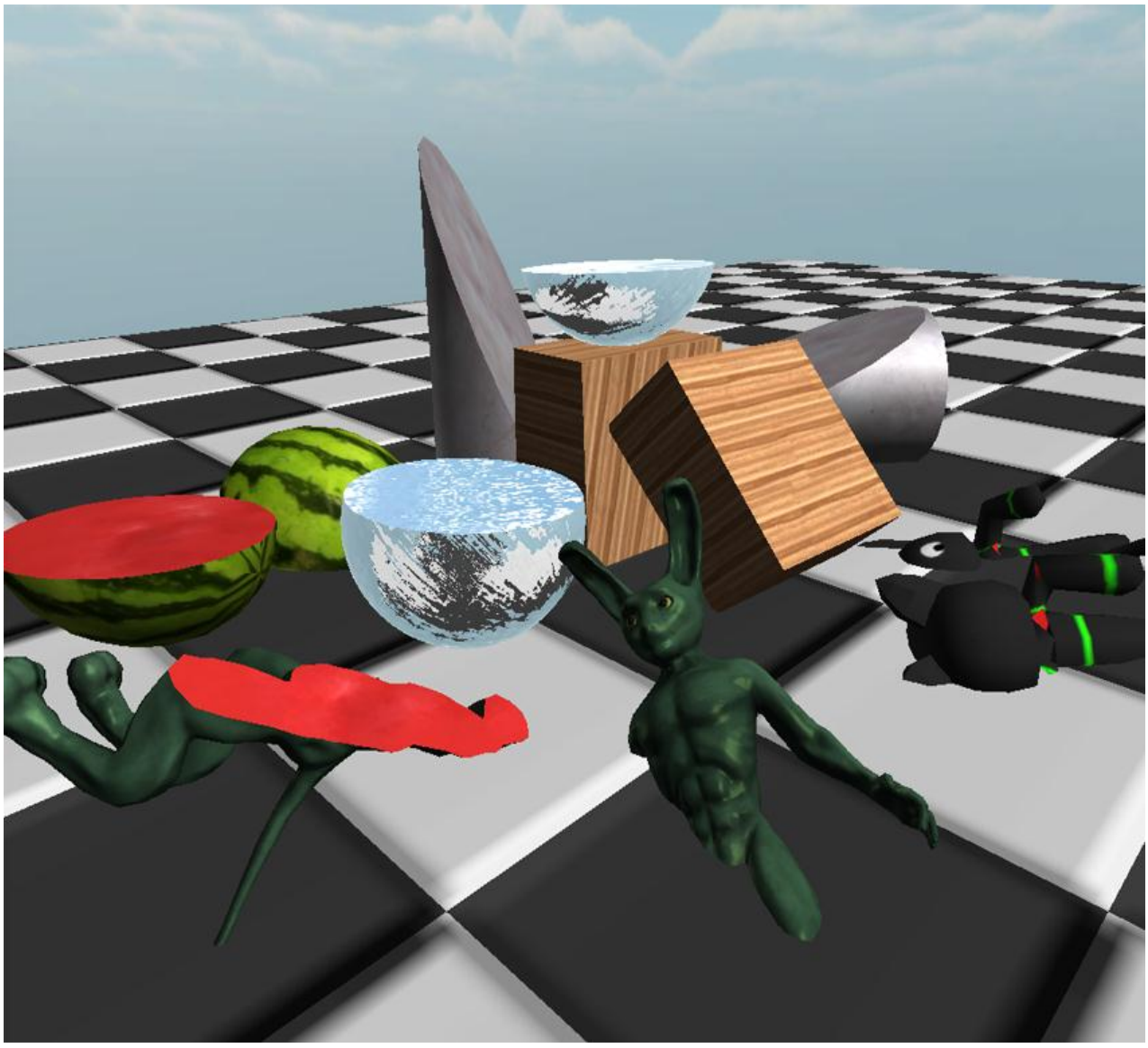

Procedural Mesh Splitting

An M.Sc. thesis
by Danni Bransholm Schou

01/06/2012
IT University of Copenhagen
Supervisor: Mark J. Nelson

Email: ddarkdingo@gmail.com
Website: http://danni.foxesgames.com

Author: Danni Bransholm Schou
Supervisor: Mark J. Nelson

Abstract

The purpose of this project is to create procedural mesh splitting algorithms for games, for both convex and concave meshes. These algorithms should then be used to create a middleware style solution that can be used by a developer to create a game that features this technology.

The need for this technology has already been shown by a hit mobile game and an upcoming AAA game. Creating good algorithms for procedural mesh splitting poses a series of challenges. A middleware solution for procedural mesh splitting would enable game developers to focus on creating interesting gameplay using this technology, instead of spending time developing it.

In this report I will explain the theory behind the algorithms I came up with, and show how I chose to implement it in a game engine. I'll explain the challenges I faced during the development, how to overcome them, and which challenges that are still to be solved.

Table of Content

Author: Danni Bransholm Schou
Supervisor: Mark J. Nelson

Introduction

This thesis began as an experiment in my spare time. I wanted to spend a weekend creating an algorithm for a simple mesh split on a convex object. It was a lot more complicated than I initially thought and in the end I had spent 3 weeks creating my first working prototype. The feedback I got people was so overwhelming that I had to expand this into something bigger and I decided to make my thesis about it.

The goal of this thesis will be to come up with algorithms for procedural mesh splitting. To successfully split a mesh there is two steps: The first one being the actual mesh split dividing a single mesh in two. The other step is to add geometry to close off the hole(s) created from splitting the mesh.

The simplest way to split a mesh would be using a plane that instantly splits the mesh. I've chosen to use this type of mesh splitting in order to focus on the mesh splitting algorithms. When looking at the mesh we want to split there is a big difference between convex and concave objects. A plane split of a convex object will always leave us with a single convex hull. When we split a concave object by a plane we could get several hulls of both convex and concave shapes. I want to address both cases and create a middleware style solution that can be used by a developer to create game featuring procedural mesh splitting.

When creating a cap for the split meshes we also need to think about texture mapping. A middleware solution should allow for the developer to map the new surface to a texture in a meaningful way, making the visualization of the new surface look intentional. When splitting a mesh we also need to split its game object and make sure the game state for the new object is maintained. At the same time we should also adjust the new game objects colliders and weight to reflect the mesh change.

Performance is an important part of game development. Therefore I should look into optimizing the mesh splitting algorithms, and create some performance guidelines that would help a developer to use this technology in the best way for a given project.

Related Work

While developing my thesis I became aware of a few projects that are similar to what I wanted to do. The first of these is a game currently being developed by Platinum Games called Metal Gear Rising, which is a part of the Metal Gear Solid franchise. The other is the widely popular mobile game Fruit Ninja developed by Halfbrick Studios.

Metal Gear Rising first featured a live demonstration[1] of mesh slicing with a Samurai sword where they sliced bowling pins, melons, and a small robot. They also released a trailer[2] later on with in-game footage where they sliced enemy soldiers and robots in combat situations.

Figure 1 - Metal Gear Rising Live Demo, slicing bowling pins and melons.

There is a couple of interesting parts to notice from what they've shown. First thing to notice is how good the uv mapping of the melon is for the sliced surfaces in Figure 1. The next thing that's interesting is a bit harder to notice but the collision detection for the two robots isn't very precise as the upper robot is sliced it falls directly through the lower one and afterwards the still standing robot puts its arm through the sliced one.

Figure 2 - Metal Gear Rising Live Demo, Collision Problem

For the trailer the most interesting part to notice is

[1] Metal Gear Solid: Rising – Slicing Gameplay - http://www.youtube.com/watch?v=FMdu_dUzjfE
[2] Metal Gear Rising – Revengeance Trailer - http://www.youtube.com/watch?v=5SNo8h-KfAU

how they've chosen to map the sliced surfaces. During the live demo we saw beautifully mapped melons, and in the trailer when we look at the pre-rendered scene they put a lot of effort making it look good, but during the in-game footage we see how the sliced surfaces of the enemies looks more like tinfoil. This mapping difference is clearly shown in Figure 4.

Figure 4 - Metal Gear Rising Trailer, Sliced Surface Mapping Difference. Left side in-game real-time, Right side pre-rendered.

The slicing being done in Metal Gear Rising is very close to the mesh splitting I began creating. The biggest difference is that they're doing a solution specifically for their game, where I want to find a more general solution that can easily be applied to any game where it would be suitable.

Fruit Ninja is a mobile game were they are using the touch surface of mobile devices to slice fruit as their main gameplay. As of March 2011 they exceeded 20 million downloads across all their platforms[3]. This means it has been hugely popular and the main gameplay is closely related to my mesh splitting.

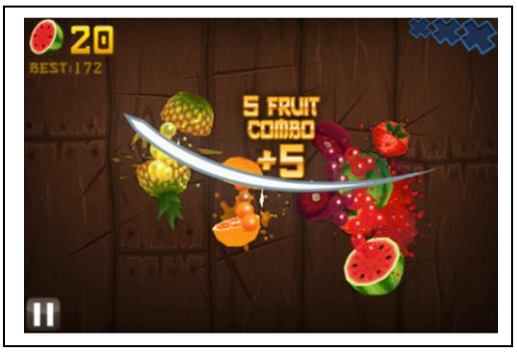

Figure 3 - Fruit Ninja

The big difference between my mesh splitting and what they do in Fruit Ninja is that mine will work in real-time on any object. They're using pre-sliced objects, so when you slice an object it just replaces the whole piece of fruit with two sliced pieces instead. This is quite fast and easy to do, but at the same time you're also limited to only exchanging objects.

[3] Fruit Ninja surpasses 20 million total downloads, Joystiq, 3rd of March 2011 - http://www.joystiq.com/2011/03/03/fruit-ninja-surpasses-20-million-total-downloads/

Demonstrations

Before explaining how all the algorithms works I want to show the results so it will be easier to understand what I do and why when I go into the theory behind it and the implementation.

When I first began this project I made a small tech demo that showed convex mesh splitting in a simple setup. After developing convex mesh splitting further I made a game for a Unity competition that used convex mesh splitting for multiple animated convex objects called Gladiator Tailes. I made a new version of the tech demo after I finished concave mesh splitting and made examples for the different cases the current state of my mesh splitting covers.

Tech Demo

The tech demo is a simple showcasing scene. It basically consists of a checker floor and a simple sunny skybox. I didn't want to focus on creating a specific control scheme for the tech demo as the controls in a game using this technology could be very different. This lead to me using standard fps controls for moving around (WASD + mouse look). I also wanted completely user controlled splitting, so to split an object the user first has to get in position and then hold space to be in split mode. When in split mode the mouse look is locked and the user can then hold left mouse button and drag to create a split that will be executed on releasing left mouse button. These controls are not suited for a game and are only for the purpose of showing the actual mesh splitting.

The demo itself contains 7 different mesh splitting cases and 6 different materials to cycle through. Every object can have its material switched by pressing Q or E, and a new object can be spawned by pressing 1 through 7. One thing that should be noted that isn't visible to the user is that the only thing required to make the user able to split objects is putting a component on them called Splitable. So setting these objects up to enable mesh splitting is very quick and easy for all cases.

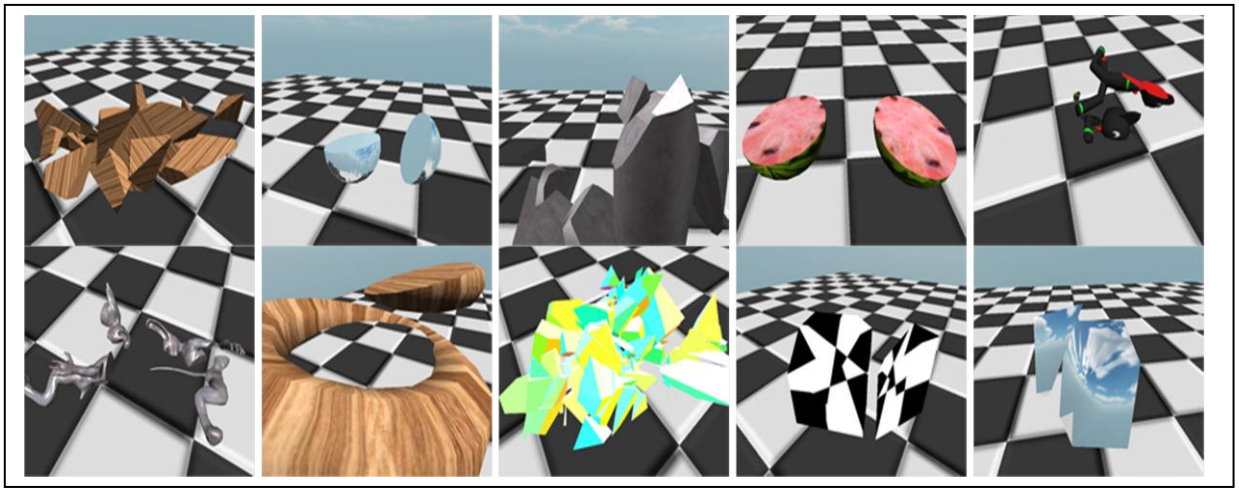

**Figure 5 - Procedural Mesh Splitting Tech Demo, Objects 1-7 including 3 extra materials.
First row left-right: Wooden Cube, Metal Sphere, Marble Cylinder, Melon Ellipsoid, and Convex Game Character.
Second row left-right: Game Character, Torus Problem, Debug Normals Material, Checker Cube Material, and Environment Cube Material.**

The first four objects are simple convex objects which are all using my convex mesh splitting algorithm. The first one being a cube and one of the simplest ways to demonstrate mesh splitting. The second object is a sphere. This case is very similar to the cube but it's a good way to show how the physics are maintained after a split. This is visible if you first push the sphere and then split it.

The third example is still a convex shape, this time a cylinder. The difference here is that when the mesh is split there will be added force to the two new pieces pushing them away from each other. This becomes more visible the smaller the two new objects are. The forth object is an ellipsoid textured like a melon. The interesting part here is that the inside of the melon is mapped to a specific part of the texture, so it looks like you cut open then melon.

The next two objects (5 and6) are sample game characters. The first one was made for a Unity competition when only convex mesh splitting was working. This means that the character had to be built of several convex meshes and then animated like one big mesh. This is not a great solution but it served its purpose for the competition. The other object is a more standard game character where you have only one animated mesh. Splitting this character is similar to the slicing being done in Metal Gear Rising.

The last object is a torus. This concave mesh shows one of the problem areas for my current solution. If it's split vertically then it'll work fine, but if it's split horizontally then the hole in the middle will cause a problem and the area will be covered on the side facing the split as seen in Figure 5.

Besides the different objects there are 5 different materials to cycle through and try on all the models. The first three objects contains the first three materials: Wood, reflective metal, and marble. I've then included a material that shows the normals of an object. This material is only for debugging purposes and never something you would use in a game.

The last two materials are another take on mapping the inside of an object. Instead of using a normal texture I use a cubemap that is mapped according to vertex position rather than uv coordinates. The two samples are a simple checker pattern and the environment map of the scene. This way of texture mapping is not necessarily better, but it was something I tried out to show alternative ways of mapping a texture. The method is a lot less flexible than the normal uv mapping and you can't batch multiple objects together when rendering. On the other hand as long as your cubemap matches up, then you'll always have new surfaces were the edges are matching. The method also require the object to be centred around the local origin (0, 0, 0), and this is also why batching won't work because it groups multiple objects together as one and offsets their placement.

Gladiator Tailes

Gladiator Tailes was I small game I developed together with James Testmann for a Unity competition. We had two weeks to create a flash game using the Unity engine. I wanted to use this opportunity to try and develop a game that used my mesh splitting. At this point though, I had only developed convex mesh splitting. Since we didn't have a lot of time we decided to go for a simple arena gameplay, and we would use mesh splitting for when enemies were killed.

Figure 6 - Gladiator Tailes, screenshots

Mesh splitting ended up being used for two different things in the end. We used it for when enemies got hit by a weapon and for the menu. As for the menu I simply used metal cubes that will get split vertically when they're being clicked on. This is mostly to feature the mesh splitting and give the menu a nice finish. The split happening here is a simple convex split without creating a cap. The main reason for not creating a cap were because Unity had a bug in their flash exporter and since the competition required the game to be in flash we had to go without caps.

For splitting enemies we decided on having two different weapons a ranged energy disc you would throw to split them in two, and a melee energy sword that you would swing to split the enemy in two. Since only convex mesh splitting was working at this point all the enemies were build up by multiple convex meshes animated together. To increase the splitting effect further the meshes are split and then the splits are applied on two ragdoll versions of the enemy. This turns the one enemy into a split ragdoll upon death.

Both the tech demo and Gladiator Tailes are just simple examples of how to use this technology. The design of the Splitable component allows a designer or programmer to easy utilize mesh splitting for almost any purpose without any customization. I will touch more on this in the next chapter.

Procedural Mesh Splitting

I've divided mesh splitting into 4 sections, to best explain how the algorithms I've created and what they do. First I'm going to talk about the general theory behind each step. I will explain the math I required to perform mesh splitting. After giving a theoretical overview and explain the math I will then talk about my implementation in Unity. Ending this chapter there is a few things that I'd like to discuss that didn't fit in during the other sections.

Theoretical Overview

First I'm going to talk about the difference between convex and concave shapes as the application of this concept to 3D meshes is very relevant for this thesis. I'll then give a short introduction to how Unity uses objects and components in the engine as my component design is based on this type of game engine. If you're already familiar with component based game engines then this part can be skipped. The rest of Theoretical Overview will explain how I perform mesh splitting on a theoretical level.

Convex and Concave Meshes

Before talking about anything else it is important to address the difference between convex and concave objects and how it affects mesh splitting and cap creation. A convex shape is a shape in which any two points in it can be connected by a straight line and still be within the object. A concave shape is the opposite of convex. In a concave shape not any two points can be connected within the shape as seen in Figure 7. This concept also applies to a 3D mesh, it can be convex or it can have a more complex shape which we'll call concave to keep it simple.

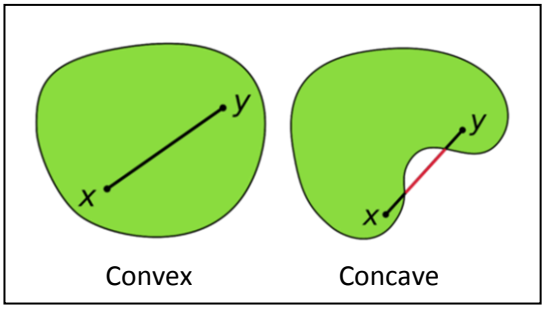

Figure 7 - Convex vs. Concave

The difference between these two is not important when we're splitting a mesh since this process is almost identical. The difference is when we need to create a single convex cap or multiple concave caps after the

mesh has been split. When a convex mesh is split it will always leave a single convex hull. This case makes it simple since we can cap this hull by sorting all the points and then make a triangle fan. On the other hand we could have a mesh that is far more complex, it could even consist multiple separate shapes. In this case we won't be able to predetermine the shape of the hull. We can be end up with a convex hull, a concave hull, or even multiples of both.

We can also have torus shapes with "holes" in them like a donut. If the torus is split horizontally we would wind up with a case where we have two hulls and we would need to create a cap between them, instead of creating one cap per hull. This creates a new type of problems since we now can have hulls inside hulls. I'm going to address this problem more in depth during the discussion as this is quite complex.

This gives us a quick overview of the challenges in mesh splitting and I'll go into more detail as I present how to split meshes and cap their hulls.

Game Engine Basics

Since I'm using Unity for my implementation it is important to understand the basics of how they handle game objects and how my classes interact with them. This is only a very brief explanation that should give a rough understanding of these concepts if they are unfamiliar.

Game Objects

The Game Object is the base of every object in a scene. Each Game Object contains a list of Components that adds functionality to the object. An object will always have a Transform component since this component keeps track of the objects position, rotation, and scale in the scene but more important it's a part of the object hierarchy making it obligatory.

Components

A Component is best described as a piece of functionality that can be added to a Game Object. Unity comes with a series of built in components that utilises the engines different subsystems, such as the Rendering

pipeline, physics system and audio system. You can create your own Component in Unity by creating a class and inheriting the Mono Behaviour class.

Mesh Splitting Component Design

My implementation of the mesh splitting designed around using two main components. The most interesting one of these is the *Splitable* component. This component has the responsibility of splitting the objects mesh and creates two new objects from it. The other component is the *Splitter* component. This component triggers the *Splitable* component and gives it information about how the object should be split.

The Splitable Component

The Splitable component is a basic implementation that handles both convex and concave mesh splitting. I've chosen to create a single component that serves as the default mesh splitting component. This makes mesh splitting easy to use as you just add the Splitable component and then it works. It is also possible to inherit the *ISplitable* interface and make a custom *Splitable* implementation that still gets activated by Splitters.

My implementation of the Splitable component covers most general cases. But I found that it's important to allow different implementations during my work with Gladiator Tailes. So giving the developer the option to create other ways to handle mesh splitting was important. The responsibility of the Splitable component is to wait for a Splitter to trigger it. When this happens it should split its mesh, create caps for the hulls, and create two new objects containing each part of the split mesh.

It's important to understand that a mesh in a game engine only renders one side of its triangles. This can be best visualized as a one way mirror. On one side you would see your reflection but on the other side you would see through it. It's possible to set both sides to render, but this is off as default to improve performance.

In the case that a mesh split result in two new meshes there is two options. If the object represents a solid object then you would want to create a cap where the mesh was split. You could also have a case where you wanted a hollow box; in that case we wouldn't want to create a cap. In the last case you'd either want to have a double sided mesh or turn on rendering for both sides of the mesh.

If splitting the mesh resulted in two mesh parts then new objects should be created. The new game objects should be almost identical to the original. The mesh should be replaced with the new generated mesh and components related to the mesh. The collider needs to be updated with new collision data. If a rigid body is being used then adjusting the weight of the new objects will give it a better feel. It's also important to transfer the current physics state to make the objects continue their movement. After the new objects have been created we just have to remove the original to finish the split of the object.

The Splitter Component

The *Splitter* component is in itself very simple. It is attached to the game object you want to affect other object into being split. The way to use the splitter component is very dependent on what kind of action should split the desired object. In the game *Gladiator Tailes* I use the Splitter in two different ways. The energy disc has a persistent splitter component on it that splits everything it collides with. The energy sword creates an invisible object with a splitter that only exists for a single frame.

The splitter requires a few other components to work. First it needs a collider to detect when it collides with objects. We want to set the collider to be a trigger, since the collision should cause the mesh to be split instead of the normal physics behaviour. It's also a good idea to have a rigid body component even if we don't want it to act like a rigid body. That way it will detect collision with all objects that has a collider. We can turn off gravity and set it to be kinematic in case we don't want the normal rigid body behaviour.

The Splitter component will listen for trigger events. When an object is triggered it searches the other object for an ISplitable component. The ISplitable component is an interface for mesh splitter components.

The Splitter component will then tell the ISplitable component to split its mesh using the Splitters Transform component.

Mesh Splitting Theory

The next parts will contain the theory needed to know in order to create a mesh splitting implementation. The things covered here are caching of mesh data, splitting the mesh, creation convex and concave cap, creation new meshes, and creating new game objects.

Mesh Data Caching

Unity gives us access to a copy of the data we need through a property. This is both slow and it's unnecessary to request this data multiple times. We need to access the mesh data a lot and we would want it to be as fast as possible. Secondly we need the world space coordinates for all vertices to split the mesh, so these needs to be calculated and stored for the mesh splitting.

The caching of mesh data is very simple. I would suggest the cached data to be structured in the same way as the engine stores meshes to so that mesh creation later on will be faster. It's also important to create dynamic arrays for new vertices added during mesh splitting and cap creation. We also need a triangle lists for the new upper mesh and one for the new lower mesh. These will be used to construct the new meshes after the mesh has been split.

After we've cached the mesh data we then need to calculate the world space position for each vertex. It's important to use world space coordinates instead of object space coordinates for two reasons. First we need world space to support non-uniform scaling. We can adjust for position and rotation by inverting those from the splitter, but we can't invert non-uniform scaling. Therefore we need the object to be at the splitters position and not the other way around. The second reason is that we need it to support skinned animation. We want the split to happen on what we see so we need to calculate the position for each vertex in the current animation state when using skinned animation.

To calculate world space coordinates for a non-skinned mesh, I just create the world matrix for my objects using its transform and then I multiply it with each vertex position. Calculating world space coordinates for a skinned mesh is a bit more complicated. First we need to calculate the matrices for all bones. We do that by multiplying the world matrix of the bones current transforms with their bind pose. Then we need to create the world matrix for each vertex. We use the vertex bone weights and bone indices for this. The bone indices tells us which bone matrices to use, we then multiply each bone matrix with its bone weight and add the matrices together to create the world matrix. After the world matrix is created then we just multiply it with the vertex position to get the world space position.

The Mesh Split

Splitting a mesh into two parts by a plane is the same no matter how the mesh is. The only difference in this part when handling convex and concave meshes is how cap vertices are stored. The calculation is based on using a plane as the splitter. If we wanted a finite sized plane or another shape then we would need to adjust our calculations for it but this is outside the scope of this thesis.

Splitting the Triangle

When talking about games every mesh is a series of triangles. To split our mesh we have to process one triangle at a time. Each triangle consists of 3 lines and I by testing for Plane-Line intersections on each of these we can find out if the triangle is intersected by the splitter plane. If no lines were intersected then we can check which side of the plane one of the vertices in the triangle is on, to determine if it should be sorted in the upper or lower triangle list and add them. If on the other hand there is a line split in the triangle then we need figure out which lines are split.

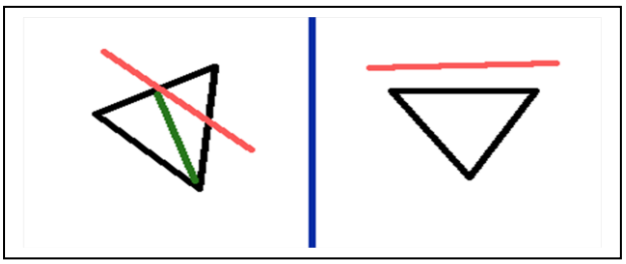

Figure 8 – Triangle Split. Left: Plane intersection. Right: No intersection.

To split the triangle we can use the interpolation value from the plane-line intersection and use it to create two new linear interpolated vertices, one for each intersection. Then one side has a triangle that we can

add to the side of the plane it belongs to. On the other side we have a quadrangle that we need to turn into two triangles instead, and then we can add these two triangles to their appropriate side.

Cap Index Storing

In order to create a cap we need to store a reference to the interpolated vertices as these are the vertices making up the new cap. For convex meshes the index of each vertex is stored in an array as this is all we need to create a cap. For concave meshes we need the split edge. The split edge is made up of the two interpolated vertices. The reason we need edges instead of just vertices for concave is because the hulls are far more complex and we need edges to recreate the hulls properly.

Creating the Convex Cap

When creating the cap we first want to transform all our cap vertices into 2D space. It makes the math a lot easier to only have two dimensions and it's easy to convert the coordinates. To transform the coordinates I can use the inverse rotation of the splitter plane. This virtually eliminates the Y component of the vectors and we now have a 2D vector by using the X and Z components. I store this data in 2D vector structures as X and Y to remove the unnecessary component.

Preparing Vertices for Sorting

Then the vertices need to be sorted. I've looked into a few convex hull algorithms in order to find a good way to organize and sort vertices. Usually a convex hull has more points than just the hull part, so I ended up making my own algorithm based on the Graham Scan[4].

My algorithm first finds the lowest vertex. This is the vertex with the lowest Y value, in case there are two vertices with the same Y value it will be decided on the lowest X value. I then need to calculate the angle from the lowest vertex to the rest so they can be sorted. I create an "up" vector (0, 1) and then I calculate the dot product between the up vector and the direction from the lowest vertex to each other vertex.

[4] Graham Scan, Wikipedia - http://en.wikipedia.org/wiki/Graham_scan

Doing this will give me that any vertex in the direction “up” will be exactly 1 and the directions left (-1, 0) or right (1, 0) will result in 0. This means that we can’t tell the difference between right and left of our lowest vertex. To correct this problem I subtract the dot product from 2, if the vertex I’m calculating the angle for has a lower X value than the lowest vertex. Figure 9 shows an example of how the angles of a hull could look after being calculated with this method.

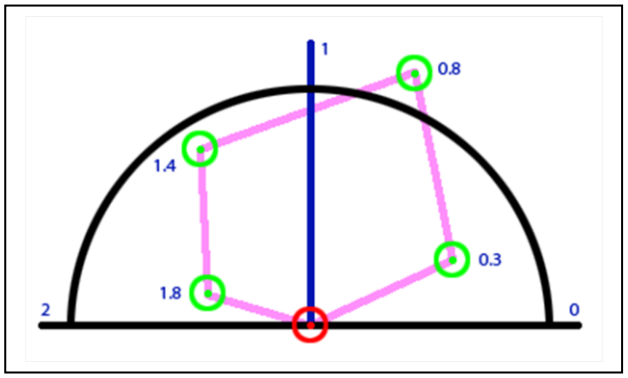

Figure 9 - Convex Hull Angles

Sorting the Vertices

Now that everything is prepared for the sorting we just have to use a sorting algorithm to sort the data based on the angles we just calculated. In my implementation I ended up using gnome sort[5]. This is far from the best sorting algorithm and it has a terrible average and worst case complexity that’s $O(n^2)$. I choose this because it was very fast to implement and because the focus of this thesis isn’t to implement the best possible sorting algorithm. In a commercial mesh splitting solution this algorithm should be replaced by a better one.

After the sorting array has been sorted according to the angles we still have one problem. Multiple vertices in the beginning and in the end can have the same angle value, and this is something that occurs regularly. An example of this can be seen in Figure 10. We need to sort the beginning and end separately after the initial sort to correct this

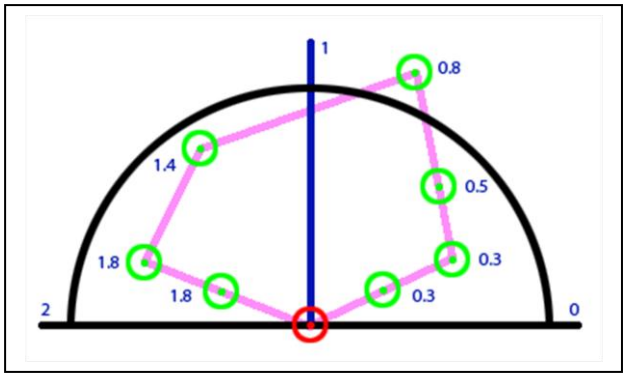

Figure 10 - Convex Hull, even beginning and end.

[5] Gnome Sort, Wikipedia - http://en.wikipedia.org/wiki/Gnome_sort

problem.

For this I've created a modified gnome sort that only sorts the beginning while the angle is the same, and then uses the Z value to sort the vertices into the right order (if the Z values are the same I also use the X values to determine which should be first). I've also created a modified version of the gnome sort that uses a reversed order to sort the end only. Like the sorting algorithm for the beginning, this one tests the vertices with the same angle and sorts them according to their Z value, where the lowest Z value is sorted into the latest positions. For the end vertices the X values doesn't matter since the only scenario where it would matter is if the angle is 2 (180°), but this can never happen since the lowest vertex is the one with the lowest Z value followed by the lowest X value making this scenario impossible.

Removing Redundant Vertices

When all the cap vertices are sorted in the right order, then we can create the triangle fan for the cap. Before we do this there is an optional thing we could do to give us a better result. The cap indices are just added from the mesh splitting whether they are actually needed or not, so at this point we could remove all the redundant vertices. All the vertices that a placed between two vertices with a 180 degree angle doesn't add anything to the shape of the hull. An example of this can be seen in Figure 10 where three redundant vertices are present.

Whether we leave these remove them the visual appearance will be the same. The choice we have is whether we want to spent a little more time removing those extra vertices now, or if we want to spend more time drawing the object to the screen each frame. In my opinion there is no correct answer for all solutions here. If we have simple mesh I would suggest not removing the redundant vertices since there won't be much to remove and therefore not much to gain in drawing speed. If we have a mesh with a lot of vertices I would suggest taking the performance hit up front and remove the redundant vertices. Since this choice is only for convex shapes most situations it will be fine just leaving them in.

To remove redundant vertices we want to test each vertex, and check if the direction from the previous vertex to the current, and the direction from the current vertex to the next vertex is the same. If they are the same it means that the current vertex is redundant and can be removed.

Adding Triangles

Before we can add the new triangles we first need to create the actual vertices for the cap. Currently we have indices for the vertices that define the hull of the cap but we want to add new vertices for the actual cap triangles.

The biggest reason for this is that the vertices need the normal of the cap surface to render lighting correctly. Therefore we use the cap indices to get the vertex data and create the new vertices with the same properties, but with the normal of the splitter plane. We need to do this twice since we also need vertices with the inverted normal of the splitter plane. This is because we need to create two different cap vertices, one for the upper mesh and one for the lower mesh.

Now we have everything we need to add the triangles to the two new meshes. We can then add all the triangles to the upper and lower mesh. The triangle list is a triangle fan so to add triangles I add the first index followed by index 2 and 3, then I take the first index again followed by index 4 and 5 and so on. One thing that's very important when adding triangles is that they need to be clockwise ordered. The wrong ordering will result in triangles only being drawn on the other side (which should already be covered by the rest of the mesh). At this point the cap is done and the new meshes are ready to be created.

Creating the Concave Cap

Like we did with the convex cap we first need to transform all the cap vertices, only this time we have edges instead of vertices. To transforming the edge we just have to transform both of its vertices.

Linking Edges

After the edges have been transformed into 2D space, we now need to link the edges together. Linking edges together will create one or more borders. To link edges together we take two edges test if the edges vertices match up. It sounds simple but creating an algorithm for this is a bit more complicated.

I use a linked list were each node contains a linked list of edge references (looking like this: LinkedList<LinkedList<int>>). I then try to match the outer nodes together and if there's a match then I'll connect the two inner nodes. When I have an outer node where the edges can make a border I then add it to my borders and remove it from the matching process. This is fairly complex to keep track of in mind so I've created Figure 11to give a rough example of how it works.

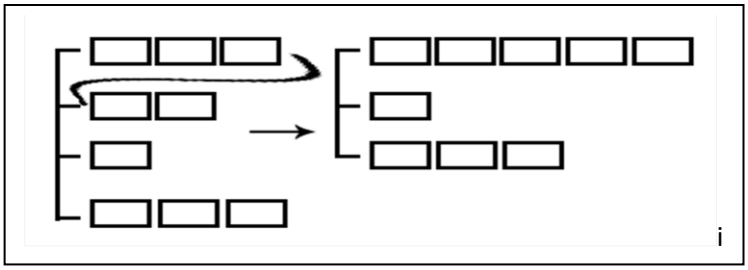

Figure 11 - Concave Cap, linking process.

After we're done finding all possible matches I then check to see if there's any uncompleted borders left. If there are any and they contain at least two edges, then I create a new edge to close off the gap and finish the border.

Understanding the way this works can be quite complex, and the complexity of this algorithm is also quite poor. In the best case scenario where everything is already ordered the right way we can get away with O(n), but the average and worst case scenario will be $O(n^2)$. Roughly speaking this is a slow sorting algorithm, but as we remove nodes from edge holder this will get better making it better than a normal $O(n^2)$ algorithm.

Checking Normal Direction

At this point we have all our complete borders. We now want to make sure all the edges in all borders have their normal facing outwards. Usually an edge wouldn't have a normal but for my algorithm to work we need to know which side is up or outwards. To have a normal I've chosen to call the two vertices in my

edges for left and right. To make the normal face in the correct direction we need to do two steps. First we find the edge in a border that has the highest right vertex, and by highest I'm testing for the biggest Y value.

I then find the edge with the left vertex that matches the highest right vertex. It is either the edge right before or right after the current one, and we need to do a comparison to find out which one it is. Now that we have our two edges we can make a test to see if we should flip all edges. I call this check an "inner side" test and it's one of the tests that allow me to create triangles from concave borders. The inner side test creates a plane out of one edge and then test which side the furthest vertex of the other edge is on. If both normals are on the same side then we want to flip all edges in that border since the normals are currently pointing inwards.

Creating Triangles

To create triangles from our borders we focus on one border at a time since each border is independent from the others. Too fill the concave border I came up with a way where I add triangles using the existing vertices where it's possible. This keeps filling the hull making it more and more manageable until the hull is filled.

I'm using two important methods to create my triangles. First I use my inside check, it's a way to check that the angle between two edges is less than 180. I do this because if it's less then means that I can create a triangle. If the angle between two edges is exactly 180 degrees then we need to melt the two edges together as the connecting vertex between the two edges is redundant. Creating a triangle with two edges that has a 180 degree angle would result in a triangle that's not visible as it would just be a straight line. The other method I'm using will check if a new edge will intersect with any of the existing edges, if there is any intersections then we can't use the new edge.

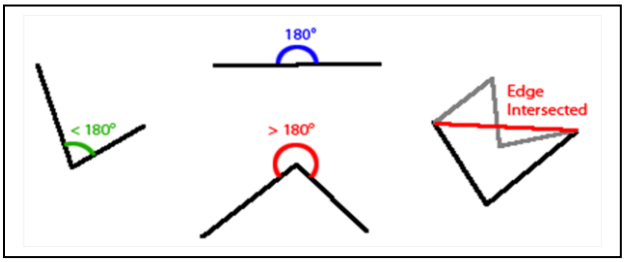

Figure 12 - Concave Edge Checks. Left inside check passed. Top inside check results in edge melt. Bottom inside check failed. Right edge intersection failed.

If the angle of two neighbouring edges is 180° then we melt the two edges together to one edge. If the angle is greater than 180° then we try to test in the other direction. If both directions fail then we move on to test another edge as this edge currently be part of a triangle. If the "inside check" passes and we have an angle less than 180° then we should first check if a closing edge is equal to the previous or next edge. If there is a match then we use the two edges and the matching edge as the triangle and remove all three from the border. If there was no match then we test if the closing edge would intersect any other edge, if it would cause intersection then we can't make a triangle and we try in the other direction. If there weren't any intersections then we can add the two edges and the closing edge as a triangle, we then remove the two edges from the border and replace them with the flipped version of the closing edge.

This process will go on until there's only 3 edges left which will make the final triangle and then we have a triangle list for the concave cap surface. We need to do this process for every border but can add all the triangles to the same triangle list since each triangle is always independent.

Adding Triangles

Adding our triangles is almost the same as with the convex cap. We first create the cap vertices by using the indices from the triangle list. We create two vertices for each index in the triangle list, one for the lower mesh with the normal from the splitter plane, and one for the upper mesh with the inverted normal.

With the convex cap we had a triangle fan but this time we just have a normal triangle list. So we can just add the three first indices, and then add the next three indices and so on. I suggest adding indices on a per triangle basis since you need to make sure the ordering is clockwise.

UV Mapping

The newly created cap for both convex and concave cap would use the UV component from the cap vertices they were referencing if we don't do anything. But if we want to then we can go ahead and improve the UV mapping of the cap surface. This is something that's optional to do but will almost always improve graphics quality.

To redefine the cap vertices UV's I begin by calculating the bounds of all the vertices. When I have the bound then I can offset and scale all cap vertex positions into a normalized space (from 0 to 1). We save the normalized positions as the UV coordinates. If we want the cap surface to be mapped to the entire surface of a texture then we can leave it as is. In my solution I let the user be able to specify an area of a texture it can be mapped to. If such an area is specified I then take the UV coordinates and then scale and offset them into the selected area. This is sort of the reverse process of normalizing the UV coordinates. The calculated UV coordinates is then used when creating the cap vertices instead of just copying the data from the referenced vertex.

The UV mapping process for convex caps should be done after the triangle fan has been created and before creating cap vertices as we need to UV coordinates for the cap creation. For concave splits we have two options. We could do it the same way as with convex caps and calculate UV's between triangle creation and creating the cap vertices. We could also do this on a per border basis, which I think looks far better and is what I've implemented in my solution. To do this I calculate the UV's for each border as the last thing I do before triangle creation. I think this solution is better as we will end up with less texture stretching in cases with multiple borders.

I've also experimented with a shader that maps texture based on vertex position and then uses a cubemap instead of a normal texture. The result of this is highly depending on the local vertex placement and will not look good if batching is enabled as this modifies the local vertex position in runtime. In my tech demo I have two samples of this. It's not an optimal way to map a texture as it requires model control to make it look good and it is very limited. But the upside to it is that the edges will always match up as long as the chubemap's edges matches up.

Creating the Meshes

During mesh splitting and cap creation all the vertex indices have been added to either an upper triangle list or a lower triangle list. When creating a new mesh I just take the wanted triangle list. I then create new

arrays for all the vertex structures based on the number of unique indices in the triangle list. I then copy all the needed vertices from the original mesh into the new mesh. The most important part of this process is that the indices of the new triangle list needs to be adapted to the new vertex arrays. To solve this I just create a lookup table that can translate the old index to the new index and create a new triangle list for the new mesh with the correct indices. When I have all the new mesh data I can just create a new mesh instance and assign all the data to it and return the final mesh.

Creating Game Objects

To create the new game objects we could create two copies of the current object and remove the current one, which is what should happen. To optimize this we could also just create one copy and adjust the current one instead which is a better way to do it. We then need to assign the upper mesh to one game object and the lower mesh to the other game object.

Besides the mesh I adjust colliders to reflect the new mesh. If the object also contains a rigid body then I copy the velocity from the original object onto the copy so the new objects will continue their physics behaviour. I also recalculate the mass of objects based on mesh bounds. This is a very rough approximation, but it's very fast and gives a good feeling to see objects being affected differently depending on their sizes.

In my Splitable component I also handle multiple meshes in children objects. This supports cases Like in Gladiator Tailes where I used multiple convex meshes to create one character. Instead of using one Splitable component on each mesh we can just use a single Splitable component on the parent. Then the Splitable searches for all meshes in its own object and in its children and splits all of them. As long as the parent object is copied with children there will be no major difference other than setting meshes for multiple objects instead of one. This also enables a trick where I can use a capsule mesh for the parent collider and have that split as well even though it's not visible.

Mathematical Theory

Here I want to address the math equations that I use for mesh splitting. Most of mesh splitting is about creating the right algorithms, but I use a bit of math that I'd like to cover here. These equations are not something I've created but they are some of the main calculations to perform mesh splitting.

Before showing the actual equations I'd like to talk a bit about rounding errors and the imprecision that was a big problem during the development of my thesis.

Rounding Errors and Imprecision

When I began working with mesh splitting and cap creation I quickly realised that even vertices that were supposed to be in exactly the same position often had an offset small enough that there was virtually no difference but still big enough that you wouldn't be able to do a normal equal test.

This resulted in using a threshold value. So when testing a vertex or any floating point that was connected to mesh splitting and cap creation, I would always calculate the difference and test if the difference was bigger or within the threshold. As soon as I had implemented this threshold I could remove duplicated vertices from the cap list before they got added. And also it would now detect virtually equal vertices as equal vertices. This threshold was also an important part as the line-line intersection in the edge class, as it would return too many false positives if I only tested for bigger than 0 and lesser than 1.

I've accepted $1*10^{-5}$ as the error I'm willing to accept when using a threshold. In game engines 1 unit is often considered 1 meter, this translates to an error of a tenth of a millimetre. If you don't use meters as units in your game engine then you would want to adjust the error threshold.

My best guess is that the reason behind these problems is that we only need vertices to be visually in the same positions combined with the precision of floating points is to blame for this mathematical imperfection.

Plane Math

We need two different equations to determine how each triangle of the mesh should be split by a plane. The first equation is to determine if a line is intersecting the splitter plane. The second equation is to determine on which side of the plane a point is. The two equations can be seen in Equation 1.

The intersect equation returns a value between 0 and 1 if the line intersected the plane. If there was an intersection, then you can make a linear interpolation with the result between *LineStart* and *LineEnd* to find the exact position of the intersection.

Plane-Line Intersect:

$$IntersectLerp = \frac{PlaneNormal \cdot (Point - LineStart)}{PlaneNormal \cdot (LineEnd - LineStart)}$$

Plane-Point Side:

$$PlaneSide = PlaneNormal \cdot (TestPoint - PlanePoint)$$

Equation 1 - Plane-Line Intersect and Plane-Point Side

We also need to sort all triangles into either an upper or lower mesh both when there was an intersection and when there wasn't. *PlaneSide* will be 0 if *TestPoint* is on the plane, positive if it's above the plane, and negative if it's below the plane. The plane direction is depending on the normal of the plane, so the upper and lower designation is just to give a logical overview and is not depending on the actual position of the mesh.

Edge Intersection

To perform edge intersection test I use a line-line intersection as seen in Equation 2. It calculates the linear interpolation for the intersection point. If both u_a and u_b is within the 0-1 range then there is an intersection between the two lines.

$$u_a = \frac{(x_4 - x_3) * (y_1 - y_3) - (y_4 - y_3) * (x_1 - x_3)}{(y_4 - y_3) * (x_2 - x_1) - (x_4 - x_3) * (y_2 - y_1)}$$

$$u_b = \frac{(x_2 - x_1) * (y_1 - y_3) - (y_2 - y_1) * (x_1 - x_3)}{(y_4 - y_3) * (x_2 - x_1) - (x_4 - x_3) * (y_2 - y_1)}$$

Equation 2 - Line-Line Intersection

Implementation

The implementation is split into two parts. The first part is an explanation of my implementation of the components the developer will have to enable mesh splitting in their game. The other part is the mesh

splitting classes that does the actual work. I will show some code examples which will mostly be slightly modified version of the actual code, but altered to help understanding them. I've been using C# as this is my most preferred language of the languages supported by Unity.

Components

To make it as easy as possible for developers there's only two components to think about. The Splitable component, this object is added to object you'd want to be able to split. And the Splitter component that is added to objects that trigger splitting.

Splitable Component

The Splitable component inherits the ISplitable interface. This interface defines a Split method that takes in a transform. I use an interface here so either I or another developer easily could make other ISplitable implementations and still get triggered by splitters.

The Splitable will use its Split function and wait for a splitter to trigger it. When this happens it creates a MeshContainer and an IMeshSplitter (could be either a MeshSplitterConvex or a MeshSplitterConcave depending on if the Splitable is set to convex or not) for each mesh in its own object and its children. Through my initial testing it seemed like it slowed down the physics system when splitting at this point. Instead I set a flag and make the component split the next frame.

The Splitable component then tells the mesh container cache mesh data and the mesh splitter to split the mesh and create a cap. If the mesh was split into two meshes then it creates the new objects. The code for it looks like the code in Code 1.

```
if (_splitMesh)
{
 _splitMesh = false;
 bool anySplit = false;

 for (int i = 0; i < _meshContainers.Length; i++)
 {
    _meshContainers[i].MeshInitialize();
    _meshContainers[i].CalculateWorldSpace();
    _meshSplitters[i].MeshSplit();
```

```
    if (_meshContainers[i].IsMeshSplit())
      {
         anySplit = true;
         if (CreateCap) _meshSplitters[i].MeshCreateCaps();
      }
   }

   if (anySplit) CreateNewObjects();
   _isSplitting = false;
}
```

Code 1 - Splitable Split

To create the new game object I use Unity's Instantiate method to create a copy of the current object. I then go through both objects meshes and replace them with either its new upper or lower mesh. If one of the split meshes only has either an upper or a lower mesh, then I remove the Render and MeshFilter for the object that doesn't have a new mesh. I do this to support multiple meshes within the same parent object while still only having to use one Splitable component.

If the parent object has a MeshCollider then I set this to the objects new mesh. If the object has a Rigidbody then I adjust its mass by multiplying the mass of the original rigidbody with the ratio of the mesh bounds. I also transfer the velocity to the new rigidbody to keep the new objects physics consistent with the old object. I've also put in an optional split force, if this force is set to something else than zero then it will add extra force pushing the objects away from each other. This split force is multiplied by its mass to keep it consistent as objects get smaller.

The last thing I do is to synchronize animation components in case the object contained one or more animations. This will make sure both objects continue playing the same animation and at the same time.

Splitter Component

The splitter is extremely simple and it is simply waiting for a trigger event and then it tells the object that triggered it to split if it contains an ISplitable class.

```
private void OnTriggerEnter(Collider other)
{
   MonoBehaviour[] components = other.GetComponents<MonoBehaviour>();
   foreach (MonoBehaviour component in components)
   {
```

```
        ISplitable splitable = component as ISplitable;
        if (splitable != null)
        {
            splitable.Split(transform);
            break;
        }
    }
}
```

Code 2 - Splitter Trigger

Mesh Splitting Classes

The design for my mesh splitting classes has been developed through iterations while developing my algorithms for mesh splitting. This means that the design is not optimal and there are a few areas that could be improved, but I still feel that it's a solid design where the responsibilities is well defined and distributed.

The base class to allow mesh splitting is the Mesh Container class. This class caches mesh data, calculates world space coordinates, and can create new meshes based on a triangle list. Then there's the Mesh Splitter Convex class. This class first splits triangles and then creates a convex cap. To work with concave caps I've created an Edge class that handles all the necessary edge operations that will be needed. Finally there's the Mesh Splitter Concave class.

Mesh Container Class

The mesh container is the class I use to cache mesh data, calculate world space position for its vertices, and create a new mesh based on a triangle list.

I cache the mesh data in static arrays that's an exact copy of Unitys mesh data. When the mesh is being split I will need to add several new vertices to both adjust the split vertices and create the cap. I Use dynamic arrays for these new vertices and make sure their size is big enough so the arrays don't have to be resized during splitting (but still has the option if necessary). I also create two new index lists for the upper and lower triangle lists.

Calculating the world space position for a static mesh is very easy in Unity. The transform component can return a world matrix for the object and then we can just multiply each vertex with this matrix to get its world space coordinates. It's a bit more complicated to calculate world space coordinates for a mesh using skinned animation. The process of this can be seen in Code 3.

```
int boneCount = bones.Length;
Matrix4x4[] boneMatrices = new Matrix4x4[boneCount];
for (int i = 0; i < boneCount; i++)
{
   boneMatrices[i] = bones[i].localToWorldMatrix * bindPoses[i];
}

Matrix4x4 localBone0, localBone1, localBone2, localBone3;
float bw0, bw1, bw2, bw3;
BoneWeight boneWeight;
Matrix4x4 worldMatrix;

int count = wsVerts.Length;
for (int i = 0; i < count; i++)
{
   boneWeight = boneWeights[i];
   // cache weights
   bw0 = boneWeight.weight0;
   bw1 = boneWeight.weight1;
   bw2 = boneWeight.weight2;
   bw3 = boneWeight.weight3;
   // cache matrices
   localBone0 = boneMatrices[boneWeight.boneIndex0];
   localBone1 = boneMatrices[boneWeight.boneIndex1];
   localBone2 = boneMatrices[boneWeight.boneIndex2];
   localBone3 = boneMatrices[boneWeight.boneIndex3];

   for (int k = 0; k < 3; k++)
   {
      for (int j = 0; j < 4; j++)
      {
         int index = k + j * 4;
         worldMatrix[index] =
            localBone0[index] * bw0 +
            localBone1[index] * bw1 +
            localBone2[index] * bw2 +
            localBone3[index] * bw3;
      }
   }

   wsVerts[i] = worldMatrix.MultiplyPoint3x4(wsVerts[i]);
}
```

Code 3 - MeshContainer, World Space Coordinates for Meshes using Skinned Animation

The last responsibility the mesh container has is creating a new mesh using one of its triangle lists.

Explaining this in words is a bit difficult so I'll cover it briefly and then show code parts that can explain it far

better than I can in words. I begin by creating a translation lookup table this will translate old index values to the index value for the new mesh. I set all its values to -1 so I can check if it has already been set. I then fill in the lookup table while I count how many unique indices there is. I then create arrays for the new mesh copy the needed vertices into the new arrays. I then create a new Mesh instance and assign the new arrays to it and return the mesh. The code showing this can be seen in Code 4.

```
private Mesh CreateMesh(List<int> tris)
{
   int triCount = tris.Count;
   int[] localTris = new int[triCount];

   int[] translateIndex = new int[vCount];
   for (int i = 0; i < vertexCount; i++)
      translateIndex[i] = -1;

   int uniIndexCount = 0;
   for (int i = 0; i < triCount; i++)
   {
      int triIndex = tris[i];

      if (translateIndex[triIndex] == -1)
            translateIndex[triIndex] = uniIndexCount++;

      localTris[i] = translateIndex[triIndex];
   }

   […] // create local mesh data arrays

   uniIndexCount = 0;
   for (int i = 0; i < triCount; i++)
   {
      int triIndex = tris[i];
      if (translateIndex[triIndex] >= uniTriCount)
      {
         localVerts[uniTriCount] = vertices[triIndex];
         if (normals != null) localNormals[uniIndexCount] = normals[triIndex];
         if (tangents != null) localTangents[uniIndexCount] = tangents[triIndex];
         if (uv != null) localUv[uniIndexCount] = uv[triIndex];
         if (uv2 != null) localUv2[uniIndexCount] = uv2[triIndex];
         if (colors != null) localColors[uniIndexCount] = colors[triIndex];
         if (boneWeights != null) localBoneWeights[uniIndexCount] = boneWeights[triIndex];
         uniIndexCount++;
      }
   }

   […] // create mesh instance and assign local mesh data to it

   return newMesh;
}
```

Code 4 - MeshContainer, Mesh Creation

Mesh Splitter Convex Class

Mesh Splitter Convex uses the IMeshSplitter interface. This interface contains a method for setting cap properties, splitting the mesh, and for creating a cap. Mesh splitting and cap creation is separated since you don't always need or want to create a cap. The reason for using an interface is that we also need a concave implementation and separating these two divides responsibility better as you would never need both when splitting an object. Each mesh splitter has two responsibilities first it should split the mesh in two, and then it should be able to create a cap for the hull left by the split.

Splitting the mesh consists of two methods; the first one is Mesh Split that gets called by the Splitable component, the other one is Split Triangle that splits the current triangle based on an offset. Mesh Split calculates if the plane intersects the triangle and find out which edges are intersected. It then uses Split Triangle with the offset it has detected. If no split was detected then it calculates what side of the plane the triangle is placed and adds it to that mesh. The Mesh Split method can be seen in Code 5.

```
int triCount = _mesh.triangles.Length - 2;
for (int triOffset = 0; triOffset < triCount; triOffset += 3)
{
   triIndicies[0] = _mesh.triangles[triOffset];
   triIndicies[1] = _mesh.triangles[1 + triOffset];
   triIndicies[2] = _mesh.triangles[2 + triOffset];

   lineLerp[0] = _splitPlane.LineIntersect(_mesh.wsVerts[triIndicies[0]],
                                           _mesh.wsVerts[triIndicies[1]]);
   lineLerp[1] = _splitPlane.LineIntersect(_mesh.wsVerts[triIndicies[1]],
                                           _mesh.wsVerts[triIndicies[2]]);
   lineLerp[2] = _splitPlane.LineIntersect(_mesh.wsVerts[triIndicies[2]],
                                           _mesh.wsVerts[triIndicies[0]]);

   lineHit[0] = lineLerp[0] > 0f && lineLerp[0] < 1f;
   lineHit[1] = lineLerp[1] > 0f && lineLerp[1] < 1f;
   lineHit[2] = lineLerp[2] > 0f && lineLerp[2] < 1f;

   if (lineHit[0] || lineHit[1] || lineHit[2])
   {
      if (!lineHit[2])
         SplitTriangle(0); // tri split at 0 & 1
      else if (!lineHit[0])
         SplitTriangle(1); // tri split at 1 & 2
      else
         SplitTriangle(2); // tri split at 2 & 0
   }
   else
   {   // tri not split
```

```
        if (_splitPlane.PointSide(_mesh.wsVerts[triIndicies[0]]) > 0f)
        {
            […] // add triangle to upper mesh
        }
        else
        {
            […] // add triangle to lower mesh
        }
    }
}
```

Code 5 - Mesh Splitter Convex, Mesh Split

The Split Triangle method uses an offset to run the same piece of code for all 3 variations of the same split. It first adds the linear interpolated vertices and then it adds the indices from these to its cap indices. Now it's just creating the new triangles and tests which side which triangles should be on. The Split Triangle method can be viewed in Code 6.

```
private void SplitTriangle(int offset)
{
    int i0 = offset % 3;
    int i1 = (1 + offset) % 3;
    int i2 = (2 + offset) % 3;

    int indexHit0 = _mesh.AddLerpVertex(triIndicies[i0], triIndicies[i1], lineLerp[i0]);
    int indexHit1 = _mesh.AddLerpVertex(triIndicies[i1], triIndicies[i2], lineLerp[i1]);

    […] // add cap indices

    smallTri[0] = indexHit0;
    smallTri[1] = triIndicies[i1];
    smallTri[2] = indexHit1;

    bigTri[0] = triIndicies[i0];
    bigTri[1] = indexHit0;
    bigTri[2] = indexHit1;
    bigTri[3] = triIndicies[i0];
    bigTri[4] = indexHit1;
    bigTri[5] = triIndicies[i2];

    if (_splitPlane.PointSide(_mesh.wsVerts[triIndicies[i1]]) > 0f)
    {
        […] // add smallTri to upper mesh and bigTri to lower mesh
    }
    else
    {
        […] // add smallTri to upper mesh and bigTri to lower mesh
    }
}
```

Code 6 - Mesh Splitter Convex, Split Triangle

To create the caps for the convex hull I first transform the cap vertices into 2D. To do this I take the inverse split rotation (represented as a quaternion) and multiply it with each cap vertex. I then find the lowest

point so I can calculate angles and sort the rest of the vertices. This lowest point is determined by the Y component, in a situation with two even lowest Y components we would then go with the lowest X component.

To prepare for the sorting we need to calculate the angle from the lowest vertex to each other vertex. I use an up vector and the normalized direction from the lowest vertex to the other vector and calculate the dot product for it. If the X component of the current vertex is less than the X component of the lowest vertex then the angle is 2 minus the dot product instead. This gives us an angle where an up vector would give a value of 1 which equals 90°, a right vector would give a value of 0 which equals 0°, and a left vector would give a value of 2 which equals 180°. I also set the lowest vertex to have an angle value of -1 to make sure it will be sorted first. The angle calculation code can be seen in Code 7.

```
// calculate angles
float[] angles = new int[capCount];
Vector2 sVec = rotVerts[sorted[0]];
for (int i = 1; i < capCount; i++)
{
   Vector2 vec = rotVerts[sorted[i]];
   float dot = Vector2.Dot(Vector2Up, (vec - sVec).normalized);
   if (sVec.x <= vec.x)
      angles[sorted[i]] = dot;
   else
      angles[sorted[i]] = 2f - dot;

   if (angles[sorted[i]] < 0) angles[sorted[i]] = 0;
}
angles[sorted[0]] = -1;
```

Code 7 - Mesh Splitter Convex, Calculating Angles

Now that we have a value we can use to sort the vertices we go ahead and sort them based on the angles we just calculated. I use a gnome sort since this was fast and easy to implement. After the sorting I we have a problem with even numbers in the beginning and in the end. To fix these I sort them but this time using position to sort them. Now that the order is correct I move on to removing redundant vertices. I make a check where for each vertex I take the normalized direction from the previous vertex to the current vertex, and the normalized direction from the current vertex to the next vertex, and calculate the dot product. If the dot product is 1 then I can remove the current vertex.

At this point I now have all the final vertices sorted into a triangle fan. Now it's time to calculate the UV coordinates of they're needed. When that's done all that's left is adding the new vertices and adding the triangle fan. Code 8 shows this process. What should be noted is that I used the inverse rotation for the object and multiply that with the normal of the split plane. This is because we've done all our calculations on a world space representation and we need to transform the normal back into local space for the mesh. The last part of the code also shows the ordering the triangle fan should be in.

```
//create cap verticies
Vector3 normal = Quaternion.Inverse(_ownRotation) * _splitPlane.Normal;
Vector3 invNormal = -normal;
int[] capUpperOrder = new int[capsSorted.Length];
int[] capLowerOrder = new int[capsSorted.Length];

for (int i = 0; i < newCapCount; i++)
{
   capUpperOrder[i] = _mesh.AddCapVertex(capsSorted[i], invNormal, capsUV[i]);
   capLowerOrder[i] = _mesh.AddCapVertex(capsSorted[i], normal, capsUV[i]);
}

int capOrderCount = capUpperOrder.Length;
for (int i = 2; i < capOrderCount; i++)
{
   _mesh.trisUp.Add(capUpperOrder[0]);
   _mesh.trisUp.Add(capUpperOrder[i - 1]);
   _mesh.trisUp.Add(capUpperOrder[i]);

   _mesh.trisDown.Add(capLowerOrder[0]);
   _mesh.trisDown.Add(capLowerOrder[i]);
   _mesh.trisDown.Add(capLowerOrder[i - 1]);
}
```

Code 8 - Mesh Splitter Convex, Adding Cap Vertices and Triangle Fan

This is all the Mesh Splitter Convex does and the rest will be handled by the Splitable and Mesh Container.

Edge Class

The Edge class is a part of the Mesh Splitter Concave class, but I wanted to single it out and talk about it before talking about the Mesh Splitter Concave class. The Edge class keeps track of all the needed information for an edge and also gives us a lot of tools to better handle edges during concave cap creation.

The *Edge* class contains two vertices making up the edge. And it also contains indices pointing to the original cap vertices. To create our triangles later on we'll also need the normal of the edge.

Talking about an edge with a normal is a bit weird. The normal is representing the normal of the triangle we split during the mesh splitting. We could just grab the normal directly from the split vertices, but we risk having a normal that will actually be pointing in the wrong direction. This can easily occur when we interpolate the normal because we only work in 2D and can't guarantee which direction the normal points.

At a later point it will be important to have the correct normal, and know which of the two vertices are the right and the left, but at the current moment we won't be able to tell which is which, so I just store the vertices in a random order and save the normal for later.

When we need the normal we can easily calculate it. The two vertices in the edge are named right and left, so we can make a direction vector from right to left and then rotate it 90 degrees clockwise (around the Z-axis) and then normalize it to give us the normal. My transformed vertices are stored as 2D vectors with X and Y components, this means that we can rotate around the Z-axis to make clockwise or counter-clockwise rotations.

I've also made three utility methods. One that flips the edge so it "points" in the other direction. Flipping an edge is just swapping the right and left vertices and indices and then recalculating the normal. The other method compares the current edge with another edge, testing if both left vertices are equal and both right vertices are equal.

The last method is a basic line-line intersection test; it tests whether the current edge intersects with another edge. The Edge class also contains a static method to melt two edges together and return the result, and another static method taking in two edges and producing a closing edge. The last one is useful for when we have two edges and we want to close them off to create a triangle.

Mesh Splitter Concave Class

The Mesh Splitter Concave class inherits the IMeshSplitter interface just like Mesh Splitter Convex. The implementation of mesh splitting for concave meshes is also similar to splitting a convex mesh. The only

difference is when we add the cap indices. To create concave cap(s) we need the edges, so I take the cap indices and store them in an edge. All the edges are then stored in a dynamic array for cap creation after the mesh splitting has finished. To see how I split meshes you can look under Mesh Splitter Convex Class.

To create the cap(s) for the concave mesh we first need to transform the edges into 2D space, just like we did with the convex mesh. I just multiply the inverse rotation (the rotation is represented by a quaternion) with each vertex in all the edges.

I then proceed to linking all the edges together. This process is fairly complicated so I'll show the code for it in Code 9. Before this code I've filled a double linked list of integers (LinkedList<LinkedList<int>>) called "edge holder". The outer linked list represents a list of borders. The inner linked list represents a list of edges in the border. The integer is the index value of the referenced edge. I've filled the outer list with in such a way that each edge equals one border. I now need to link the edges together to create complete borders. Each time I find a match pair of borders I then link those borders together. I then check if the border is complete and add it to the final border list if it is. This process is continued until no more matches can be found. The linking will end as soon as edge holder is empty or when no more matches can be made.

```
while (edgeHolder.Count > 0)
{
   Vector2 curLastLeft = _edges[currentNode.Value.Last.Value].Left;
   Edge testFirst = _edges[testNode.Value.First.Value];
   bool otherTest = Compare(curLastLeft, testFirst.Right);
   bool sameTest = !otherTest ? Compare(curLastLeft, testFirst.Left) : false;

   if (otherTest || sameTest)
   {
      if (otherTest)
         AttachLinkedList(testNode.Value, currentNode.Value);
      else
         AttachLinkedListFlip(testNode.Value, currentNode.Value);

      edgeHolder.Remove(testNode);
      testNode = currentNode.Next;

      Vector2 curFirstRight = _edges[currentNode.Value.First.Value].Right;
      if (CompareVector2(curLastLeft, curFirstRight))
      {
         LinkedList<int> tmpList = new LinkedList<int>();
         AttachLinkedList(currentNode.Value, tmpList);
```

```
            linkedBorders.AddLast(tmpList);
            edgeHolder.Remove(currentNode);

            if (edgeHolder.Count == 0)
               break;

            currentNode = testNode;
            testNode = currentNode.Next;
         }
      }
      else
      {
         testNode = testNode.Next;
      }

      if (currentNode == testNode)
      {
         if (currentNode == edgeHolder.Last) break;

         currentNode = currentNode.Next;
         testNode = currentNode.Next;
      }
   }
```

Code 9 - Mesh Splitter Concave, Linking Edges

After the linking has ended I test if any uncompleted borders have two or more edges. If that's the case then I create and add a closing edge to close those off and add them to the linked borders list.

Now we have all the borders and we need to make sure that the normals are pointing outwards. To do this I find the edge with the right vertex that has the highest Y component in a border. I then find the neighbour that connects to this right vertex. Now I can use the "inside test" to see if the normal are pointing the right direction. If they point the wrong way then I just flip all edges in that border. This process is repeated far all borders.

If we want to have custom UV's on a per border basis then this is where we should calculate it. In my opinion this gives the best result and this is where I calculate my UV coordinates. This is calculations to create custom UV's are the same as for the convex cap.

The next step is creating all the triangles to cover the borders. I'm going to explain it in short and then include the entire code for it. It's a lot of code but this is the key algorithm for creating a concave cap so I thought it was too important to not be included. The code for the triangle creation can be seen in Code 10.

The algorithm creates triangles to fill a concave hull so I'm running it once for each border in the linked borders list. The algorithm takes two edges and uses the inside test to check the angle between them. If the inner test returns 0 then we have a redundant vertex connecting them, I melt the two edges together and replace them with the new melted edge.

If the inner test is 1 then we have a potential triangle. I create a closing edge and try to see if it matches either the previous or next edge. If there's a match then we can add those three as a triangle and remove them from the border. If there wasn't a match I will then test if the closing edge intersects any of the other edges. If there's an intersection then we can't make a triangle and need to try the other direction. Else if there isn't any intersection then we add the triangle ad remove the edges and replace them with a flipped version of the closing edge.

If the inner test returned 2 then there can't be a triangle between the two edges and I try the other direction. If the direction has been switched twice without creating any triangles then I move on to the next edge since the current edge can't make any triangles at this moment. When only 3 edges is left in the border I exit the loop and add the final triangle.

```
foreach (LinkedList<int> border in linkedBorders)
{
   LinkedListNode<int> curNode = border.First;
   bool cwIsNext = Compare(_edges[curNode.Value].Right, _edges[curNode.Next.Value].Left);
   bool cw = true;
   int insideCheckCount = 0;

   while (border.Count > 3)
   {
      bool xorCWs = cw ^ cwIsNext;
      LinkedListNode<int> testNode = xorCWs ? curNode.Previous : curNode.Next;
      Edge curEdge = _edges[curNode.Value];
      Edge testEdge = _edges[testNode.Value];
      int innerTest = TestInnerSide(curEdge, testEdge, cw);

      if (innerTest == 0) // melt current node and test node
      {
         Edge meltEdge;
         if (cw)
            meltEdge = Edge.MeltEdges(curEdge, testEdge);
         else
            meltEdge = Edge.MeltEdges(testEdge, curEdge);
```

```
            LinkedListNode<int> meltNode = border.AddAfter(curNode, _edges.Count);
            _edges.Add(meltEdge);
            border.Remove(curNode);
            border.Remove(testNode);
            curNode = meltNode; // set our melted node as the current node
            continue;
        }
        else if (innerTest == 1) // test node is inner side
        {
            Edge closingEdge; // make closing edge
            if (cw)
                closingEdge = Edge.CloseEdges(curEdge, testEdge);
            else
                closingEdge = Edge.CloseEdges(testEdge, curEdge);

            LinkedListNode<int> prevNode, nextNode; // find next and previous edges
            if (xorCWs)
            {
                prevNode = curNode.Next;
                nextNode = testNode.Previous;
            }
            else
            {
                prevNode = curNode.Previous;
                nextNode = testNode.Next;
            }

            Edge prevEdge = _edges[prevNode.Value];
            Edge nextEdge = _edges[nextNode.Value];

            LinkedListNode<int> matchingNode = null;
            if (closingEdge.SameVectors(prevEdge))
                matchingNode = prevNode; // previous edge was a match
            else if (closingEdge.SameVectors(nextEdge))
                matchingNode = nextNode; // next edge was a match

            if (matchingNode != null) // if a match was found
            {
                if (cw)
                    AddTriangle(curEdge, testEdge, closingEdge);
                else
                    AddTriangle(curEdge, closingEdge, testEdge);

                border.Remove(curNode);
                border.Remove(testNode);
                curNode = matchingNode.Next; // set new current before removing matching
                border.Remove(matchingNode);
                insideCheckCount = 0;
                continue;
            }

            if (TestNewEdgeIntersect(closingEdge, border)) // Test for edge intersection
            {
                insideCheckCount++;
                cw = !cw; // invert direction
            }
            else // no intersections
            {
                if (cw)
                    AddTriangle(curEdge, testEdge, closingEdge);
```

```
            else
               AddTriangle(curEdge, closingEdge, testEdge);

            LinkedListNode<int> addedNode = border.AddAfter(curNode, _edges.Count);
            closingEdge.Flip();
            _edges.Add(closingEdge);

            border.Remove(curNode);
            border.Remove(testNode);
            curNode = addedNode;
            insideCheckCount = 0;
            continue;
         }
      }
      else
      {
         insideCheckCount++;
         cw = !cw;
      }

      if (insideCheckCount >= 2)
      {
         curNode = cw ^ cwIsNext ? curNode.Previous : curNode.Next;
         insideCheckCount = 0;
      }
   }

   if (border.Count == 3) // Add the last triangle (safety check)
   {
      Edge first = _edges[border.First.Value];
      Edge mid = _edges[border.First.Next.Value];
      Edge last = _edges[border.Last.Value];

      if (cwIsNext)
         AddTriangle(first, mid, last);
      else
         AddTriangle(first, last, mid);
   }
}
```

Code 10 - Mesh Splitter Concave, Triangle Creation

After we have the new triangle list I just create the cap vertices from the index references in the triangle list. Creating these cap vertices is exactly the same as with convex cap creation. Adding the cap indices to the upper and lower meshes is a bit different since we now have a triangle list instead of a triangle fan. Adding a triangle list can be seen in Code 11.

```
for (int i = 2; i < capVertexCount; i += 3)
{
   _mesh.trisUp.Add(capUpperOrder[i - 2]);
   _mesh.trisUp.Add(capUpperOrder[i]);
   _mesh.trisUp.Add(capUpperOrder[i - 1]);

   _mesh.trisDown.Add(capLowerOrder[i - 2]);
```

```
    _mesh.trisDown.Add(capLowerOrder[i - 1]);
    _mesh.trisDown.Add(capLowerOrder[i]);
}
```

Code 11 - Mesh Splitter Concave, Adding Triangles

This concludes the Mesh Splitter Convex and the Splitable and Mesh Container can now create the new objects and meshes for concave cases.

Discussion

This section will cover some important points that didn't fit in either theory or implementation.

Splitter Type

Using a plane to instantly split a mesh is the simplest way to perform mesh splitting. I have therefore focused on this case only, since the objective of this thesis was to create a working mesh splitting algorithm. This type does provide some limitations and I'd like to discuss two of them here. The effect is instant so you won't be able to direct your split through the mesh in real time. Secondly a plane is infinite in size which means that you also won't be able to do partial mesh splitting.

Doing real time mesh splitting would be more complex and performance heavy than my solution. It would require the mesh to be altered every frame where you're cutting, instead of just one single time with an instant mesh split. The interesting part in mesh splitting is when you see the two objects coming apart. If we made a game with real time mesh splitting, then we would also like to see the objects gradually come apart. But while we're cutting the object would still be one single object. To make a single object react to the mesh splitting it would then also require us to have a more complex physics simulation. An example of this type of mesh splitting in use would be cloth simulation with fracturing. In unity this is already supported in their physics system. Using cloth simulation would be fine for small cloth like meshes. But that kind of physics simulation is very too slow to work with game character sized meshes, which easily can be around 5000 triangles.

Partial mesh splitting could still be an instantaneous split, and use the game objects transform for defining its size thus not require altering of the current splitter component. The concave mesh splitter could be extended into handling this kind of scenario. The challenge in this would be finding out how to divide and group the split part into a new object or multiple new objects. This case is something that could be worked on later but is out of the scope of this thesis.

Mesh Splitting Problems

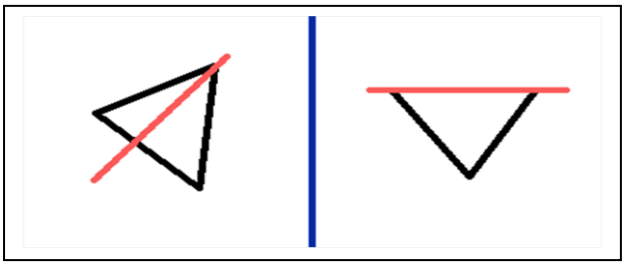

Figure 13 - Mesh Splitting Problems, 2 Problematic Cases.

The method I'm using for mesh splitting works in practice. But to make this airtight there is two issues with the current way I'm doing things. When calculating line intersections I handle two line intersections, but we could have a situation where the plane passes exactly through a vertex and the opposite side. The chances of this happening are very small since the plane would have to be exactly on the vertex, but in theory it could cause some triangles to be split wrong. The other issue is when testing an un-split triangle, I just test the first vertex what side it's on by testing if it's above zero. In the case that it's zero another vertex should be tested since the current vertex is exactly on the splitting plane and a vertex that is not on the splitting plane should define its side instead. Again I haven't had many problems and I only rarely get cap errors, but both these issues are something that should be taken care of in the future to increase the quality of the mesh splitting.

Inner Borders

When splitting concave meshes we risk running into a problematic scenario that produces an unwanted result. This problem arises if we have one border inside another border. An obvious case of this would be a torus split horizontally. The hole in the middle will leave us with a border inside another border. If we moved on to triangle creation with those two borders then we would end up filling the hole in the torus, but only one the side facing the split.

This scenario made me come up with inner and outer borders. The way we have borders at this point makes them all outer borders, and as long as we don't have objects with closed holes inside them then we'll never run into problems. But as soon as we have objects with the same sort of hole as a torus then we should detect it and act accordingly.

In my current implementation I don't handle this case. But I have a theoretical solution for how to handle it. When we detect an inside border then we can flip all its vertices making the normals point inside, instead of outside the border. Then we can create two new edges that connect each end and beginning of the two borders with each other as seen in Figure 16. This will result in one outer border and this is why we wanted to flip the edges first. Having one outer border instead of an outer and inner border fixes the problem.

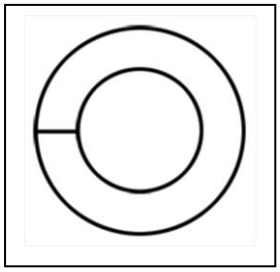

Figure 14 - Outer and Inner Border Connected

Handling one inner border within an outer border is one thing. But if we have several inner borders within the same outer border, then it gets a lot more difficult to connect the inner borders to the outer border without intersecting each other. To add to an already bigger headache we could have an object that have an outer border with an inner border and inside that border there could potentially be an outer border. Detecting this would be crucial to giving the correct result. The problem here is that we need a very complex algorithm to handle all these inner and outer borders which can further contain more inner and outer borders.

The good news is that in general this won't be a problem as most objects that you would want to split will never have the torus structure. We could also make a solution of that supports a single inner border, and not allow outer bounds to be inside inner bounds to limit our complexity.

Evaluation

This thesis works as a sort of middleware so to evaluate it I will look at how it affects performance of a game, how it can be used in game, and what could be done to improve this technology even more. Because the nature of this thesis is procedural this means that it's very hard to give precise benchmarks since every case is different. It will also be very dependant a lot on the end users system so I'm going to talk about these areas with this in mind rather than just giving a table of numbers.

Performance

Having a lot of tables with numbers here wouldn't help mush since the nature of mesh splitting is procedural meaning that every case will be different and heavily depending on the hardware it's run on. I have tried to get a sense of the general performance and the relation between the different kinds of mesh splitting. In the current state I haven't worked on optimizing concave cap creation, so it's very likely that performance can be increased for this part. The rest have been optimized heavily since their first versions.

I think it's important to note that I've developed and tested this thesis on a two year old gamer laptop. It has a 1.6 GHz quad core processor and 6 GB of RAM. The tests I've done is to get a sense of what affects the mesh splitting and to have some guidelines on how to optimize the usage for the project it's used in. My main tool for performance testing has been the profiler in Unity.

The two main things that are interesting as I see it is to define the two ends of our complexity spectrum. A static convex mesh with a low triangle count would be the fastest case. And an animated concave character mesh with a high triangle count would be the slowest (my test model contained about 4000 triangles).

I've tried to do some testing on how many objects I can split simultaneously. With the fast case I ended up with significant slowdowns when I got up to around 50+ objects split. With the slowest case I started having slowdowns when I got to around 10 objects split in the same frame. There are a lot of differences between

these two results such as reason for slowdown, object complexity, static vs. animated, and convex vs. concave cap creation.

As for the slowdown for the fast split case the slowdown is caused by the physics system whereas the slow case is slowed down by the cap creation. The object complexity in itself also plays a huge part since the fast case has is a cube having 12 triangles compared to the 4.000 triangles so processing the slow case will take up a lot more time.

The difference between static and animated meshes is only present during calculation of world space positions. Calculating animated world space coordinates is about 70 times slower than the calculations for static meshes. Even though it's that much slower it's still only a small part of the total time for mesh splitting and this is the only slowdown between animated and static meshes.

With cap creation we don't have a choice if the mesh itself is concave. But we could use concave cap creation for convex meshes if this was faster. I've tried to do a rough performance test of convex vs. concave cap creation for the fast case. Using concave cap creation took a bit more than twice as long as convex cap creation.

The mesh splitting is of course taking up a lot of time during that single frame. But the physics system will spend even longer during the following frames to settle in all the new physics objects created by the mesh splitting. It's important to differentiate between these as we can try to optimize the mesh splitting if that's the problem but the physics system creates a bottleneck that we can't optimize us out of. If the split makes the game freeze until its split and then resume smoothly then the mesh splitting is the cause, but if the game is lagging for a while then the problem is the physics system spending a lot of time over multiple frames.

Uses

The uses for this kind of technology are many and could be applied to numerous games either as a main feature or just as an added touch. This technology is designed for easy implementation and just through Unity it could be built for games on the following platforms: PC, Mac, Browsers (IE, Firefox, Safari, and Chrome) , Andriod, iOS, Wii, Playstation 3, and Xbox 360.

The main thing to do when using this mesh splitting technology would be to write a custom way to use splitters, because this is the way you interact with mesh splitting. When you have the needed Splitter creation then you can just use the Splitable component and that should take care of the splitting the mesh when needed. When deciding on a platform it's very important to design what kind of mesh splitting that would be relevant and then limit amount of simultaneous splits it to make sure the performance doesn't slow down the game.

I feel at this point that the product I've created here could be used to build a game around, and with a bit more polishing and testing it could definitely be sold as middleware for developers to use to create their own game using this technology.

Future Development

There are a few things I would have added if I had had more time for this project. The most pressing task would be to complete the mesh splitting triangle test with the last two cases. With that finished there shouldn't be any cases not covered for mesh splitting and convex caps should always fill the entire surface. At this point I feel it's close to a first version commercial quality.

Another thing I would like to address is the possibility to have the Splitable component check for if a concave mesh has been split into more than two objects and act accordingly. I suspect that if the mesh has a ragdoll like colliders set up for it then these could be used to detect such cases and further subdivide the part that produces multiple meshes. This would require some more setup from the user and I see this as a

nice to have optional thing to give to the developer. The reason why I want to use colliders is because they're a lot faster to check against than testing every triangle to see if they are all connected to the same mesh. The complexity of such a method I would assume is at least $O(n^2)$ and might not be viable for a game that's running real time. I also tried to see if Metal Gear Rising had solved this and from the gameplay footage in the trailer it seems like they don't handle this, and instead they "cheat" by making a lot of splits quickly to minimize this from being visible. This is actually visible in the left picture in Figure 4 where the soldier's right hand had been sliding to the right side of the picture, together with the rest of the upper body. If they had split it correctly, then it would have fallen straight down instead of sliding with the upper body.

The last thing I think could really improve the performance would be a resource manager to handle all Splitables. I imagine such a manager would make sure that only a set amount of Splitables executed their mesh splitting at the same time and then delayed the rest for the following frames. It could even go in and handle their physics objects so all rigid bodies wouldn't be activated at the same time. Having a manager to handle when to split according to available resource could greatly improve performance and allow you to virtually split a lot more objects simultaneously without having any slowdown.

Appendix

Accompanying Disc

The disc will contain the following:

- Report
- Demos
    - Gladiator Tailes
    - Tech Demo
- Related Work
    - Fruit Ninja Trailer
    - Metal Gear Rising (Slicing gameplay + Trailer)