Methodology to Utilize Pre-Computed Voronoi Diagrams to Enable Dynamic Deformation and Destructibility of Environmental Meshes Within A Simulation Environment

Noah Nam, Kyle McCullough, Raymond New
University of Southern California Institute for Creative Technologies
Playa Vista, California
{nam,mccullough,new}@ict.usc.edu

Ryan McAlinden
Army Futures Command
Orlando, Florida
ryan.e.mcalinden.civ@mail.mil

ABSTRACT

Runtime environments commonly use "sleight-of-hand" techniques to approximate destruction and convince a user that their actions have deformed or destroyed an object. These "sleight-of-hand" processes rely on hand-creating general-use "pre-shattered" objects that may not accurately reflect the effects of every explosion, given the possible variations in radius and force. In a simulation, munitions must deform and fragment the environment believably in order to provide analysis and visual feedback to the users. At I/ITSEC 2019, the authors explored methodologies for increasing the number of autonomous entities within a simulation using existing game industry solutions (McCullough et al., 2019). For this work, the authors have utilized a similar approach to mesh deformation by exploring the commercial game industry and attempting to capitalize on existing methods, while taking into account the necessity for removing the human-in-the-loop required by those techniques. With the global coverage of environment meshes expected to be used in the Synthetic Training Environment (STE), it would be too time consuming to create each mesh's destroyed variants by hand. In order to overcome this challenge, we have explored a method of using precomputed Voronoi diagrams to deform and fragment 3D meshes in real-time while maintaining performant simulation speed and fidelity. This method allows destruction to be customized to specific scales and positions on a given mesh based on the type and location of an ordnance. Utilizing additional research presented at I/ITSEC 2019 for creating simulation terrain (Chen et al., 2019) the workflow takes into account semantic classification enabling the system to deform ground meshes into craters and support attributed material types (e.g. concrete, wood) for object mesh fragmentation. With this research, we can bypass a lengthy process of manually preparing destruction meshes while also having a destruction system decoupled from mesh generation; both systems can be independently updated without requiring modification of the other.

ABOUT THE AUTHORS

Noah Nam is currently a programmer at USC-ICT working on the One World Terrain project. His work involves implementing Unity's Entity Component System to enable simulating many entities within a large-scale dynamic simulation environment. He has a B.S. in Computer Science from Drexel University. Email: nam@ict.usc.edu

Kyle McCullough is the Director of Modeling & Simulation at USC-ICT. His research involves geospatial initiatives in support of the Army's One World Terrain project, as well as advanced prototype systems development. His work includes utilizing AI and 3D visualization to increase fidelity and realism in large-scale dynamic simulation environments, and automating typically human-in-the-loop processes for Geo-specific 3D terrain data generation. Kyle received awards from I/ITSEC and the Raindance festival, winning "Best Interactive Narrative VR Experience" in 2018. He has a B.F.A. from New York University. Email: mccullough@ict.usc.edu

Raymond New is a senior programmer at the USC-ICT working on various projects including the One World Terrain. He acquired his BS. in Computer Science from the California State University Long Beach in 2007. He has been developing games and simulation software for the institute for over 12 years, including the Dark Networks game for the Naval Post-Graduate School which won the Silver Medal Award at a Serious Games competition in 2018. Email: new@ict.usc.edu

Ryan McAlinden is a Senior Technology Advisor for the US Army's Synthetic Training Environment (STE), part of Army Futures Command (AFC). He is the Cross Functional Team (CFT) lead for the One World Terrain (OWT) initiative, which seeks to produce a high-resolution, geo-specific 3D representation of the surface used in the latest rendering engines, simulations and applications. Ryan previously served as Director of Modeling, Simulation & Training at the University of Southern California's Institute for Creative Technologies where he led several initiatives related to training modernization across the Services. Ryan rejoined ICT in 2013 after an assignment at the NATO Communications & Information Agency (NCIA) in The Hague, Netherlands. There he led the provision of operational analysis support to the International Security Assistance Force (ISAF) Headquarters in Kabul, Afghanistan. Ryan's research interests lie in the design, development, implementation and fielding of solutions that are at the cross-section of 3D rendering, geospatial science, and human-computer interaction. Ryan earned his B.S. from Rutgers University and M.S. in computer science from USC. Email: ryan.e.mcalinden.civ@mail.mil

Methodology to Utilize Pre-Computed Voronoi Diagrams to Enable Dynamic Deformation and Destructibility of Environmental Meshes Within A Simulation Environment

Noah Nam, Kyle McCullough, Raymond New
University of Southern California Institute for Creative Technologies
Playa Vista, California
{nam,mccullough,new}@ict.usc.edu

Ryan McAlinden
Army Futures Command
Orlando, Florida
ryan.e.mcalinden.civ@mail.mil

INTRODUCTION

Overview

There is a need for removing human-in-the-loop processes for large scale simulations, and whenever data may need to be rapidly generated at the Point of Need (PoN) with limited resources and man-hours. The focus of this work considers these factors in relation to environmental destructibility for uses such as training, battle damage assessment and prediction, and mission planning. Destructibility in this context is defined by the ability to deform or break apart 3D models that represent terrain features such as buildings, trees, and the ground. The Army's efforts on the Synthetic Training Environment (STE) and particularly the One World Terrain (OWT) Program, provide an exceptional use case for this need. STE will feature global-scale terrain, with high resolution small unmanned aircraft system (sUAS) photogrammetric reconstructions often happening at the PoN, where the operators will be without the access or time required to utilize existing destructive mesh methodologies, such as pre-shattered or hand-crafted damage states. As we began to explore this capability, there was relatively little related literature to this work directly, especially in regards to game-engines, though there is a long history of commercial solutions to these problems, which we'd imagine are often proprietary and not well documented publically. Two related works of note that we found were unpublished theses covering procedural destruction (Van Gestel, 2011) and efficient methods for destructibility (Dobransky, 2017). Dobransky's work goes into detail on how they found uses for Voronoi methodology to be useful. However, our work is different in many ways, most notably that we are using photogrammetrically derived flat mesh data, implementing the Voronoi methods on segmented models based on material classification, and automating the processes using the OWT geo-specific datasets and pipeline.

Justification

This project seeks to provide a capability for large-scale and PoN destructibility and deformation of 3D meshes in a runtime agnostic and highly performant manner. This capability will fill in gaps and seek to meet the requirements of demanding users and programs of record, providing a capable and reproducible methodology that achieves a sufficient level of fidelity while requiring no customized setup or specific mesh models. Typically, the level of fidelity this work seeks to achieve is only realizable via artist-created damage states or pre-shattered 3D models. We recognize that artist-created destructible objects are not feasible when there is a need for them to be generated at a large scale or when the data has been captured at a location that doesn't allow for the time or presence of an artist, such as would be the case for a forward deployed unit. Destructibility must be programmatically achieved with minimal, if any, human intervention to support these datasets.

Scope, Planned Method

In order to best understand the metrics and systems involved in creating fully automated procedural destructibility we have chosen a discrete research line for our initial work, focusing on the Unity Game Engine and the readily available mesh rendering and filtering components it provides. We have also leveraged previous work for generating photogrammetric mesh data (Spicer et al., 2016) as well as the Semantic Terrain Points Labeling System Plus (STPLS+) (Chen et al., 2019) which is a fully automated pipeline for taking raw photogrammetric terrain data and outputting semantically classified and segmented datasets. For this work our main focus from the semantic hierarchy are the Building, Ground, and Tree classifications. Our goal is to design a system that will take the terrain information

into account to create realistic destructive effects based on the properties for each of the semantic layers. Buildings made of concrete would crumble, the ground would crater, and trees would splinter.

DESTRUCTIBILITY IN THE COMMERCIAL VIDEO GAME INDUSTRY

History

Destructibility has long-been a hallmark of immersion and interactivity within the commercial video games industry. From early arcade classics to today's first-person shooters, the use of destructible and deformable objects and structures has been paramount to a game's challenge, mechanics, and even perceived value. Beyond entertainment however, destructibility brings a realism to a gaming and simulation environment that when implemented with real-world material and physical attributes can actually provide a level of analysis and review for the simulated systems.

A number of industry techniques currently exist for implementing destructive and deformable environments and objects, but after investigating them it was determined they wouldn't be practical for our intention and purposes. One technique that we explored was the idea of scripted art swaps, which generally require a designer creating pre-fragmented objects or damage states that are substituted during a destruction event. We also looked at the use of generic fragmentation, where scripted events inject pre-fabricated mesh models with generic, though usually artistically relevant elements via an action trigger. Lastly, and most similar to our method were destruction systems that had been implemented to be procedurally generated, though even these proved to be insufficient for our purposes, as they still required a great deal of human design and interaction to achieve the overall destructibility goals, which served a narrow scope for single games.

A great example of a game that offers procedural destruction is Red Faction: Guerilla (2009). According to the creators, the game itself is from the ground up "built around destruction" (Orry, 2009), focusing a majority of gameplay around destructibility and a player's ability to interact with environments. The developers and designers created the Geo-Mod 2.0 (THQInsider, 2009) engine incorporating data from structural elements and systems that would allow players to strike at load-bearing walls and columns in order to cause large buildings to completely crumble, and when targeting the ground, to create cratering and deformation. While this is a wonderful and entertaining aspect of such games, the singular factor that connects the underlying mechanics is that these buildings were designed with that precise feature and outcome in mind requiring a specific physics engine, Havok. A technical artist built the 3D meshes and data specifically to allow for the destruction. A game programmer designed the systems and code that caused the buildings to crumble based on intricate systems for this one game engine and this one purpose. A level designer helped to ensure that destroying this building would be part of the fun, or part of the challenge, on the way to completion. All of this is manual effort in order to achieve a goal of fully destructible environments for entertainment value. However, attempting to bring this capability into another game, even by the same developer, proved "literally impossible" as senior producer Jim Boone referenced in an interview by saying "*you'll notice we don't have a tremendous density of structures – talking about very large structures – and that's because we push these platform to the nth degree just to be able to do those buildings in the way that we're doing them, in order for them to break apart in the way that we do."* (Cook, 2013)

However, what do you do if you're attempting to create this type of interaction at a global scale for high-fidelity realistic simulation?

Impact

The more qualities within a simulation that can be an analogue to the real-world - the higher the fidelity of that simulation. Physical properties are critically important in order to provide the necessary data to simulation systems in order to convey high levels of realism. Take destructibility for instance. Using classic methods of destructibility from the video game industry, an artist would generally create the pre-shattered analogue to a physical object or structure. The technical artist that is designing these destructible elements already knows the properties of the material and uses their own creativity to design the shattered forms accordingly. For instance, if a wooden fence is meant to be affected by in-game explosives, the technical artist would create the shattered pieces in the style of broken wood, a game programmer would create the system to instantiate the shattered pieces in an act of sleight of hand, applying the necessary physics and motion to give the illusion that the fence has been destroyed. While these same techniques

could be applicable to a simulation capability, managing this at scale becomes terrifically unwieldy. The hours required for an artist to interpret shattered forms, and a programmer to create realistic systems for each of the objects and structures, is just not feasible. The method this work is exploring remove the need for meticulous hand-crafted artistry and utilizes prior work in semantic classification (Chen et al., 2019) to derive the physical properties using a fully autonomous methodology, computing the destructed elements on-the-fly based on the existing mesh geometry and semantic identification. In a high-fidelity simulation, it's expected that destructible environments provide tangible feedback to its users. In a video game, destructibility makes a user's actions more satisfying. In our case, destructibility and the resulting debris allow for a wide range of applications directly related to realistic training, planning, and intelligence operations. For instance, realistic destructibility allows a trainee to see the cause and effect between their munitions and the environment, preparing their expectations during combat. It also provides a realistic effect on potential changes in terrain traversal, navigation, and trafficability. Terrain modification through destructibility and deformation directly informs enhanced Battle Damage Analysis (BDA), breaching operations, and mission planning capabilities. By being able to effectively link a weapon, munition and target with all of their actual physical properties, we're able to generate a sufficient picture for the full effect of ordnance and projectiles on targets.

Critical Factors

One of the goals of STE is to allow users to run realistic simulations on 3D datasets of real world locations derived from algorithmically processing geographic information system (GIS) data capture. As such, a majority of the 3D objects and terrain data within STE are not created by artists; they are derived directly from the GIS data without a human-in-the-loop. In the case of destructibility, while it would be possible for an artist to pre-fracture the generated objects in a modelling software, the sheer volume of objects generated for STE makes this unviable, as the program seeks to create a global database and provide a capability for rapidly updatable datasets through organic acquisition. To help to achieve these requirements, destructibility systems, in the same way as the terrain data, must be computed by an algorithm. Precomputing all of the fragmented objects isn't feasible due to the myriad of ways an object can be deformed. Therefore to account for different munitions and positions of destruction, fractured objects must be calculated in real-time based on parameter inputs that can be derived from real-world terrain information, material classification, and munitions.

Variable vertex count is a critical factor in destructibility calculations. Mesh deformation directly affects mesh vertex position, therefore running increased calculations across a high-number of vertices and the simulation frame-rate will slow or even stall completely. Too few vertices and the effect of deformation will look unconvincing. Some objects use multiple LOD (Level Of Detail) models in which objects far from the camera display a lower quality mesh to save resources. A solution for automated and procedural destructibility must be able to handle the varying vertex counts that a terrain dataset and its child objects can have.

A solution must also be modular, as it will be used as a 'plug-and-play' library for in-house development, it cannot be reliant on outside information and should be as agnostic as possible towards the runtime environment. Of note are Unity's tags and layers, commonly used to differentiate between physics objects. Because it cannot be guaranteed that a given Unity project uses the tags and layers that destructibility systems would prefer it to have, the solution cannot use tags and layers at all without risking breaking simulation physics.

METHOD

In creating a real-time destruction and deformation system, Voronoi diagrams were utilized to describe fracture patterns and mesh vertex transformation was used to deform plane meshes into craters. Destructibility and deformation were tested against 3D meshes derived from GIS through the Institute for Creative Technologies (ICT)'s OWT Pipeline. Terrain tiles were pre-classified to separate ground, building, and tree, with each becoming their own game objects. While ground and building objects were directly created from captured data, trees were inserted by placing geotypical SpeedTrees at the locations trees were classified in the dataset.

Differentiating Objects

Before an object can be destroyed or deformed, the object needs to be known. Unfortunately for this destructibility system, Unity's tags and layers cannot be used for object identification. The destructibility system is intended to be

imported into any Unity project so the tags and layers in a given project cannot be guaranteed. So instead a **TerrainInformation** class was created to attach to destructible objects. TerrainInformation contains two enums, one to define the object's terrain type: Ground, Building, LODTree, TreeWood, and TreeLeaf, and one to define the object's destruction type. Destruction type defines a fragmented object's appearance. Currently two types are defined. **Concrete** breaks into rock-like polyhedrons, and **wood** breaks into splinters. We differentiate between three different tree objects (LODTree, TreeWood, TreeLeaf) for optimization. SpeedTrees use different LOD's (Level Of Detail) objects as the camera moves closer or further away from them. As the camera gets farther, the tree switches to a lower quality model to conserve resources. This means the object can contain multiple models at once, which is unwanted for destruction. Destroying multiple tree models would be detrimental towards performance, so one LOD is chosen and the others are discarded when a tree is destroyed. The tree that was marked "LODTree" splits into several GameObjects, the woody objects becoming "TreeWood", and the leaf ends becoming "TreeLeaf". There are many leaf objects on trees, and it would be too time consuming to fracture them. In addition, leaves are often a single texture stretched on two triangles, it would not look realistic for a bundle of leaves to be cleanly sliced in half. So "TreeLeaf" objects simply fall to the ground on destruction, but "TreeWood" objects do break apart.

Craters

If an explosion occurs on the ground, the ground is deformed into a crater. The crater's size is determined by the radius of the explosion. Vertices near the center of the explosion are pushed downwards on the y axis. The amount of depression into the ground decreases as the distance from the center increases, resulting in a crater shape. The maximum depression is equal to the radius of the explosion.

We start by using Unity's built-in physics library to run **OverlapSphere()**, which returns the physics colliders of objects within a given sphere. If an object contains a TerrainInformation component and that TerrainInformation's terrain type is "Ground", it will be deformed by a crater.

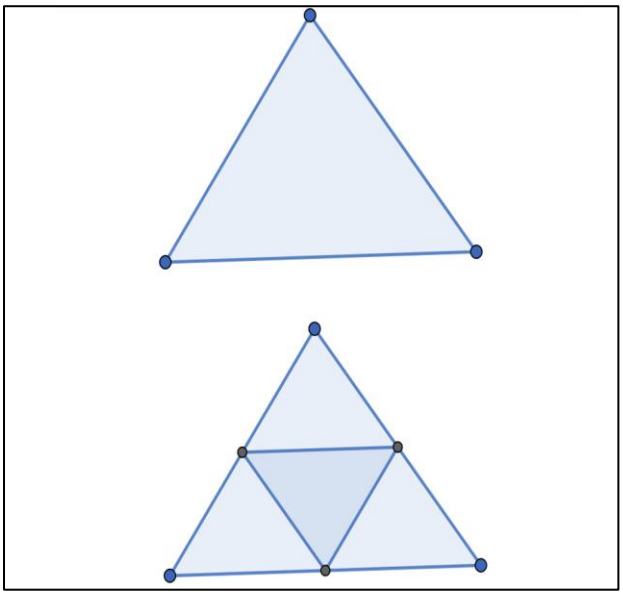

Figure 1: A triangle and its medial triangle.
Adapted from "GeoGebra" by GeoGebra. 2020.

Given a mesh to deform, its vertices must first be checked. The mesh to be deformed needs to have at least a certain amount of vertices within the radius of deformation or the deformation will not be noticeable. This is because crater deformation does not inherently add new vertices to a mesh, it simply alters the Y value of existing vertices. If there are not enough vertices, more vertices must be added. To increase the amount of vertices, the existing mesh triangles will be divided into multiple smaller triangles by calculating for each triangle's medial triangle. A medial triangle is a triangle whose vertices are the midpoints of the edges of a containing triangle (Figure 1). Adding medial triangles is as follows:

Given the **mesh**, the deformation **center**, and deformation **radius**, A HashSet<int> **verticesInRadius** is created to record vertices within the radius of deformation. A Dictionary<(int,int), int> **vertexMidpointMap** is also created to map tuples of two vertex indices to the vertex index of their midpoint. This dictionary is necessary to differentiate between midpoint vertices that exist in the same physical position but are mapped to different UV coordinates. An integer **vertexCountRequired** is calculated by multiplying **radius** with a predetermined multiplier. **vertexCountRequired** is used to check if the function can be exited: the radius must contain at least this many vertices. Finally a HashSet<int> **verticesInRadius** is created to record the indices of vertices within the radius. The mesh's list of vertices is iterated through and vertices whose distance from **center** is less than or equal to **radius** are added to **verticesInRadius**. If **verticesInRadius** is greater or equal to **vertexCountRequired**, the function exits here; there are already enough vertices for deformation. Otherwise, until the vertex count requirement is met, three new vertices will be created for every triangle in the mesh, so long as that triangle is not smaller than a certain size. Because

Unity has a limit on vertices per mesh (65,534), and because terrain tiles can be large enough in world space that adding enough medial triangles to satisfy **vertexCountRequired** would easily hit this limit, it is important to ensure large triangles receive medial triangles and that sufficiently small triangles are not wastefully subdivided any further. The new vertices exist at the midpoints of each triangle's edges. These vertices are added to the mesh, and their corresponding UV coordinates are also added to the mesh with each new UV being the midpoint of the UVs corresponding to the triangle's vertices. Four new triangles will be created from the new vertices and the triangle's vertices, forming three triangles connected at their vertices with an inverted triangle in the center (Figure 1). The original triangle is discarded from the mesh, and the new triangles replace it. New medial triangles are added to the mesh until **verticesInRadius** is greater than or equal to **vertexCountRequired**, or until adding any more vertices would surpass Unity's vertex limit per mesh.

With the mesh prepared, its vertices can now be deformed into a crater. A crater comprises two features, a depression and a lip that borders the depression. The depression's radius is the same as the previously initiated **radius**, centered on **center**. Its maximum depth **maximumDepth** is equal to **radius**. For every vertex within the depression's radius, its Y value is linearly interpolated along a sine wave, starting at the center at $\sin(\pi/2)$ and ending at the lip at $\sin(0)$, forming an upwards opening parabola. The Y value is then multiplied by **maximumDepth**. The lip's radius is 1.2x that of the depression, but only the vertices beyond the depression will be affected by lip calculations. For every vertex within the lip radius that is also outside the depression radius, its Y value is linearly interpolated along a sine wave, starting at the edge of the depression at $\sin(0)$ and ending at the radius of the lip at $\sin(\pi)$, forming a downwards opening parabola. The lip has a **maximumHeight** of 0.15x the **maximumDepth**. The lip is further clamped from being taller than 0.5x the **maximumHeight.** This clamp is to prevent intersecting lips of multiple craters additively forming unnatural appearing spikes. Craters of varying radii can be seen in Figure 2. The intersecting craters on the left of the figure show an example of the lip clamp preventing the intersecting lips from being too tall.

Figure 2: Craters of varying radii.

Buildings and Trees

To simulate the destruction of buildings and trees, Voronoi diagrams were used to break meshes apart into realistic appearing fragments. A Voronoi diagram divides a space containing **n** points into **n** cells, in which a cell encompasses the area that its point would be closer to any position in that cell than the point of any other cell in the diagram would. The resulting diagram results in an image of uniquely shaped polygons that do not overlap, and combine to fill the area of the original space (Figure 3). These polygons are made of bounded and unbounded edges. An unbounded polygon extends out into infinity in the direction of its unbounded edges, as there are no other polygons in those directions. The Voronoi diagram can also be calculated in 3D so 3D diagrams were used to calculate uniquely shaped polyhedrons that would give the approximation of a chunk of concrete. The following method of calculating a 3D Voronoi diagram follows the steps outlined in Ledoux's (2007) paper.

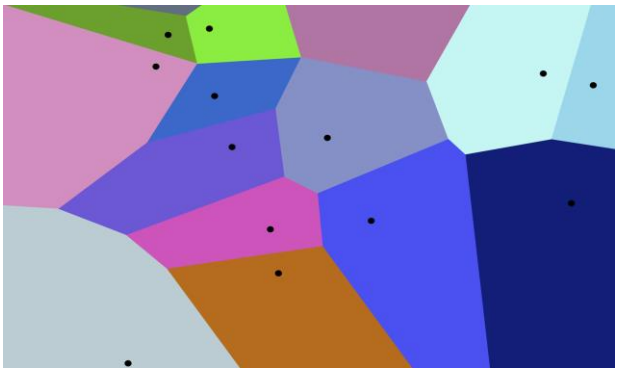

Figure 3: A Voronoi diagram.
Adapted from "Euclidean Voronoi diagram.svg" by Balu Ertl. 2015. Creative Commons Attribution-Share Alike 4.0 International license.

A property of Voronoi diagrams is that they are dual to Delaunay triangulations, that is, one can be converted to the other and back again. This is useful to us because it is simpler to calculate Delaunay triangulations than it is to calculate Voronoi diagrams (Ledoux, 2007). Delaunay triangulations work with triangles (which always have three points). The polygons within a Voronoi diagram

have an arbitrary amount of points. After calculating a Delaunay triangulation, its dual can be calculated to get a Voronoi diagram.

For 3D diagrams, polyhedrons are desired, not polygons, so first calculated will be a Delaunay tetrahedralization instead of a Delanuay triangulation. The difference between calculating for triangles and tetrahedrons is simply raising the dimension of the data structures within the algorithm. Since the Unity engine is being used, that would mean that the Vector2's used to describe 2D points would now become Vector3's. Instead of iterating on triangle edges, iteration is on tetrahedron faces.

To calculate the Delaunay tetrahedralization, the Bowyer-Watson algorithm was used, modified for 3D use as described above. The Bowyer-Watson algorithm starts with a set of random points **R** in 3D space, and a "super-tetrahedron" that contains all points in **R**. The points in **R** will become the vertices of the resulting Delaunay tetrahedralization. **R** must contain at least one point. More points will increase the amount of resulting tetrahedrons. We insert one point in **R** into the super-tetrahedron and draw new lines that connect the vertices of the super-tetrahedron to the added point, creating four new tetrahedrons. The super-tetrahedron will not be in the output, it is merely a staging space required to contain the created tetrahedrons. We now iterate through the rest of **R**, adding points and drawing new tetrahedrons. If a point is ever within the volume of a tetrahedron's circumsphere, that tetrahedron is removed and recalculated with the new point. Once the algorithm is complete, the tetrahedralization can be converted into a Voronoi diagram. Figure 4 shows the results of Delaunay tetrahedralization. Each yellow sphere represents a randomly generated vertex of the tetrahedralization. Each white line represents the tetrahedron edges.

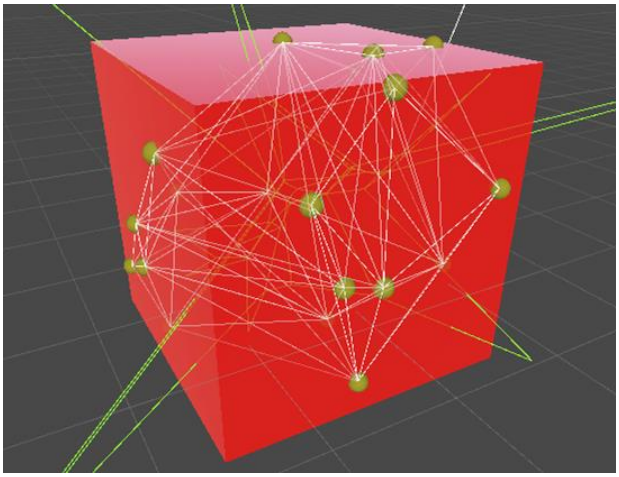

Figure 4: Generated Delaunay tetrahedralization.

A Delaunay tetrahedralization is dual to a Voronoi diagram as follows: Each vertex of a Delaunay tetrahedralization corresponds to the center of a Voronoi cell. The circumcenter of each Delaunay tetrahedron corresponds to a vertex in the Voronoi diagram. If two Delaunay tetrahedrons share a face, their circumcenters are connected with a face in the Voronoi diagram (Ledoux, 2007).

To convert the Delaunay tetrahedralization into a Voronoi diagram, we begin with four givens: the tetrahedralization itself (a list of tetrahedrons), a dictionary that maps tetrahedron edges to tetrahedrons that contain them, a dictionary that maps tetrahedron faces to tetrahedrons that contain them, and the original super-tetrahedron. We first determine the unbounded Voronoi edges. For each tetrahedron face, we check if two tetrahedrons share it (signaling adjacency between the tetrahedrons). If it does, we then determine if either of the tetrahedrons shares a vertex with the super-tetrahedron. If it does, this tetrahedron face's dual Voronoi edge is unbounded, and we add this edge to the cells dual to the face's vertices as a ray. Now we will record the bounded Voronoi edges. We first remove all tetrahedrons which share a vertex with the super-tetrahedron. For each remaining tetrahedron edge within the tetrahedralization, we determine its dual Voronoi face by calculating the circumcenters of the tetrahedrons that contain the edge and containing the circumcenters in a list representing the face. The face is then sorted so that traversing it in order would form a single closed path. In cases that the face has less than 3 vertices, sorting is not needed. Voronoi edges are created by iterating through the Voronoi face and assigning the current element with

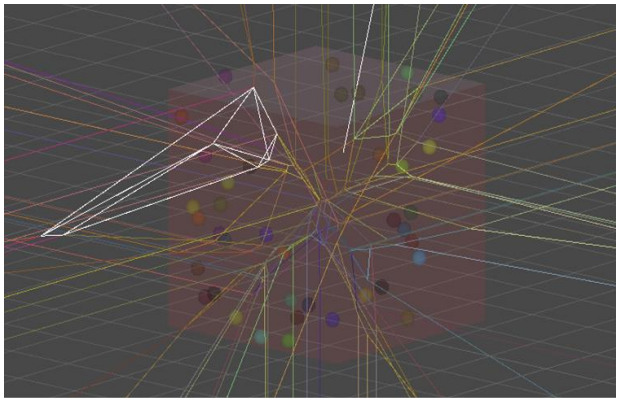

Figure 5: Generated 3D Voronoi diagram.

the element proceeding it to a new edge. The Voronoi edges are added to the Voronoi cells dual to the tetrahedron edge's vertices as line segments. We now have a Voronoi diagram, which contains cells, which contain line segments to represent bounded edges and rays to represent unbounded edges.

Figure 5 shows the results of 3D Voronoi mesh generation. Each Voronoi cell is represented by a set of lines sharing the same color. Lines extending into infinity represent unbounded Voronoi edges. Whether a cell is bounded or unbounded does not change how mesh fragments will be calculated from them; A fragment is always the intersecting volume between the mesh and a cell. One cell is highlighted in white. Meshes will be sliced according to the diagram, each cell carving a piece of the mesh by its cell volume.

We now store the diagrams as JSON files for use at runtime. It is not necessary to save individual Voronoi edges; we only need to associate a cell with its faces. Furthermore, we don't need to save a face's vertices as we will be treating them as planes during mesh cutting. So each face is represented as a plane in point-normal form, and each cell is a collection of planes. Several diagrams are stored to provide a variety of destruction shapes at runtime. Diagrams do not differ in severity of destruction; the selected diagram will be modified in scale to match the munition used. When destruction occurs, one of the stored diagrams will be randomly selected. Each destruction type has its own set of diagrams. Currently that includes concrete and wood, with plans for more types in the future.

To handle munition types, two user interface (UI) sliders are provided to the user to adjust destruction power and destruction radius. Power modifies the amount of force that fragments are blown from the epicenter with. Range modifies the scale of the Voronoi diagram, modifies the radius of the ground crater, and modifies the radius within which fragments can break off the mesh. In the future, presets for specific munitions will be available.

At runtime when destruction is called, we first check if the object within the radius of the explosion is capable of fracture. If its TerrainInformation component's TerrainType is "Building" or any tree variant, it can be destroyed. We then determine which destruction material should be used, "Concrete" or "Wood". (Though it sounds repetitive to have building and tree types at the same time as having concrete and wood materials, in the future a building may be made of wooden beams or glass.) We then randomly select a Voronoi diagram appropriate for the destruction material. For each cell in the diagram, we iterate through the cell's planes, stored in point-normal format, and slice the mesh by each plane. We are using a third party tool, UnityAssets, to handle mesh slicing (Whirle, 2020). A new game object is created from the sliced mesh. Each sliced face is assigned a texture appropriate for its destruction material. Each game object, a fragment, represents a Voronoi cell and fills the cell's volume up to the boundaries of the mesh. This process repeats for each cell. We discard the original GameObject and are left with fragments that combined take up the same volume as the original GameObject did.

If a fragment's center is within the radius of the explosion, a Unity Rigidbody will be added to that fragment. Rigidbodies allow Unity GameObjects to interact with the built-in physics system. The fragment is given a mass based on its size. A force is applied to the fragment, pushing it away from the epicenter. The closer to the epicenter it is, the larger the force is. Fragments that are outside the radius of the explosion instead stay in place, but keep track of other non-mobile fragments adjacent to them. Adjacencies are tracked to avoid a fragment floating in the air without being connected to the ground. Every destruction, fragments that track adjacencies check if they are still attached to the ground via their neighbors (and their neighbor's neighbors, etc.). If a fragment is not attached to the ground, it is given a Rigidbody and is removed from the adjacency matrix.

Concrete uses a 3D Voronoi diagram to simulate the effect of a rock-like object breaking apart (Figure 6). Wood uses a 2D Voronoi diagram stretched into three dimensions) to simulate the effect of a long grained object breaking apart (Figure 6).

Figure 6: Concrete and wood destruction, respectively.

Wood destruction runs on the same principles of building destruction, but uses a different diagram. Whereas

concrete uses an unmodified 3D Voronoi diagram, wood uses a 2D Voronoi diagram stretched into 3D, then further sliced. To begin, we calculate a Delaunay triangulation given a random set of points. We convert the triangulation into its dual Voronoi diagram. At this point we have a 2D diagram of polygons. We then add a third dimension to all the vertices of the diagram, and duplicate the vertices twice. The first duplicate's vertices' Z points are set to 1, and the second duplicate's vertices Z points are set to -1, giving us a 2D diagram stretched into 3D space. It appears as a set of long, conjoined columns. The tops and bottoms of each column contain lines stretching up and down to infinity respectively to represent unbounded edges in the Voronoi diagram. Each of these columns will represent a bundle of wood fibers that have split from the tree. To further increase the realism of the fracture, we split the columns along their length. Each column splits into a top, middle, and bottom piece. The position of each split is randomized along the axis of the length. All columns are then saved to JSON as lists of normal vectors for later use. At run-time, destruction appears as in Figure 6. The 'middle' columns are close enough to the epicenter to detach from the mesh, but not the top and bottom columns, resulting in a splintering effect.

Findings

Mesh deformation at run-time is a computationally non-trivial operation. It is important to optimize whenever possible to reduce the chance of the simulation holding while it waits for an operation to complete. We found that using optimal data structures and caching calculation output greatly reduced the time it took to apply destruction on a target object. We also found that upscaling a model's vertex count was needed when it was too low so that mesh deformation would appear as expected.

Any valid Voronoi diagram can be used to cut any valid mesh; it is unnecessary to calculate Voronoi diagrams at runtime. We stored our diagrams in JSON format, with each cell holding a list of rays, which each represent a plane to cut the mesh with. This way we do not need to bloat the data with cell vertices, edges, faces, etc. At runtime, we simply choose a diagram and carve the mesh accordingly.

Destruction creates a large number of new objects, many now interacting with the physics engine. This puts heavy strain on the computer's resources. To keep frame rates stable, the simulation deletes certain fragments after a period of time. Deleting fragments is undesirable to the realism of the simulation, so we prioritize fragments smaller than a certain threshold as they are too small to significantly obstruct the environment. Tree leaves are automatically marked for deletion, as they have flat meshes and exist in very high numbers per tree. In the case that large fragment count grows too large, large fragments begin spontaneously breaking apart into small fragments until large fragment count falls below the count limit. The new small fragments will delete themselves over time.

Deleting fragments has a 950% framerate improvement from keeping them in the scene in the tested environment. The workstation utilized for testing was running Windows 10 Pro 64-bit, with an Intel Core i7-8750H CPU @ 2.20 GHz, 16 GB RAM, and an NVIDIA GeForce GTX 1070 with Max-Q Design. 25 objects in a given terrain were fragmented across all trials.

Table 1. Comparing Fragment Deletion Performance

	Framerate (frames per second)	
	Small fragments are deleted	**Small fragments are not deleted**
Trial #1	59	4
Trial #2	55	7
Trial #3	57	7
Average	57	6

The SpeedTrees we use contain the whole tree within one mesh (discounting LODs). Because destruction affects leaves differently from wood, it is more convenient to separate the mesh into trunk, branch, and leaf meshes. Since we know which trees will be used for a given scenario, we can separate the tree meshes beforehand and cache them in memory. When a tree is destroyed, we don't calculate its separated form and instead look up its pre-broken variant, saving additional time.

Caching reusable separated trees has an 1863.66% calculation speed improvement from separating trees individually on the tree tested on. The workstation utilized for testing was running Windows 10 Pro 64-bit, with an Intel Core i7-8750H CPU @ 2.20 GHz, 16 GB RAM, and an NVIDIA GeForce GTX 1070 with Max-Q Design. The same tree was fragmented across all trials. The time recorded represents the time it took to get separated tree data; the cached version looks up a cached separated tree, while the non-cached version calculates separations itself.

Table 2. Comparing Tree Separation Performance

	Time to separate tree (ms)	
	Trees are cached	**Trees are not cached**
Trial #1	2.62	63.73
Trial #2	2.98	61.89
Trial #3	4.41	60.56
Average	3.33	62.06

Limitations

Our destruction system is being developed in parallel with the classification system that determines if a given GIS mesh is the ground, building, or a tree. As the classification grows more features, we will be able to be more specific with the destruction. A building won't simply be made of concrete; walls, windows, etc. will be able to be differentiated.

Due to float precision error, we occasionally run into cases where data comparisons return unequal when they should have been equal. This can result in overlapping Voronoi cells. We implemented more lenient equality methods to greatly reduce the chance of an error, but a stronger solution is being considered.

FUTURE WORK

Unity Data-Oriented Technology Stack (DOTS)

Our work on modifying mesh information would be a good candidate for modifying for use with Unity ECS (Entity Component System) and Unity Burst Compiler. Both are used in a data oriented programming ideology, a departure from Unity's standard component oriented design.

Within the Entity Component System, data such as mesh information is stored separately from the objects they represent. Instead of an object containing its own vertices and triangles, all vertices of all objects are stored within one data structure, and all triangles in their own data structure. By doing this, like data is guaranteed to be sequential in physical memory. When calculations are done en-masse on vertices, lookup times are reduced due to much less frequent cache misses. The Burst Compiler creates strict requirements of code design that allows the compiler to make time saving assumptions that would otherwise cause errors in a standard compiler.

Previous work (McCullough et al., 2019) showed significant gains in calculation speed when using Unity DOTS. Using DOTS to calculate transformations for hundreds of thousands of identical entities traversing across a map had significant performance improvements (7400% faster than conventional object oriented programming) (McCullough et al., 2019). Destructibility exists in a similar situation, in which many calculations must be done on a large amount of objects of the same type.

Geotypical, Geospecific Interiors

Currently when a building is fragmented, it is treated as if the entire interior were filled with concrete. Buildings in real life are rarely solid blocks of concrete and have interior spaces. It would greatly increase the realism of the

simulation for buildings to have interiors. A trainee would no longer treat buildings as thick walls. They would have to be mindful of the ability to enter buildings, and the effects of destruction on those who may be inside.

Two options exist in moving forward with interiors. Geotypical interiors would be procedurally generated, either possibly completely, or by stitching geotypical interiors together to fit the building space. This would automate populating interiors but wouldn't be able to replicate real world buildings. Real world, geospecific interiors would require GIS capture of the building interior, and then matching up the exterior 3D mesh with the interior mesh.

Interior generation would be a non-trivial task. Geospecific interiors would need to be properly scaled and positioned to fit within the building it represents. The increased vertex count would affect calculation time. Geotypical interiors would require a knowledge of a building's volume, when currently our buildings are created of multiple game objects.

Material and Feature Attribution

As the system works now we apply varied techniques to buildings, ground, and trees. Furthering our integration with the STPLS+ pipeline as it develops will allow us to begin taking advantage of higher precision attribution into deeper levels of a classification hierarchy. In the future we will be able to identify if a building is made of wood, steel, concrete, clay or any other building material. We will be able to utilize additional ground material classification to understand if the affected target is dirt, sand, or road. Taking these factors into account will allow us to expand the programmatic destruction models to more realistically represent a destruction event, as well as beginning to account for more complex materials such as glass and ductile objects such as vehicles.

Effects on the Navigation Mesh

Fragments created from destructibility contain Unity's Rigidbody as a component. Rigidbodies affect an object's mass and allow it to interact with Unity's physics system. However, fragments do not affect the navigation mesh used for artificial intelligence (AI) pathing. An AI entity would walk into a hunk of concrete in its way instead of pathing around it. Implementing a way for mesh fragments to alter the navigation mesh would increase destructibility's interaction with the environment.

CONCLUSION

Implementing destructibility in a simulation allows users to visualize the effect of munitions on real world terrains. Users will be able to visually confirm destruction and will interact with a changed environment as a result of their actions. A real time simulation that uses a large number of algorithmically derived models cannot feasibly use pre-destroyed models, so this solution was developed as a way to calculate mesh destruction and deformation in real-time. By optimizing choice of data structure and caching as many calculation results as possible, the performance of this destructibility was improved. While the Unity engine was used for prototyping, destructibility is designed to be engine agnostic only requiring replacing physics function calls for functions used in the given environment. In the future, this system will further enhance its capabilities and performance.

ACKNOWLEDGEMENTS

The authors would like to thank the two primary sponsors of this research: Army Futures Command (AFC) Synthetic Training Environment (STE), and the Office of Naval Research (ONR). This work is supported by University Affiliated Research Center (UARC) award W911NF-14-D-0005. Statements and opinions expressed and content included do not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

REFERENCES

Chen, M. Feng, A. McCullough, K. Bhuvana Prasad, P. McAlinden, & R. Soibelman, L. (2019). Fully automated photogrammetric data segmentation and object information extraction approach for creating simulation

terrain, *In Proceedings from the Interservice/Industry Training, Simulation and Education Conference (I/ITSEC) 2019*

Cook, D. (2013). *Full Geo-Mod Enabled Saints Row Is "Literally Impossible" in This Gen, Says Volition*. 12 Aug. 2013, www.vg247.com/2013/08/12/a-fully-geo-mod-enabled-saints-row-is-literally-impossible-in-this-gen-says-volition/.

Dobransky, M. (2017). *Efficient simulation of environment destruction in games.* (Unpublished Bachelor Thesis). Charles University, Prague, Czechia.

Ertl, B. (2015). *Euclidean Voronoi diagram* [SVG]. Retrieved from https://commons.wikimedia.org/wiki/File:Euclidean_Voronoi_diagram.svg

GeoGebra. (2020). *Geometry*. https://www.geogebra.org/geometry

Ledoux, H. (2007). Computing the 3D Voronoi diagram robustly: an easy explanation, *4th International Symposium on Voronoi Diagrams in Science and Engineering*

McCullough, K. New, R. Nam, N. & McAlinden, R. (2019). Exploring game industry technological solutions to simulate large-scale autonomous entities within a virtual battlespace, *In Proceedings from the Interservice/Industry Training, Simulation and Education Conference (I/ITSEC) 2019*

Orry, T. (2009). *Red Faction: Guerrilla Interview*. 14 Apr. 2009, www.videogamer.com/previews/20090414111852-red-faction-guerrilla-interview

Spicer, S., Mcalinden, R., & Conover D. (2016). Producing Usable Simulation Terrain Data from UAS-Collected Imagery. *In Proceedings from the Interservice/Industry Training, Simulation and Education Conference (I/ITSEC) 2016*

THQInsider. (2009). *Red Faction: Guerrilla - Geo-Mod 2.0* [Video]. YouTube. https://www.youtube.com/watch?v=lICurOVsNv0

Van Gestel, J. (2011). *Procedural Destruction of Objects for Computer Games.* (Unpublished Master Thesis). Delft University of Technology, Delft, the Netherlands

Whirle, D. (2020). *UnityAssets, GitHub repository, https://github.com/BLINDED-AM-ME/UnityAssets*