Udder Destruction: Breaking Things and Other Explorations in Blender

Kiwi Sheldon and Leo McElroy

CS500

12/17/17

Abstract

The aim of our study was to create a tool to animate destruction sequences in Blender and to use said tool to create such a destruction sequence for the short film *Estrellita*. We modified an existing fracture tool to make it suitable for animation uses but encountered limitations when applying the tool to the intricate yet geometrically flawed mesh used in the shot of interest. Ultimately, we used a forked version of Blender to create the finished destruction shot, one that composes roughly fourteen seconds of the film. We also created a collection of tools and scripts to assist with the various needs of other animation studio members. By the semester's end, we successfully adapted an accessible fracture tool to animation purposes; wrote several custom tools that will be useful to the Middlebury animation studio; learned to navigate the inner-workings of Blender; and gained valuable experience contributing to two large projects, Blender and *Estrellita*.

Semester Objectives

We began the semester with ambitious, yet flexible, goals of what we hoped to accomplish before December.

- First, we wanted to make a meaningful contribution to the film *Estrellita*. We expected this to come in the form of one to three destruction animations that take place during the climax of the film.

- We wanted to create tools that might be useful to other animators, both in the larger blender community and in the Middlebury animation studio. More specifically, we were interested in creating tools which facilitate destruction animations.
- We wanted to learn about the inner-workings of the Blender software: the data structures that store mesh and animation data, tools that edit meshes, algorithms that generate apparent randomness, and how scripts could be effectively employed in animation workflows.
- We wanted to gain experience working on projects whose scales extended beyond our own contributions - in this case, both the software and the film.

As is to be expected with open-ended independent projects, we refined our goals over the course of the semester as we became increasingly aware of the challenges we faced and how we could feasibly overcome them in the timespan available to us.

A Note on the Shot

As a whole, the film offers a child's perspective on deportation in the United States. When her father is taken in by U.S. Immigration and Customs Enforcement, Estrellita's fantastical imaginary world comes crashing down around her as the crushing reality sets in. In the film, this transition is conveyed by the collapse of three structures that Estrellita had built up in her imagination: the barn where her father worked (the film is set on a dairy farm in Vermont), the trailer where they lived, and the cow-shaped mountain in the distance.

We spent almost the entire semester focusing on this last shot, the collapse of the cow-shaped mountain. The object itself, "cow_mountain," was built over the summer using a clay model and a 3D scanner. This procedure resulted in an unusually complex mesh with prominent non-manifold geometry, a fact that became a serious issue as the semester progressed.

Pre-existing Destruction Tools

We began our destructive efforts by investigating existing fracture tools. Such a tool takes a single mesh object as input and outputs a set of shard meshes, a “broken” version of the original mesh (see Fig. 1). With such a tool we hoped to fracture the cow_mountain object, generating shards that would fall apart or explode outward.

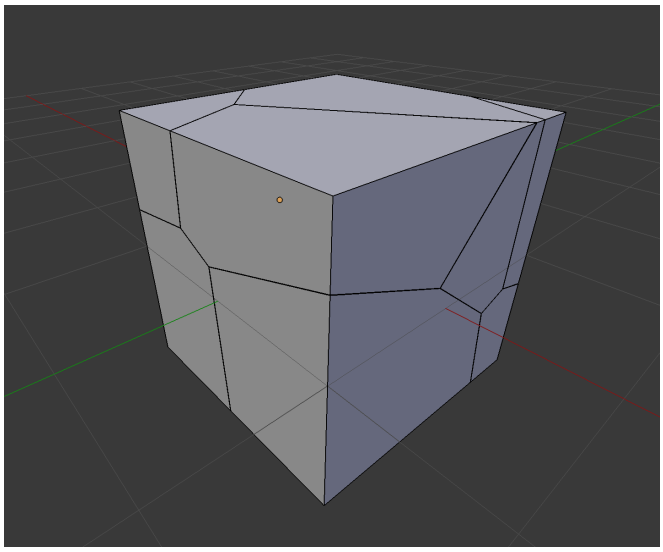

Figure 1. Shards. The shards resulting from a cube fractured using the cell fracture add-on. With these particular settings, eight shards are produced

The default version of Blender has no fracture tool of its own, but as Blender is an open-source project, two working modifications offer fracture capabilities. The first is an add-on available from within the default blender. The second is scorpion81’s Fracture Modifier Build, a forked version of the entire software package. After an examination of the capacities of each, we decided to begin working in-depth with the add-on. This decision was made primarily on the basis that using the Fracture Add-On allowed our work to be more easily integrated into existing project files. Use of the Fracture Modifier requires the animator to commit to an integrated Fracture Modifier workflow which involves mesh creation, physics simulations, and keyframing. The Fracture Add-On code was also far more accessible because the Modifier code included numerous tweaks and optimizations to core Blender files which obfuscated the salient pieces of code relevant to fracturing objects.

Working with the Add-On

The bulk of the add-on was written several years ago by pildanovak. Though there exist some online tutorials concerning its basic use, the support community is small compared to that of other frequently used Blender add-ons. The add-on is directly available within default Blender and can be enabled in 'User Preferences.' Because of the widespread availability of the add-on to other members of the Blender community, and because of the apparent accessibility of its source code, we decided it would be a suitable base from which to develop our project.

The add-on works by creating a bounding box - a cube that encompasses the entire mesh - and then fracturing this bounding box in one of several ways, each based on a different set of points. The resulting fractured bounding box is then combined with the original mesh using Boolean modifiers. This constructive solid geometry creates a set of shards, the output of the add-on.

The bounding box fracture relies first on a set of points in 3D space. In the cell fracture add-on, these points are chosen by the user. The add-on offers four preset options for deriving these points from the geometry of existing objects (see fig. 2) and a fifth option to create a custom set of points using the grease pencil tool. From these points an algorithm generates the new meshes of the fractured bounding box. To produce these intermediate meshes, most programs make use of the Voronoi tessellation method through the Voro++ library, a library written in C++ and tailored to three dimensional geometry. Unusally, the Cell Fracture Add-on implemented its own Voronoi method.

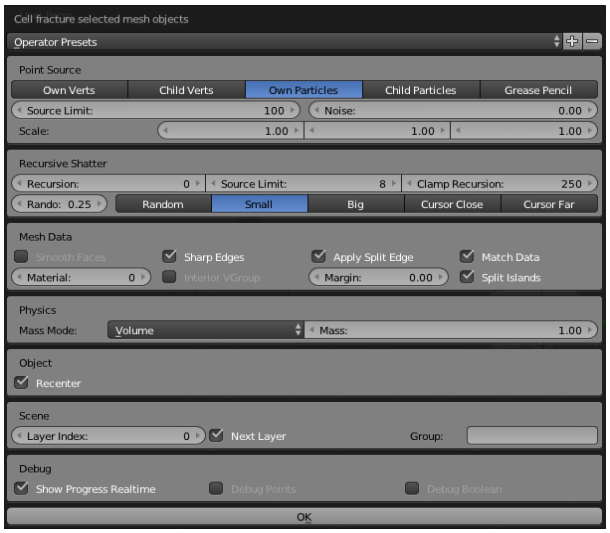

Figure 2. **GUI of Cell Fracture Add-on.** The bar at the top (own verts, child verts, own particles, child particles, grease pencil) offers five options of point sets on which to perform a Voronoi tessellation.

The first and most obvious issue to address in the Cell Fracture Add-On was a flaw in its application to animated objects: resulting shards did not conserve the momentum of the original object.

> *A Note on Animation in Blender:* Blender simulates all animation using keyframes. A keyframe belongs to an object and consists of two pieces of data: a particular frame and a particular characteristic of an object (its location, rotation, scale, etc.). A location keyframe on frame 10 means that object will be in that particular location at that frame. Animation is the result of having multiple keyframes governing the same characteristic. Blender will interpolate and extrapolate data from two or more keyframes. The kinematics of an object is contingent on that object's animation data.

An object with animation data, when fractured using the cell fracture add-on, passed along none of that animation data to the resulting shards. For our purposes - imitating objects which shatter on impact with the ground - this unexpected feature of the add-on resulted in a quite unnatural appearance; falling objects seemed to lose all their momentum on the keyframe where we performed the fracture. A significant portion of our time this semester was devoted to addressing this issue.

To solve this issue, we first implemented a workaround within Blender's user interface. Once we had our methods settled, we implemented the workflow in pieces using scripts, then integrated this code into the add-on. Our workflow essentially consisted of copying animation data from the original object to the shards. We achieved this by parenting the shards to the original mesh (a constraint which ties the location of one object to that of another), keyframing the shards in two different frames immediately before the intended fracture, removing the parent constraint, and then extrapolating motion from keyframe data.

More formally, our solution can be broken down into the following steps:

1. Add resulting shards to the rigid body group. This means Blender will automatically cache animation data generated by a built-in physics engine.
2. Parent shards to original object, a constraint which applies transformations of the parent object onto its children.
3. Set frame to moment of fracture. This changes the frame of the shot which moves shards to appropriate position.

4. Remove parent constraint.
5. Set keyframes on shards with linear interpolation animation.
6. Reset parent constraint.
7. Move backward some frames (we choose ten).
8. Remove parent.
9. Set keyframe.
10. Set extrapolation for all keyframes.
11. Swap original object for fractured shards by hiding original object and revealing shards at the moment of fracture.
12. Clean-up - deselect and deactivate all objects.

We found our modifications were successful in simulating the conservation of momentum and allowed the fracture to be applied without generating unnatural physics. However, iterative application of the add-on as well as its use on more complex meshes caused other issues.

Encountering and Investigating Limits

As we began to apply the cell fracture recursively and to more complex meshes, we began to run into problems with the appearance of the shards. Applying the add-on to more complex meshes would frequently result in visually flawed shards. Recursive use of the add-on with even the simplest meshes would also produce these flawed shards (see Fig. 3).

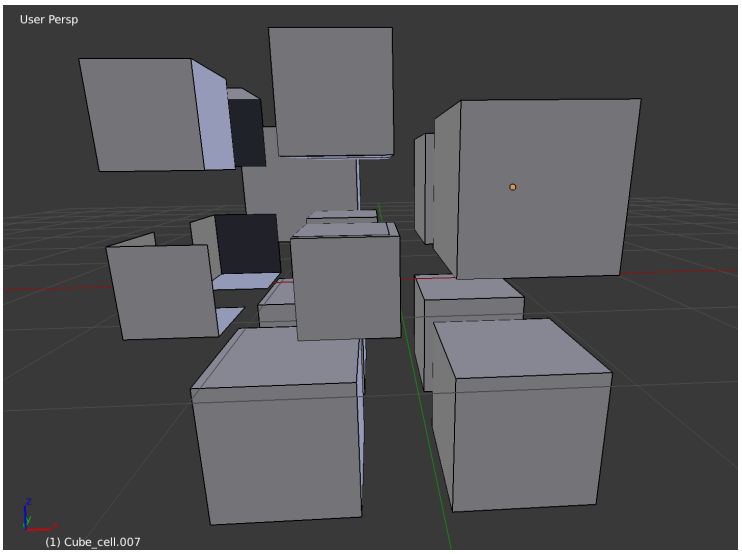

Figure 3. Iterative Fracture. Iterative use of the Cell Fracture Add-on applied to a cube. The first fracture produces eight visually correct but geometrically flawed (non-manifold) shard meshes. The second fracture, applied here to one such flawed mesh, produces shards that are both visually and geometrically flawed.

It was puzzling that high-quality, manifold meshes subjected to the add-on would output low-quality, non-manifold meshes that were unsuitable for further application of the add-on. (In this context a manifold mesh is essentially a watertight mesh without overlapping edges and faces.) We spent some time investigating the root cause of the issue with the intent of fixing the problem within the add-on's code. The issue lay within the add-on's use of Boolean modifiers. These modifiers are built into Blender and make use of a library for constructive solid geometry (CSG) called Carve. Carve is improper for use on meshes with non-manifold surfaces or high-connectivity points (verices at the confluence of many intersecting edges). These two requirements contributed to our issues with iterative application, but the primary problem we observed arose from the application of Boolean modifiers on objects with overlapping edges or edges that were too close together. Because the Boolean modifiers are a form of CSG, their application to overlapping faces cannot be expected to generate solid objects (watertight manifold surfaces). Realizing that the issue we were facing was caused by core features of Blender we also discovered the motivation for forking Blender to create an improved fracture tool. This lead us to decide the best approach to producing our shot was to use the Fracture Modifier forked version of Blender we had discovered and studied earlier.

In the course of our investigation into the failures of the cell fracture add-on we learned about how mesh data is represented in Blender as well as alternative topological data structures. As of the 2.63 release, Blender represents mesh data in a structure called bmesh. A bmesh structure is composed of four parts: vertices, edges, faces, and loops. Vertices, edges, and faces store data defining their locations

as well as links to cycles, connections between topological entities that enable the structure to maintain persistent adjacency information.

Working with the Fracture Modifier Build

Accepting that we would use the Fracture Modifier also entailed committing to animating the elements of the shot that made use of the Modifier. This is because we had to output a file which could be opened and edited in “out of the box” Blender. This commitment was reaffirmed when one of our studio peers attempted to edited some shots in the forked version and found he was unable to do so without the program crashing.

When we began working in the cow_mountain.blend file, we began to spend time considering the design of the shot. Would the whole mountain explode upward? Would ominous cracks grow from the base? Eventually we settled on a concept consisting mostly of the collapse of the cow-shaped portion of the peak.

The process of collapse began with a separation of the cow head from the rest of the mountain. This separation was achieved with the use of a growing crack. Though the slow cracking sequence did not make it into the final cut of the shot (the shot became too long), it featured prominently in earlier versions and is still there for those quick enough to spot it. The cracking effect was created with the use of nested Boolean modifiers. The visible mountain object is the result of cow_mountain mesh minus a crack mesh object that we created. From this crack mesh, in turn, we subtracted a cube. As the cube moves, the crack mesh grows, translating into a negative space in the cow_mountain mesh.

From here the head falls downward, a motion that is partly manually animated and partly directed by Blender’s rigid body physics simulation. The cow head splits in two, a manual animation achieved with Boolean modifiers. As these two meshes reach the ground (that is, a carefully placed plane object - there is no real ground in the spot where they fall) the fracture modifier takes over. At the moment of impact, the two head objects are swapped out for two sets of shards generated by the fracture modifier. These shards interact with each other and with the ground plane object as rigid bodies. To stabilize the shot, we converted the physics simulation data to keyframed object data. This also makes the shot easy to

integrate with the rest of the movie, which is made in “out of the box” Blender as opposed to the forked version.

Though the fractured head was aesthetically pleasing, the shot still looked somewhat unnatural; the objects seemed small and ceramic rather than large and mountainous. To counter this appearance, we rigged several particle systems to simulate the snow, dust, and medium-sized debris of the collapse. These ‘smoke and mirrors’ particle systems add a great deal of realism to the shot.

Script writing for other people

Throughout the semester studio members encountered a variety of issues which could be addressed by writing Blender scripts. We wrote three such scripts.

When creating a film in Blender artists often create two versions of models - a low resolution for animating and a high resolution version for rendering. The high resolution model will appear in the film. Working with a high resolution model during animation would create lag which interferes with an animator's sense of timing and their ability to animate effectively. This problem is ameliorated by introducing a low resolution model. This introduction however creates a new issue - the necessity to manually swap models at render time. We wrote a script which automated the process so when a shot was render all models were automatically swapped with their high resolution version and then automatically swapped back with their low resolution version after rendering. The script was designed in such a way that a user could easily specify which models have multiple versions which should be swapped at render time.

Figure 4. Resolution Swapping. Left: low-res version of Estrellita’s hair; right: high-res version.

The next script we wrote was to simplify the modelling process. Another artist in the studio was working with models that had a number of lattice modifiers applied to different components of the models. This type of modifier deforms meshes. While editing the models she had to filter-search for these modifiers and then manually toggle the modifiers on/off so she could work on the original meshes. She asked us to develop a tool which would automate this process. We wrote a script for this which also generated a button that could be used to toggle all lattice modifiers on or off.

Figure 5. Lattice Visibility Toggle. Left: Chevy Tahoe model with lattice modifier visible; right: without.

The last script we wrote was to handle a cycle dependency issue which caused animated milk constrained to a path to shoot out of the tube it was travelling through. This issue was easily resolved by automatically shifting frames forward and back to the start shot on render.

Figure 6. Path Dependency Cycle Issues. Single frame of the shot with milk ejection fixed.

Learnings and reflection

To review, our goals for the semester were to:

- Make a meaningful contribution to the film *Estrellita*.
- Create tools that might be useful to other animators.
- Learn about the inner-workings of the Blender software.
- Gain experience working on projects whose scales extended beyond our own contributions.

We felt some measure of success in achieving each of these goals.

We completed the animation of a destruction shot which is ready for lighting and material refinement, and will set the mood of the film's ending.

We created tools which helped other animators in the studio achieve their goals. This success deviated from how we perceived it would be. We expected the studio to use our destruction tool, but ultimately it was our custom tools which were used. Developing these custom tools gave us the valuable opportunity to think about the relationship between technical and intuitive interfaces and when it is appropriate to introduce automation into a workflow. The distinction between technical and intuitive is well represented by the scripting and 3D View environments in Blender. The scripting environment requires a user to write programs in python and the 3D View environment is a WYSIWYG (What You See Is What You Get) interface. Although I would not claim that either is intuitive, the 3D View is certainly more so. What was interesting about developing our tools was that they crossed the threshold between working in these two environments. In Blender there are many workflows which lend themselves more to one environment than the other. Keyframing large numbers of shards was monotonous and called for automation but timing pivotal cracks required feeling the movement of the shot and animating by hand. Building tools which bridge these two environments allowed us to empower users who may be more comfortable in one environment with the capabilities of the other. Through this work we found a reflection within Blender of the cross-disciplinary work we were doing in the studio and the value of integrating computer science with art. The opportunity to augment each with the advantages of the other. Although we did not end up using the our modified destruction tool for the *Estrellita* film we still produced a distributable modified Cell Fracture Add-On which conserves momentum.

We learned how to script in Blender, which entailed learning how Blender works - specifically the way different object properties are stored with meshes in dictionary data structures and how attribute accessibility changes with context. We also learned about bmesh object representation and how Boolean modifiers are implemented using the carve library.

Lastly we gained a lot of experience working on two very large scale projects: Blender itself and the *Estrellita* film. Working on projects of such scale entailed many challenges which are not typically encountered in regular coursework. One such challenge was the necessity to “feel around in the dark” or to pursue dead ends. This process allowed us to discover the context into which our work fit and to critically evaluate what contributions we could feasibly make in the time available to us. Some dead ends could have been avoided by performing more research but a majority of the time the only way to truly understand the problems was to encounter them for ourselves. Working on a large open-source project also provided us with the opportunity to read and study thousands of lines of code, written by others.

Overall, we both considered the experience of working in the animation studio highly valuable, enjoyable, and uniquely different yet complementary to the our other coursework.