Tech-Art - Lyra Recherche et développement GP

Loïc BRAUD – Antoine FONTVERNE

Décembre 2025 – Janvier 2026

Sommaire

Contents

1 Introduction

1.1 Présentation du projet

Ce document explique la démarche ainsi que la recherche technique pour l'implémentation et la mise en place d'un système de dégâts et de découpe dynamique de meshes. Nous aborderons des notions en rapport avec les materials, les compute shaders, l'écriture dans des textures en runtime ainsi que des algorithmes de mathématiques 3D. Vous trouverez aussi d'autres techniques que nous avons expérimentées mais auxquelles nous n'avons pas donné suite.

Ce projet a été réalisé dans Unreal Engine 5.6, en étroite collaboration entre une équipe de deux Game Programmers : Loic BRAUD et Antoine FONTVERNE ainsi qu'une equipe de Game Artists Eliott CHU, Ana GOERRIAN, Gabriel HAMCHA et Lou LEBOUCHER.

Ce document détaille de nombreuses techniques et recherches de bas niveau. Bien que de nombreuses illustrations soient disponibles, une image ne vaut pas toujours mille mots. Nous nous excusons si le document peut sembler dense ou lourd à la lecture.

Bonne lecture !

2 Dégats dynamiques

2.1 Volonté

Notre volonté pour le système de dégâts était de développer un système se basant sur celui de Dead Island 2, c'est-à-dire quelque chose de dynamique et modulable. Le système F.L.E.S.H utilise des splatter maps ou textures de dégâts pour enregistrer des informations et permettre au mesh qui lit cette texture de représenter les dégâts subis. Pour cela, il faut dans un premier temps déplier les UVs du mesh dans une texture, puis écrire une brush de dégâts au bon emplacement dans cette dernière, puis envoyer au mesh la splatter map.

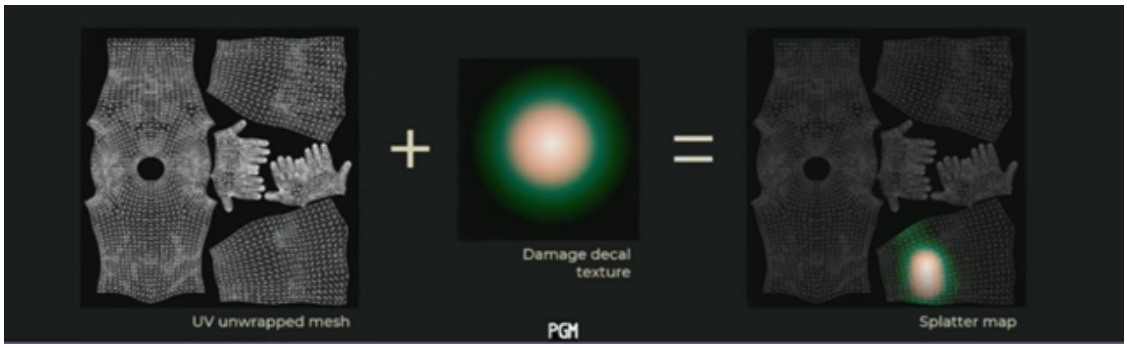

Figure 1: Splatter map et brush de dégât du système F.L.E.S.H

2.2 Appliquer une brush de dégât

Nous avons interprété que les splatter maps étaient des render targets qui étaient liées au material du zombie lors de la création de l'instance du material. Or, pour écrire dans une render target, il existe 2 solutions : la node Draw Material to Render Target et le component Scène Capture 2D. Après plusieurs tests avec la première node, on ne pouvait pas indiquer le mesh qui devait être rendu par le material, la node draw material était donc inutilisable, car elle n'utilisait pas les UVs du mesh. Nous nous sommes donc penchés sur le component de capture. Ce component possède une fonction "Capture Scene" qui permet de prendre en photo la scène avec une caméra fictive et d'écrire le résultat dans la render target.

Notre première approche utilisait la node Find Collision UV : elle permet de donner les coordonnées UV d'un mesh touché sur Hit. Or, nos tests se portaient, dans un premier temps, sur le mesh de base d'Unreal, un skeletal mesh. Et après recherche, la node renvoyait toujours un Vector2 nul, car elle ne fonctionnait pas avec ce type de mesh. Nous avons donc écarté cette hypothèse, car nous voulions un système qui s'applique à n'importe quel type d'objet. De plus, avec cette coordonnée UV, nous aurions eu le centre du decal, mais pour ensuite sampler la brush sur la surface, la tache aurait été difficile et demandait de la recherche sur la manipulation des UVs et autres.

Figure 2: Node blueprint qui récupère les UVs grâce au hit

Nous nous sommes ensuite penchés sur les decals internes d'Unreal. L'avantage est qu'ils s'appliquent très bien sur les surfaces, sont faciles à faire apparaître sur un point d'impact et donnent un résultat plutôt satisfaisant sur l'empilement et sur un mesh animé. En modifiant quelques paramètres dans le material pour limiter le stretch sur l'application de surface, tels qu'un character, nous pouvions réfléchir à comment écrire les informations dans une splatter map. Cependant, l'utilisation de la capture du component pour écrire dans une splatter map ne pouvait pas fonctionner avec l'approche des decals, car elle ne représentait pas une texture des UVs dépliés. Une autre piste était de sampler les decals grâce à une node dans les materials nommée DBuffer. On peut récupérer des informations telles que la Base Color, les normales, la roughness et autres. Or, lors du passage à la représentation en enfonçant les vertex grâce au World Position Offset (WPO), en multipliant la valeur de profondeur stockée dans les decals par les normales des vertex, une erreur interne d'Unreal empêche la compilation. Nous n'avons pas réussi à trouver de solution à ce problème même en demandant à nos professeurs. Nous avons donc exploré une nouvelle piste, qui sera celle utilisée pour le reste du projet.

Figure 3: Test avec les decals d'Unreal et lecture de leur data

Cette troisième technique consiste à créer un material qui permet de déplier les UVs en world space, puis de continuer sur la volonté d'utiliser les render targets, en capturant avec la Scène Capture 2D la projection des UVs en world space, avec l'application entre-temps de la brush de dégât.

Figure 4: Capture du material d'unwrap, avec application d'une brush, via le component de Scène Capture 2D

Dans un premier temps, lors de nos recherches, on ne montrait pas comment appliquer une brush personnalisée dans le material, ils utilisaient la node Sphere Mask.

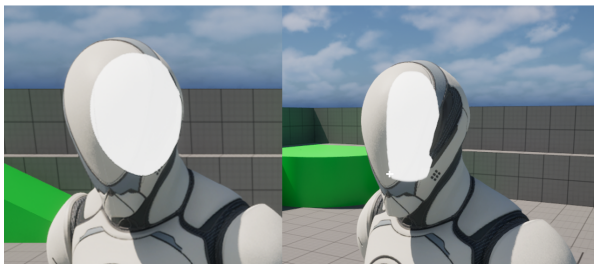

Figure 5: Différence entre l'utilisation du sphère mask d'unreal et un masque circulaire personnalisé

Nous avons ensuite réussi à appliquer un motif en particulier, en manipulant les vecteurs directeurs des UVs ainsi que des opérations pour gérer le radius de l'impact.

Figure 6: Représentation de la brush avec correction des UVs

Figure 7: Application de la splatter map avec le decal étoile

Figure 8: Application de la splatter map avec un decal circulaire

2.3 Représentation des dégâts

Dans le système F.L.E.S.H, les brush de dégâts enregistre des informations dans chaque canal RGBA. Lors de ce projet nous avons eu le temps de nous pencher uniquement sur le renfoncement des blessures. Pour améliorer le rendu de l'enfoncement on a d'abord créer une brush de dégât personnalisé pour donné la forme voulu puis on a appliqué un dégradé sur l'alpha afin de creuser de façon smooth le mesh.

Figure 9: Brush de dégât avec dégradé

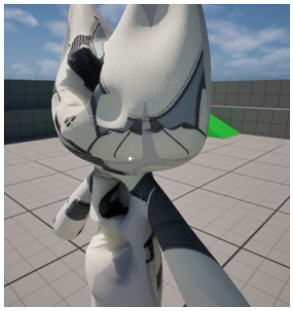

Figure 10: Résultat lors de l'application de la brush dégradée

Nous avons ensuite voulu essayer la tessellation au lieu du WPO pour modifier la position des vertex. Or, en activant Nanite pour la tessellation, nous avons eu des problèmes avec notre système précédemment implémenté, malgré le résultat convaincant en utilisant des valeurs par défaut sans splatter map. Nanite modifie les vertex de base du mesh ainsi que ses UVs.

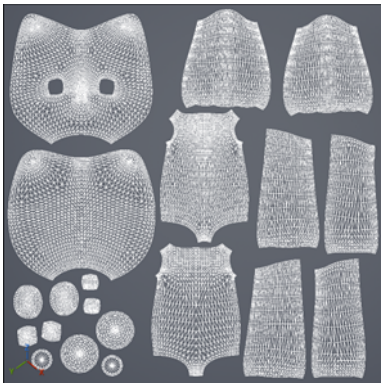

Figure 11: UVs dépliés du mesh de base

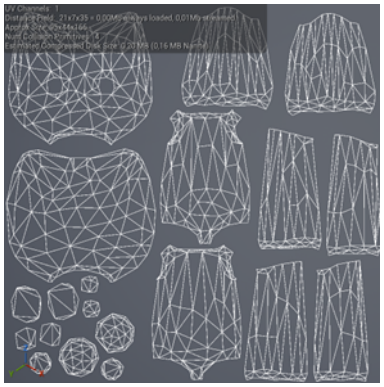

Figure 12: UVs dépliés du mesh avec Nanite

Figure 13: Artefacts lors de l'unwrap

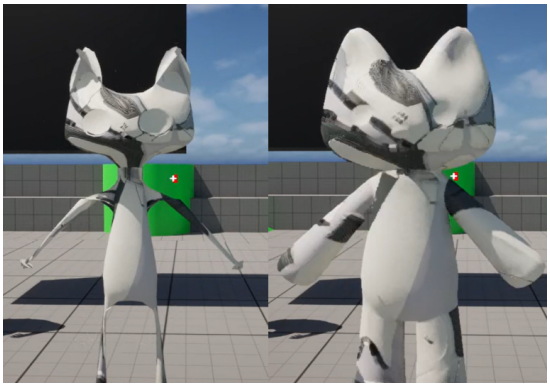

Figure 14: Résultat de la tessellation

Nous sommes donc revenus sur le WPO, mais en ajoutant une texture qui décrit en nuances de gris à quel point on peut creuser dans le mesh. Dans notre source, ils utilisaient une morph target, or nous n'avons pas eu le temps d'implémenter ce système, et la texture d'épaisseur donnait un résultat qui nous convenait.

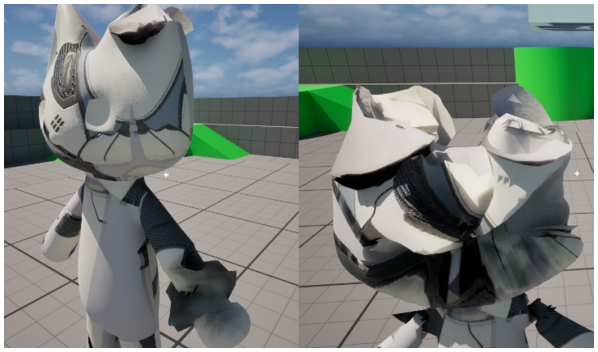

Figure 15: Problèmes avec les extrémités

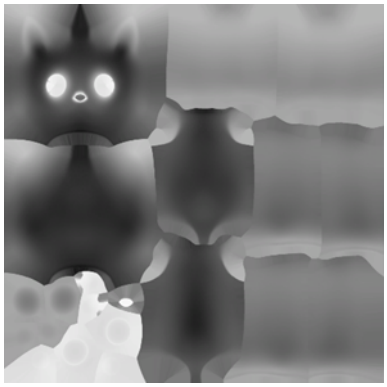

Figure 16: Map d'épaisseur

Figure 17: Résultat avec application de la map d'épaisseur

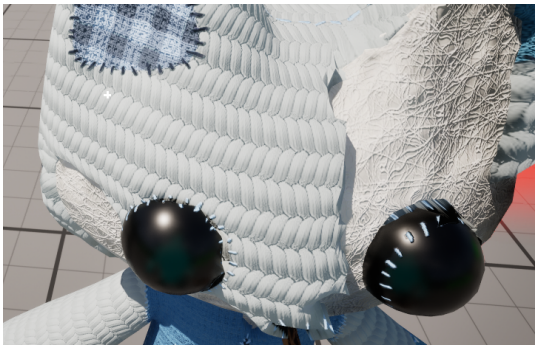

Figure 18: Résultat peluche

Figure 19: Résultat cookie

3 Découpe dynamique

3.1 Volonté

Les premières envies pour la découpe de mesh étaient d'avoir un outil d'abord capable de couper n'importe quel mesh à l'aide d'un plan. Nous envisagions d'avoir une seconde version permettant de découper le mesh avec des formes géométriques plus complexes. Cette idée fut écartée après avoir vu l'envergure que risquait de prendre la première version du système.

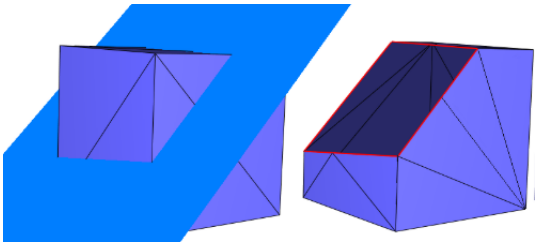

Figure 20: Exemple de coupe avec un plan

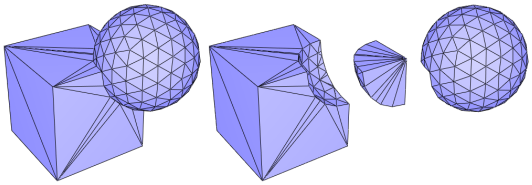

Figure 21: Exemple de coupe avec un autre mesh

Nous savions que de tels outils existaient déjà dans Unreal Engine. Cependant, notre objectif était de comprendre comment pouvoir faire un outil similaire à celui déjà présent. De plus, celui-ci pouvait ainsi faire office de comparaison.

L'un des premiers objectifs avec le système de découpe était qu'il soit rapide. En regardant la conférence sur le système de Dead Island 2, nous avons décidé de partir sur des compute shaders comme eux. Cette option nous permet de profiter du parallélisme de la carte graphique et nous permet aussi de découvrir une nouvelle technologie que nous n'avions jamais utilisée auparavant.

Nous tenons à préciser que notre solution n'a pas bénéficié de beaucoup d'itérations, celle-ci étant relativement bien documentée et ayant marché du premier coup.

3.2 Outil

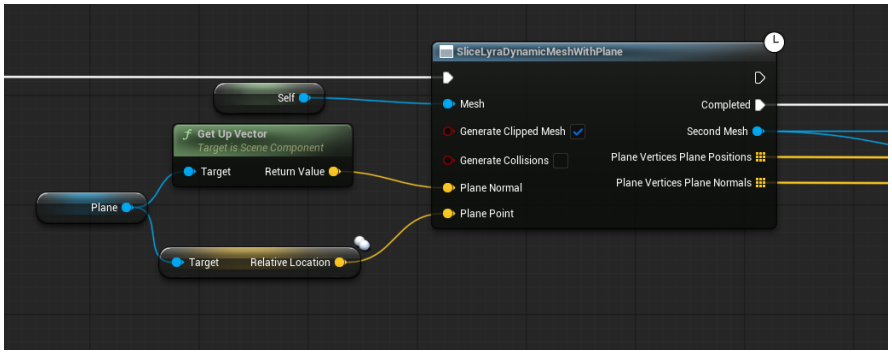

Figure 22: Screenshot de notre outil de clipping

Voilà l'outil mis en place. Très simple, il prend en entrée le Mesh à découper, un point et la normale du plan. Les informations du plan sont converties dans l'espace local du mesh (Espace dans lequel tous les calculs ont lieu). Il est possible de demander ou non la seconde partie du mesh, ce qui simplifie ou non le shader (Moins de buffer d'entrée et de sortie) et s'il faut régénérer les collisions du mesh (Cette partie est entièrement gérée par le mesh du moteur). En sortie, il donne le mesh nouveau créé s'il y en a un et la liste des points et des normales (En world space) du mesh sur le plan. (Ils devaient être utilisés pour la mise en place d'une solution abandonnée)

3.3 Clipping

Afin de couper le mesh, nous appliquons un simple algorithme de clipping sur celui-ci.

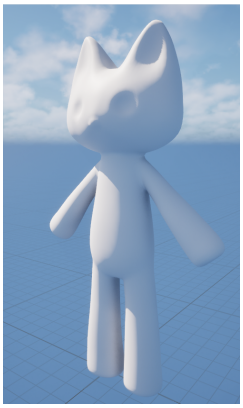

Figure 23: Exemple d'un mesh

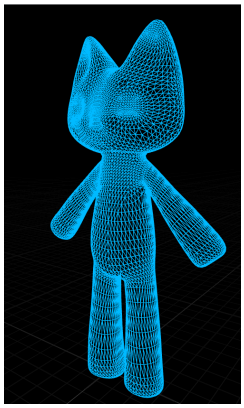

Figure 24: Exemple de la consitution d'un mesh

L'algorithme consiste à seulement couper un triangle à l'aide d'un plan. Avec l'aide du compute shader, chaque thread de la carte graphique s'occupe d'un triangle à la fois.

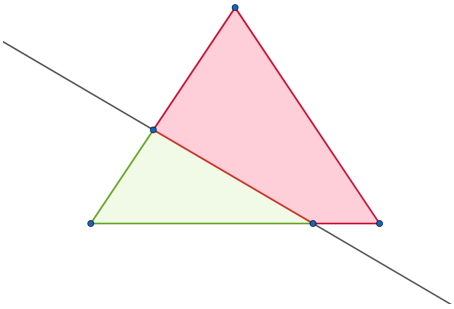

Figure 25: Premier résultat de coupe du triangle

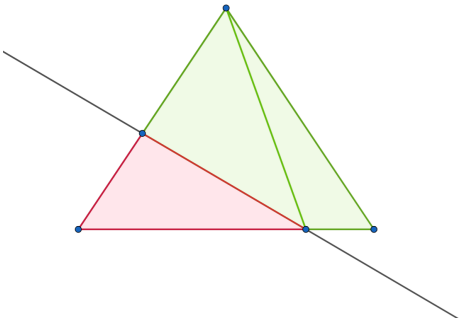

Figure 26: Deuxième résultat de coupe du triangle

Afin de mettre en place la découpe, nous avons juste suivi deux tutoriels différents que nous avons combinés pour effectuer notre solution. Ce qui est important à savoir, c'est que lorsque l'on coupe un triangle, il nous donne soit un nouveau triangle, soit un parallélogramme qu'il faut rediviser en deux triangles. (Voir au-dessus)

Afin de savoir si un plan coupe un triangle, il faut déjà définir ce qu'est un plan :

$$N \cdot X - c = 0$$

N est la normale au plan, c un scalaire et X Un point quelconque sur le plan. Il est possible de savoir facilement où se trouve un point par rapport au plan grâce à cette formule. Pour n'importe quel point Y, ce point est du côté positif du plan si :

$$N \cdot Y - c > 0$$

Du côté négatif du plan, si :

$$N \cdot Y - c < 0$$

Sur le plan si :

$$N \cdot Y - c = 0$$

En se basant sur cette formule et ces propriétés, il est très simple de mettre en place notre algorithme. Celui-ci se déroule en trois grandes étapes.

La première étape est le calcul des distances signées. Il y a une par point. Elles représentent la situation du point par rapport au plan. Comme vu précédemment par les trois propriétés, elles peuvent être du côté positif ou négatif du plan ou bien sur celui-ci. Le calcul que nous utilisons dans le code est une version modifiée de la précédente formule :

$$d_S = N \cdot (Y - X)$$

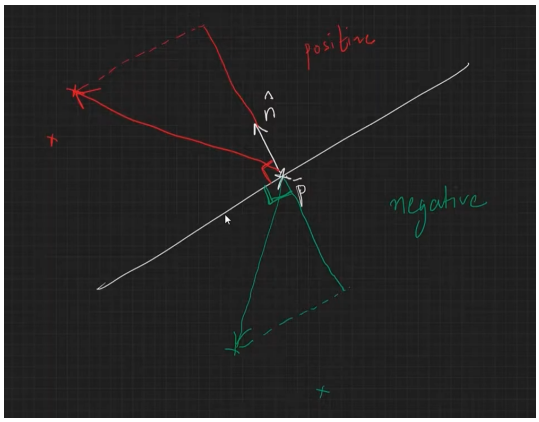

Figure 27: Schéma des distances signées

Une fois les distances signées il est possible de récupérer les points d'intersection. Pour savoir si une des arêtes du triangle est intersecté il suffit de multiplier les distances 2 par 2 pour chaque arête. Si le résultat est positif alors les arrêts sont du même côté du plan, ce qui signifie qu'il n'y a pas d'intersection. Dans le cas d'une intersection, il suffit de faire une interpolation linéaire des deux points :

$$t = d_{s1}/(d_{s1} - d_{s0})$$

$$V_I = V_0 * t + V_1 * (1 - t)$$

V_I correspond au point interpolé entre V_0 et V_1. d_{s0} et d_{s1} sont les deux distances signées correspondant à l'arête traitée.

La dernière étape consiste à recréer les triangles. Pour ça, nous devons savoir comment le triangle est coupé et comment le reconstruire. Il existe 8 possibilités de coupe pour le triangle. Chaque cas utilise des indices et des intersections particulières.
Afin de savoir dans lequel des 8 cas notre triangle se trouve, nous allons former un masque de triangle. Nous créons ce masque grâce à des déplacements binaires. Si une distance signée est négative, on déplace un bit dans le masque pour chaque sommet, ce qui donne la configuration du triangle (allant de 0 à 7).

Une fois la configuration indentifiée il suffit de reconstruire le triangle à l'aide des indices et des intersections du triangle actuellement traité.

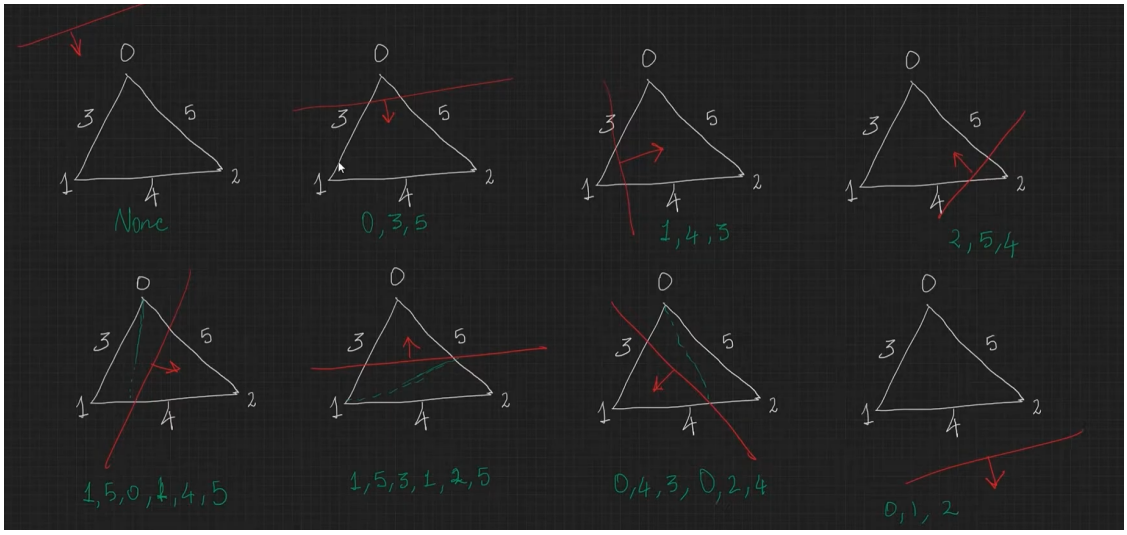

Figure 28: Schéma des 8 différents cas de coupe du triangle

3.4 Capping

Le capping ou la triangulation du mesh était un de nos objectifs pour ce projet. Le but de cette méthode est de re-former la géométrie d'une section ouverte d'un mesh. En effet, après l'application du clipping de notre mesh, nous sommes laissés avec un trou béant à l'intersection du plan.

Figure 29: Notre rendu sans capping

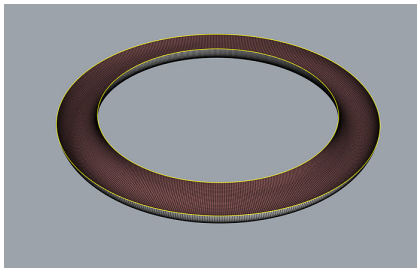

Figure 30: Exemple de mesh coupé

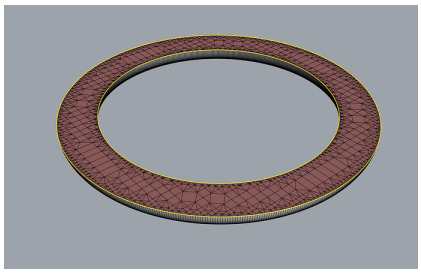

Figure 31: Exemple de géométrie après capping

Il y avait deux solutions que nous souhaitions explorer, mais que nous n'avons pas pu implémenter par manque de temps. La première est l'Ear Clipping, une approche simple pour générer de la géométrie. Cependant, cette méthode est majoritairement séquentielle, chaque nouveau triangle sert aux étapes suivantes.

Étant donné notre utilisation des compute shaders, nous recherchions une approche parallélisable. La triangulation de Delaunay répond mieux à cette contrainte, car il est possible de créer des sous-ensembles de points indépendants, de les trianguler en parallèle, puis de les fusionner.

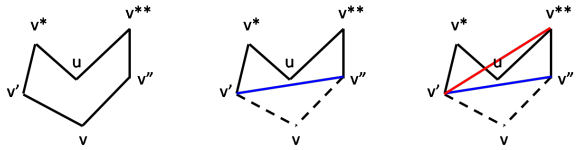

Figure 32: Ear Clipping

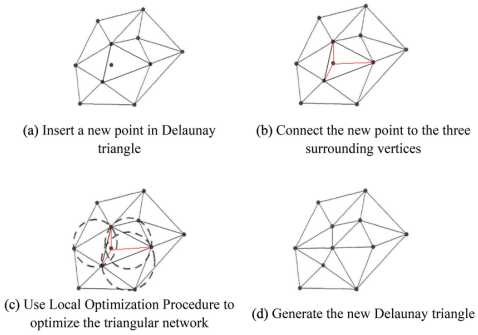

Figure 33: Triangulation de Delaunay

3.5 Compute Shader

Comme expliqué auparavant, tout notre algorithme est exécuté sur un compute shader. La mise en place en elle-même du compute shader ne fut pas compliquée et n'a pas posé de problème. En revanche, nous avons fait face à un souci concernant l'allocation des buffers de sortie du compute shader. Nous envoyons en entrée plusieurs informations au compute shader (Vertices et indices). Le problème est que nous ne savons pas à la création du shader combien de vertices ou d'indices vont sortir du compute shader après la coupe du mesh.

La première solution à laquelle nous avons pensé est de réserver un buffer de sortie de la même taille que le buffer d'entrée. Le souci est que la coupe d'un triangle peut en créer un de plus au maximum. Si la géométrie est mauvaise, nous risquons de dépasser en mémoire.

La deuxième solution que nous avons adoptée pendant un moment est de doubler ou d'augmenter le buffer d'une marge fixe par rapport à celui d'entrée. La limite de cette solution réside dans une utilisation mémoire qui devient trop importante. Un seul mesh high poly pourrait faire exploser l'utilisation mémoire. Le risque est aussi que cet espace mémoire va souvent être gâché et vide.

La dernière solution que nous avons implémentée est un peu plus lourde mais permet de pallier les deux dernières solutions. Nous décidons de pré-calculer la taille requise du buffer. Nous avons construit une succession de compute shader. Le premier pré-calcule les tailles des buffers requis, puis nous exécutons le deuxième shader pour couper le mesh.
Nous avons fait des recherches afin de bien comprendre comment utiliser le RHI (Render Hardware Interfece) et le RDG (Render Dependency Graph) d'Unreal. Cela nous a permis de créer des "pipelines" de compute shaders. Nous voulions aussi, comme expliqué précédemment, enchaîner un troisième compte shader générant le capping.

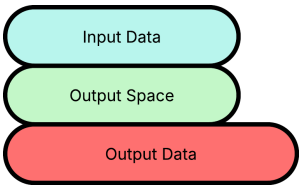

Figure 34: Première solution (Espace dépassé)

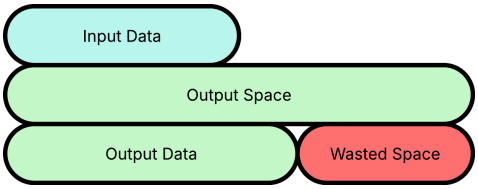

Figure 35: Deuxième solution (Espace gâché)

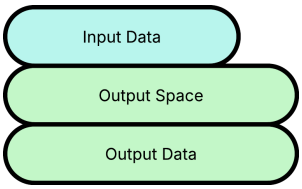

Figure 36: Troisième solution (Espace exact)

Il est toutefois possible de noter une probable perte de performance dans notre shader. Ne pouvant prédire combien de triangles une exécution va sortir, nous devons synchroniser un compteur pour savoir où écrire notre triangle. Cette synchronisation prend du temps et peut ralentir notre écriture. Une solution pourrait être de réserver deux emplacements mémoires dans le cas où un deuxième triangle est écrit, évitant ainsi la synchronisation mais prenant plus de place.

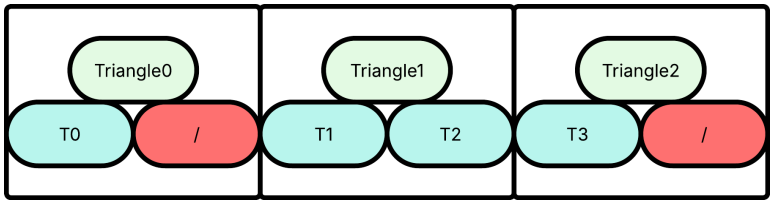

Figure 37: Solution de sortie de mesh

La dernière limitation de notre approche est le readback. Le readback est la lecture des données calculées sur la carte graphique sur le processeur. C'est une opération qui prend malheureusement beaucoup de temps afin de transférer ces données d'un composant à l'autre.

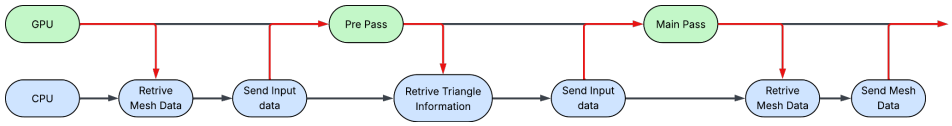

Figure 38: Première version des échanges GPU - CPU

Ci-dessus était la première version de notre pipeline de compute shaders avec beaucoup de readback.

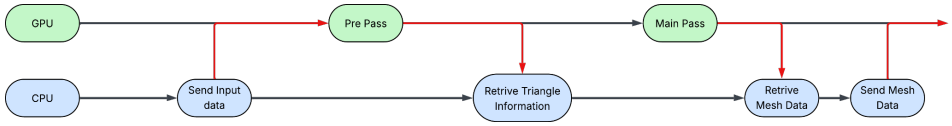

Figure 39: Deuxième version des échanges GPU - CPU

Nous avons réussi à réduire les interactions en envoyant une seule fois des données partagées entre les deux shaders sur la carte graphique.

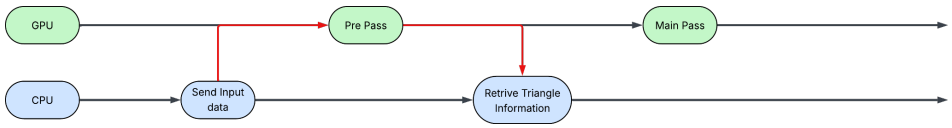

Figure 40: Hypothèse de version des échanges GPU - CPU

Une forme d'optimisation serait d'arriver à trouver un moyen de récupérer les données du vertex shader directement depuis la carte graphique et de les mettre à jour de la même manière. Si une telle option est possible, nous pourrions arriver à cette optimisation ci-dessus.

Au vu du choix du compute shader, voici quelques benchmarks que nous avons effectués entre l'outil du procédural mesh component d'Unreal (CPU only, single thread) et notre version (GPU only, multi threads). À noter ici que notre benchmark ne prend pas en compte le temps de readback entre le GPU et le CPU, ce qui le rend un peu plus lent que ce qui en ressort sur le benchmark, mais reste toujours plus efficace que la version du moteur.

Figure 41: Notre clipper 15K triangles

Figure 42: Unreal clipper 15K triangles

Sur 15 000 triangles, nous sommes 48 fois plus rapide que le moteur. 33.8 microsecondes contre 1.6 millisecondes.

Figure 43: Notre clipper 2M triangles

Figure 44: Unreal clipper 2M triangles

Sur 2 000 000 triangles nous sommes 4800 fois plus rapide que le moteur. 1.5 millisecondes contre 7 secondes.
De manière logique, plus nous avons de triangles à traiter, plus le multithreading nous est bénéfique et efficace en notre faveur, contre le seul thread alloué pour cet outil par le moteur.

4 Reconstruction du mesh

4.1 Volonté

La reconstruction a pour but de pouvoir réassembler le mesh dans un format interprétable et pouvant être rendu à l'écran par le moteur. Notre shader nous permet d'obtenir un ou deux meshes selon la section et le choix de l'utilisateur, mais le moteur ne sait pas comment le lire.

Cette section se divise en deux parties. D'abord, nous parlerons de la représentation du mesh en mémoire dans le moteur au plus bas niveau. Nous verrons ensuite comment cette reconstitution marche au plus haut niveau du moteur pour l'utilisateur.

4.2 Mesh - Bas niveau

Le but de notre projet était de faire des dégâts et une découpe dynamique sur le mesh. Même si c'est une solution, nous ne voulions pas faire notre propre format de mesh dans le moteur. Nous avons privilégié utiliser des outils préexistants.

Un point important à mentionner : aucun de nos meshes n'utilise Nanite d'Unreal Engine. Nous avons choisi de travailler avec ce que nos artistes nous ont fourni, sans ajustement de niveau de détail.

4.2.1 Static Mesh

La première solution que nous envisagions était le Static Mesh Component. Vu que celui-ci ne change pas en mémoire, il utilise une pass de rendu rapide. Cependant, il devient compliqué de modifier ses données en exécution.

Nous pouvions le reconstruire grâce à la fonction **UStaticMesh::BuildFromMeshDescriptions()**. Le problème est que construire une statique mesh est long et pas adapté pour du runtime. (C'est ce qui est fait quand le moteur charge)

4.2.2 Dynamic Mesh

La deuxième solution était le Dynamic Mesh Component. C'était un très bon choix. Il est plus lent en rendu car son buffer est update pour envoyer les changements. Cependant, c'est un mesh component fait pour être modifiable.

Plusieurs points nous ont bloqués. Premièrement, le mesh est d'abord pensé pour les outils éditeurs du moteur. Certaines des fonctions que nous utilisions ne pouvaient pas marcher en build. La création du mesh ne prenait pas facilement certains paramètres comme les tangentes ou plus d'un UV channel. Nous avions aussi des problèmes de rendu avec. Celui-ci apparaissait plus sombre que les autres meshes. Dernièrement celui-ci dépendait d'un plugin externe au moteur de base. Nous voulions éviter d'avoir une dépendance supplémentaire.

4.2.3 Procedural Mesh

La dernière solution étudiée et qui fut gardée est le Procedural Mesh Component. Similaire au dynamic mesh mais sans ses points négatifs. Prend beaucoup de paramètres à la création, est pensé entièrement pour le runtime, est de base dans le moteur et n'a aucun problème de rendu.

Nous l'avons aussi pris car : Unreal Engine propose déjà des fonctions de découpe de mesh et de copie de static mesh sur le Procedural Mesh Component, deux fonctions que nous avons donc recréées. En cas d'abandon ou d'échec de notre part, cela nous permettait de nous rabattre sur une solution préfaite. Il nous était aussi facile de benchmarker les performances si nécessaire.

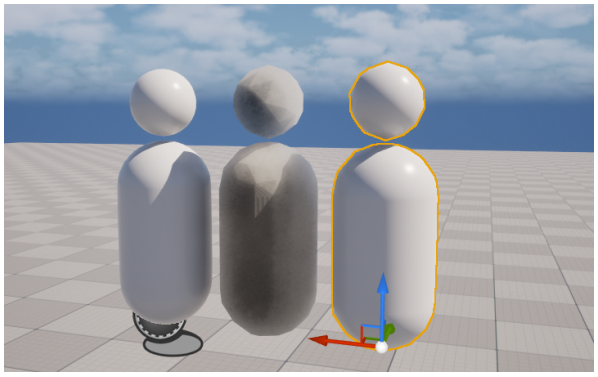

Figure 45: Différences entre les trois meshs : Static, Dynamic, Procedural

4.2.4 Skeletal Mesh

Un de nos objectifs était aussi le Skeletal Mesh. Nous voulions pouvoir découper des mesh animés. Nous avons décidé de ne pas nous attarder plus longtemps sur ce problème à partir du moment où reconstruire un simple mesh sans animation nous posait des problèmes.

Pendant nos recherches, nous avons constaté que certaines personnes avaient déjà tenté d'effectuer des tentatives similaires à nos intentions sans grande réussite. Comme évoqué au début de la section, la solution la plus propice serait sûrement de recréer une structure en interne beaucoup plus bas niveau nous permettant d'influer sur ce type de mesh.

Un autre problème auquel nous avons pensé est la question du skinning. Quand nous découpons un mesh non animé, nous transformons le plan dans l'espace local du mesh. Cependant, pour un mesh animé, si nous faisons cette méthode, nous obtiendrons un mesh dans une poisition par défaut. Une solution que nous avons imaginée serait d'appliquer le skinning dans notre shader, calculer l'intersection et d'enlever le skinning une nouvelle fois pour obtenir l'intersection sur le mesh local. Ce n'est bien sûr qu'une hypothèse et rien de tout ça n'a été testé.

Figure 46: Idée de solution pour le Skeletal Mesh

4.3 Mesh - Haut niveau

Chaque mesh découpable conserve une référence vers le blueprint au dessus qui agit comme manager. Lors d'une découpe, le mesh à séparer notifie, à la fin de l'exécution de ses compute shaders, que la seconde moitié du mesh a bien été générée, et fournit une référence vers le nouvel objet. Une fois que l'ensemble des meshes du blueprint concernés par la découpe ont été traités, les différentes parties sont recombinées dans un blueprint du même type que le blueprint manager initial. À l'exception de certains cas particuliers comme le blueprint du chat en peluche, qui contient un squelette en métal. Dans ce cas précis, la découpe ne génère pas un `BP_Cat_Yarn_Sliceable`, afin d'éviter la duplication du squelette, mais un `BP_LyraSliceableMesh_Ext_Int`, qui correspond à la classe parente de ce blueprint. Une fois cette étape terminée, les collisions sont activées et la simulation physique est lancée. Enfin, le nouveau blueprint manager partage les même splatter maps que le blueprint d'origine, ce qui permet à la fois d'économiser de la mémoire et de conserver les dégâts déjà appliqués.

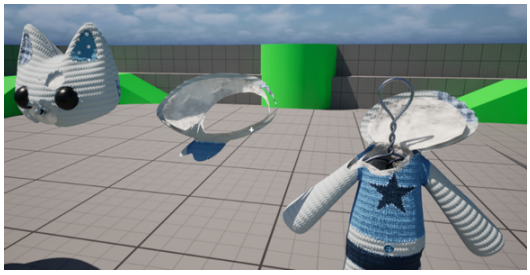

Figure 47: Découpage multiple et démonstration BP_Cat_Yarn_Sliceable

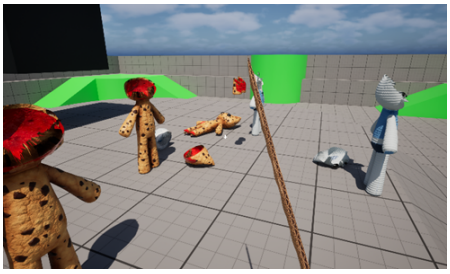

Figure 48: Activation des collisions et de la simulation physique

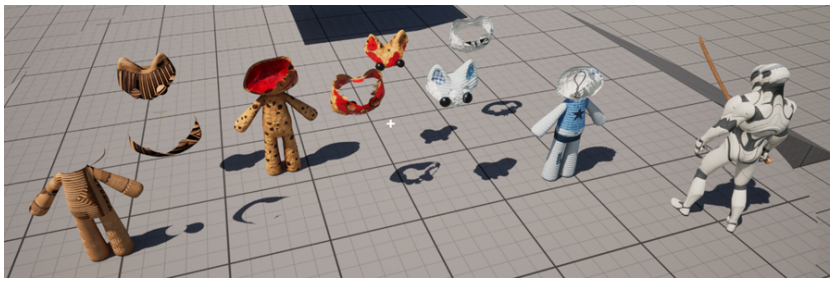

Figure 49: Exemple de découpe

À la suite de la découpe, nous avons voulu augmenter le détail en appliquant sur tout le tour de la découpe une brush de dégâts. Or, le système ne pouvait pas effectuer cette tâche sur 300 points voire plus sans que les performances soient impactées. Une solution aurait été que le material dégrade tous les vertex à une certaine distance du plan de découpe, toutefois, le temps nous a empêchés de mettre cette solution en place.

Figure 50: Dégâts sur le tour de la coupure

5 Conclusion

Nous avons beaucoup appris lors de ce projet, que ce soit au niveau technique, avec la manipulation approfondie des decals, des materials, ainsi que des render targets pour la partie des dégâts dynamiques. Pour la découpe, nous avons commencé à plonger dans le monde des compute shaders, avec leur mise en place dans Unreal, l'utilisation du RHI et du RDG. À cela s'ajoutent les recherches effectuées au bas niveau sur tous les types de mesh components, ainsi que des algorithmes mathématiques 3D pour le clipping et le capping. Mais ce projet nous a aussi apporté une expérience très enrichissante, de par la collaboration avec des artistes.

5.1 Documentation

Ici se trouve la documentation utilisée pour mettre en place les différents objectifs du projet.

Principale inspiration :
- The Innards of F.L.E.S.H: Dead Island 2's Gore System Dissected

Dégâts dynamiques :
- Série de vidéos pour utiliser les decals Unreal
- Tuto sur l'utilisation des decals Unreal et la récupération de leurs données
- Tuto sur les dégâts sur un personnage et la technique de dépliement d'UV
- Tuto sur la tessellation et d'autres techniques de déformation

Découpe dynamique :
- Clipping a Mesh Against a Plane
- Mesh-Plane Clipping — GeomAlgoLib

Meshs dynamiques :
- Mesh Generation and Editing at Runtime in UE4.26
- Dynamic Mesh Component
- UProceduralMeshComponent

Compute Shaders :
- Simple compute shader with CPU readback
- Modern OpenGL Tutorial - Compute Shaders
- Coding Adventure: Compute Shaders
- Getting Started with Compute Shaders in Unity

Triangulation / Capping :
- Triangulation by Ear Clipping
- Polygon Triangulation [1] - Overview of Ear Clipping
- Visualizing Delaunay Triangulation
- Delaunay Triangulation