University of Waterloo

Faculty of Engineering

Material-based Prefracturing for Rigid Body Destruction in Visual Effects

Side Effects Software

123 Front Street, Toronto, ON

Prepared By

Wayne Wu

ID 20563585

4A Department of Systems Design Engineering

September 20, 2018

Abstract

Material-based prefracturing is a critical step to obtaining visually realistic destructions in visual effects. Houdini, as it is, already provides tools to handle rigid body destruction. However, due to a complicated workflow coupled with various technical difficulties in the software, the process to fracture geometries and set up a correct foundation for simulation can be challenging. In this project, we provide an automatic tool to prefracture geometries based on three material types: concrete, glass, and wood. In doing so, the general destruction workflow in the pre-simulation stage is redesigned to optimize in performance, art-directability and the overall efficiency of the system. The development of the material-based fracturing toolkit will serve as the basis for future adaptations around rigid body destruction in Houdini.

Contents

List of Figures

1

Introduction

Hollywood movies are recognized for their astonishing visual effects. The use of visual effects brings liveliness and realism that elevates the overall perception of the art and the stories within, which is commonly used in the entertainment industry such as films, animations and games. Visual effects, or VFX, are created with the incredible combination of computational physics and computer graphics where physical phenomenon, such as fluid motions, cloth buckling, muscle deformation etc. are simulated and rendered pixel by pixel for visualization. This process is typically packaged as a 3D software, such as SideFX's very own *Houdini*, which internally deals with all the mathematics, programming, and hardware acceleration required while exposing controllable parameters through a graphical user interface. This allows VFX artists to model and control the simulation from an artistic point of view and produce the desired result without advanced technical background.

Among all VFX categories, rigid body dynamics/destructions (RBD) is one of the most frequently used elements. Common uses of RBD include explosion, collapse, and object impact etc. With the help of Bullet [1], a physics engine that simulates rigid body dynamics and handles collision detection, the destruction process after an object has been fractured is relatively trivial. However, a realistic destruction requires the fracturing of object to follow the properties of material and behave according to fracture mechanics. Although physically-based computational methods, such as FEM, have been developed to model the fracturing behaviour accurately, the computational cost is beyond practical feasibility thus avoided in VFX. Instead, the widely used approach is to model each fractured piece first then sticking it back together, hence the term *prefracturing* in the industry. The prefractured object will then break apart during the simulation.

However, prefracturing object to realistically portray the actual fracture behaviour of the specific material is a very challenging task. Although Houdini has tools making it possible to obtain a realistic result, the complexity makes the process extremely time consuming and not at all beginner-friendly. While larger studios have the resources to augment native Houdini tools and ease the process, smaller studios are hit most severely with the bottleneck. In this project, new tools and framework is introduced as part of the next release version of Houdini, H17, to reduce the time spent on the pre-simulation setup of RBD and make the whole destruction process in Houdini rudimentary to anyone.

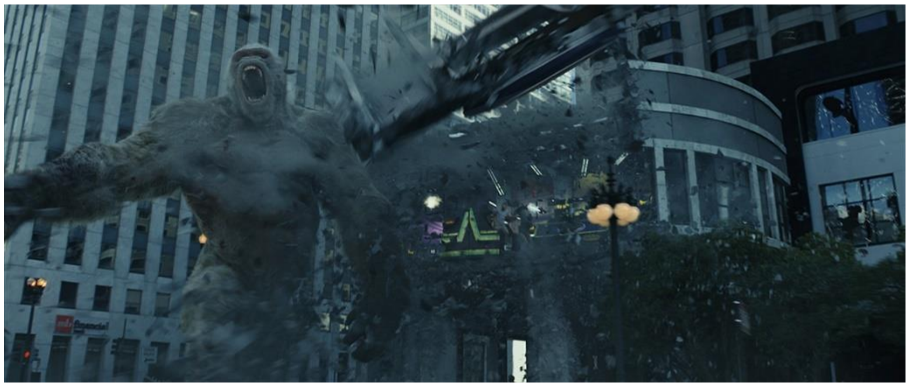


Figure 1: Building destroyed from the movie Rampage (2018).

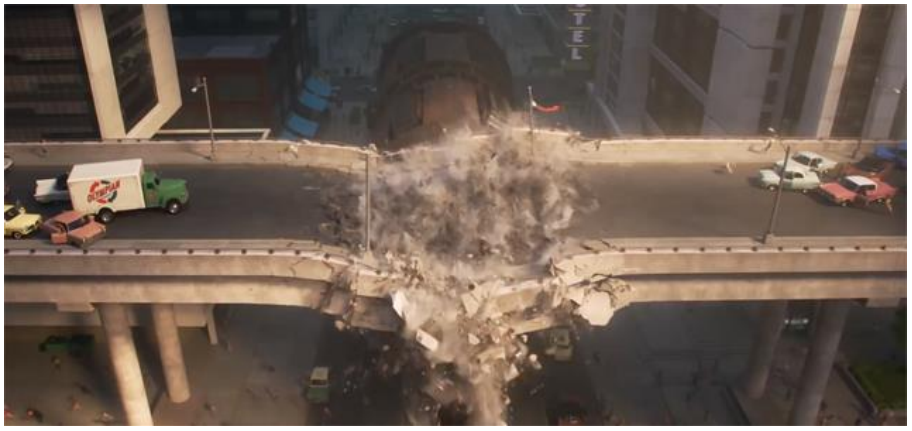


Figure 2: Bridge collapsing from the feature animation The Incredibles 2 (2018).

2

Background

2.1 Proceduralism in Houdini

Breathing for more than two decades in the industry, Houdini has become one of the most prominent software applications for VFX and 3D animation. Though often considered as a very technical software, it appropriately comes with extremely powerful tools that allow artists to create virtually anything in the 3D world. In addition, the procedural workflow used in Houdini gives transparency to the flow of data in the software, which is visually presented as a hierarchical network in the software, known as the Network Editor (Figure 3). With proceduralism, artists can make changes to the data or operations at any stage of the network, and the following processes will reflect the new changes accordingly. This reduces drastically the iteration time of a complex system.

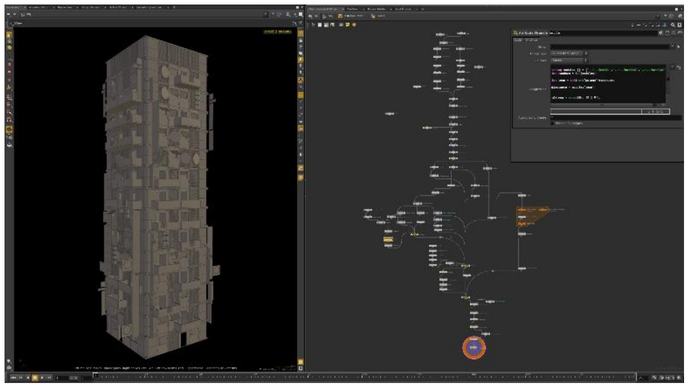


Figure 3: Interface of Houdini. The viewport (left) and network editor (right).

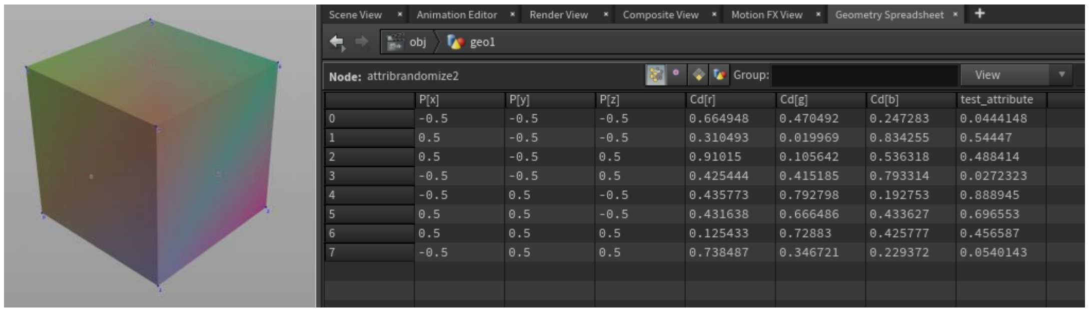
Nodal Operators

Houdini is a node-based software. Each node represents an operation, thus also sometimes referred to as an "operator". With over 500 unique operators, each node is categorized into a specific node type that indicates the context in which the node can be used. For example, the Geometry and Dynamics node types are used predominantly in this project. The Geometry nodes, also referred to as SOP, can only be used in the Geometry network to modify geometry data. Any operations that changes the geometry, such as modeling tools, are SOPs. The Dynamics nodes, or DOP, are nodes that deal with simulations and dynamics, such as the integrated Bullet solver. They can only be used in the Dynamics network. Proceduralism is achieved by connecting nodes in a sequential order. The network created presents how the data are processed by each node procedurally one after another. Figure 3 shows the network of SOPs for the procedural building. Each node is either implemented in C++ or built from other nodes to create a subnetwork.

Geometry Data

Geometries in Houdini are represented as *points*, *vertices*, and *primitives*. A primitive is made up of vertices to represent either a polyline or a polygon. For example, a primitive with two vertices would represent a line. Likewise, a primitive with four vertices forms a quadrilateral. Each vertex of a primitive is unique to that specific primitive. Therefore, two or more primitives cannot share the same vertex. Primitives can, however, share the same point which is referenced by the unique vertices. Complex geometries are essentially made up of thousands of primitives and points to form a defined mesh.

Each geometry type carries *attributes*. Attributes are data that can be added and modified by SOPs. These data can be a single value, or an array, of integer, floating point, or string data type. Each point, vertex, or primitive carries its own values of the attributes, which can be accessed by knowing the specific point, vertex, or primitive number. A very important attribute is the position of points, which is stored as a vector of float. One can see all the attributes in a geometry using the Geometry Spreadsheet as shown in Figure 4. Almost all operations done in the context of Geometry involve manipulating these attributes, which is fundamental to using Houdini.



Node: attribrandomize2

	P[x]	P[y]	P[z]	Cd[r]	Cd[g]	Cd[b]	test_attribute
0	-0.5	-0.5	-0.5	0.664948	0.470492	0.247283	0.0444148
1	0.5	-0.5	-0.5	0.310493	0.019969	0.834255	0.54447
2	0.5	-0.5	0.5	0.91015	0.105642	0.536318	0.488414
3	-0.5	-0.5	0.5	0.425444	0.415185	0.793314	0.0272323
4	-0.5	0.5	-0.5	0.435773	0.792798	0.192753	0.888945
5	0.5	0.5	-0.5	0.431638	0.666486	0.433627	0.696553
6	0.5	0.5	0.5	0.125433	0.72883	0.425777	0.456587
7	-0.5	0.5	0.5	0.738487	0.346721	0.229372	0.0540143

Figure 4: Geometry spreadsheet (right) showing the point attributes of the box (left).

2.2 Fracture Modeling

Crack propagation of materials is governed by the studies of fracture mechanics. While accurate physically-based computational methods have been developed, the cost of computation makes it unsuitable for the VFX industry [2]. Alternatively, the prefracturing technique is used. Prefracturing usually refers to the pre-simulation setup required to achieve the desired visual outcome. This includes not only the shattering of objects, but also the establishment of a correct constraint network between fractured pieces for simulation [3]. In this section, the relevant processes of prefracturing will be introduced. These concepts are crucial to designing the solution.

Shattering

The first and most important step of prefracturing is breaking the geometry apart. The two most commonly used approaches are Voronoi Fracture and Boolean Fracture. Each method has distinct advantages and disadvantages which will be compared more thoroughly in the later chapters.

Voronoi Fracture

Voronoi Fracture, as the name suggests, uses Voronoi diagram to partition the object into fragments. This process takes in a set of defined *generators* (cell points), and output regions (cell) for every generator, which any point inside the given region is closest to its corresponding generator. If a point is equidistant to two or more generators, a boundary is formed [4]. Figure 5 shows an example of a 2D Voronoi diagram based on the cell points.

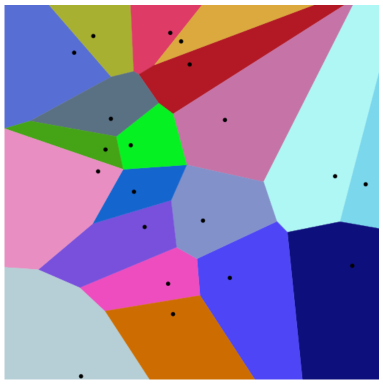


Figure 5: 2D Voronoi diagram.

Using Voronoi diagram to fracture object is trivial as it simply takes a set of points, which can be randomly generated or generated based on a specific distribution and yields naturally sized fragments. Typically, a point cloud will be created within the object from which the Voronoi fracture can be executed. Figure 6 shows a simple Voronoi-fractured box in Houdini.

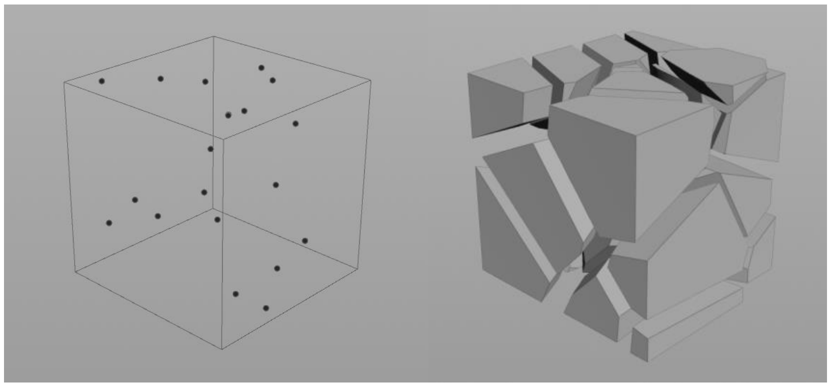


Figure 6: Voronoi-fractured box (right) from 20 cell points (left).

Boolean Fracture

Boolean Fracture uses the concept of Constructive Solid Geometry (CSG) to break object apart. CSG, in computational geometry, is an extremely powerful way of modeling by applying Boolean operations, such as XORs, add, subtract etc., between models. This allows 3D modelers to create very complex model from simple shapes and do so repeatedly, as shown in Figure 7.

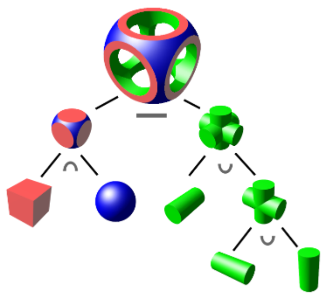


Figure 7: Concept of CSG.

CSG can be used for fracturing by applying the "cookie cutter" technique. Instead of applying Boolean operation between two solids, the solid geometry to be fractured is subtracted from a cutting surface. This allows the object to shatter based on the intersection with the cutter. Figure 8 shows a simple Boolean fractured box in Houdini.

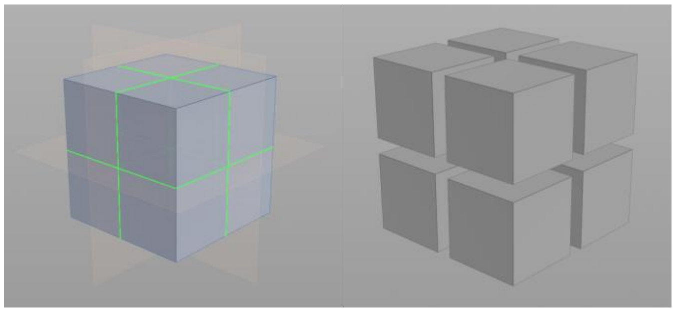


Figure 8: Boolean-fractured box (right) using the cutting planes (right).

Clustering

Once a geometry is fractured down to pieces, a user can then cluster the pieces together to create a larger piece. There are a couple reasons for doing so. One may cluster pieces together to reduce the overall number of pieces in simulation, thus making the simulation more performant. Using compound convex hulls, it also allows the calculated convex hull of the cluster to be more accurate for collision detection. The more practical reason for clustering is to obtain interestingly shaped pieces. By combining multiple pieces together, a new shape is formed, which can be more visually realistic in some cases.

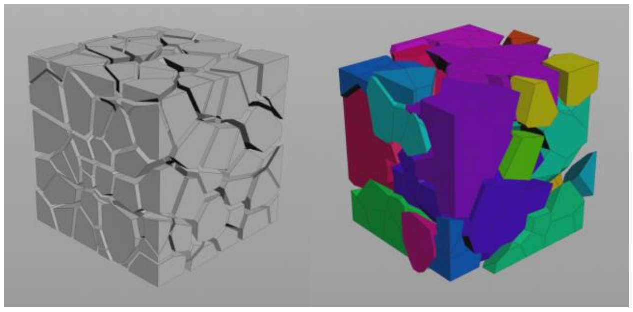


Figure 9: Original (left) vs clustered (right) pieces

Constraining

Clustering combines pieces together geometrically as one new piece. Constraining, on the other hand, defines relationships between fractured pieces without modifying the geometry, using a constraint network. To "stick" the pieces together during the simulation, each piece is constrained to its adjacent pieces with properties assigned to each connection depending on the type of constraint used. The constraint network itself is a geometry as well, where each connection is represented as a line primitive and each point is the piece itself. Constraint properties are assigned as attributes to the line primitives.

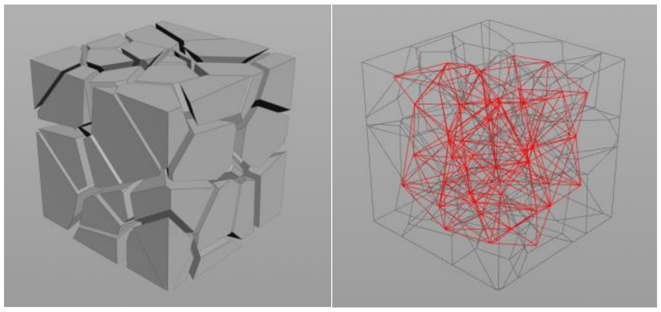


Figure 10: Fractured box (left) and its constraint network (right)

2.3 RBD Simulation

Bullet is the main physics engine used in Houdini for rigid body dynamics. While understanding the workings of Bullet is not required for this project, it is important to realize how the RBD simulation processes the geometries, and how the rigids interact with each other.

Name Attribute and Packed Primitives

Every fractured piece is tagged with a name attribute to uniquely identify the piece. This name is used by the simulation as well as many SOPs to partition the geometry based on the fractured pieces. Before passing the pieces into the simulation, each piece is turned into a *packed primitive*. A pack primitive, in this case, embeds the geometry information for the corresponding piece inside them, and retains only the name attribute explicitly. As a rule of thumb, the name attribute is stored as a primitive attribute when unpacked and promoted to a point attribute when packed.

Instead of using the whole geometry of a piece, a single packed primitive is used. This reduces the amount of memory used for a heavy destruction scene with large number of pieces and makes the simulation and rendering more performant. However, the geometry data inside the packed primitive cannot be changed without *unpacking*. This generally does not concern with RBD, since the fractured geometry is pre-generated and therefore not changed through out the simulation. Only transformation is applied to each piece during the simulation. As a good practice, packing of the geometry should only be done at the very end of prefracturing before the simulation.

Render vs. Collision Geometry

Bullet handles collision detection using convex hulls [1]. By default, every fractured piece is approximated by a convex hull. If the piece is geometrically composed of multiple smaller pieces, like a cluster, compound convex hull can be used, which approximates the collision geometry by combining the convex hulls of all sub-pieces. This increases the accuracy of the collision detection.

Typically for a highly detailed geometry, a simplified proxy geometry is used instead for the simulation. The use of a proxy geometry makes the simulation faster as the geometry is simpler compare to the highly detailed geometry. Depending on how the proxy geometry is generated, it can also improve the result of the collision detection. One can easily generate the proxy geometry by using convex decomposition. Voronoi-fractured pieces can be used directly as proxy geometry as they are convex. After the simulation, the transformation of the collision geometry at each frame is transferred to the more detailed geometry for rendering.

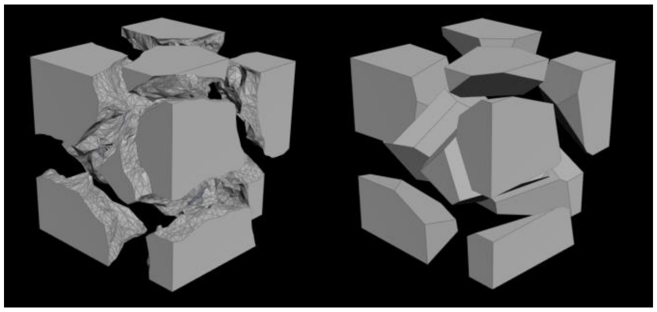


Figure 11: Render geometry (left) vs. collision geometry (right).

Constraint Types

There are many constraint types that can be used with Bullet in Houdini. Each constraint type has different *constraint properties* to define the break configuration between pieces. Among all, the most frequently used one is the *glue constraint type* which simply connects the pieces together based on a strength value. The connection will then be broken if the impact exceeds the glue strength. Another powerful constraint type that is recently introduced is the *soft constraint*. Soft constraint provides bending and spring-like connections as shown in Figure 12. Spring properties such as stiffness, damping ratio etc. are available to adjust for soft constraint.

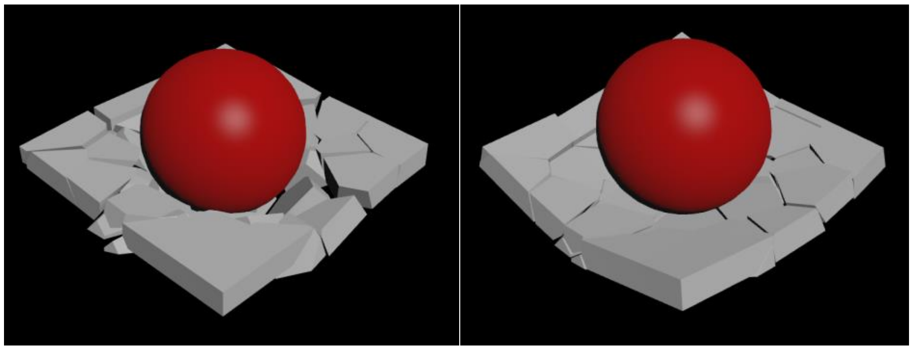


Figure 12: Glue constraint (left) vs. soft constraint (right)

3

Design Overview

This project focuses on the pre-simulation stage in the RBD pipeline, where a significant amount of work is required. Without learning the current prefracturing workflow involving many advanced techniques, it is difficult for users to create realistically fractured geometry and set up a correct constraint network. The goal of this project is to improve upon the prefracturing tools based on material-based fracturing, and establish an optimized, state-of-the-art workflow that can be extended further to easily build any destruction scene.

3.1 Motivations

The two objectives of prefracturing are creating realistic fracture appearance and following accurate break configuration. The former deals with the visual result of the fractured geometry, while the latter is achieved with a correct setup of constraint network. In both cases, there are many apparent problems complicating the overall process of RBD.

Existing Problems

Prefracturing in Houdini is complicated for various reasons. First and foremost, it is difficult to obtain visually realistic fractured geometry. In most cases, the "Voronoi-look" that a basic Voronoi fracture produces is undesirable. The lack of details on the edges and interior faces of the pieces makes it unrealistic. However, given the nature of Voronoi fracture, it is difficult to control the shape of the edges. Various techniques such as clustering and edge manipulation can be used for polishing, but they are still limiting to certain extents. While Boolean fracture can easily produce edge details, the challenge with Boolean fracture is knowing how to cut the geometry correctly to produce realistically-sized pieces — something that Voronoi fracture handles well. It is worth noting that edge noising is still considered an unsolved problem in the industry.

Once a detailed fractured geometry is obtained, the next challenge arises when a user begins to deal with the constraint network. As mentioned in section 2.2, the existing way of creating constraint network relies on the Connect Adjacent Pieces node to create the constraint network, after all geometries have been fractured. However, when working with a more complicated, heterogenous, multi-material setup, it becomes difficult to isolate specific connections and assign constraint properties for simulation. The assignment of the constraint properties itself is also a non-intuitive process as users must create the attributes manually, not knowing exactly what parameters are available to the users.

Finally, when fracturing for a largescale destruction scene, the iteration time is often very long. In a large destruction scene, such as a city falling apart, the total number of pieces can easily go over 100,000 pieces. To fracture each building would take hours of processing time. This means that every tweak of a parameter would require hours before it is reflected to the whole scene. In most studios, where fast turnaround time is required, this becomes a heavy bottleneck to deal with [3]. On top of fracturing, there is also the actual simulation to process which can also be hours of computing time depending on the resources available.

Related Works

To address some of the problems above, larger studios have created tools to ease the process, such as Weta Digital's *whRigid* [5] and Industrial Light & Magic (ILM)'s *SimpleBullet* [6]. SimpleBullet was created back in 2015 and has since been integrated into Walt Disney Animation Studio (WDAS) and Pixar's pipelines. These tools were presented recently in SIGGRAPH '18. While whRigid focuses more on asset integration with Weta's internal pipeline, SimpleBullet is a more general toolkit that wraps around the complexity of destruction by abstracting processes and shifting away from DOPs to SOPs. It provides a streamlined, artist-friendly way to deal with the simulation while also providing useful tools to perform fracturing.

At this point in time, SimpleBullet is no longer being actively developed. Its maturity makes it an excellent source of reference for this project to understand the requirements of large studios. However, only limited access to its implementation is available since it is a proprietary tool. Nonetheless, for this project, the destruction team at SideFX has the privilege to work with artists and technical directors from WDAS to come up with the design requirements. This allows the team to understand the benefits and challenges of SimpleBullet from users' perspectives. It is not an over statement to say that the long-term goal for this project is to replace SimpleBullet with a solution provided natively from Houdini. However, to fully replace SimpleBullet would require many iterations of work that is simply beyond the current target of the project.

3.2 Objectives

With a clearer understanding of the problem space, six main design objectives are listed to drive the development of the solution. These objectives are created based on discussions with WDAS as well as analysis on the existing workflow.

Aesthetic

As emphasized repeatedly, achieving the desired visual result is the purpose of fracturing and destruction in general. In the context of live action films, this translates to how realistic the destructions look in comparison to real-life footages. For animation studios, such as WDAS and

Pixar, more stylized and artistic look is often desired [6]. The solution must therefore strive to allow both realistic and stylized fracturing. It is important to reference both real-life footages as well as exiting shots from films throughout the development.

Art-Directability

Art-directability is a measure of easiness to control the result of simulation with artistic inputs. Parameters and properties that are exposed for the simulation are usually very specific and technical. From an artist's perspective, parameters like damping ratio, impact propagation rate etc. are hard to conceptualize in relation to the result of simulation. Since artists are visual-oriented, painting and drawing with colour feedback are usually the most intuitive way for them to drive a simulation. However, to translate between technical and artistic inputs is an extremely challenging task, thus not often seen in many 3D software. Art-directability also works against proceduralism making it extra difficult for Houdini. Nonetheless, this project will aim to tackle the area of art-directability to allow studios to meet artistic requirements more efficiently [7].

Flexibility

This project focuses on destruction of buildings as a starting point. While modern skyscrapers are simpler in geometry, cultural and artistic architectures tend to be more complex. It is important to provide flexibility to fracture objects in various forms, shapes and sizes. Additionally, the ability to customize the fracture pattern is also essential for artists to randomize and obtain the desired appearance.

Functionality

The full destruction pipeline includes pre-simulation, simulation, and post-simulation setup. Providing a one-for-all solution is the long-term goal of the destruction team, with this project as a starting point that focuses in the pre-simulation stage. Within pre-simulation, there are many sub-functions that can be improved and automated. The solution coverage of the destruction pipeline will determine the overall usefulness of the feature.

Performance

Working in a fast-paced environment, the iteration time of doing shots becomes the bottleneck for studios [7] [3]. For a large-scale scene with simulation involved, the iteration time is usually affected by the computation performance of the software. In the context of destruction and Houdini, there are two main performance driven processes. The first is the processing time for fracturing all geometries in a scene to be destroyed. The second is the computation time for the simulation. Since this project focuses on the prefracturing workflow, the primary performance constraint will be the fracturing time. However, as the simulation time is affected by the number of fractured pieces as well as the complexity of the collision geometries, the computation cost for simulation will be considered as well.

Extendibility

While providing a functional tool that performs fracturing automatically is crucial, it is just as important to allow users to learn and adapt from the solution provided. As a native feature in Houdini, clients will tend to reference it as an example for custom usage. Therefore, the solution must be carefully designed to be optimized and extendible by users. Even if larger studios may have tools like SimpleBullet, it will still serve as an important building block for them.

4

Implementation

The implementation starts by understanding the current prefracturing process and the improvements that can be made to satisfy the design objectives. This is done by fracturing, from scratch, different materials that are commonly used for destruction, after which improvements are made to the system. The fracturing setups will then be made available to use directly as a material-based fracturing tool — the RBD Material Fracture SOP.

In this project, concrete, glass and wood are chosen to investigate further, which are common materials found in buildings. These materials have distinctively different properties, making them excellent choices for fracturing. While metal is also abundant in structures, its deformable characteristic makes it slightly more complicated to prefracture. For simplicity, metal is not included in the current implementation, though the workflow designed should be applicable to metal as well as any other material-based fracturing.

4.1 Concrete Fracture

Concrete is the most widely used material type in construction of buildings [8]. Unsurprisingly, it plays the most significant role in a lot of destruction scenes. The common approach of fracturing concrete consists of a few steps that are applicable to any fracturing in general. The process begins by converting the geometry into a volume. Since the scattering of points is driven by the density distribution of the volume, a noise can be applied to the volume to obtain a non-uniform density distribution. This generates reasonably-sized pieces. However, as mentioned previously, the lack of detail makes it unrealistic. Therefore, an enhanced technique of fracturing is used.

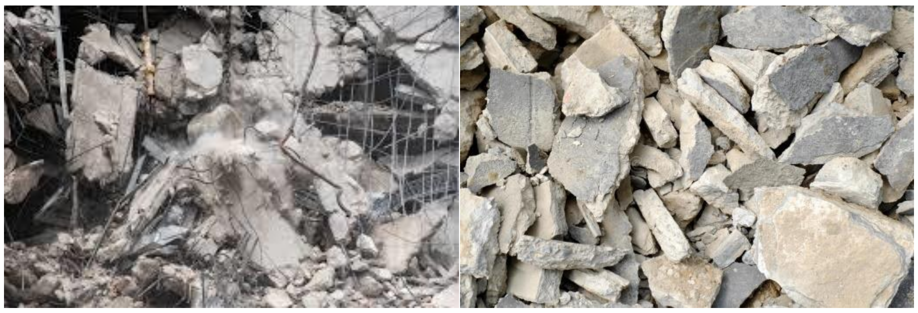


Figure 13: Reference images of fractured concrete.

Size Variation

The first feature of fractured concrete is the variation of size in pieces. This can be seen from Figure 13 as well as the crack pattern in Figure 15. Although variation can be obtained by using a non-uniform distribution of cell points, it is inefficient and difficult to control the noise, via its frequency or offset, for obtaining the desired size variation. The nature of Voronoi diagram also makes it challenging to control the size of specific cells.

Another approach to achieve size variation is the use of clustering. By using clustering based on cellular noise and randomly detaching pieces from selected clusters, one can obtain varying sizes of clustered pieces such as the fractured box in Figure 9. Typically, the geometry is fractured down to the smallest desired size before clustering them back together to yield larger size variation. Despite producing a nice variation in size, the merging of multiple pieces makes the surface very jagged and more like crystal than concrete, as shown below.

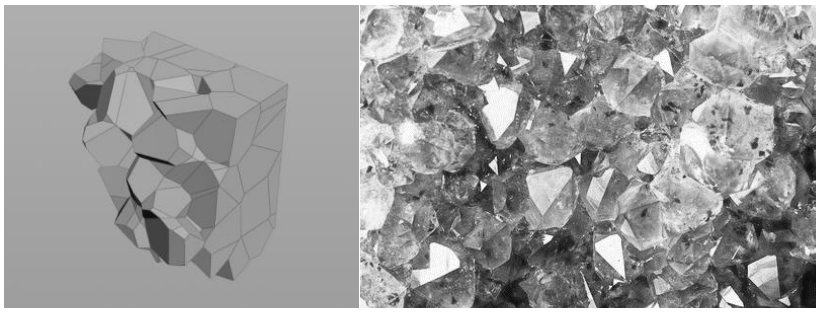


Figure 14: The surface of the clustered piece (left) and the surface of crystal clusters (right).

The better approach uses sub-fracturing, which applies another round of fracturing to some fractured piece. This is applicable to the fracturing of concrete because the crack pattern (Figure 15a) can be viewed as multiple levels of sub-fracturing. In Figure 15b, the cracks are traced out based on a rough estimation of multiple sub-fracturing. The primary fracture is indicated by the red lines producing few large pieces. Each piece is then fractured again based on the orange lines to create the secondary cracks. Finally, another level of sub-fracturing can be added to obtain finer details as shown with the yellow lines. To provide size variance, only some of the pieces are sub-fractured down to smaller pieces leaving some large pieces untouched.

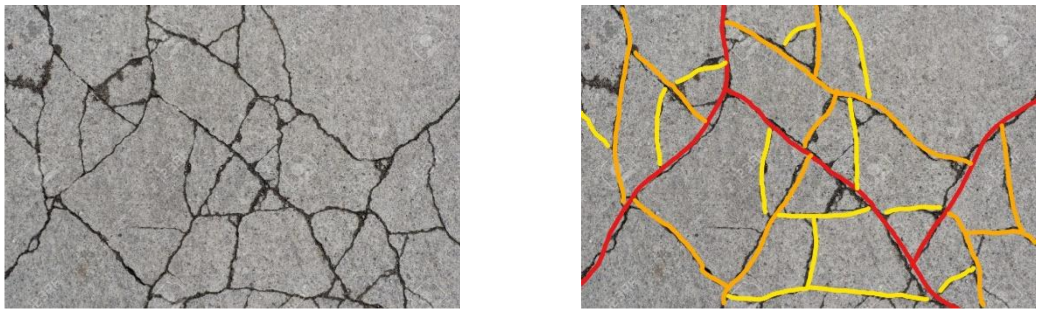


Figure 15: Original fracture pattern (left) vs. traced out pattern (right).

Chipping

Closely related to size variation is the idea of chipping. In the crack pattern of Figure 15, certain pieces have their corners "chipped" off to create small pieces. Providing the ability to chip away the corners adds in an extra level of detail to make the fracturing of concrete more realistic. To chip off the corners, Voronoi fracture is used in a specific way.

To begin with, the corner points of a fractured piece are randomly selected to be chipped off. If a corner point is selected, another point is added with small offset in the direction to the centroid of the piece. The corner points and offset points are then merged together to be the cell points used for Voronoi fracture. The boundary between a corner point and the offset point creates a cut for the corner, thus creating the chip. However, the piece is also cut by the boundary between offset points. Thus, all the pieces that are not chips are merged back together to form the original piece.

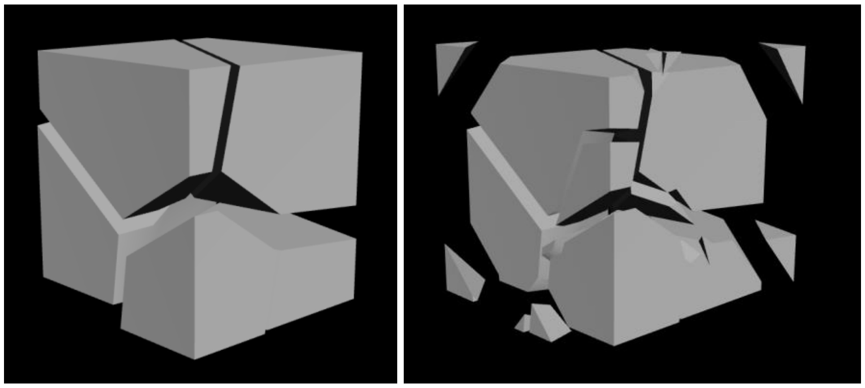


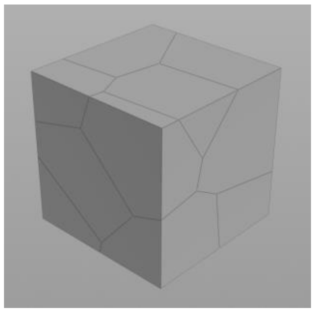
Figure 16: Fractured box without (left) and with (right) chipping.

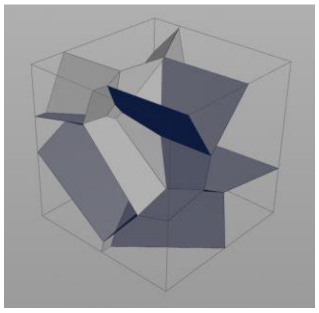
Edge Noise

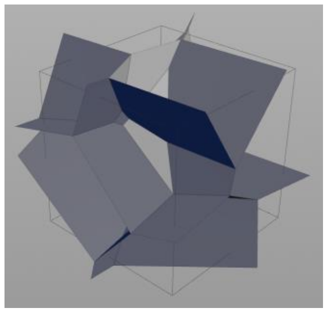
The other important feature identified from the reference images is the noise along the cracks. The higher frequency noise can be produced with shaders at render time, the lower frequency noise, however, must be reflected in the render geometry upon fracturing. Edge displacement, as mentioned in 3.1, is still widely regarded as an unsolved problem of fracturing. There are many ways of doing edge displacement, yet none of them is robust enough to be crowned as the best solution. The challenge arises from the fact that the original outline of the geometry must be retained after fracturing, for it to appear unfractured. When dealing with complex geometries, the directions to displace the points along the edges become very much ambiguous and error-prone. If a point is shifted in the wrong direction, the shape of the geometry may change, or a visible gap can be created between edges.

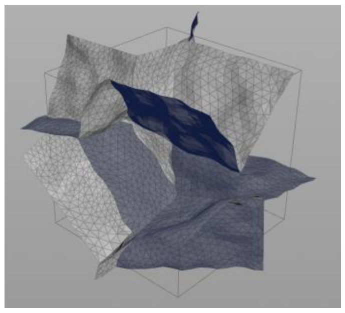
Instead of directly manipulating edge points for noising, another method is used that takes advantages of both Voronoi and Boolean fracture, as they complement each other nicely. Voronoi fracture has the advantage of faster processing and producing nicely sized pieces with trivial input, especially with the sub-fracturing mechanism. Boolean fracture, on the other hand, requires multiple custom cutting planes, which would normally yield *sliced* pieces instead of *fractured* pieces. However, Boolean's power comes from its flexibility to change the shape of geometry based on the cutting surface. If the surface is noised, the pieces will become noised along the edges as well. Therefore, the method leverages Voronoi's capability for size variation, while using Boolean for edge details.

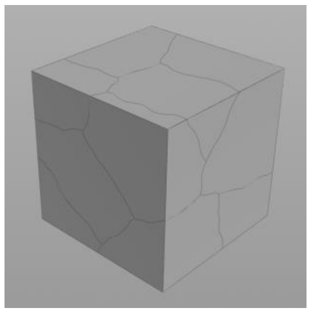
The basic process is as followed: after Voronoi fracturing the geometry to obtain the correctly sized pieces, the inside faces of each piece are extracted. This gives us a Voronoi-based cutting surface, which if fed into Boolean fracture as it is, would return the exact same output as the original Voronoi fracture. Before applying Boolean fracture, the cutting surface is extruded along the borders to be slightly greater than the geometry. This ensures that the geometry is fully cut by the cutting surface. Finally, the cutting surface is noised to create the desired edge displacement before using it in Boolean fracture. Figure 17 illustrates this process graphically.


a) Voronoi fracture


b) Inside faces


c) Extrusion


d) Resample and noising


e) Boolean fracture

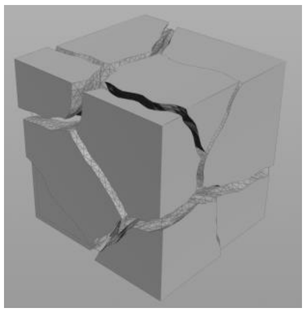

f) Exploded view

Figure 17: Steps for CSG Voronoi.

To make this method more robust, instead of using the original Voronoi-fractured pieces as the cutting surface (Figure 17b), the Voronoi fracture of the scaled bounding box of the geometry is used. The exact same scattering points are used to fracture the oriented bounding box that is scaled slightly larger than the object to ensure the cutting surface cuts through the object. Manual extrusion of points is no longer needed, which would have been a similar case as edge manipulation where the direction of extrusion is ambiguous. The new cutting surface also makes the cleaning of duplicated inside faces much more robust. Altogether, this method handles complicated geometry very well. Figure 18 illustrates this concept.

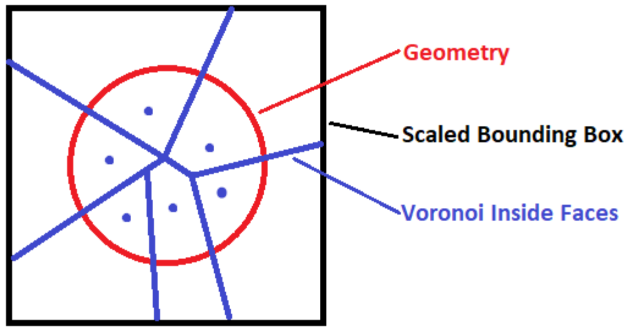


Figure 18: A more robust mechanism for obtaining the Voronoi cutting surface.

This technique is believed to be what SimpleBullet uses internally for fracturing, referred to as *CSG Voronoi* [6]. During the talk at SIGGRAPH 2018 on SimpleBullet, ILM briefly mentioned the usage of CSG Voronoi which, according to Pixar and WDAS, is one of the most useful features from SimpleBullet. However, since the implementation is not publicly available, the version implemented here is based entirely on the team's own research and development thus differ from SimpleBullet. In fact, according to a quick discussion with Will Harrower from ILM, the whole CSG Voronoi in SimpleBullet is implemented in C++ using HDK whereas the implementation here is purely using native Houdini nodes. Regardless of the differences, this hybrid method has proven to be much more desired and powerful for edge noising.

4.2 Glass Fracture

Glass fracturing can exhibit many different fracture patterns depending on the type and shape of the glass, as well as the cause of fracture [9]. To simplify the scope of glass fracturing, this project focuses on flat plates, which encompasses for most windows and glass panels used in buildings. Furthermore, only impact-driven fracture will be considered, ignoring other less common causes in destruction such as thermal and torsional fractures [9]. More generic fracturing applicable to curved glass and glass containers should be implemented in the future to provide more flexibility.

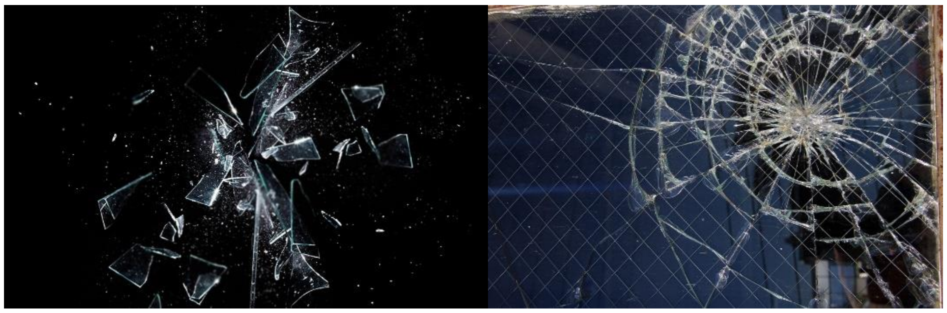


Figure 19: Reference images for glass fracturing.

In the context of fracturing, glass panels can be considered as either a float/annealed glass or tempered glass [10]. The former has more obvious fracture paths that traces back to the point of impact. The fragments produced are usually sharper and slivery as shown in Figure 20. Tempered glass, on the other hand, exhibits a more uniform pattern called *dicing,* regardless of the location of impact [9]. The fragments from dicing are more uniform and polygonal. Given this characteristic, dicing can typically be done with a simple and dense Voronoi fracture. The fracturing of float glass, however, is harder to produce, thus specifically analyzed here.

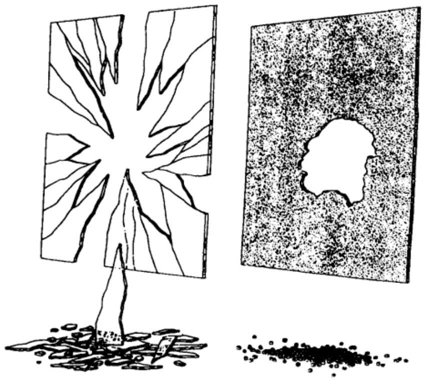


Figure 20: Annealed vs tempered glass

Cracks

The point of impact, or the *fracture origin*, determines how the overall fracture pattern should look like [9]. The propagation of cracks generates smaller fragments closer to the origin and larger slivers further away from it. Multiple fracture origins can also exist, which the order of impact can be determined based on the dominance of the cracks. The primary cracks, or *radial cracks*, are the first cracks to form instantaneously after the impact. These cracks extend radially outward from the fracture origin all the way to the edges. Any radial crack can be traced back to its corresponding fracture origin. In some cases, *forking* can also occur near the end when new cracks are branched out from a radial crack [10]. The secondary cracks, or *concentric cracks*, are formed immediately after the radial cracks. These cracks surround the fracture origin concentrically with higher density closer to the origin.

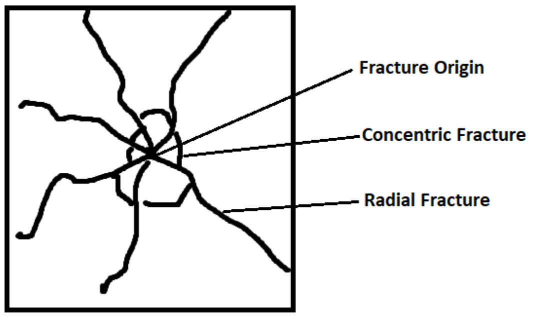
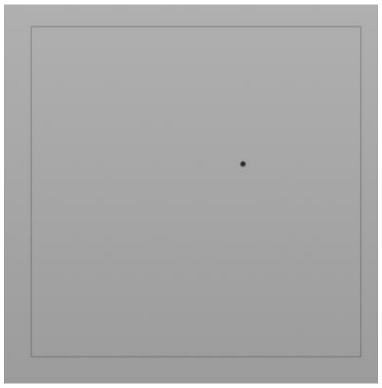


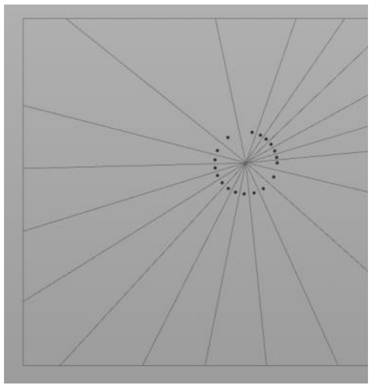
Figure 21: Impact-driven fracture pattern of normal glass.

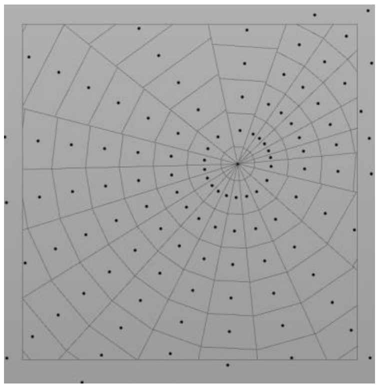
Fracture origin, radial and concentric cracks are the key features of glass fractography that should be reflected in the fractured geometry. Voronoi fracture is selected here as the fracturing method for its faster processing time and the discontinuous characteristic of the concentric cracks. Given the specific hierarchical fracturing order, the same sub-fracturing technique used in concrete can be applied here. However, unlike concrete, cell points are not scattered randomly. Instead, the points are placed in specific ways to force the desired pattern.

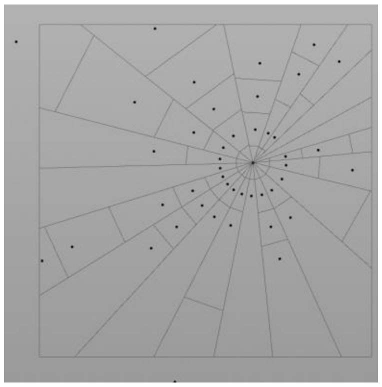
The tool begins by identifying the plane of cracking, which is normal to the direction of impact. When dealing with a flat plane that is as simple as four points and six faces, the two faces with the maximum area are automatically assumed here to be the surfaces of cracking. The rest of the faces are the borders that the radial cracks extend to. By default, the tool scatters points across the fracture surface to indicate the fracture origins. Users can optionally override the fracture origins and specify the exact locations. To generate radial cracks for each fracture origin, cell points are added around the fracture origin all with the same distance from it. This ensures that the boundaries of the Voronoi diagram are forced to extend towards the origin, thereby forming the radial crack.

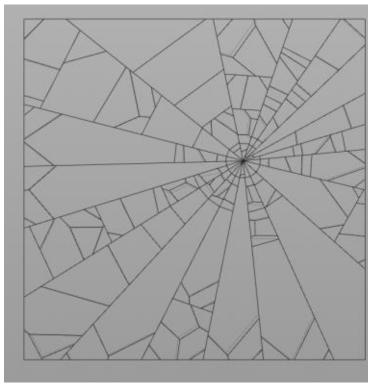
After the radial fracture, each piece is fractured individually to obtain the concentric cracks. Cell points are placed from the fracture origin all the way to the end of the radial piece at equidistance. This distance is the smallest possible distance between concentric cracks. At this point, a spider web is produced as shown in Figure 22c. To generate a more irregular pattern, a strength value is assigned for each concentric cell point and a noise is applied to the strength value based on its position. Points with strength value below certain threshold are discarded. To ensure that the area around the origin has more pieces, the strength is amplified closer to the origin for more concentric cracks. Users can tweak the noise, as well as the amplification, to get the desired concentric fracture pattern (Figure 22d). Finally, chipping is also added to provide an extra level of detail similar to concrete.


a) Impact point


b) Radial Crack


c) Spider Web


d) Discontinuity


e) Chipping

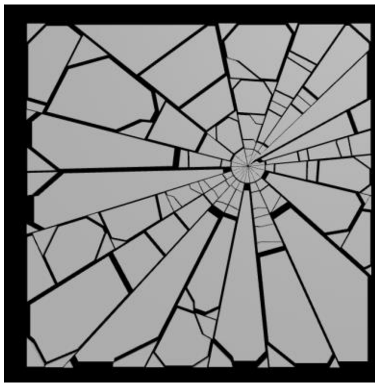

f) Exploded View

Figure 22: Glass fracture process.

Edge Displacement

Edge noising is also required for glass. However, CSG Voronoi is not necessary in this case as the geometry for a glass panel is simpler and assumed to be flat. The edge points are displaced along the fracture surfaces in fixed planes. For simplicity in noising, the z-axis is assumed to be the impact direction, while the x-y plane is the plane of cracking. Prior to fracturing, the glass panel is rotated to have the plane of cracking to face the z-axis. After fracturing, the cracks are resampled to increase the resolution for noising. Each point's position is shifted by a noise value, seeded by its position. The z component of the noise value is set to 0 as the displacement is confined to only the x-y plane. Finally, the panel is rotated back to its original orientation by applying the inverse transformation.

To avoid shifting the fracture origin, or the points along the border, a scaling function of 0 to 1 is applied to the noise value based on the points' distances from the origin and from the border. The scaling function can be defined, per point, as the following:

$$scale = S_{origin} * S_{border}$$

$$S_{origin} = smooth\left(0, 1, abs\left(P_{origin} - P\right), R_{origin}\right)$$

$$S_{border} = smooth(0, 1, mindist(borders, P)\ , R_{border})$$

The function *smooth* computes a value between 0 to 1 based on the value provided in the third argument. The interpolation is controlled by the roll-off value, *R*. The function *mindist* returns the minimum distance from a position to a set of primitives. In this case, the minimum distance from a point's position *P* to the borders is returned. Using this scaling function, the closer the points are to either the origin or the borders, the smaller the noise values. Points right at the origin or the borders are not moved. In Figure 23, the scaling factor is indicated by the red colour.

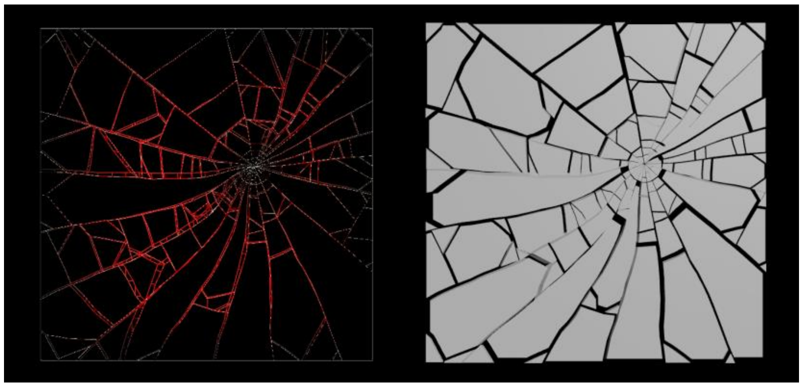


Figure 23: Edge noising amplitude (left) and the result (right).

4.3 Wood Fracture

Wood fracturing are traditionally done with Voronoi fracture. Given a geometry, such as a wooden plank, it is first scaled down in the fiber direction, fractured, and then scaled back up to obtain the lengthened pieces. However, this method is difficult to control since the fracturing is applied at the scaled geometry rather than its original size. Using Voronoi fracture also creates very straight (Figure 24b) or chunky pieces (Figure 24a) that are not realistic. In this project, a more realistic method is developed using Boolean fracture based on the macrostructure of wood. Convex decomposition is used to obtain the collision geometry.

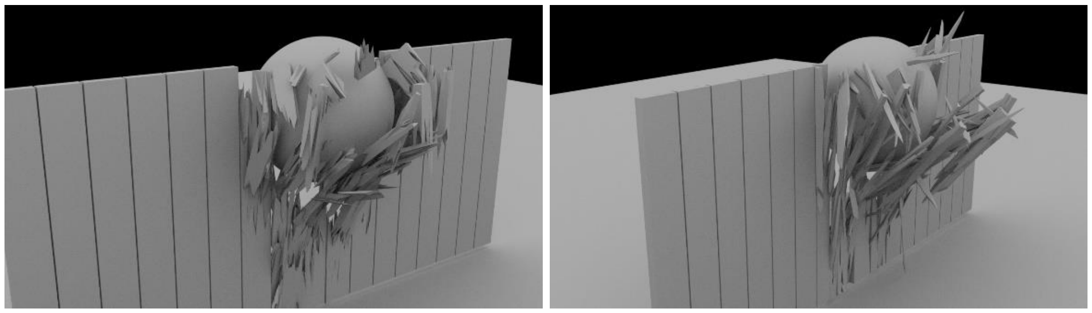


Figure 24: Early test of wood fracture using Voronoi fracture.

Fracture Directions

In the fracture mechanics of wood, there are three directions involved: the longitudinal (L), radial (R) and tangential (T) directions [11]. The L direction is the same as the fiber direction, while the R and T directions are with respect to the growth ring. In this implementation, only the L and R directions are considered. Additionally, wooden planks are treated as the primary object of fracturing, which simplifies the development of the solution.

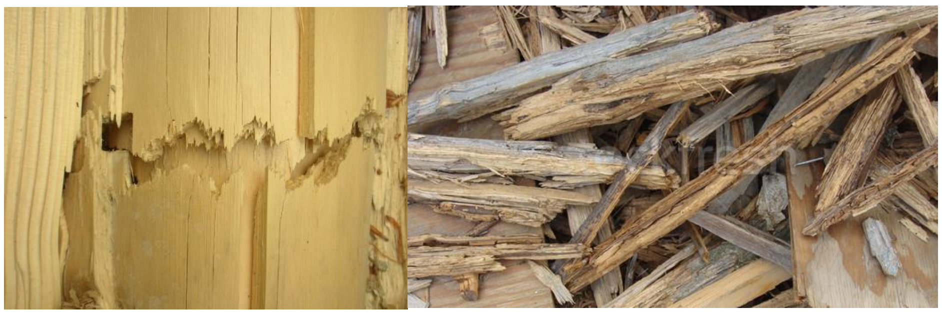


Figure 25: Reference images for wood fracturing.

The ability to determine the fracture directions of wood automatically is essential when they are randomly oriented. When fracturing with SimpleBullet, objects are fractured at the origin before moving it back to world space [6]. For this solution, objects are instead fractured in space. To do so, the oriented bounding box of the wood is used to determine the fracture directions. One point of the bounding box is located, and the lengths of the three edges connecting to the point is measured. Three edges, exactly, is guaranteed since the bounding box is always rectangular, and any point can be used for this calculation. The longest edge calculated is the L direction, and the second longest edge is the R direction. The shortest edge, in this case, is considered as the depth of the wooden plank. The L and R directions are used later for aligning the cutting planes. Users can optionally override the directions with specific direction vectors.

Grain

The most important feature of wood that affects the shape of the fractured pieces is the grain. Depending on the part of the growth ring from which the wooden plank is cut, the grain pattern can be different [12]. Since only the R direction is used for this implementation, the grain is assumed to be parallel lines rather than concentric rings. This makes the cutting surface to be as simple as rectangular planes.

Rather than creating a plane and then orienting it correctly to match the world space location of the wood, the faces from the oriented bounding box, used for determining the fracture directions, are extracted and used as the cutting planes. These faces are conveniently aligned to the calculated LR directions. Since the grain lines are added in the R direction, the face that is normal to the R direction is essentially the cutting plane to generate the grain. Therefore, the face is extracted and copied along the R direction to generate the grain. The spacing of the grain lines can be controlled depending on the level of detail desired. Finally, the cutting planes are displaced with a low frequency noise to yield curved, natural looking grain lines.

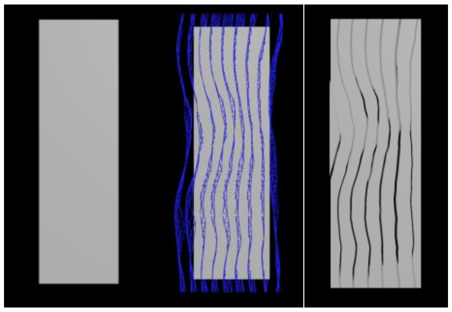


Figure 26: Generating grain using Boolean.

Cuts and Splinters

The fracturing of wood yields splinters along the line of breakage, referred to here as *cut*. While tiny splinters can be generated as a post-simulation effect, larger and visible splinters must be physical. Before the splinters, cuts are added along the L direction to indicate the lines of breakage. In the same fashion as the grain, the face of the oriented bounding box that is normal to the L direction is used as the cutting plane for the cuts. The face is extracted, copied along the L direction, and then finally noised in low frequency for more natural cut lines.

To generate the splinters, the points on the cutting surface for each cut are dramatically and randomly pushed or pulled in the L direction. This yields an extremely noised plane with sharp peaks, from which the splinters will cut out. Users can change the density and length of the splinters based on the desired visual result. While higher density of splinters will usually yield more realistic result, it risks failing the Boolean fracture with numerical or geometrical issues, not to mention the additional computational cost required.

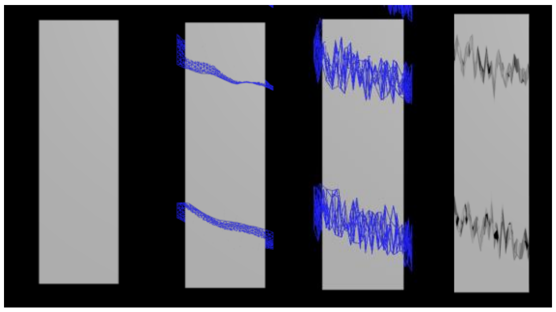


Figure 27: Generating cuts and splinters using Boolean.

Clustering

Clustering is used optionally for wood. Without clustering, the fractured pieces are all separated and do not look exactly like wood. However, once pieces are clustered together, wood-like pieces are formed. The reason why clustering can be optional is because the constraints will hold the pieces together during the simulation thus ultimately getting similar effect as clustering.

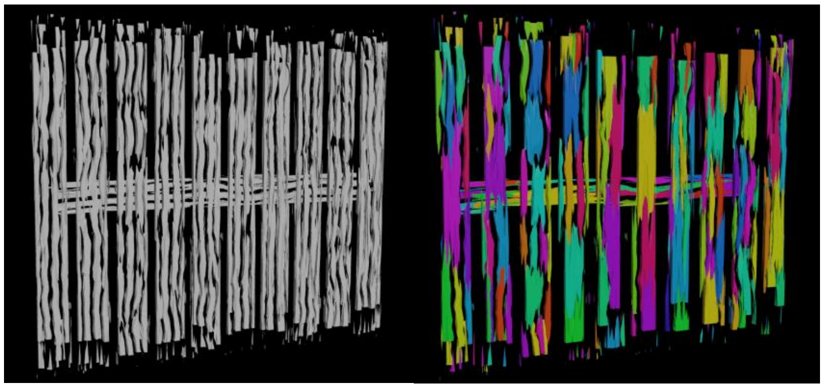


Figure 28: Unclustered vs clustered pieces of wood

4.4 Modularity and Polymorphism

Based on the implementations for concrete, glass and wood fracturing, optimizations are made to make the overall process of fracturing easier. These optimizations are mostly based on the repeated processes for all three implementations, which can be refactored using a more modular workflow.

RBD Building Blocks

Prior to H17, Voronoi Fracture SOP was the only fracturing tool available for users that wraps up, in a limiting way, the manual work required such as naming and clustering. Boolean fracture could be achieved manually by using the generic Boolean SOP and handling the name attributes after fracturing. In H17, Boolean Fracture SOP is introduced in equivalence to the Voronoi Fracture SOP. Additionally, the two nodes now output the constraint network so that users are no longer required to manually create it from the fractured geometry. These changes are the foundations to the new framework introduced for working with RBD. Boolean Fracture and Voronoi Fracture, as mentioned in section 2.2, are the two main mechanisms of fracturing. This idea is made clear in H17 as they are the heart of any material-based fracturing.

RBD Recursive Fracture

In all three materials, the idea of sub-fracturing is used. Sub-fracturing is performed by using for-loops that loop through each piece and apply fracturing to each, using either Voronoi or Boolean. The network can become very large and complicated if many sub-fracturing levels are needed. Therefore, RBD Recursive Fracture SOP is introduced with the aim to simplify the network and make any fracturing processes as generic as possible.

The core of RBD Recursive Fracture is based on a callback mechanism, or the Invoke SOP. When looking at the three material-based fracturing processes, it is realized that they follow a distinct pattern. Given an input geometry, points or cutting surface are generated that feed into either Voronoi or Boolean Fracture SOP respectively. Once the geometry is fractured, an additional processing may be required to either increase or lower the detail of a geometry to produce the render and collision geometries. This process is repeated when sub-fracturing a fractured geometry.

RBD Recursive Fracture builds a recursive network using a nested for-loop. While the current implementation only uses Voronoi fracture, the same idea can and should be applied to Boolean fracture such that users can select the desired fracturing method. Users can specify the number of recursions, which corresponds to the number of sub-fracture level. For each fracture level, one can specify exactly how to scatter points for the incoming geometries at that level by passing in a callback to the scattering mechanism defined elsewhere. Furthermore, a callback is also used for the detail processing of the geometries. Users can optionally specify how to process the fractured geometry to obtain either the render or collision geometry. Using glass fracturing as an example, radial, concentric and chipping fracture can all be done using one RBD Recursive Fracture node instead of three massive for-loops. The node has three recursions each with different scattering mechanisms based on the properties discussed in section 4.2.

The use of RBD Recursive Fracture makes the usage of Voronoi and Boolean Fracture SOPs much lower level. In a way, the RBD Recursive Fracture inherits from both Voronoi and Boolean Fracture and is promoted to be the more common tool to used for fracturing. Each material-based fracturing would use RBD Recursive Fracture to create the desired fracture pattern, while RBD

Material Fracture wraps around all material-based fracture nodes as the main tool. Figure 29 demonstrates this concept.

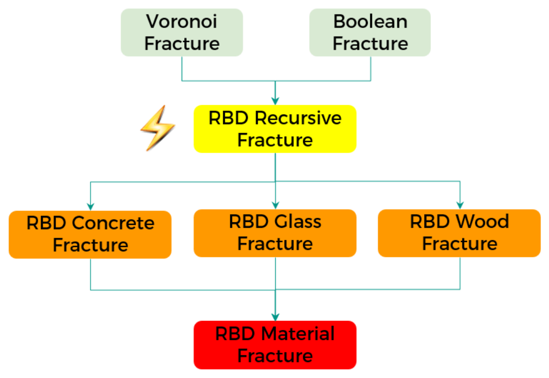


Figure 29: The polymorphic structure of RBD fracture nodes.

RBD Inputs and Outputs

For standard RBD processes, three geometries are involved: the render geometry, the constraint network and the collision geometry. All three geometries are closely tied together with the name attribute. To modularize the workflow, a standard I/O system is established for RBD related processes. For standard RBD nodes, there are three inputs and outputs which correspond to *geometry*, *constraint geometry*, and *proxy geometry*. While the first and third inputs can be understood as render and collision geometries, they are named differently because the collision geometry is optional. One can work exclusively with just the first two inputs or, in some cases, only the geometry (like the old workflow). For this reason, the naming of the first input/output is kept as ambiguous as possible such that it can be interpreted as both render geometry or collision geometry depending on the usage.

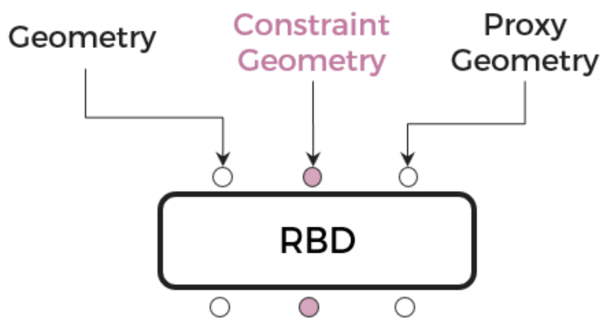


Figure 30: The multi-input/output paradigm for RBD nodes.

For RBD Fracture SOPs, such as RBD Material Fracture and RBD Recursive Fracture, a fourth input is available for extra inputs. In the current implementation, only concrete and glass fracture have the option to use the fourth input for scattering additional points or specify the impact points. In the future, however, more options will be available that allow users to drive fracturing using custom geometry inputs. Since the number of inputs can grow indefinitely, any extra inputs will have to be merged into the fourth input and later separated using a unique group identifier, such that the inputs to the fracture SOPs are not crowded in the network. It is designed in hopes that only one or two custom inputs would be required per fracture node.

4.5 Smart Constraints

Once a geometry is fractured, the constraints between pieces are needed for a correct break configuration during the simulation. Reiterating the existing problems of constraints setup from 2.1, the three main issues identified are:

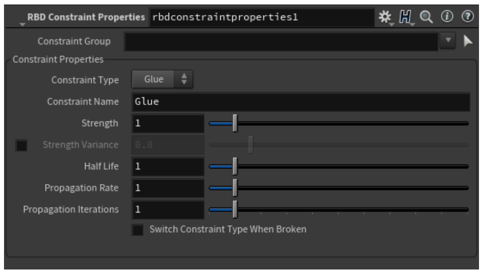
1. The creation of constraint network with larger number of pieces.
2. The isolation of specific connections in a massive constraint network.
3. The assignment of constraint properties for simulation.

In this section, solutions are developed to alleviate the problems above.

RBD Constraint Properties

The existing workflow requires users to use VEX to explicitly assign constraint properties, such as rest length, glue strength etc. as an attribute to each line primitive/connection. This is a tedious step not only because of the expressions that users are required to code, but most annoyingly, having to know exactly what constraint properties are available to tweak and what the corresponding attribute names are. This information is currently found in the DOP network or the documentation online. With the initiative to shift away from DOP to SOP workflow, a better user experience is required.

To enhance this process, RBD Constraint Properties SOP is introduced, which is a simple node that hides away the manual attribute assignment for constraint properties and expose all the controllable properties in the user interface. The current implementation only supports assignment of glue and soft constraint properties, however more constraint types can be supported in the future. This allows the users to drop down a single node and control the constraint properties of a specified connection group without writing any VEX code. Figure 31 shows the constraint properties that are exposed in the user interface.



RBD Constraint Properties rbdconstraintproperties1

Constraint Group

Constraint Properties

Constraint Type: Glue

Constraint Name: Glue

Strength: 1

Strength Variance: 0.8

Half Life: 1

Propagation Rate: 1

Propagation Iterations: 1

Switch Constraint Type When Broken

Figure 31: The parameters of RBD Constraint Properties.

Connection Groups

Having RBD Constraint Properties would be useless without useful *connection groups* to assign the properties to. A connection defines the constraint between two pieces. Multiple similar connections can be grouped together to form a connection group, which can then be assigned with the same constraint properties using the new node. In this workflow, taking advantages of the connection groups is the key to successfully establishing a correct and accurate constraint network for simulation.

Instead of isolating specific connections, each material-based fracturing outputs useful connection groups as part of the constraint network. To begin with concrete fracturing, each fracture level has a unique connection group allowing users to specify different constraint properties based on the recursion level. For glass, the connections between radial, concentric and chipping cracks are available to use. Similarly, the grain and cut connections are outputted for wood fracturing. Using these connection groups along with RBD Constraint Properties, users can easily define the break configuration during simulation for the specific connections to obtain the desired visual result and fracture behaviour.

Auto Constraint Update

In H17, Voronoi and Boolean Fracture SOPs both create the constraint networks along with their fractured geometries. This idea of having the constraint network outputted together with the fractured geometry is the essence of the new workflow. At any given stage of an RBD process, the constraint network, from the middle input of the multi-output paradigm, should correspond to the fractured geometries in the first and third outputs. In another word, the RBD outputs can be used directly in the simulation without any further processing.

To support this workflow, RBD Rewire Constraints SOP is implemented which updates the constraint network every time a geometry is fractured. This node takes the new constraint network generated after fracturing and the original constraint network before fracturing, and rewire the constraints based on new fractured pieces. This ensure that the constraint network is always kept updated at any fracturing step. Both RBD Material Fracture and RBD Recursive Fracture use it internally to update the constraints and support this workflow. To benefit from this workflow, users are encouraged to setup the constraints of the heterogenous setup before fracturing to define the structural relationship of the geometry. The constraints will remain intact and updated for every fracturing step. Figure 32 compares the old and new workflows.

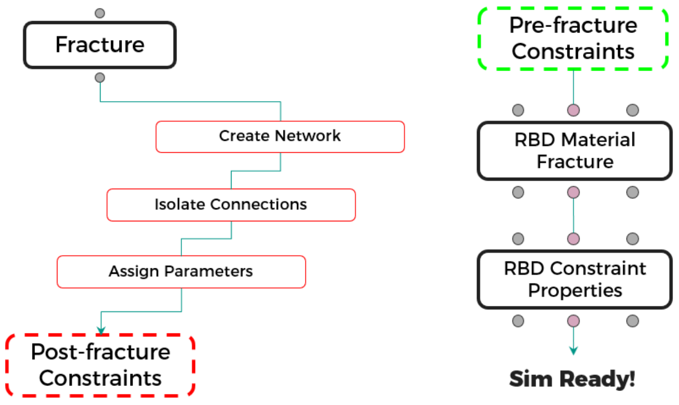


Figure 32: The old (left) vs. new (right) workflow for managing constraints.

4.6 Art-Directing Fracturing

The final feature included in this project addresses the art-directability issue in RBD, which can be applied to almost every stage in the pipeline. This project focuses primarily on art-directability for fracturing. There are two general art-directable features that are frequently requested for fracturing — crack drawing and fracture density painting.

Crack Drawing

Crack drawing is the ability to directly draw an explicit crack on top of an existing fracture pattern. This allows artists to specify exactly where the cracks are, instead of relying on the control of scatter points, for the case of concrete fracturing. In this implementation, only a conceptual prototype is developed which is not integrated to the overall system.

Crack drawing can be achieved with both Boolean and Voronoi fracture. The input from user is a set of curves defining the cracks. For Boolean fracture, the curves can be extruded into the fracture surface and turned into a cutting plane. However, using this method would require the curves to be extended and extruded correctly such that the cutting surface cuts through the geometry completely. This poses the same problem as CSG Voronoi where the extrusion of cutting planes can become very tricky.

Using Voronoi fracture to form specific cracks is also challenging. The method used here is inspired by [13], which adds pairs of points along the crack lines to force the boundaries to align and form the cracks. Figure 33a demonstrates this concept with the red cell points. This generates many small pieces along the cracks which are then merged together based on the piece it belongs to, as defined by the cracks. Figure 32b is an early test result of using this method to fracture concrete. Multiple crack lines were drawn on the surface.

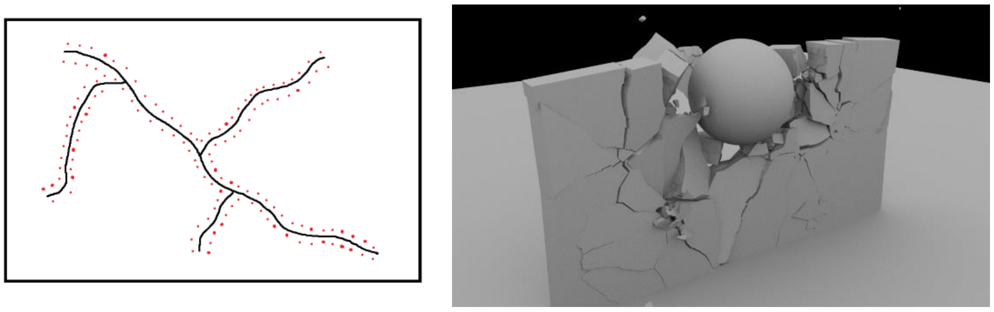


Figure 33: Crack drawing using Voronoi fracture.

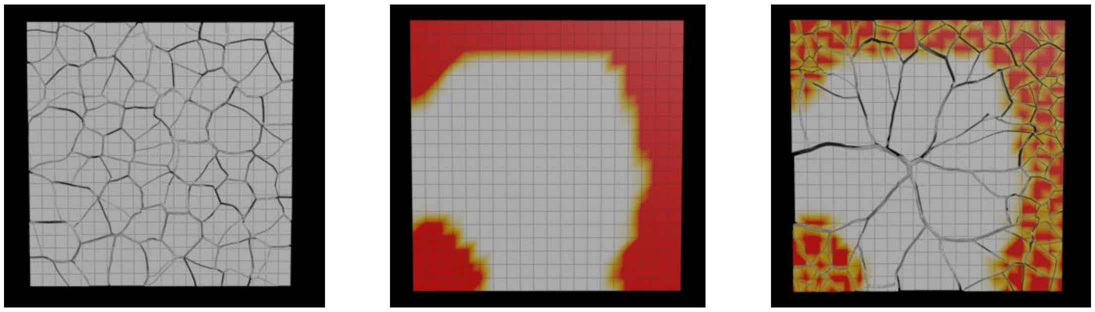
However, the above method is also problematic because many pieces are generated along the cracks, which increases the fracturing and simulation time significantly. The smoother the crack, the more cell points are required. It is also challenging to correctly combine the pieces along the crack when dealing with a more complicated geometry. More research and testing are required to support crack drawing robustly, therefore it is not included as part of the release of H17.

Density Painting

Fracture density painting is the ability to paint areas of a geometry to indicate the density of fracturing. This allows artists, for example, to specify the weaker regions in a geometry that would break into smaller pieces. This workflow must be supported using a painting mechanism as preferred by most artists. Therefore, in this section, the making of RBD Paint SOP is discussed.

Painting is generally not procedural as the painting data is baked into the geometry. To cope with this, Houdini has introduced the Stroke SOP to fake the painting behaviour using multiple curves to indicate each stroke. Thus, every stroke that an artist draw is represented internally as a curve fixed in the world space. Since the number of curves can grow significantly large after multiple strokes, a smart caching mechanism is integrated to purge all previous curves and only retain one active stroke at a time. This workflow is used for many existing paint SOPs in Houdini to retain proceduralism, including Heightfield Paint, from which RBD Paint is based from.

RBD Paint SOP is created as a separate node, instead of using the fourth input of RBD Material Fracture, to make this tool as generic as possible for future development. The current state of RBD Paint simply allows users to paint an attribute, such as density, on the surface of the geometry to be fractured. This attribute is then carried over during fracturing to specify exactly how to interpret the data. For example, concrete fracturing has the option to scatter points based on attribute on the surface, instead of the volume of geometry. Therefore, by first painting a “density” attribute and then using it to scatter points, users can essentially paint the desired fracture pattern. In addition, as concrete fracturing uses recursive fracture, each recursion level can use a different attribute to drive the fracturing of that level. This encourages artists to paint layers of density, or any attribute, to customize the fracturing.



a) Normal fracture b) Density Painting c) Density Scatter

Figure 34: Fracture density painting

Since painting is operated on the surface of the geometry, it has no sense of depth which is needed when scattering points. While scattering points on the surface based on density is easy, scattering points with a specific depth is ambiguous. In this implementation, the depth is handled internally by randomly pushing the points into the geometry. However, depth should also be defined by the artists if needed. One way of doing this would be to paint another attribute for depth, which will be used to determine how far to push the points in. Concrete fracturing must be modified separately to handle depth based on attribute.

Although the application of RBD Paint is currently limited to just concrete fracturing, it is a crucial development as it opens the door for art-directing, not only the fracture pattern, but also other areas in the pipeline such as constraint properties. RBD Paint is designed as a generic attribute painting tool that can be used for any purpose among RBD operations. The next iteration of work will be identifying more use cases for it as well as improving the workflow based on users’ feedback.

5

Testing and Evaluation

In this chapter, the result of the fracturing and the new framework developed for RBD is discussed. These tools will be evaluated against the design objectives that were established in Chapter 3.

5.1 Material-based Fracture

For each material type, relatively simple geometries are fractured and simulated with simple RBD setups. Emphasis is placed on the visual result of fracturing itself rather than the simulation and constraints setup.

Concrete

The use of recursive fracture for size variation and CSG Voronoi for edge noising yields much more realistic fracture result than simple Voronoi fracture. In one test case, a row of concrete columns is fractured each with slightly different fracture pattern. The crack patterns generated for each column matches closely to the reference from Figure 15.

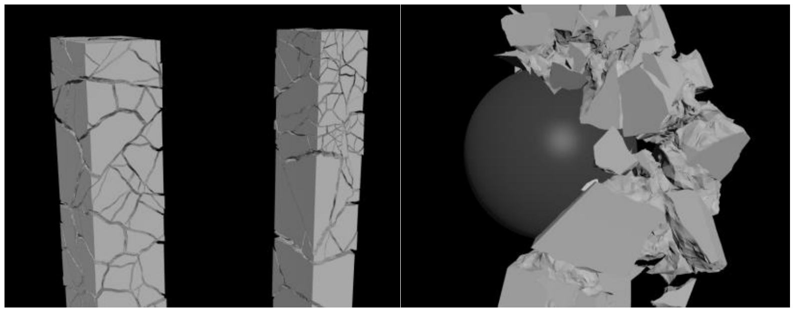


Figure 35: Concrete fracture (left) with wrecking ball (right)

A more complicated test case using a pig head geometry is also conducted to prove the complexity in geometry that the bounding box method of CSG Voronoi can handle.

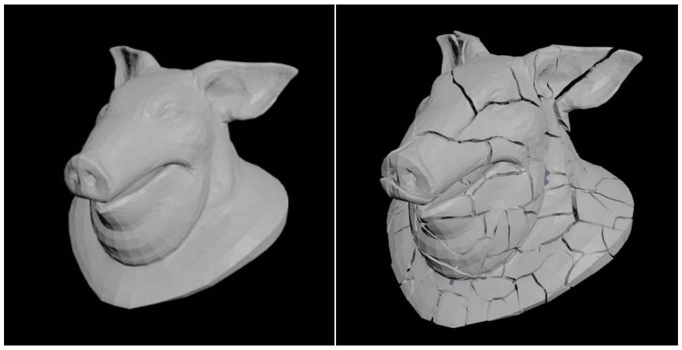


Figure 36: Applying concrete fracture on a pig head geometry.

In some cases, however, artifacts are generated by CSG Voronoi, as shown in Figure 37. The implementation of CSG, in general, is very sensitive to ill-shaped geometry, which causes Boolean to misinterpret the geometry thus shattering incorrectly. For this reason, clean geometry must be guaranteed for CSG Voronoi to work correctly. In the current setup of recursive fracturing, geometries can become very distorted after multiple rounds of fracturing coupled with intense noising. The distortion is also most likely due to the use of proximity to map Boolean pieces back to the corresponding Voronoi proxy geometry. Small geometries that are created from Boolean can potentially be merged to a wrong piece and making the geometry dirty. This effect is especially amplified with recursive fracturing.

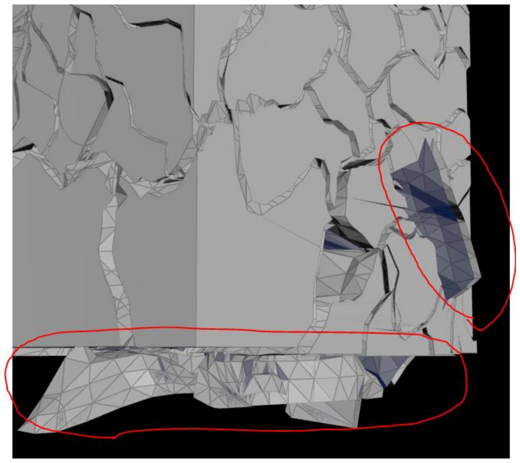


Figure 37: Artifacts generated from CSG Voronoi.

The use of CSG Voronoi also makes the fracturing of concrete slower. Without CSG Voronoi, only one Voronoi fracture per fracture level is used. Using CSG Voronoi, however, requires two Voronoi fractures (one for proxy geometry, one for bounding box fracture) and one Boolean fracture. The fracture time is unsurprisingly slower. It is recommended to implement CSG Voronoi in C++, or as a lower level tool that is optimized in performance.

Glass

In this implementation, glass fracturing is only applicable to flat glass panels. As such, the test cases are very trivial. Most test cases involve multiple windows of a building, which are all fractured together using one RBD Material Fracture node. Each window has slightly different fracture origin and pattern.

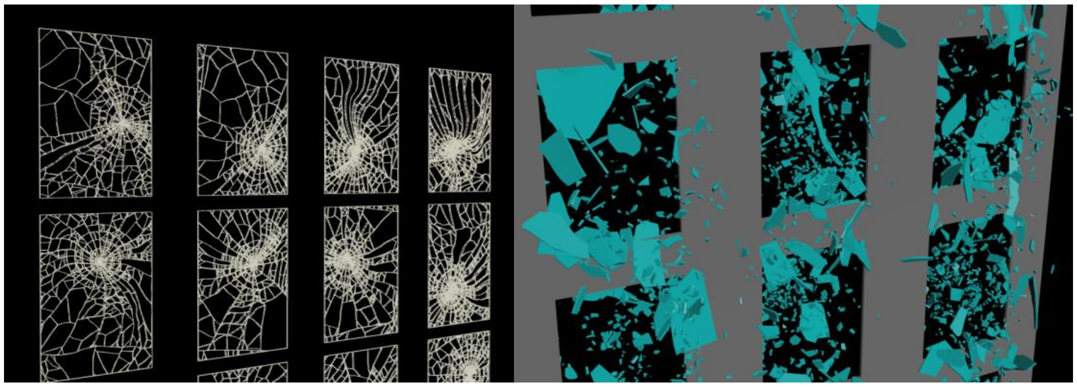


Figure 38: Glass fracturing of multiple windows.

The overall visual result for glass fracturing is good. Radial and concentric fractures are both clearly visible in the fracture pattern. The use of chipping also provides an extra level of randomness making the fractured pieces less similar. The chipping is, however, also applied to the small pieces closer to the fracture origin. This creates even smaller pieces that may be insignificant to the overall visual result but adds in additional cost for the simulation. A more defined chipping area, rather than randomly chipping corners away, will help alleviating this problem. Edge noising can also be improved as all radial cracks seem to be following the same noise pattern thus not looking as natural as it could be. This is expected as the same low frequency noise is applied to all radial cracks. Each radial crack should have its own noise.

Multi-impact points per glass can be achieved as shown in Figure 39. However, the fracture order is not reflected; distinct lines of separation between fracture origins can be seen. Several test cases have suggested that the fracture pieces are dense enough such that the fracture order would not make significant difference. In fact, random multi-impact fracture can be used to generate a generic fracturing of glass that is not caused by impact.

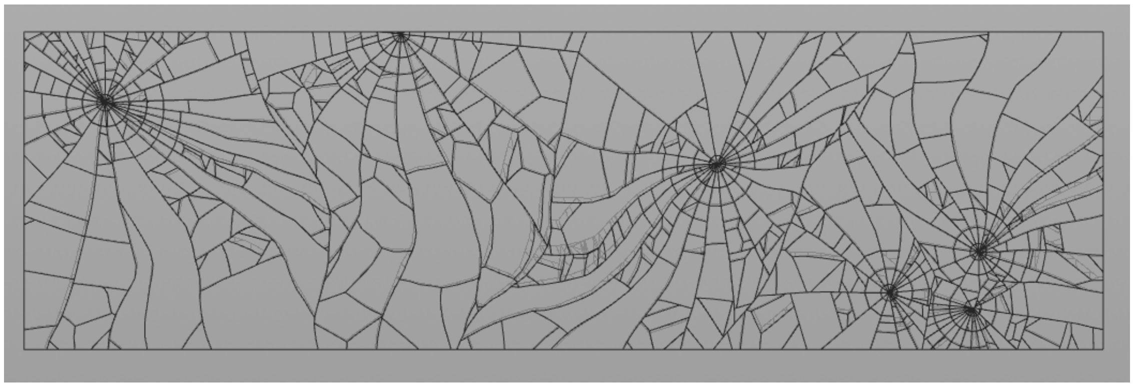


Figure 39: Multi-impact fracture pattern of glass.

Wood

The test cases for wood fracturing are also simple, mostly with wooden planks. The auto-determination of fracture directions allows fracturing an array of planks oriented in any direction. In the case of wooden planks, the fracture pattern is convincing. With the combination of grain, splinters and the use of clustering, the fracture pieces obtained are much more detailed in comparison to the result obtained from Voronoi fracture. The pieces exhibit a more natural shape with small curveture following the grain line and sharper, denser splinters at the ends. Users can also lower the detail in favour of performance.

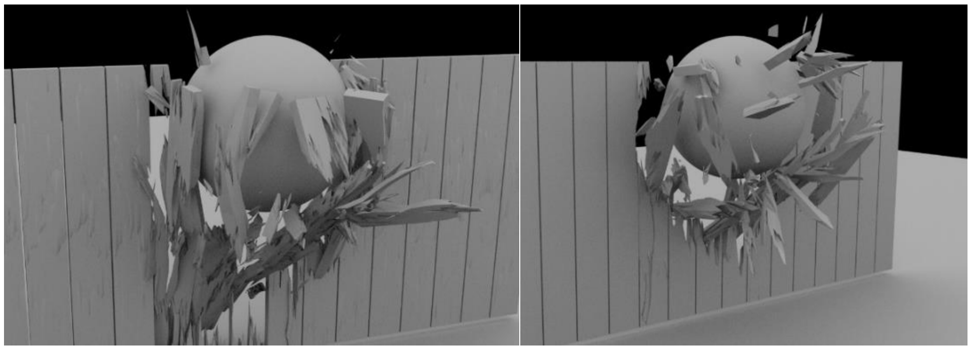


Figure 40: RBD Simulation with wood fracturing at different levels of detail.

The fracturing should, however, be more impact driven. Unlike concrete and glass, no additional user input is currently used to drive the fracturing. As a result, the fractured pieces around the impact area can be unrealistically large. It is recommended to use the fourth input to either supply additional cuts or indicate the impact region at which to avoid large pieces.

For more general wood fracturing, the current implementation also produces reasonable results for tubular and square-based support, as shown in Figure 41. However, as the grain is assumed to be unidirectional, the result is not as physically realistic as the crack patterns of tree trunks or support beams. The current solution should be extended to change the grain pattern based on the section of the growth rings.

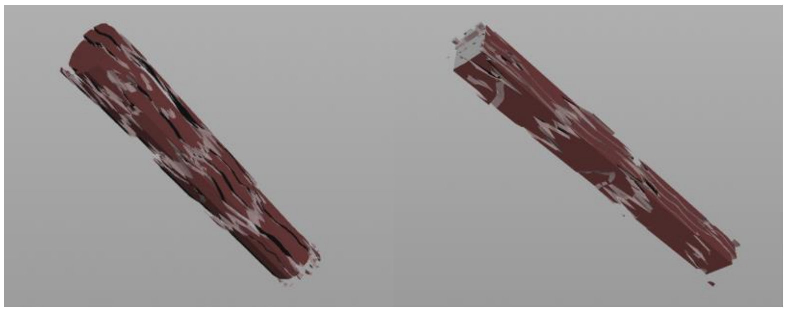


Figure 41: Wood fracturing of tubular geometry (left) and support beam (right).

5.2 Workflow Example

One of the main objectives of this project is to improve the workflow for fracturing a multi-material, heterogenous setup. To evaluate against this objective, a more complicated test case is set up to test the new framework, in addition to the fracturing tool itself. In this test case, a simple house geometry, consisting of all three materials, is fractured sequentially using RBD Material Fracture. The emphasis in this section is on the overall workflow rather than the visual result of fracturing.

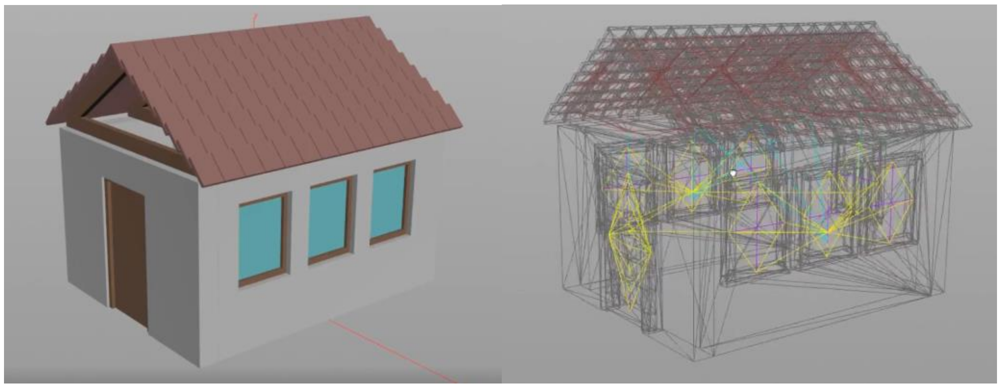


Figure 42: House geometry (left) and its constraint network (right) before fracturing.

Fracturing

The fracturing process is trivial. Each part of the house is fractured based on its material type. Repeated objects, such as the windows, roof tiles and supports are fractured together in one RBD Material Fracture node. Figure 43 shows the final fractured geometry of the house as well as the sequence of nodes used to generate the fracturing result. Since the house is one single geometry, users can filter out specific part of the house to fracture using the group identifier. This allows users to fracture the part of the house in place, rather than having to isolate the specific part of the geometry and merging it back together later.

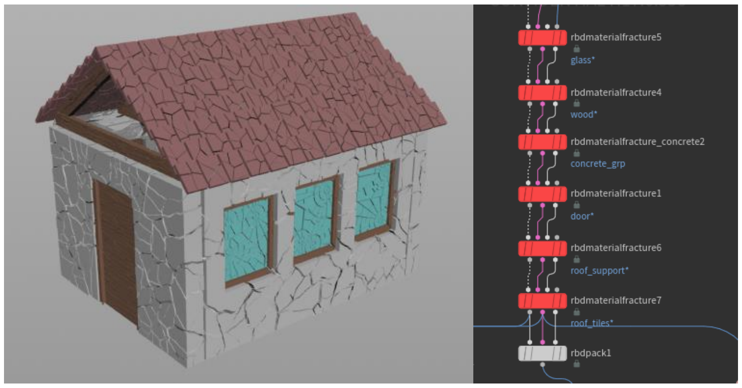
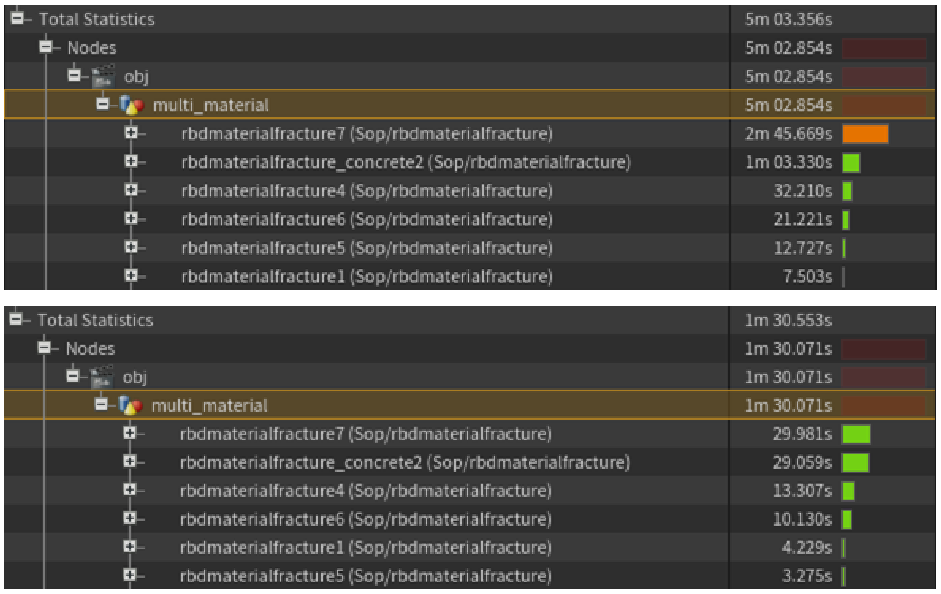


Figure 43: Fractured house geometry (left) and the series of RBD Material Fracture nodes, in red, that performs the fracturing (right).

The visual result of material fracturing is discussed in 5.1. However, the overall performance of the process has yet to be discussed, which is also a crucial design objective. In this implementation, all fracturing nodes are multi-threaded. Multi-threading with Voronoi fracture was not possible prior to H17. As H17 enables multi-threading for Voronoi fracture as well as Boolean fracture, a significant effort has spent in ensuring that every fracturing process within RBD Material Fracture, including Recursive Fracture, is multi-threaded. As a result, an average of 3-5 times performance gain is seen with the new fracturing tool. Depending on the number of cores available in the workstation, the performance gain may vary. Figure 44 compares the time to fracture the house with and without multi-threading. As seen from the speed breakdown, different material types benefit from multi-threading differently.



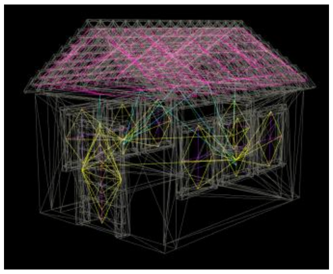
```
Total Statistics                                                  5m 03.356s
  Nodes                                                           5m 02.854s
    obj                                                           5m 02.854s
      multi_material                                              5m 02.854s
        rbdmaterialfracture7 (Sop/rbdmaterialfracture)            2m 45.669s
        rbdmaterialfracture_concrete2 (Sop/rbdmaterialfracture)   1m 03.330s
        rbdmaterialfracture4 (Sop/rbdmaterialfracture)                32.210s
        rbdmaterialfracture6 (Sop/rbdmaterialfracture)                21.221s
        rbdmaterialfracture5 (Sop/rbdmaterialfracture)                12.727s
        rbdmaterialfracture1 (Sop/rbdmaterialfracture)                 7.503s

Total Statistics                                                  1m 30.553s
  Nodes                                                           1m 30.071s
    obj                                                           1m 30.071s
      multi_material                                              1m 30.071s
        rbdmaterialfracture7 (Sop/rbdmaterialfracture)                29.981s
        rbdmaterialfracture_concrete2 (Sop/rbdmaterialfracture)       29.059s
        rbdmaterialfracture4 (Sop/rbdmaterialfracture)                13.307s
        rbdmaterialfracture6 (Sop/rbdmaterialfracture)                10.130s
        rbdmaterialfracture1 (Sop/rbdmaterialfracture)                 4.229s
        rbdmaterialfracture5 (Sop/rbdmaterialfracture)                 3.275s
```

Figure 44: The time taken to fracture the house without (top) vs. with (bottom) multi-threading.

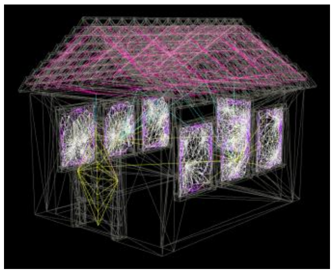
Constraints

Prior to fracturing the house, a simple constraint network is established as shown in Figure 42b. This constraint network defines the physical connections for the house such as connection between wooden frames and windows, between roof supports and the wall etc. Each type of connection is colour coded to visualize how the constraint network is changing at each fracturing step.

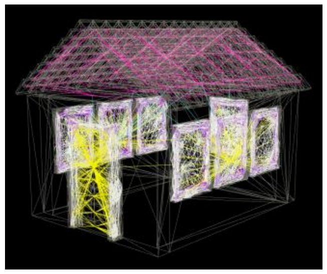
As seen from Figure 45, the connections are retained and updated based on how the geometry is fractured. White constraints are intra-material constraints generated by each RBD Material Fracture, while the colour coded constraints are connections of the house structure, as defined before fracturing. Without any further modification, the constraint network is ready to be used.



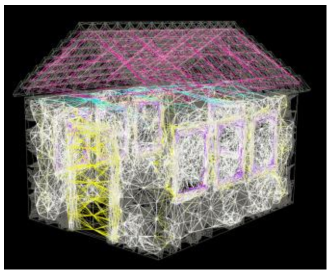
a) Original Constraints



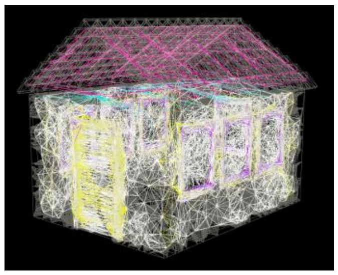
b) Window



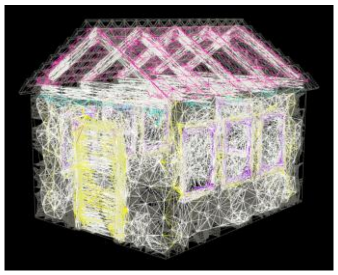
c) Window Frames



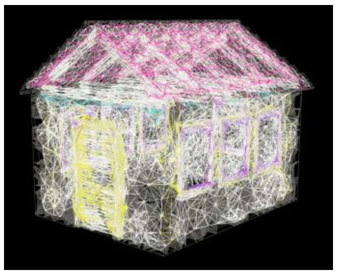
d) Concrete Wall



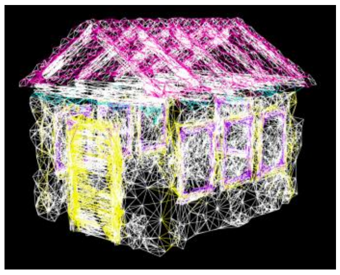
e) Door



f) Roof Support



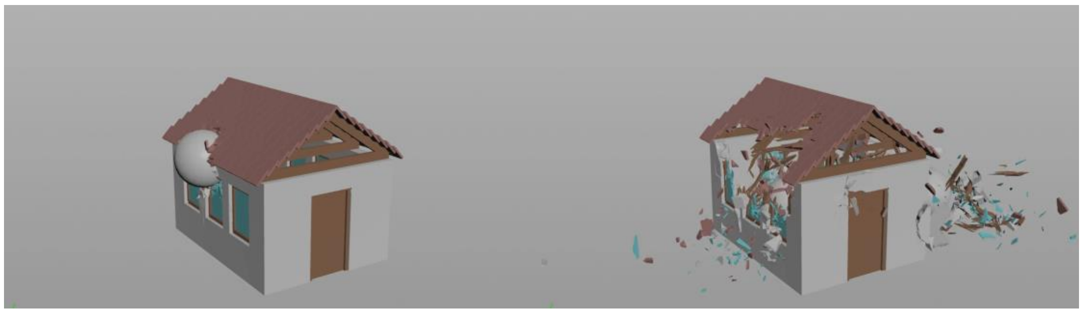
g) Roof Tiles



h) Final Constraints

Figure 45: The constraint network at each fracturing step without any additional modifications.

Finally, a series of RBD Constraint Properties nodes are used to assign constraint properties to each connection types, before passing everything into the simulation. Figure 46 contains a few still frames from the result of simulation. The simulation can be more refined by tweaking the constraint properties until the desired visual outcome is obtained.



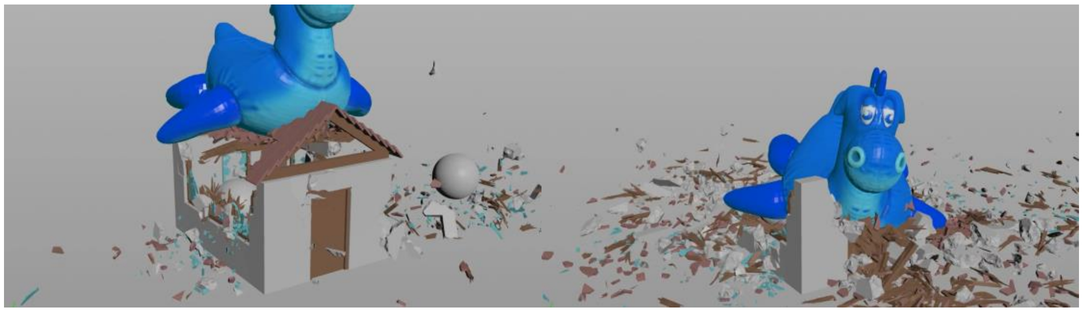


Figure 46: Four still frames extracted from the RBD simulation of the house.

Overall, the constraint network is created and updated as expected for both intra and inter connections of fractured objects. This reduces drastically the time for users to manage constraint network and the complexity in the network editor.

5.3 User Feedback

User feedback is received from two sources — direct discussion with WDAS and the H17 beta forum, which the latter is still an on-going process. Before releasing the project to the beta forum, a meeting is conducted with WDAS in which a demo of the house setup is given. The overall feedback from WDAS is very positive and many artists have shown excitements over the new framework. However, since the workflow differs from that of SimpleBullet, it will take some time for them to adapt to it and provide more useful feedback. It is noticed, however, that the presence of the connection groups is not as obvious as it should be. Since connection groups correlate to recursive fracturing, it is important to make them more obvious for users to leverage. Furthermore, lots of emphasis is placed on the ability to isolate regions of a geometry to do fracturing separately. While the sequential setup of fracturing simplifies the nodal network, having the ability to fracture geometries in parallel, such that each fracturing is not dependent on the previous fracturing, will be extremely beneficial with respect to efficiency.

The feedback from the beta forum is more focused on the multi-input/output system. A lot of users seem to be overwhelmed by the change in the workflow and having to deal with three geometries constantly. This can most likely be alleviated by providing a user-friendlier way of dealing with multi-input/output nodes in the network editor, which is in the scope of Houdini's general user experience. Some users have also found the customization of constraint network to be limiting, since the constraints creation is now hidden from users. More controls over the creation and update of the constraint network is strongly recommended.

6 Future Work

In this chapter, future work is suggested to further enhance the project. The enhancements discussed here are general improvements rather than specific bug fixes. Based on the result of testing and user feedback, the next iteration of work will look to further align the solution with the established objectives.

6.1 Generic fracture structure

The use of RBD Recursive Fracture has generalized the fracturing workflow and reduces the network complexity. However, its usage can be redesigned, in the next iteration of development to provide a more intuitive, extendible, and polymorphic structure.

To support material-based fracturing as well as the new constraint workflow, a lot of manual processes have been integrated to RBD Material Fracture and RBD Recursive Fracture. These manual processes, such as clustering, rewiring constraints, also appear to follow a distinct pattern that can be further refactored to provide the most generic fracturing structure.

The motivation of providing a generic fracturing *template* is to provide extendibility for users to create other material-based, or even object-based, fracturing tool. Rather than copying the same processes from an existing fracture node, users can inherit the generic fracture template such that the structure is already setup. Users would only be required to specify how to fracture the object, either using Voronoi, Boolean, or another fracturing method that inherits from the generic template. A pseudo-code of the template (RBD Fracture) and its usage can be found in Appendix A. With RBD Fracture, RBD Recursive Fracture can be replaced by explicit usage of RBD Fracture per fracture level. This makes debugging much easier while also separating each fracture level to be recooked separately rather than together as in RBD Recursive Fracture.

6.2 Fracturing as a post-process

So far in the project, all RBD Material Fracture has been treated as a pre-simulation only tool. However, many advanced techniques in productions involve taking the simulation result and remodify the fractured geometry as a post-process [14]. One example of this usage is the fracturing of metal. Fracturing metal requires a first pass of simulation to first determine the deformation of the object before applying the actual tears and cracks.

Another example is the concept of secondary simulation as demonstrated in [14]. An object can first be simulated with large fractured pieces to have a base animation, after which another round of fracturing is done to add in smaller pieces. Impact data from the previous simulation can be used to drive the secondary fracturing. After smaller pieces are obtained, another round of simulation is processed. Using multi-pass fracturing and simulation provides an extra level of art-directability and decreases significantly the iteration time in a fast-paced production [14].

Providing a generic fourth input for RBD Material Fracture opens the door for accepting simulation result directly, on top of just user inputs. However, the work involved here requires much more sophisticated understanding of the simulation itself and how to process the data from the solver. More research and development are needed to support RBD Material Fracture as a post-process tool.

6.3 Constraints

The new workflow for dealing with constraints have proven to be much more efficient and trivial for simulation. However, the system has yet to matured and requires more enhancements over the next few iterations to allow users to fully appreciate the benefit of it. These improvements include more controls over the way constraints are created internally by RBD Rewire Constraints. No control is currently given to the users, which is essential based on the feedback from the forum. For example, the ability to create edge-to-edge connection instead of the current center-to-center (of pieces) connection is needed to establish a more realistic break configuration. Furthermore, more constraint types must be handled by RBD Constraint Properties. Based on SimpleBullet, it is recommended to develop a unified constraint type that can handle most of the constraint behaviour required for destruction. Finally, applying RBD Paint beyond fracturing to drive constraint properties, such as the glue strength, will further enhance the art-directability of the system.

6.4 Caching and Parallelism

In the next design iteration, the solution must be applied to a larger scale destruction setup to stress test its functionality and efficiency. When dealing with large geometric data, caching and parallelism are the key to fast iteration time, as emphasized in the meeting with WDAS. As modular as the system has become, no test has been conducted to use RBD Material Fracture in parallel rather series. A way of merging multiple RBD Material Fracture nodes and updating the constraints correctly must be developed to support this workflow.

Alternatively, since the RBD Material Fracture isolates geometry using the group identifier, parallelism may not be required if it can be detected that the current fracture has no dependency to the previous fracture. In the house setup, for example, the fracturing of the roof tiles and the fracturing of the windows have no dependency with each other. However, since they are connected in series, a change in the fracturing of one will trigger a reprocessing for the other, thus making it inefficient. Separating dependencies and caching unchanged geometries are the next set of work required for performance and efficiency.

7

Conclusion

In this project, the design objectives are developed based on issues identified in the RBD pipeline, specifically at the pre-simulation stage. The project can be broken down into two main developments — the material-based fracturing toolkit and the redesign of the RBD workflow.

The material-based fracturing toolkit, or RBD Material Fracture, allows automatic fracturing of objects based on three material types: concrete, glass and wood. Using Voronoi and Boolean fracture as the building blocks, various techniques such as recursive fracturing and CSG Voronoi are developed to yield realistic fracturing result. Although the implementation of glass and wood are simpler and limited to certain cases, they do encompass most structural geometries in buildings and the techniques can be further extended to fracture more complicated setup.

The redesign of the RBD workflow is based on the existing problems with managing the constraints. By automatically updating and creating constraints and outputting them along with the fractured geometries, users are no longer required to deal with constraints as a post-fracture step. This reduces the complexity for heterogenous destruction setup and makes the overall pre-simulation process much simpler. However, the change in the workflow, including the introduction of multi-input/output system, still need to be adapted by existing users. Lots of improvements are needed to bring back old functionalities and provide more controls over the creation of constraints.

Overall, the implementation of RBD Material Fracture based on the new fracturing techniques and the redesigned workflow has proven to be beneficial and game-changing in certain aspects. It serves as a solid foundation, around which many more destruction tools can be developed to address important issues such as efficiency and art-directability. While users explore the new tools and adapt to the framework, the next chapter of RBD continues with many more improvements and innovations, ultimately letting artists make breathtaking destruction in a snap of fingers.

References

[1] E. Coumans, "The bullet physics library," 2018. [Online]. Available: http://bulletphysics.org. [Accessed 2018].

[2] Lien Muguercia, Carles Bosch, Gustavo Patow, "Fracture modeling in computer graphics," *Computers & Graphics,* vol. 45, pp. 86-100, 2014.

[3] M. Seymour, "Art of Destruction (or Art of Blowing Crap Up)," *fxguide,* 13 December 2011.

[4] Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, Sung Nok Chiu, Spatial Tessellations: Concepts and Applications of Voronoi Diagrams, 2nd Edition, Chichester, West Sussex, England: John Wiley & Sons, 2000.

[5] Johnathan M Nixon, Sebastian H. Schmidt, "Rampage: A Pipelined Approach to Managing Large Scale Character Driven Effects," in *SIGGRAPH '18,* Vancouver, 2018.

[6] Will Harrower, Ferdi Scheepers, Marie Tollec, "SimpleBullet: Collaborating on a Modular Destruction Toolkit," in *SIGGRAPH '18,* Vancouver, 2018.

[7] Jeffery A. Okun, Susan Zwerman, The VES Handbook of Visual Effects, Burlington, MA: Focal Press, 2010.

[8] M.-H. Education, McGraw-Hill Encyclopedia of Science & Technology, 10th Edition, McGraw-Hill, 2003.

[9] G. D. Quinn, Fractography of Ceramics and Glasses, Gaithersburg, MD: NIST Spec. Publ. 960-16e2, 2016.

[10] R. C. Bradt, "The Fractography and Crack Patterns of Broken Glass," *J Fail. Anal. and Preven.,* vol. 11, pp. 79-96, 2011.

[11] Marcia Patton-Mallory Steven M. Cramer, "Fracture mechanics: a tool for predicting wood component strength," *Forest Products Journal,* vol. 37, pp. 39-47, 1987.

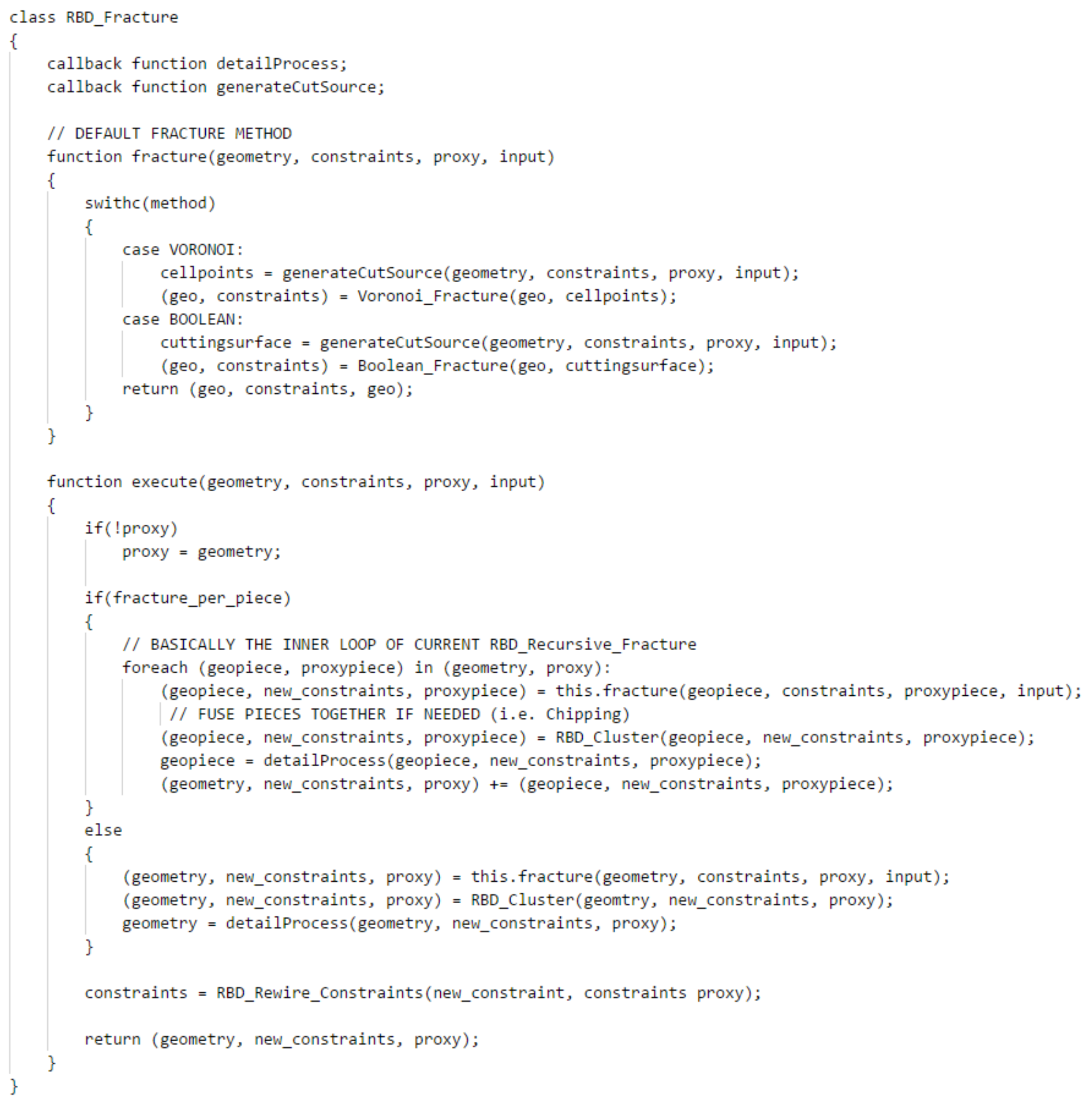
[12] Thomas Nilsson, Roger Rowell, "Historical wood - structure and properties," *Journal of Cultural Heritage,* vol. 13S, pp. S5-S9, 2012.

[13] D. Asahina, J.E. Bolander, "Voronoi-based discretizations for fracture analysis of particulate materials," *Powder Technology,* vol. 213, pp. 92-99, 2011.

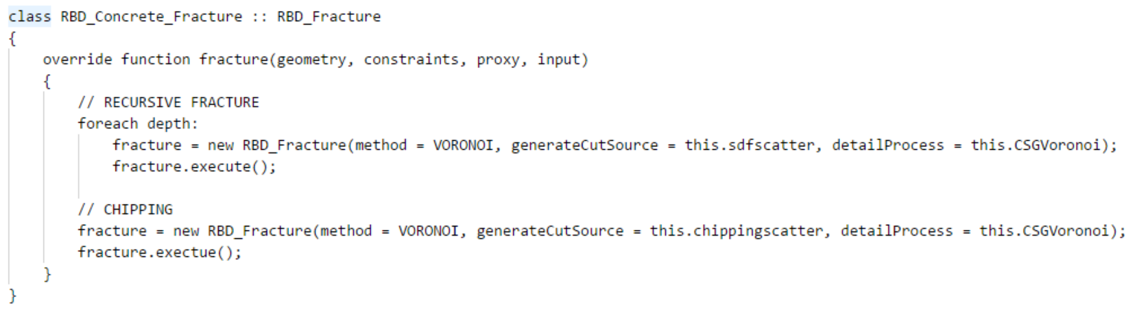
[14] F. Lee, "Art Directing Rigid Body Dynamics as a Post-process," in *SIGGRAPH '14,* Vancouver, 2014.

Appendix A

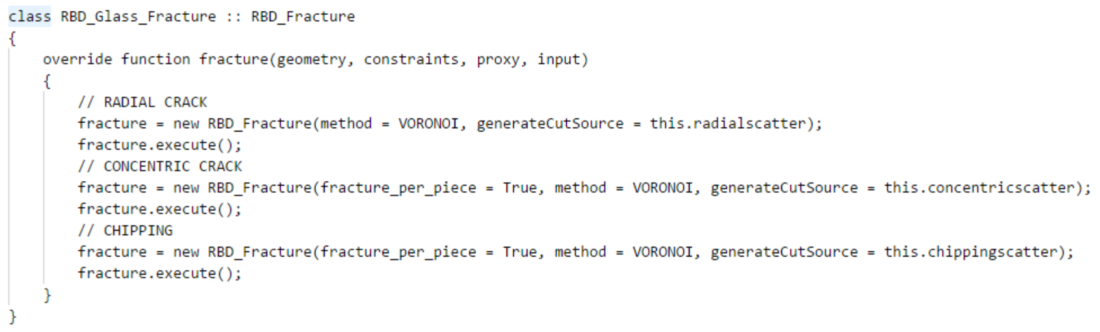
RBD Fracture Pseudo-Code



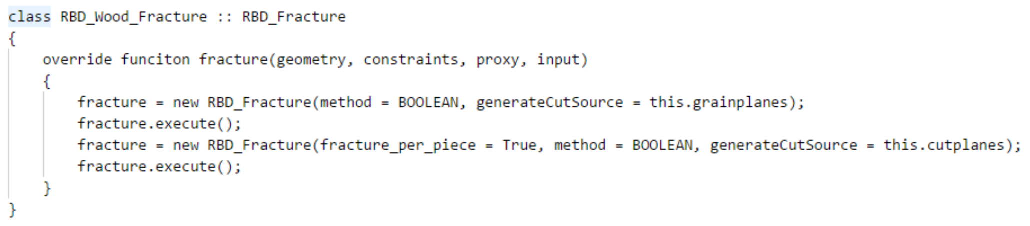
```
class RBD_Fracture
{
    callback function detailProcess;
    callback function generateCutSource;

    // DEFAULT FRACTURE METHOD
    function fracture(geometry, constraints, proxy, input)
    {
        swithc(method)
        {
            case VORONOI:
                cellpoints = generateCutSource(geometry, constraints, proxy, input);
                (geo, constraints) = Voronoi_Fracture(geo, cellpoints);
            case BOOLEAN:
                cuttingsurface = generateCutSource(geometry, constraints, proxy, input);
                (geo, constraints) = Boolean_Fracture(geo, cuttingsurface);
            return (geo, constraints, geo);
        }
    }

    function execute(geometry, constraints, proxy, input)
    {
        if(!proxy)
            proxy = geometry;

        if(fracture_per_piece)
        {
            // BASICALLY THE INNER LOOP OF CURRENT RBD_Recursive_Fracture
            foreach (geopiece, proxypiece) in (geometry, proxy):
                (geopiece, new_constraints, proxypiece) = this.fracture(geopiece, constraints, proxypiece, input);
                 // FUSE PIECES TOGETHER IF NEEDED (i.e. Chipping)
                (geopiece, new_constraints, proxypiece) = RBD_Cluster(geopiece, new_constraints, proxypiece);
                geopiece = detailProcess(geopiece, new_constraints, proxypiece);
                (geometry, new_constraints, proxy) += (geopiece, new_constraints, proxypiece);
        }
        else
        {
            (geometry, new_constraints, proxy) = this.fracture(geometry, constraints, proxy, input);
            (geometry, new_constraints, proxy) = RBD_Cluster(geomtry, new_constraints, proxy);
            geometry = detailProcess(geometry, new_constraints, proxy);
        }

        constraints = RBD_Rewire_Constraints(new_constraint, constraints proxy);

        return (geometry, new_constraints, proxy);
    }
}
```



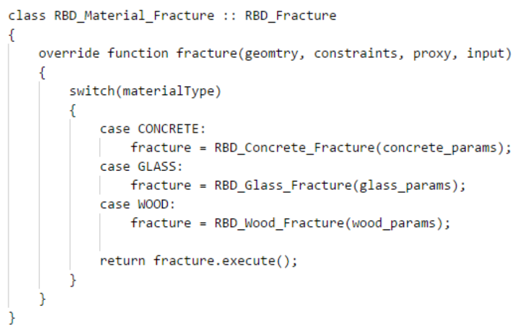
```
class RBD_Concrete_Fracture :: RBD_Fracture
{
    override function fracture(geometry, constraints, proxy, input)
    {
        // RECURSIVE FRACTURE
        foreach depth:
            fracture = new RBD_Fracture(method = VORONOI, generateCutSource = this.sdfscatter, detailProcess = this.CSGVoronoi);
            fracture.execute();

        // CHIPPING
        fracture = new RBD_Fracture(method = VORONOI, generateCutSource = this.chippingscatter, detailProcess = this.CSGVoronoi);
        fracture.exectue();
    }
}
```



```
class RBD_Glass_Fracture :: RBD_Fracture
{
    override function fracture(geometry, constraints, proxy, input)
    {
        // RADIAL CRACK
        fracture = new RBD_Fracture(method = VORONOI, generateCutSource = this.radialscatter);
        fracture.execute();
        // CONCENTRIC CRACK
        fracture = new RBD_Fracture(fracture_per_piece = True, method = VORONOI, generateCutSource = this.concentricscatter);
        fracture.execute();
        // CHIPPING
        fracture = new RBD_Fracture(fracture_per_piece = True, method = VORONOI, generateCutSource = this.chippingscatter);
        fracture.execute();
    }
}
```



```
class RBD_Wood_Fracture :: RBD_Fracture
{
    override funciton fracture(geometry, constraints, proxy, input)
    {
        fracture = new RBD_Fracture(method = BOOLEAN, generateCutSource = this.grainplanes);
        fracture.execute();
        fracture = new RBD_Fracture(fracture_per_piece = True, method = BOOLEAN, generateCutSource = this.cutplanes);
        fracture.execute();
    }
}
```



```
class RBD_Material_Fracture :: RBD_Fracture
{
    override function fracture(geomtry, constraints, proxy, input)
    {
        switch(materialType)
        {
            case CONCRETE:
                fracture = RBD_Concrete_Fracture(concrete_params);
            case GLASS:
                fracture = RBD_Glass_Fracture(glass_params);
            case WOOD:
                fracture = RBD_Wood_Fracture(wood_params);

            return fracture.execute();
        }
    }
}
```