down.

There are numerous data structures used to keep track of changes and events that require updating the current state of the diagram. They are as follows:

- Sweep line: Priority Queue of events sorted on y-position (higher y-value puts the event earlier in the queue as it will be swept by the line going to top to bottom of the plane).
- Beach line: Red-Black Tree of breakpoints between arcs that tracks the beach line. A cell vertex is located at the point two breakpoints meet (i.e. collapse of a beach arc section).

- Diagram: Graph containing list of half-edges that bisect each pair of sites (each with a reference to associated cell, previous/next half-edges in CCW chain), list of sites, and list of cells (each with reference to half-edges it contains).

Updates are tracked by events in the sweep line queue, containing site and circle events. A site event occurs when a new site is encountered by the sweep line. The beach line tree is then appropriately updated. A circle event occurs when an empty circle (contains no sites) touches three or more sites with its border, and results in the removal of an arc [8]. After all events have been handled, not all edges will have finite end points so they are clipped on the plane's bounds.

Since we rely on a balanced binary tree (RB-tree) and priority queue of similar implementation, updating the beach line will always run in $O(log(n))$ time with all other event operation being constant. In addition, there are a maximum of $O(n)$ events in the sweep line queue to cycle through before completing the diagram. This yields an overall complexity of $O(nlogn)$ as expected.

After the Voronoi diagram has been completed, the vertices of each cell are then applied to a mesh to create fracture pieces. The mesh is give a z-thickness to provide some element of three-dimensionality. It is created via simple triangle fan approach for the front and back faces and a triangle strip for the sides connecting the two differing z-faces.

2.2 Fan-Fracture

The second method of fracture implemented only takes place immediately in the mesh generation phase. The fractured Voronoi cell piece or further subdivided element has its mesh split up in a triangle fan pattern to create smaller fractured pieces. This is far less costly ($O(V)$ where V is the number of vertices in the front face of the mesh) and works well on already fractured pieces since they will be smaller and less noticeable uniform in their division.

2.3 Jump Flood

One other suggested path in the proposal was implementing a Voronoi fracture algorithm in a shader such that it could be computed in parallel on the GPU. Fortune Voronoi depends on already found elements making it a poor choice to parallelize. To combat this, the Voronoi diagram was computed using the Jump Flooding Algorithm (JFA) [3] where each pixel within a texture can be handled independently to find which Voronoi cell it belongs to.

In JFA, sites are mapped to a texture at random pixel locations and given a color that will represent its cell. During each pass in the fragment shader, each pixel will check all (eight) pixels "surrounding" it and apply current pass step size of offset to them in the x/y directions. The pixel is then colored according to the closets site found at the surrounding locations. The step size doubles on each pass until it reaches the size of the maximum width or height dimension of the texture. Two extra passes, with step sizes one and two respectively, were added to reduce error and potential island pixels (colored different than surrounding Voronoi cell).

Resolving the texture to a diagram was tricky, as Unity does not have any build in feature detector. As a consequence of this, a fairly naive approach was used to find the Voronoi vertices constituting cells. The method is as follows:

The corners of the texture were automatically mapped to a list of vertices of the same color. Any color changes when looping over the top or bottom rows, and when looping down the first and last columns were mapped to lists of vertices with the first incurred color in the color change. Finally, every other row of the texture was looped across jumping two pixels at a time, with the current pixel keeping track of the immediate pixels above and in front of it. If there were three or more different colors (including the current pixel) a Voronoi vertex occurs at that location [7], so it was added to the lists of each different color found in the map.

After the Voronoi diagram has been completed, the vertices of each cell are then applied to a mesh to create fracture pieces as with the Fortune Voronoi method.

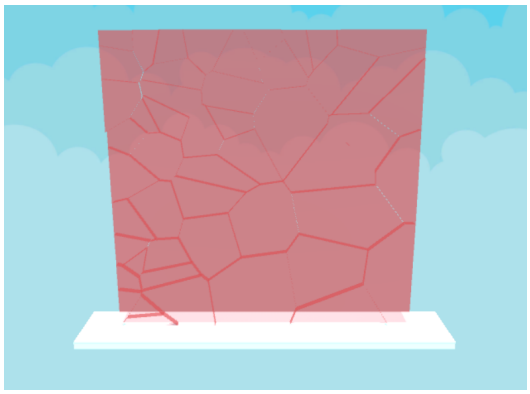
Since JFA doubles the step count on each pass of the shader, it takes $O(log(n) + 2)$ time, where n is the largest dimension of the texture, to compute the Voronoi texture with two additional passes for error correction. The naive Voronoi vertex and cell detector is the bottleneck here, as it does not run on the GPU and requires looping over the entire image, yielding $O(n^2)$ runtime on an $n \cdot n$ sized texture.

Due to time limitations this method was not as flushed out as the Fortune Voronoi implementation and supports fewer fracture features and is not as quick, as it requires first quickly rasterizing the Voronoi diagram then expensively extracting the vertices/cells.

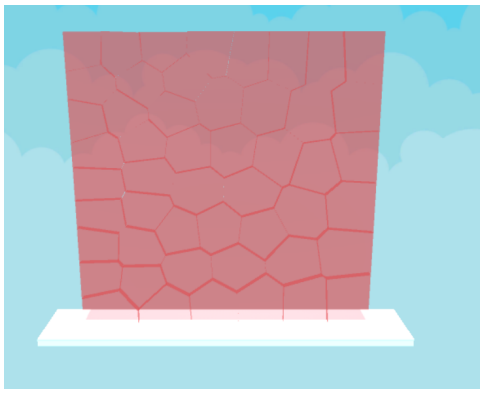
3 Results

The fracture methods above were tested under various conditions to get somewhat convincing fracture animations. Below are images from various test cases.

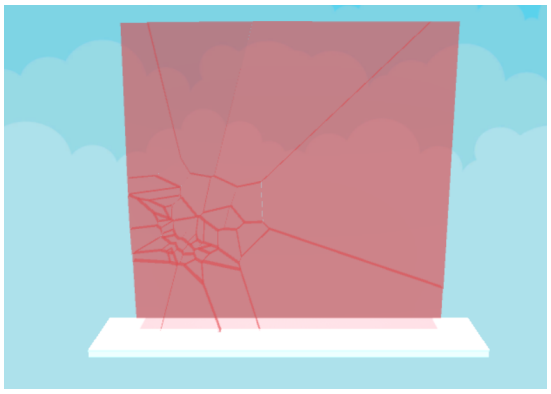
(a) Glass pane before fracture.



(b) Glass pane fractured into pieces using Voronoi diagram generated by Fortune's algorithm.

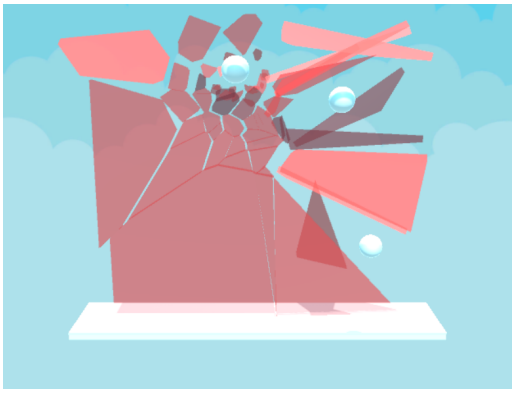


(a) Relaxed site positions of previous fracture using Fortune's algorithm.

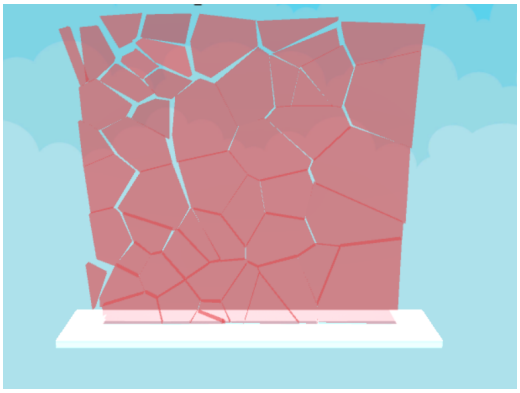


(b) Concentrated site positions about a fracture point.

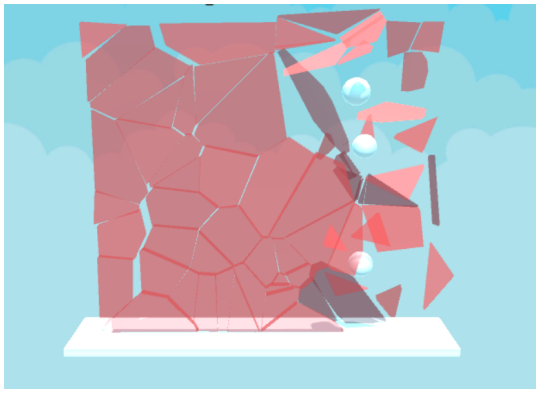
(c) Glass fractured by ball procedurally on collision using Fortune's algorithm.



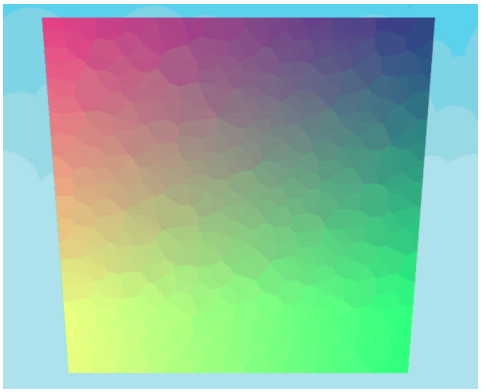
(d) Glass fractured by ball with additional subdivision of fractured pieces on further collision with the ball.



(e) Glass pane fractured into pieces using rasterized Voronoi diagram generated by the Jump Flooding algorithm.



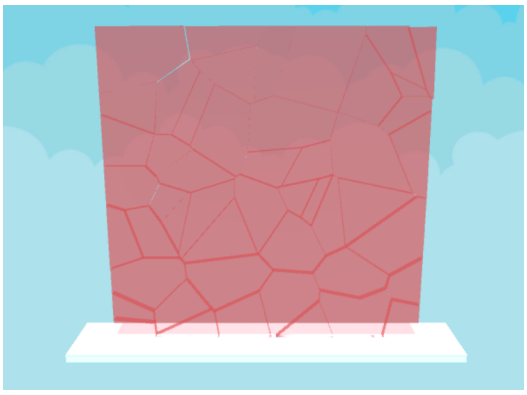
(f) Glass fractured by ball procedurally on collision using Jump Flooding algorithm.



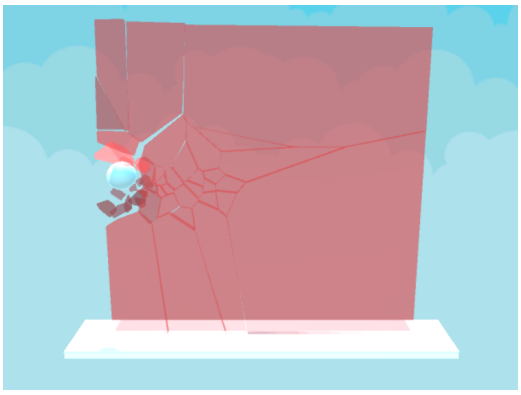
(g) Voronoi diagram texture generated using Jump Flooding algorithm.

4 Additional Findings

Randomization was used a few times in this project, however it does not always yield better results. For example, in the case of generating sites for the Voronoi diagram upon impact of the plane with a projectile, fully randomized sites does not look realistic. As such, the sites were concentrated closer to the area of impact, while still maintaining randomization in where they fell within that zone.



(h) Glass fractured with fully random sites.



(i) Glass fracture with ball using concentrated sites around collision position.

In contrast, however, having a random site center of the triangle fan in the naive subdivision fracture method could greatly help create more realistic fracture as each cell will fracture at a different point (not always at the cell centroid as used in this project). This lack of uniformity would make the fracture seem more natural (not just "pizza cut" of mesh).

Finally, concerning JFA, with a faster vertex detection algorithm or only use of the texture, it is great for use with lots of sites, as you can increase the count without critically impacting runtime (lower resolution grids do however affect accuracy). This highlights the many benefits and drawbacks each of parallel and non-parallel algorithms have when performing the same task.

5 Proposal Achievement

This projects has achieved the primary objective of "semi-realistically modeling 2D fracture of objects by splitting an object's mesh into smaller pieces on collision with another object" set forth in the proposal. Additional items completed outside the scope of the proposal include

- Naive mesh subdivision fracture to create additional fracture pieces upon impact
- Extension to Quasi-3D by mapping the plane to three dimensional mesh
- Jump Flooding implementation of Voronoi fracture on the GPU

Note: 2633 lines of code were written by myself for this assignment.

References

[1] Matthias Muller, Nuttapong Chentanez, Tae-Yong Kim. *Real Time Dynamic Fracture with Volumetric Approximate Convex Decompositions.* CM Transactions on Graphics, vol. 32, no. 4, Jan. 2013, p. 1., doi:10.1145/2461912.2461934.

[2] Lien Muguercia, Carles Bosch, Gustavo Patow. *Fracture Modeling in Computer Graphics.* Computers & Graphics, vol. 45, 2014, pp. 86100., doi:10.1016/j.cag.2014.08.006.

[3] Guodong Rong, Tan Tiow-Seng. *Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform.* Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games SI3D ́06, 2006, doi:10.1145/1111411.1111431.

[4] Jeremie St-Amand. *Unity Voronoi.* 14 December 2014. <https://github.com/jesta88/Unity-Voronoi>.

[5] *Fortune's Algorithm.* Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 11 Jan. 2017. Web. 6 October 2017. <https://en.wikipedia.org/wiki/Fortune%27s_algorithm>.

[6] Dave Mount *Vor2d_1.* 13 November 2007, PowerPoint file. <http://ima.udg.es/~sellares/ComGeo/>.

[7] Dan Maljovec. *Delauney Triangulation on the GPU.* 2 May 2010, PowerPoint file. 2 May 2010, PowerPoint file. http://www.cs.utah.edu/~maljovec/files/DT_on_the_GPU_Print.pdf

[8] David Austin. *Voronoi Diagrams and a Day at the Beach.* Web. 6 Nov. 2017. <http://www.ams.org/samplings/feature-column/fcarc-voronoi>.

[9] Pixelplacement. *iTween.* 1 December 2010. <https://assetstore.unity.com/packages/tools/animation/itween-84>.

[10] *Farland Skies Cloudy Crown.* 13 April 2016. <http://www.borodar.com/>.

[11] JohanDeecke. *Glass Hit (29).* 13 November 2016. <https://freesound.org/people/JohanDeecke/sounds/368339/>.

[12] C_Rogers. *Glass Shattering 03.* 19 October 2013. <https://freesound.org/people/C_Rogers/sounds/203373/>.

[13] RoganDerrick. *Glass Break Small Jar 01.* 5 January 2015. <https://freesound.org/people/RoganDerrick/sounds/260433/>.

[14] soundscalpel. *Foley cable whoosh air 001.* 16 December 2010. <https://freesound.org/people/soundscalpel.com/sounds/110615/>.