Université Paris-VIII

Master Création Numérique

Parcours : *Arts et Technologies de l'Image Virtuelle*

Créer une fracture d’objet solide en temps réel tout en gardant la maîtrise artistique.

Swann Martinez

Encadré par

Chu Yin Chen, Professeur d’Université, Paris-VIII

Mémoire de Master 2, 2016 - 2017

« Tout acte de création est d'abord un acte de destruction ».

Pablo Picasso.

Remerciements

J'adresse mes remerciements aux personnes qui m'ont aidé dans la réalisation de ce mémoire. En premier lieu, je remercie Mme Chu Yin Chen, professeur à l'université de Paris 8. et directrice de mémoire dont l'aide a été précieuse dans mon travail. Je voudrais aussi remercier l'ensemble des membres de l'équipe INREV pour leurs précieux conseils.

Je remercie aussi mes amis ainsi que ma famille pour avoir relu ce manuscrit en quête des fautes d'orthographe, mais aussi pour leur soutient.

Résumé

La fracture d'objets solides est essentielle à toute interaction physique vraisemblable. Aujourd'hui, les destructions en temps réel sont majoritairement scriptées : elles produisent toujours le même comportement et les mêmes formes. Cette méthode nuit énormément à l'immersion du joueur par sa redondance.

Ce mémoire propose de structurer notre méthode de destruction d'objets solides sur une approche entièrement procédurale, donnant ainsi au joueur une interactivité plus poussée avec son environnement.

On vise ici à améliorer la destruction d'objet solide en temps réel tout en permettant aux concepteurs de conserver le contrôle de l'effet. Dans cette perspective, on expérimente un pipeline de destruction tridimensionnelle complet au travers de la conception d'un outil de fracture en temps réel.

Abstract

Solid object fracturing is essential for all plausible physical interactions. Today, real-time destructions are mostly scripted: they always produce the same behavior and form. This method greatly hurts the immersion of the player shares his redundancy.

The purpose of this thesis is to structure our method of destruction of solid objects on a completely procedural approach, giving a better interactivity between the player with his environment.

This thesis aims to mprove the destruction of solid object in real time while allowing the designers to retain control of the effect. A complete three-dimensional destruction pipeline is tested through the design of a real-time fracture tool.

Sommaire

Introduction

La destruction d'un objet virtuel a pour but de reproduire, grâce à différents algorithmes, un phénomène physique omniprésent nous entourant. Il est ainsi possible de calculer la destruction d'objets tels qu'une tasse, une voiture ou encore un bâtiment entier. La simulation de destruction est mise en place dans de nombreux domaines et plus particulièrement dans celui de la réalité virtuelle. Cette dernière est une science qui vise à immerger un utilisateur dans un monde virtuel, simulé. Une définition technique précisant certains détails est donnée par Philippe Fuchs [6] :

« La réalité virtuelle est un domaine scientifique et technique exploitant l'informatique et des interfaces comportementales en vue de simuler dans un monde virtuel le comportement d'entités 3D, qui sont en interaction en temps réel entre elles et avec un ou des utilisateurs en immersion pseudo-naturelle par l'intermédiaire de canaux sensori-moteurs. » p41, P. FUCHS, Les interfaces de la réalité virtuelles, 1996.

Comme cette définition le souligne, la simulation du comportement d'entités en 3D temps réel permet de mettre en place l'interactivité entre l'utilisateur et le monde virtuel. Parmi les comportements simulés en temps réel, on trouve la fracture qui ajoute beaucoup de vraisemblance à l'interaction du joueur avec son environnement. Celle-ci consiste à définir le comportement d'un objet lorsqu'il est soumis à de trop grandes forces pour conserver son intégralité. Comme le précise Fuchs, il est essentiel que l'interaction s'effectue en temps réel pour que l'utilisateur ait un retour direct et soit ainsi immergé. La destruction doit donc elle aussi se dérouler en temps réel pour ne pas nuire à l'expérience, c'est ici l'une des principales problématiques qui a motivé la réalisation de ce mémoire. Les mondes virtuels mis en place dans les expériences peuvent refléter la réalité mais aussi des mondes imaginaires exprimant une volonté artistique propre. De ce fait, lors de la conception, l'interaction de la destruction doit pouvoir être modelée afin de refléter cette volonté artistique, la seconde partie de ma problématique traite de cet aspect.

Dans ce mémoire, je m'intéresse à la fracture d'objet solide en temps réel pour permettre à un artiste de mettre en place une destruction interactive. Dans un premier temps, je réalise l'état de l'art des différentes méthodes utilisées aujourd'hui afin d'en distiller leurs composantes principales. La seconde partie traitera de la conception de mon outil rendant possible la mise en place d'une fracture d'objet solide en temps réel en tentant d'apporter une réponse à ma problématique : Comment créer une fracture d'objet solide en temps en gardant le contrôle artistique ? Enfin, les pistes découvertes lors de mes recherches et expérimentations feront l'objet de la dernière partie proposant ainsi une extension à ce mémoire.

1 La fracture d'objet, un effet spécial

La fracture d'objet solide a pour objectif d'augmenter l'interactivité du joueur avec l'environnement virtuel. De nombreuses recherches ont apporté des solutions traitant des différentes problématiques relatives au calcul de la destruction. Ce premier chapitre est primordial dans la compréhension de mes travaux et des raisons les ayant motivés, il constitue une introduction au sujet sur les points de vue esthétiques et techniques de la fracture. Dans cet état de l'art, nous commencerons par une approche préliminaire contextualisant la destruction. Dans un second temps nous développerons les différentes méthodes de destruction d'objet solide afin d'aboutir à la destruction d'objet solide en temps réel.

1.1 En quoi consiste la fracture ?

La fracture est un terme utilisé dans de nombreux domaines tels que la médecine, la mécanique ou encore le numérique. L'objet de cette section est de définir la fracture en tant qu'effet spécial tout en justifiant cette appellation par son origine historique.

1.1.1 Définition

De nos jours, l'évolution technologique nous permet de concevoir des jeux et des films de plus en plus saisissants avec de plus en plus de réalisme grâce aux **effets spéciaux.** Mais qu'est-ce qu'un effet spécial ?

> Un effet spécial est un trucage consistant à créer l'illusion de certains objets et phénomènes en les simulant.

De par, leur recrudescence ces dernières années, il m'est impossible de citer toutes les catégories et secteurs de travaux apparus dans les effets spéciaux ; cependant, nous allons ici étudier l'un des plus connus et utilisé à ce jour pour de nombreux projets visuels : la destruction.

Une destruction est l'action de casser un objet, de le séparer en plusieurs fragments indépendants. L'acte de détruire quelque chose implique plusieurs acteurs : l'objet cassé et l'objet cassant. L'objet cassant provoque avec effort, la rupture de l'objet cassé. Il existe de nombreux aspects dans la destruction utilisée dans les effets visuels : l'esthétique, la physique et les mathématiques. Cet aspect pluridisciplinaire est très intéressant.

Illustration 1 - Exemple de fracture d'une tasse – (CGCOOKIE, 2013).

Dans l'illustration 1 ci-contre, nous pouvons constater que l'objet détruit est la tasse et que l'objet cassant est la balle.
Dans cette situation, on peut aussi dire que la tasse est fracturée par la balle. L'action de fracturer un objet est synonyme de détruire ce dernier mis à part que la fracture est plus explicite car elle désigne le moment clé où l'objet va céder aux contraintes exercées par l'objet cassant.

Le processus de fracture d'un objet, dit aussi pipeline[1] de destruction ou de fracture regroupe toutes les étapes nécessaires pour l'achever. Par exemple, dans le cas de l'illustration 1, la fracture se divise en quatre grandes phases, dont certaines sont d'ordre purement graphique et esthétique, d'autres mathématiques et d'autres seulement physiques. Je détaillerai plus explicitement le pipeline de destruction dans la seconde partie portant sur les expérimentations.

La destruction est avant tout un effet spectaculaire, mis en place au cœur d'une action, elle est donc généralement éphémère et d'une courte durée. Souvent, la destruction est utilisée pour la crédibilité qu'elle peut apporter à une scène mais aussi pour sa capacité à impressionner. Lorsque dans un blockbuster un bâtiment est détruit, il est très généralement fait de manière démesurée, dans ce cas la destruction est l'outil de cette exagération.

Durant une fracture classique, l'objet cassé est traité mathématiquement, esthétiquement et physiquement afin de ressortir sous forme de multiples fragments. L'esthétique intervient lors de la détermination des matériaux des débrits ainsi que sur la forme et les FX[2] (visuels et audios). Les mathématiques quant à elles vont permettre de traiter la géométrie de l'objet détruit afin de générer procéduralement les fragments (également déterminés par certaines lois physiques selon les situations). Enfin la physique va permettre de déterminer le comportement des différents débrits générés en y appliquant diverses lois telles que la gravité.

Mais pourquoi avoir besoins de la fracture dans les effets spéciaux ?

Au cinéma aussi bien que dans le jeu vidéo, les spectateurs se sont habitués avec la démocratisation des effets à une évolution importante du vraisemblable. Un effet vraisemblable est à comprendre comme un effet ayant toutes les apparences du vrai, du plausible, mais qui n'a pas forcément été poussé jusqu'à un rendu photoréaliste. Généralement, l'utilisation de la destruction ou de fracture est faite dans un souci de vraisemblance. Par exemple, si dans le film *Transformers 4* [30] les bâtiments ne se détruisaient pas après qu'un vaisseau s'y écrase, le

1 Pipline : Chaînes d'actions ayant un objectif final commun, dans le cas présent la destruction.
2 FX : Special effects ou effets spéciaux en français.

spectateur ne comprendrait pas l'action avec autant de conviction. La notion du plausible est aujourd'hui partout dans le sens où l'œil du spectateur est devenu de plus en plus exigeant.

Historiquement les effets de destructions peuvent se classer en deux grandes catégories : temps réel et pré-calculé. Ici nous désignons majoritairement par temps réel les technologies du jeu vidéo et du multimédia interactif nécessitants de calculer un nombre élevé d'images par seconde (minimum 50 afin que cela apparaisse fluide pour l'œil humain [24]). La catégorie du pré-calculé quant à elle englobe les techniques n'ayant aucune contrainte de temps de calcul des images, ces dernières sont utilisées dans l'univers du long et du court métrage.

1.1.2 Au départ, une imitation réelle

Dans cette partie, il sera question de mettre en relation l'histoire de l'évolution de la destruction dans les effets spéciaux avec leurs limites et avantages. Du point de vue de l'histoire, c'est dans le domaine du pré-calculé que sont apparues les premières fractures d'objets avec les débuts du cinéma. Le numérique n'existant pas encore, les pionniers du cinéma ont dû inventer des techniques à base de bric et de broc.

Illustration 2: Un des premiers effets de destruction au cinéma (Geroge Méliès, 1901).

Les premiers effets spéciaux de destruction sont apparus avec les débuts du cinéma en 1901 avec Georges Méliès dans *L'homme à la tête de caoutchouc* [35] lorsque la tête de ce dernier explose à la fin du film (voir illustration 2). La particularité de sa méthode qui sera utilisée par la suite dans de nombreux films réside dans le fait que les destructions réelles sont effectuées sur maquette. Dans cette technique, l'échelle des maquettes détruites dépend du niveau de détail souhaité ainsi que du budget du film .

Par la suite, des destructions plus poussées seront mises en place dans des films tels que *le Dictateur* de Charlie Chaplin (sorti en 1940) [31] avec l'essai de l'artillerie lourde dans laquelle Chaplin essaye de tirer un obus ou encore avec le crash de l'avion. Durant ces scènes, de multiples destructions d'objets solides ont été effectuées dont :

- L'explosion du cabanon (voir illustration 3 ci-contre),
- L'explosion de l'obus tiré par Chaplin lors de l'essai loupé,
- Le crash de l'avion.


Illustration 3: Destructions dans Le Dictateur (Charlie Chaplin, 1945).

Lors des deux plans de la séquence d'artillerie, on peut constater la présence de multiples débris appartenant aux objets. C'est l'un des premiers films à monter une explosion aussi détaillée et ce grâce à l'utilisation d'une maquette à l'échelle 1:1. Cette méthode a permis à Chaplin d'imiter le phénomène avec beaucoup de crédibilité aux yeux du spectateur car il s'agit là d'une vraie explosion filmée. Cependant, l'absence d'acteurs à proximité dans ces scènes, principalement dans celle de l'avion où le pilote est absent lorsque l'avion se crashe pointe un premier désavantage de la technique maquette : l'interactivité avec des acteurs réels est limitée. Il s'agit là d'un frein important auquel les techniques numériques sauront remédier.

Dans l'histoire, de nombreux genres de films ont permis l'évolution des effets spéciaux, parmi ces derniers, le plus emblématique est sûrement la science-fiction, véritable nid de créativité et d'imagination. Parmi les différentes œuvres appartenant à ce genre *Star Wars* de George Lucas est l'une des plus importantes. En termes de technique, elle marque l'apogée de ce qui s'est fait en matière de technique maquette au cinéma. Ici je ne parlerai que du premier titre apparu, *Star Wars IV : Un nouvel espoir* [34] dans lequel les effets de destructions sont partout.

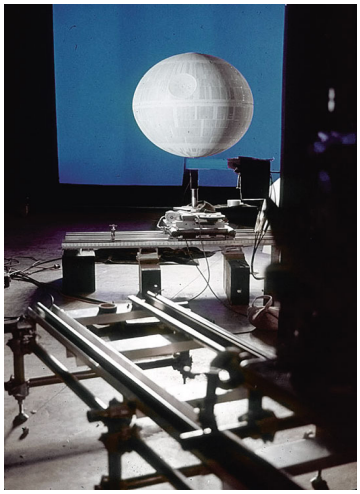


Illustration 4: technique d'incrustation de l'étoile noire (Star Wars IV : Un nouvel espoir*, George Lucas,1977)* .

> "I think the effects budget on *Star Wars* was $2.5 million. It was like no money." [3]

dit Richard Edlund pour une interview pour 3D World parue dans le numéro 189. Malgré les apparences, *Star Wars VI* [34] n'eut pas un gros budget pour les effets spéciaux, cette information montre un des principaux avantages de l'utilisation des maquettes : le faible budget requis. Malgré les faibles moyens à leur disposition, Richard Edlund et son équipe ont réussi à mettre en place de nombreuses destructions de manière inouïe pour l'époque.

La séquence la plus impressionnante en matière d'effet de destruction est celle de l'étoile de la mort (voir la figure 5) dont le faucon millénium échappe de justesse. Dans ce plan, nous pouvons observer le faucon millénium avec en arrière-plan l'explosion de l'étoile noire et l'espace en fond. Cette image se décompose donc en trois plans principaux montés les uns sur les autres. Cette technique d'incrustation des effets (dont l'explosion) représente une avancée

Illustration 5: Plan de destruction de l'étoile noir (Star Wars IV : Un nouvel espoir*, George Lucas, 1977).*

3 Richard Edlund,3D World 183.

majeure dans le monde des effets spéciaux qui sera reprise pour de nombreux autres films par la suite. Ce système apporte beaucoup de flexibilité à la technique de démolition par maquette car le lieu de destruction n'est plus un rempart, il est désormais possible de les superposer à n'importe quel lieu. L'illustration 4 montre le système de fond bleu en action, une fois filmé, ce dernier permet de générer un calque de matte[4] utilisé pour soustraire le fond et le remplacer par autre chose (dans ce cas les étoiles).

Cependant, malgré toutes ces avancées, la méthode de la maquette reste extrêmement contraignante et fastidieuse. Le contrôle des explosions est limité par la physique : il est impossible de tricher sur certaines lois. La limitation de l'interactivité de la destruction avec les acteurs bride également beaucoup les possibilités créatives. Les matériaux utilisés pour fabriquer les maquettes sont également limités : l'utilisation de matériaux nobles pour les détruire coûte cher. Cependant, avec le progrès des ordinateurs dans les années 80, une nouvelle aire où tout est possible s'amorce : l'ère du numérique.

1.1.3 Aujourd'hui, une imitation numérique

L'avènement des ordinateurs a permis de réaliser un véritable bond technologique en avant dans les effets spéciaux : de par leur puissance de calcul ils permettent de réaliser des simulations physiques et de générer des géométries assez crédibles pour être utilisés dans le film aussi bien d'animation que classique. Par leur rapidité et leurs capacités, ils vont se démocratiser jusqu'à être utilisés dans quasiment la totalité des productions audiovisuelles d'aujourd'hui.

C'est durant les années 90 que les premiers effets de destruction assistés par ordinateur sont apparus sur le grand écran avec *Jurassic Park* [38]. Steven Spieldberg et son équipe sont parvenus à faire beaucoup de progrès sur les effets spéciaux, les propulsant ainsi à un nouveau stade.

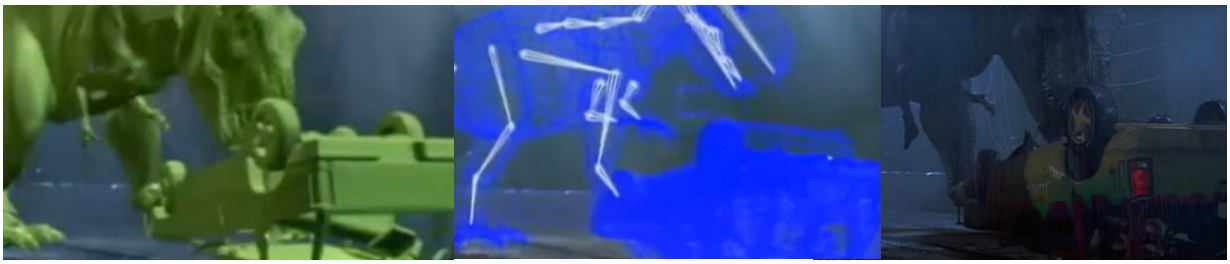


Illustration 6: L'attaque du T-Rex dans Jurassic Park *(Spiegberg, 1993).*

L'attaque du T-Rex est l'un des plans le plus significatif du film en terme de destruction, on observe le T-Rex détruisant la voiture des visiteurs. À cette époque, il est encore trop tôt pour

4 Matte : calque de transparence.

parler de simulations[5], ici tout à été animé à la main par des artistes. Lorsque les morceaux de la voiture s'arrachent sous la force des dents du T-Rex, ce sont des objets pré-séparés. L'illustration 6 montre le processus de création de l'effet ; dans un premier temps les parties fictives ont été modélisées puis un *rig*[6] à été mis en place afin de permettre l'animation des objets dans un second temps. Enfin , l'étape finale consiste à rendre les objets virtuels et à les superposer sur l'image réelle tournée. Cette méthode a un certain avantage, elle permet de contrôler très précisément les fractures car l'animation se fait par objet. Cependant, elle s'avère extrêmement fastidieuse du fait qu'il s'agit d'animation *keyframe*[7] mais également très coûteuse à cause du nombre de personnes nécessaires.

Toy Story [33] de John Lasseter est le premier film d'animation à mettre en scène une destruction lors de la scène finale avec l'explosion de la fusée permettant a Buzz et Woody de s'échapper. Dans ce plan (voir illustration 7), si l'on décompose l'effet, nous pouvons distinguer deux acteurs différents: l'objet solide qui constitue la fusée et le système de particules attaché à l'arrière. Ici le système de particules permet de simuler la combustion du carburant, qui met en valeur l'explosion de la fusée en la rendant vraisemblable aux yeux du spectateur. L'objet solide, quant à lui, se fracture en plusieurs fragments respectant ainsi la réalité physique de l'explosion d'une fusée. Malheureusement, il est impossible d'en dire plus sur les détails de la méthode utilisée pour détruire la fusée en raison de l'absence d'informations communiquées par Pixar.

Illustration 7: Explosion de la fusée lors de la fuite de Buzz et Woody dans Toys Story *(John Lasseter, 1996).*

Par la suite, l'industrie du film se développe énormément et une part de plus en plus importante d'œuvres cinématographiques utilisent les effets spéciaux. On pourrait appeler cette période la « ruée vers la vraisemblance ». Dans un souci d'exigence croissante de la part du spectateur dans l'aspect qualitatif des effets spéciaux, les grands studios de productions tel qu'ILM[8] innovent sans cesse dans l'objectif de convaincre le spectateur. Dans les films emblématiques nous pourrions citer *Le jour d'après* [32] dans lequel les éléments naturels se déchaînent contre l'homme, détruisant la ville de New York ou encore la trilogie du *Seigneur des Anneaux* [36] dans laquelle les armées de Sauron tentent de détruire le monde libre afin d'y régner.

5 Simulation : imitation volontaire ou semi-volontaire d'un trouble mental ou physique. Représentation du comportement d'un processus physique, industriel, biologique, économique ou militaire au moyen d'un modèle matériel dont les paramètres et les variables sont les images de ceux du processus étudié. Larousse.
6 Rig : armature virtuelle permettant de contrôler des personnages à l'aide d'os virtuels
7 keyframe : animation effectuée image par image.
8 ILM : Industrial Light and Magic

Parallèlement à ces avancées dans l'univers du film, un tout nouveau média a vu le jour avec l'apparition de l'ordinateur : le jeu vidéo. Avec l'évolution de la puissance de calcul des processeurs et peu après des cartes graphiques, ce nouveau média visuel interactif permet au spectateur de vivre une expérience vidéoludique (seul ou à plusieurs) rendue en temps réel par la machine. Afin que l'expérience soit immersive, de nombreux éléments entrent en jeu dont les effets visuels. Or qui dit effet visuel implique effet de destruction dans de nombreux cas. Les jeux vidéos constituent donc une mine d'or en matière d'effets de fracture.

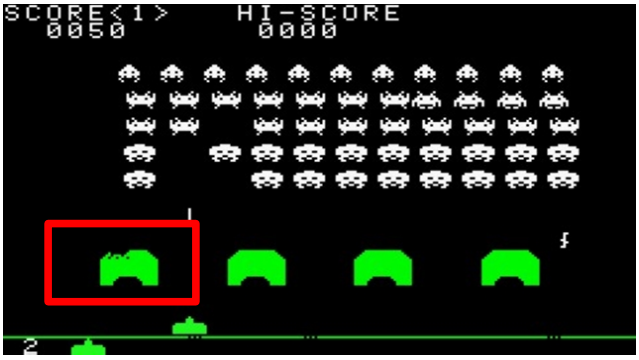


Illustration 9: Objets destructibles dans Space Invader (Taito, 1978).

Illustration 8: Explosion d'asteroïdes dans Astreroîds (Atari Inc, 1979).

Space Invader [39] est le premier jeu à mettre en place un effet de destruction en 2D donnant au joueur la possibilité de se cacher derrière des objets cassables . Ici l'effet très simple et pas moins efficace pour représenter la destruction consiste à retirer des parties de l'objet cassé sans faire de fragments. La capture d'écran ci-contre montre le résultat de l'effet (encadré en rouge, illustration 9) sur les objets. On peut constater qu'à cette époque aucune physique poussée n'est encore appliquée, les machines ne le permettant pas encore.Un an après Atari sort un futur succès mondial : *Asteroids* [40]. L'objectif du jeu est très simple, survivre et détruire le plus d'astéroïdes possible afin d'engranger le plus de points (illustration 8). Ici deux destructions sont notables, l'explosion du joueur en morceaux lorsqu'il perd et l'explosion des éléments du décor (astéroïdes, ennemis, etc.). Ce sont les premiers effets de destruction avec fragments dans l'histoire du jeu vidéo. Malgré tout, l'effet reste en 2D et les fragments n'ont aucune collisions entre eux, il faudra attendre les années 90 pour voir les premières destructions évoluer avec la démocratisation des jeux en 3D.

Le premier opus à mettre en place une fracture d'objets en 3D est *Ghen War* [43]. Dans ce dernier, le terrain est généré procéduralement permettant ainsi au joueur de le détruire en tirant dessus. C'est l'un des premiers jeux à implémenter la destruction en 3D. L'illustration 10 est une capture d'écran issue du jeu, prise au moment où la fracture de l'objet se produit. Dans cette capture on peut observer l'explosion et plus

Illustration 10: Destruction de décors dans Ghen War (SEGA, 1996).

particulièrement les fragments de l'objet cassé, c'est ici la première fois que la physique est appliquée à une destruction en 3D avec un système de *rigid body*[9] pour la physique des fragments. Je détaillerai plus précisément dans la section suivante le fonctionnement de la fracture utilisant les *rigid bodies,* cependant le concept à retenir ici est simple : l'objet détruit est pré-fracturé, les morceaux sont maintenus entre eux jusqu'au moment où l'explosion se produit. Cette méthode a quelques avantages : elle est légère à calculer, la fracture étant déjà effectuée. Ce sont les artistes qui, dès l'élaboration des objets et décors, vont effectuer la fracture de l'objet.

Par la suite, de nombreux gros titres utiliseront ce processus, parmi les plus célèbres, *Red Faction Armagon* [44] (basé sur le moteur GeoMod 2) ajoute un système de contraintes basées sur le stress appliqué aux surfaces lors de l'impact permettant ainsi de simuler des structures plus complexes tel que des bâtiments entiers ou encore des ponts.

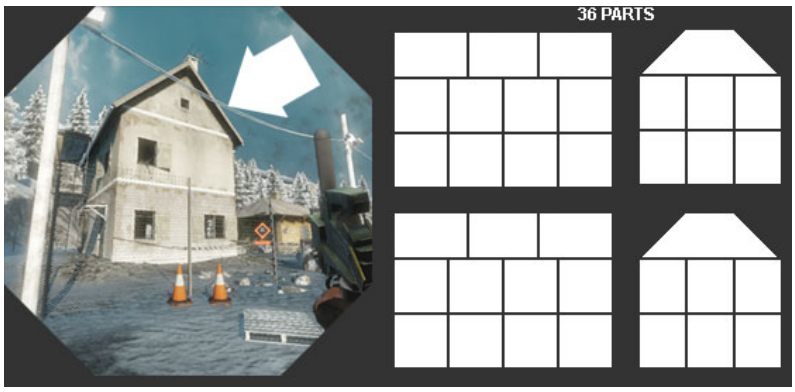


Illustration 11: patron de destruction dans Battlefield Bad Company 2 (EA, 2010).

La saga *Battlefield* est sans doute la plus célèbre auprès du grand public en terme d'effets de destruction. *Battlefield Bad Company* [42] intègre la première version du module de destruction (Destruction 1.0) mise en place dans le moteur Frosbite[10]. La technologie œuvrant derrière ce module est similaire à celle de GeoMod : les artistes préparent les *assets*[11] destructibles en les pré-fracturant. Dans le cas de larges bâtiments, afin d'optimiser les performances et de rendre l'effet modulaire, les structures sont divisées en parties distinctes liées entre elles. Le schéma 11 illustre parfaitement ce principe, on peut y voir le patron de physique de la maison. Toutes les parties sont liées entre elles permettant ainsi de simuler l'intégrité globale de la maison : si un certain pourcentage des murs est détruit, le bâtiment jouera l'animation de sa destruction complète.

Le principal problème que pose cet algorithme majoritairement utilisé dans le jeu vidéo réside non dans la performance mais plutôt dans son aspect visuel : lorsque le joueur va déclencher la destruction, elle sera toujours la même, les fragments étant pré-calculés. Nous reviendrons sur ces détails dans la prochaine partie lors de l'explication des méthodes temps réel.

9 Rigid body : Corps Rigides
10 Frosbite : moteur de jeu développé par Electronic Art
11 Asset: objets produits pour un jeu (Ex: un modèle de véhicule)

Aujourd'hui, le jeu ayant la technologie la plus avancée en matière de destruction est *Rainbow Six Siège* [41] avec le moteur REALBLAST. C'est pendant cinq ans qu'une petite équipe de développeurs a travaillé à développer le module de démolition utilisé dans le jeu. Avoir cette donnée m'a beaucoup sensibilisé quant aux objectifs fixés au départ, cela m'a poussé à revoir les étapes de développement afin de ne pas envisager quelque chose d'irréalisable. Comme Julien l'Heureux (Ingénieur dans l'équipe en charge du développement de REALBLAST) l'explique [12], il s'agit là d'une destruction procédurale effectuée en 2D. Les détails de cette méthode seront évoqués en 1.3.2.

Il faut savoir que c'est l'un des rares jeux à ne pas exploiter la destruction uniquement pour sa crédibilité mais comme élément principal du *gameplay*[12], impliquant ainsi de nombreuses problématiques de *game design*[13] dont nous discuterons dans la partie 2. La méthode expliquée pour Rainbow Six est très efficace lorsqu'il s'agit de l'appliquer sur des objets plans comme les murs et les trappes. Cependant lorsqu'il s'agit de destruction d'objets 3D complexes tels que des statues ou autres cela ne marche pas car il n'y a pas de décomposition volumétrique de l'espace.

Les effets de destructions ont beaucoup évolué ces dernières années, avec l'arrivée du numérique ils se sont complexifiés et diversifiés. J'ai volontairement omis l'explicitation de ces derniers: ils seront étudiés dans la prochaine partie, ainsi que les diverses autres méthodes existantes afin de nourrir nos réflexions dans le cadre de la réalisation d'une fracture d'objet solide en temps réel. Quelles sont les méthodes utilisées aujourd'hui pour effectuer des simulations de destruction ? Parmi ces dernières lesquelles sont utilisées en temps réel ? Quels sont les principaux outils utilisés actuellement par l'industrie du cinéma et du jeu vidéo pour les mettre en place ? La prochaine section répond à ces questions.

12 Gameplay *:* règles du jeu

13 Game design : regroupe tout ce qui est relatif à la mécanique du jeu.

1.2 Pipelines de destruction

Que ce soit dans le domaine du temps réel avec le jeu vidéo ou dans le pré-calculé avec le cinéma, la dernière section a mis en évidence l'omniprésence des effets de destruction dans les productions audiovisuelles actuelles. Les processus de fractures d'objets peuvent se classifier en deux principales catégories : les RBS[14] appelées aussi les RBD[15] et la MEF[16]. Actuellement, la majorité des studios utilisent les RBS comme solution dans leur pipeline, cependant la MEF plus récente et vraisemblable, commence à se démocratiser chez certains acteurs majeurs du domaine comme ILM.

De manière générale, nous pouvons vulgariser un pipeline de destruction classique en trois principales étapes :

1. Préparation du mesh[17], pré-fracture et création des *rigid body* – Ici l'objectif est de mettre en place la géométrie pour la destruction.
2. Contraintes et chorégraphie – Mise en place des différents acteurs de contraintes d'influences sur la destruction, donnant ainsi une marche à suivre à la destruction.
3. Simulation finale et mise en cache géométrique – Lancement de la simulation et enregistrement de cette dernière sous forme de cache (Généralement Alembic) afin d'effectuer les rendus.

Dans cette partie nous présenterons l'état de l'art de ces différentes méthodes ainsi que leurs avantages et inconvénients. Enfin, nous décrirons plus en détails celles utilisées dans le temps réel afin de comprendre les contraintes de ce domaine ainsi que leurs limitations. Lors de nos explications nous nous limiterons exclusivement aux destructions effectuées en 3D .

1.2.1 RBS

Cette méthode est aujourd'hui majoritairement implémentée dans des librairies physiques telles que Bullet, PhyX ou encore ODE sous la forme d'un *solveur*. Un *solveur* est un module logiciel utilisé comme « boîte noire », sans aucune intervention sur le fonctionnement interne. Le principe général est toujours similaire : il prend en entrée les données de la simulation physique, en sortie les résultats du problèmes à l'instant t . C'est le module de calcul de toute simulation classique.

C'est autour de Bullet que se regroupe le plus grand nombre de solutions. Dans l'industrie, beaucoup de grands studios ont recours à cette technique, parmi les plus connus, il y a

14 RBS : Rigid Body Simulations ou simulation de corps durs.
15 RBD : Rigid Body Dynamics ou dynamiques des corps durs.
16 MEF : Méthode des Éléments Finis.
17 Mesh : objet tridimensionnel constitué de polygones.

notamment Weta, Frameworks ou encore DNegative. En matière de solution logicielle grand public, Houdini 12 bénéficie d'une intégration du solveur Bullet mais a également son propre module de *solving* de *rigidbody*. Parmi les plus célèbres dans le domaine, il y a également DMM[18] de Pixelux présent sous forme de plugin dans la suite Autodesk, CBDS[19] de RealFlow ou encore Dynamica développé en 2011 par Michael Baker pour Disney Animation Studio, ce dernier étant une passerelle entre Bullet et Maya. Utilisé dans de nombreuses productions telles que *Bolt*, il a été présenté au grand public lors du Siggraph 2011.

Le pipeline RBS peut se généraliser en trois principales phases : pré-fracture, contraintes et simulation. Ces différentes étapes sont applicables aussi bien dans une production temps réel que pré-calculée. Pour des raisons de clarté, considérons une simulation RBS effectuée sur un objet *P (voir illustration 12)* pour les prochains paragraphes.

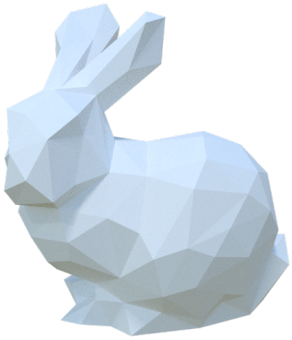


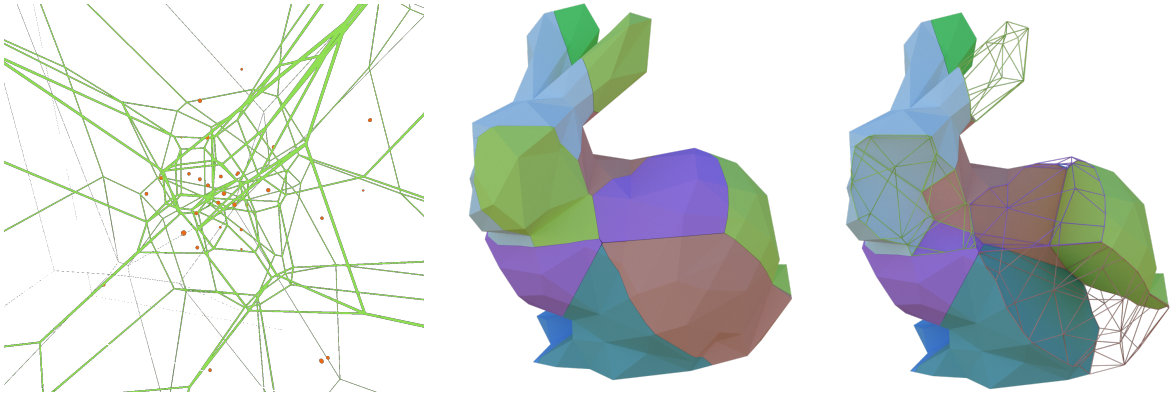
Illustration 12: Objet P utilisé comme exemple pour les prochains paragraphes.

1.2.1.1 Pré-Fracture

La pré-fracture constitue le moment clef où l'objet est séparé en fragments. Par définition les fragments sont des partitions du volume de l'objet. Dans un premier temps, il est donc nécessaire d'échantillonner le volume de *P* afin d'en déduire des fragments. Il existe différentes manières de calculer le partitionnement volumétrique d'un objet:

- Les diagrammes de Voronoï,
- La géométrie de construction de solide (CSG en anglais),
- La décomposition convexe de l'espace,
- La triangulation 3D.

18 DMM : Digital Molecular Matter
19 CBDS : Caronte Body Dynamics Solver



1 2 3

Illustration 13: Décomposition de P en suivant un diagramme de Voronoï 3D. 1 : Diagramme de voronoï. 2 : Cellule formé par l'intersection d'un diagramme et de P. 3 : Visualisation du volume des cellule.

En mathématiques, un **diagramme de Voronoï** [23] est un partitionnement d'un plan ou plus généralement d'un espace en cellules (appelées aussi région) basé sur la distance entre les points de contrôle les constituant. Comme l'illustre la figure 13.1, chaque cellule a un point (en rouge) et représente l'ensemble des points de l'espace plus proches de ce point que de tous les autres (délimité en vert sur le schéma). Dans l'image 13.2 on constate que la topologie du maillage de *P* n'est pas impactée lors de l'échantillonnage, elle reste visuellement similaire : c'est une méthode non dégradante. De ce fait elle s'avère idéale pour traiter les meshs utilisés pour le rendu. Les diagrammes de Voronoï produisent des formes naturelles de fracture s'apparentant à la roche. La photographie 14 illustre cette similarité avec les cellules de Voronoï , elles sont toutes les deux convexes avec des faces planes. Cette méthode permet de mettre en place une fracture solide contrôlée ou aléatoire en fonction de la répartition des points. Cette répartition peut être effectuée à la main ou complètement aléatoirement. Aujourd'hui c'est la technique la plus utilisée dans l'industrie.

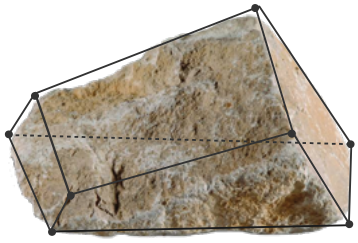


Illustration 14: Similarité rocher (photographie) / cellule de Voronoï (en noir).

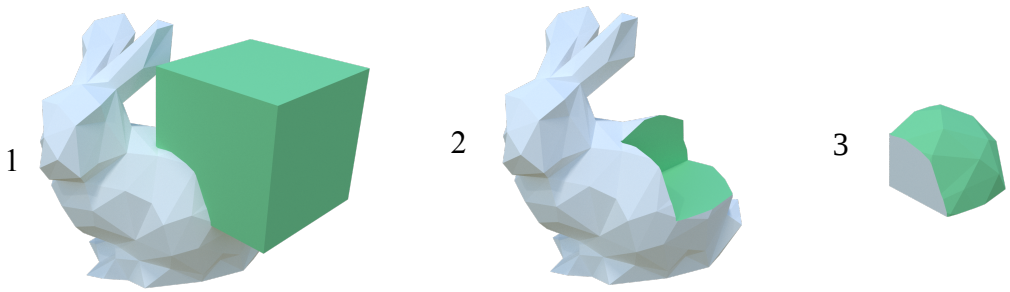


Illustration 15: Exemple d'opérations booléenne de géométrie de construction solide sur P avec un Cube.

La **géométrie de construction de solide** (abrégée GCS) permet de modéliser des surfaces ou des objets complexes en combinant des objets plus simples via des opérateurs booléens. Les GCS permettent d'effectuer des opérations booléennes volumétriques telles que l'union (Combinaison de deux objets, illustration 15.1), la différence (Soustraction de deux objet, illustration 15.2) ou encore l'intersection (Partie commune des deux objets, illustration 15.3). Ce mode opératoire permet de fracturer l'objet en plusieurs fragments similairement à un moule de découpe pour gâteau. Fréquemment utilisé lorsqu'un haut niveau de contrôle est exigé dans une destruction, ils sont également utilisés comme outils de modélisation en CAO.

La **décomposition convexe** consiste à décomposer un objet en un ensemble de sous objets convexes. Cette décomposition peut être créée à la main par l'artiste ou générée par l'ordinateur. Les figures 16.1 et 16.2 montrent le résultat de la décomposition convexe de P, on observe que les cellules formées simplifient grossièrement l'apparence de *P* : cette méthode à l'opposé de Voronoï dégrade l'apparence visuelle de *P*, elle n'est donc pas souhaitable pour échantillonner le mesh de rendu. Cependant dans le processus de fracture d'un objet, la fracture du mesh visuel et physique est généralement séparée dans un souci d'optimisation. Le mesh visuel est beaucoup plus détaillé tandis que le mesh physique ne gérant que les collisions, est une approximation du mesh visuel. La décomposition en convexe s'avère donc idéale dans la création du mesh physique. C'est aujourd'hui la technique majoritairement utilisée pour cela dans l'industrie.

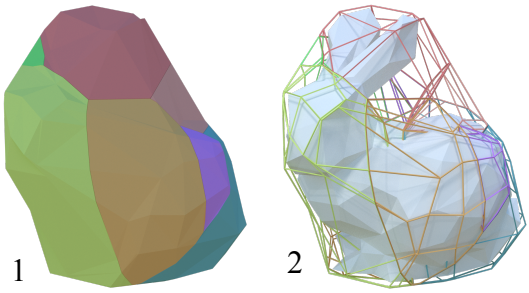


Illustration 16: Exemple de décomposition convexe de P. 1 : Visualisation des surfaces des cellules convexes. 2 : Visualisation du mailage des cellules

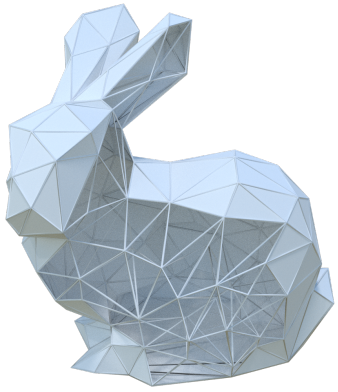


Illustration 17: triangulation 3D de P.

La **triangulation 3D** d'un objet consiste à décomposer l'espace de ce dernier en un ensemble de tétraèdre (voir illustration 17). La triangulation de Delaunay est la méthode la plus utilisée pour calculer l'ensemble de tétraèdres au moyen d'un set de points répartis dans le volume de l'objet ciblé. Cependant il faut savoir qu'il est aussi beaucoup utilisé dans la FEM (Voir 1.2.2). Par exemple, le plugin Maya DMM utilise cette cette méthode.

La pré-fracture s'effectue en suivant des méthodes différentes selon le type d'objet à fracturer ainsi que du niveau de contrôle et de détail souhaité. Dans le cas du mesh visuel le diagramme de Voronoï reste une référence, pour un mesh physique c'est actuellement la décomposition convexe qui est idéale. Le tableau 1 ci-dessous énumère les différentes méthodes parcourues en fonction des besoins.

	Diagramme de Voronoi	GCS	Décomposition en convexe	Triangulation 3D
Avantages	Rapide, facilement contrôlable		Génère un ensemble de convexes idéaux pour la détection de collision	
Inconvénients		Lourd à calculer		
Niveau de contrôle	Moyen	Élevé	Faible	Inexistant
Niveau de détail possible	Moyen	Élevé	Faible	Moyen

Table 1: Récapitulatifs des différentes méthodes de découpe volumétrique

1.2.1.2 Contrainte et animation

Une fois la géométrie préparée et pré-fracturée en de multiples fragments, leurs comportements doivent être définis. Dans un RBS classique, sans eux les fragments s'éparpilleraient immédiatement dans tous les sens avec la gravité. Il est donc nécessaire de définir des règles d'attaches entre ces derniers : **des contraintes**.

Il existe de nombreuses façons d'appliquer des contraintes, l'une d'entre elles consiste à établir une connexion entre chaque fragment. Cette méthode donne beaucoup de contrôle cependant elle ne s'avère pas du tout optimale lorsqu'il y a beaucoup de connections : tester des millions de relations à chaque image comporte un grand coût en calcul. Généralement ces contraintes sont créées procéduralement par la solution logicielle et sont ensuite influençables par les artistes. Par exemple chez Double Negative les artistes peuvent influencer ces dernières en peignant des *maps*[20] d'influence.

Un autre processus consiste à disposer ces contraintes de connexion en se basant sur la détection de collision (en ayant préalablement collé entre eux les *rigids bodies*): calculer la distance au point d'impact et ne créer des relations que lorsque le point d'impact est à une certaine distance définie par l'utilisateur. S'il y a une collision avec une force assez grande, elle se propage parmi les connections. Ces dernières peuvent être cassées ou affaiblies. Lorsque les contraintes ont étés définies, l'artiste peut alors démarrer la simulation .

1.2.1.3 Simulation

La simulation repose entièrement sur le moteur physique utilisé. L'une des composantes clefs réside dans sa capacité à détecter les objets rentrant en collision les uns avec les autres à chaque pas de temps. Des formes simplifiées des objets sont généralement utilisées afin d'alléger les temps de traitement. Parmi ces simplifications, la plus primitive pour mettre en place la détection demeure la *bounding box*[21](illustration 18): si tout objet est contenu a l'intérieur d'un simple cube il devient alors beaucoup plus aisé d'empêcher le chevauchement des objets entre eux. Cependant bien que cette méthode soit très rapide, elle demeure très imprécise, deux objets pourraient rentrer en collision sans qu'ils se touchent visuellement. Prenons par exemple *P* et un bâton : il serait très maladroit d'utiliser la *bounding box* de *P* pour calculer sa collision contre le bâton. Cependant, il serait extrêmement long de calculer les collisions exactes entre *P* et le bâton.

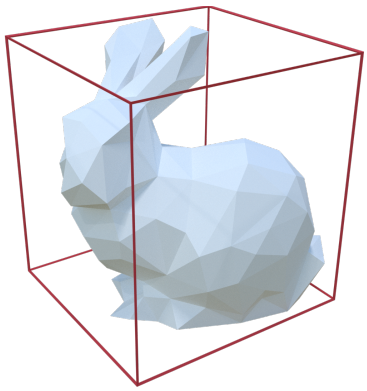


Illustration 18: Bounding box de P (en rouge).

20 Maps : Images plaquées sur la simulation.
21 Bounding Box : Cube englobant la géométrie ciblée.

Il existe différentes méthodes de détection de collision permettant de remédier à ce problème, chacune a ses avantages et ses spécificités. Parmi les plus utilisés il y a notamment :

- La détection par enveloppe convexe (Convex Hull Colision Detection en anglais).
- La détection par SDF (Signed Distance Field).

La **détection par enveloppe convexe** est plus populaire (aussi bien dans le jeu vidéo que dans le cinéma), elle détermine les paires d'objets convexes en collision à chaque pas de temps. Afin de l'utiliser, les objets doivent être préalablement divisés en un ensemble d'objets enfants, tous individuellement convexes (Voir la décomposition convexe en 1.2.1.1). Cet ensemble *S* de convexes est hiérarchisé dans une structure de données dépendant des algorithmes de traitements choisis. Comme François Lehericey l'explique [10] la détection se déroule en deux principales phases : La *Broad-phase* et la *Narrow-phase.*

La **Broad-phase** consiste à effectuer les calculs sur des versions simplifiées des objets, il existe plusieurs routines pour y parvenir. Son rôle consiste à lister les paires d'objets potentiellement en collision afin d'éviter de calculer des collisions inutiles. C'est une phase dite d'optimisation où toutes les paires qui ne sont pas en collision sont éliminées afin de localiser les objets potentiellement en collision. Dans sa thèse, Lehericey décrit trois catégories d'algorithmes de broad-phase :

- La **force brute** consiste à vérifier toutes possibilités de paires d'objets afin de tester si leur volume est en collision. Nécessitant de tester toute les paires d'objets entre eux, c'est la méthode la plus fastidieuse et rarement utilisée .
- La **décomposition spatiale** consiste à partitionner l'espace sous forme d'une structure stockant des pointeurs vers les objets contenus dans ces partitions. Une fois le partitionnement effectué, il est possible d'interroger directement cette structure afin de connaître les objets partageant des régions communes. Ce partitionnement peut être absolu et ne pas prendre en compte l'environnement ou adaptatif et s'adapter en fonction de ce dernier.

  Différentes structures de données permettent de stocker le découpage spatial (Illustrées dans le schéma 29) . Dans le cas de la décomposition absolue, il est possible d'utiliser une grille uniforme sur l'espace choisi (2D, 3D,..). Dans le cadre d'une décomposition adaptative, les structures les plus utilisée sont :

  les Kd-tree : arbre où chaque nœud a huit fils.

  Les Quadtree : même chose avec quatre fils.

  Les BSP (Binary Space Partitionning) : divise l'espace en cellules convexes qui sont ensuite ordonnées dans un arbre binaire.

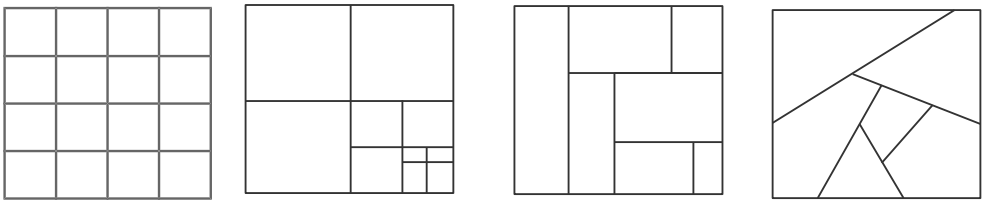


Illustration 19: Différents types d'arbres utilisés pour stocker un partitionnement spatiale.

- Le **sort and sweep** est une forme de tri spatial. Comme le met en lumière Ericson [5], cette méthode consiste à trier le début et la fin de chacun des *bounding volumes* des objets sur un ou plusieurs de leur axes (x,y,z). De cette manière il est rapide de savoir si un objet est en collision avec ses voisins : il suffit d'aller interroger ses voisins. L'illustration 20 montre ce principe : en comparant la fin *e3* avec le début *b4* on détecte rapidement la collision entre O_3 et O_4. Entre O_1 et O_2 on constate que sur cet axe ils apparaissent en collision tandis qu'ils ne s'y trouvent pas, il faut donc l'effectuer sur plusieurs axes. C'est la méthode la plus utilisée car sa complexité ne dépend pas du nombre d'objets contenus dans la scène contrairement aux deux premières méthodes décrites. Cette dernière réside dans la mise a jour de la liste des objets triés.

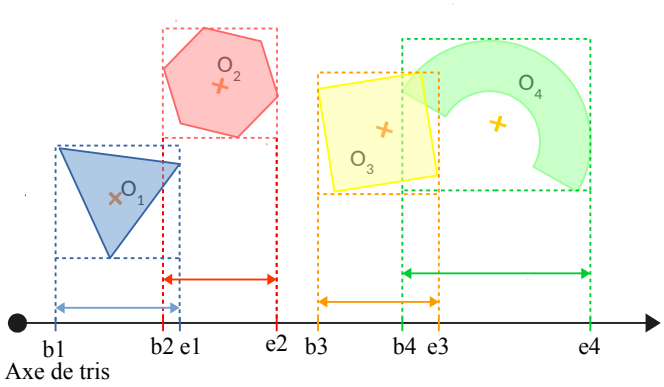


Illustration 20: Représentation de l'algorithme de Sort and Sweep sur un axe.

Après la broad-phase viens la **narrow-phase** qui effectue des tests plus poussés sur les paires d'objets sélectionnés lors de la broad-phase afin de déterminer définitivement si deux objets sont en collision. Cette étape est primordiale dans le sens où elle va également localiser la collision, ce qui est très important pour permettre aux moteurs physiques tels que Bullet d'établir une réponse physiquement correcte. Tout comme la broad-phase, la narrow-phase peut utiliser différents algorithmes. Parmi les principaux nous pouvons énumérer trois catégories:

- Les **algorithmes caractéristiques** travaillent directement sur les primitives géométriques (similaire à la force brute). Généralement le principe des ces derniers est relativement simple, il consiste à trouver les deux points les plus proches entre les paires d'objet.
- Les **algorithmes basés simplexe** travaillent sur des enveloppes convexes englobant des ensembles de points. Ici le principe de base consiste a calculer les distances entre ces deux ensembles convexes.
- Les méthodes basées sur les **volumes englobants** (ou Bounding Volume Hierarchy en anglais) qui consistent à partitionner les objets récursivement. Ils sont stockés sous forme d'arbres dans lequel chaque nœud est un volume englobant et chaque feuille une ou de multiples primitives(voir illustration 21).

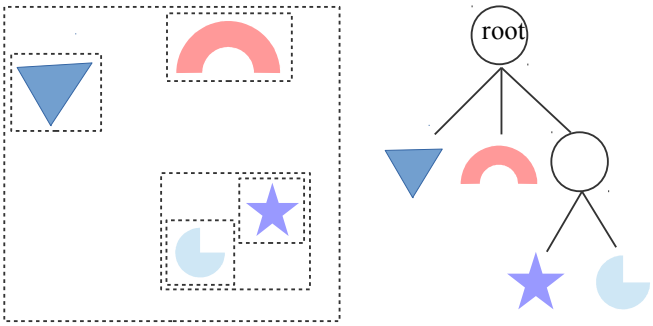


Illustration 21: Schématisation des BVH, à gauche l'espace à décomposer, à droite le BVH correspondant

La **détection par Signed Distance Field** [7] (SDF) se fonde sur les distances entre objets. Les SDF, dites aussi *distances maps* sont présentées et stockées sous la forme de grilles dont chacune des cellules représente la distance la plus courte entre elle et une autre cellule ayant d'autres propriétés. Généralement ces propriétés sont des contours comme ceux que le joueur ne peut pas passer ou encore la surface d'un mesh (Voir Illustration 22 ci-dessous).

Lorsque cette grille est formée, il n'y a qu'à l'interroger directement afin de savoir si deux objets sont en collision. Etant très fastidieuse à calculer, la passe des distances est mise en place seulement sur les objets statiques lors de la conception du jeu. De ce fait elle est généralement utilisée pour détécter les collisions entre des objets dynamiques et statiques, tel qu'un tissu (dynamique) avec un vase (statique). Dans Unreal par exemple la détection par SDF est utilisée par les particules *GPU*, la *map* des distances stockée dans le GPU sous forme de passe permet à ces dernières (les particules) d'y accéder directement pour estimer leurs collisions.

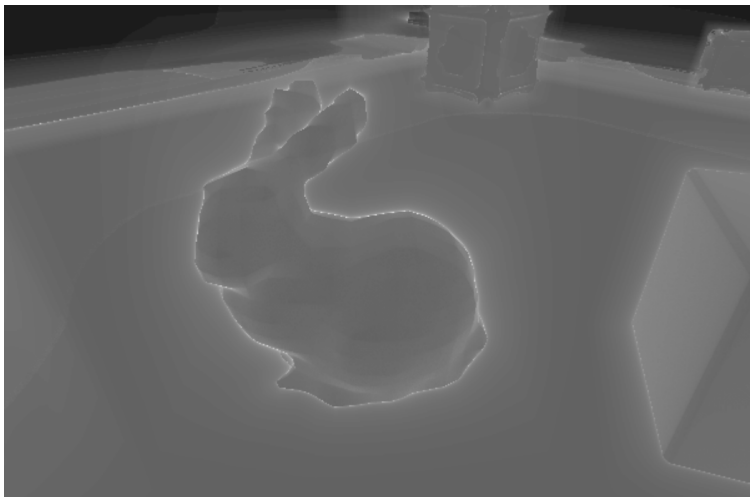


Illustration 22: Passe de Mesh Distance Field dans l'Unreal Engine 4.

Une fois la détection de collision effectuée le moteur peut établir une réponse physique sur les objets dont les collisions ont étés détectées. Cette réponse dépend de nombreux paramètres physiques définis par l'utilisateur (tel que la masse, la viscosité, etc.) mais aussi par la simulation (Objet en collisions). Les RBS sont aujourd'hui beaucoup utilisées dans le jeux vidéo et dans le cinéma. Étant concerné par le temps réel dans le cadre de mes travaux, certains cas pratiques d'utilisation des RBS en temps réel seront détaillés dans la prochaine section.

1.2.2 MEF

Aujourd'hui la méthode des éléments finis (abrégée MEF) n'est pas encore très répandu dans le monde des effets spéciaux, aussi bien dans l'univers du jeu vidéo que du cinéma. L'une des raison majeures vient de son coût en temps de traitement, cependant cela est amené à changer avec l'évolution des algorithmes et capacités de calculs des ordinateurs. MPC fait partie des rares à utiliser un système de destruction basé sur la MEF, mis en place avec DMM, utilisé pour la première fois dans *Sucker Punch* [1].

La méthode des éléments finis [11] est une approche physiquement réaliste contrairement aux RBS, elle utilise les éléments finis afin de résoudre l'équation partielle différentielle qui pilote le comportement d'un matériau déformable. Fonctionnellement elle se base sur un système complexe de nœuds constituant un maillage. Ce dernier contient toute les propriétés physiques structurelles qui vont définir le comportement du volume face aux différents facteurs. Ces nœuds sont répartis à une densité dépendant du matériau physique et dans certains cas en fonction du stress anticipé qui peut se produire dans certaines zones. Les zones soumises à un plus fort stress contiennent une densité de nœuds plus élevée que celles en endurant peu afin d'avoir une meilleure résolution dans la zone d'intérêt.

La forme que prend le maillage dépend de la décomposition spatiale utilisé, en 3D les élément les plus communs pour décomposer l'espace sont les tétraèdres et les hexaèdres (voir illustration 23). Il faut voir le maillage comme un réseau qui, lors de la simulation va permettre aux nœuds de communiquer entre eux des informations relatives à leur déformation. Dans le cas précis de la fracture, en analysant le stress subi par le maillage, la simulation détermine où commence la fracture et dans quelle direction elle se propage.

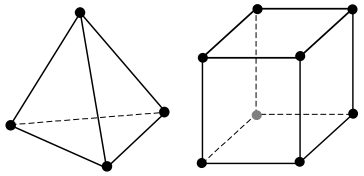


Illustration 23: Éléments volumétriques utilisées couramment pour la MEF, le tétraèdre à gauche et l'hexaèdre à droite.

Tout comme les RBS, afin d'optimiser la simulation, une broad-phase est souvent utilisée (notamment dans DDM). Mais cette dernière est plus rapide qu'en RBS : les objets sont déjà décomposés en tétraèdres ou hexaèdres contrairement aux RBS où il s'agit de convexes. Tester les distances entre tétraèdres de tétraèdres est plus rapide.

Le gros désavantage d'utiliser les RBS au détriment des MEF réside dans le fait que les RBS nécessitent beaucoup plus de préparation : la fracture s'effectue avant la simulation tandis que dans les MEF la fracture se produit à la fin. De plus, les RBS ne permettent pas de simuler des corps souples comme le plastique. Il existe des versions hybrides de RBS permettant de créer des déformations semi-rigides mais cela reste beaucoup moins optimal que les FEM qui font « tout en un ».

Le problème aujourd'hui dans les grandes sociétés telles qu'ILM réside dans le fait qu'elles ont un pipeline RBS déjà bien en place dont il serait compliqué de sortir du fait que les MEF et RBS ne fonctionnent pas du tout de la même manière : la transition d'un pipeline RBS

vers MEF demanderait de remettre en question tout le pipeline FX dans le sens ou les trois étapes du RBS ne sont plus là.

1.3 Méthodes temps réel

Dans le chapitre précédent nous avons abordé les principaux pipelines de destructions d'un point de vue générique, cependant dans le cadre de mes recherches, l'objectif étant de réaliser une destruction en temps réel nous allons ici approfondir les méthodes utilisées dans le temps réel. Dans le monde du jeu vidéo, il n'est pas possible de tricher en terme de temps, le joueur n'attend pas qu'une simulation se termine avant de jouer. La moindre discontinuité dans le nombre d'image par seconde risque de couper toute l'immersion de l'expérience. Il s'agit là d'un point auquel il est extrêmement important de faire attention lorsque l'on développe le jeu. Dans cette optique, de nombreux papiers et présentations font état des différentes méthodes de destruction optimisées pour le temps réel dans des exemples concrets. Ces dernières ont des similarités avec celles étudiées précédemment, c'est pourquoi nous ne nous attarderons que sur les détails importants. Afin de rester concret, nous baserons l'étude de deux cas de l'industrie avec Star Wars : The force Unleashed et Rainbow Six Siege.

1.3.1 Approche physiquement correcte

Start Wars est l'une des seules franchises de jeu vidéo à utiliser une méthode de destruction physiquement vraisemblable. Dans leur exposé [15], James F. O'Brien et al expliquent une technique basée sur les MEF pour mener à bien les effets de déformations et de fracture du jeu. La fracture est ici au cœur du gameplay, aux yeux du joueur, elle prend la forme du pouvoir de la force.

L'utilisation des MEF dans ce cas précis permet d'effectuer une déformation/fracture non prédéfinie qui changera à chaque interaction en fonction du comportement du joueur, elle ne sera jamais la même: on peut la qualifier de procédurale. Ils permettent également une interaction beaucoup plus poussée entre le joueur et son environnement : il peut tordre et déformer les élément du décor. Dans Star Wars, les objets visuels sont spatialement décomposés en tétraèdres .

Dans leur modèle de fracture, O'Brian et al ont défini trois principaux comportements sur lesquels pourront s'appliquer les forces afin d'ouvrir l'objet. Pour ces trois cas le comportement de la fracture est réalisé en analysant les forces situées au niveau de la pointe de la fissure : elles vont permettre de la guider. Les forces de tractions opposées entres elles prolongent la fissure dans la direction perpendiculaire à la direction de la forces de traction la plus élevée. Dans l'autre sens, les forces de compression auront tendance à arrêter les fissures auxquelles elles sont perpendiculaires. Afin de déclencher une fracture, le système établit toute les forces de compression et de traction à chaque pas dans le temps et forme un tenseur résultant pour chaque nœud. Si la force est suffisamment grande, le nœud concerné est divisé en deux nœuds distincts, le plan de fracture est alors calculé. Les tétraèdres voisins de celui coupé en deux doivent également être divisés afin de maintenir l'intégrité du maillage. Afin d'éviter certaines difficultés

dues aux faibles densités de points, O'Brian et al effectuent dans un premier temps une retopologie[22] du mesh visuel localement autour de la fracture en divisant en deux tous les éléments intersectant le plan de fracture. Dans un second temps, le nœud à l'origine de la fracture est dupliqué de manière à ce que la fissure passe entre lui même et son clone. Enfin, toutes les positions des nœuds directement attachés au nœud de départ sont testées avec celles du plan de fracture, si ces derniers ne sont pas en intersection, ils sont associés au nœud original ou au clone en fonction du coté du plan de fracture auquel ils appartiennent.

Il s'agit là d'une application temps réel des FEM, la densité du maillage physique ne peut donc pas contenir un nombre élevé de nœuds car les performances en seraient directement impactées. Dans cette optique, le découpage du mesh visuel ne peux pas s'effectuer en suivant les faces des tétraèdres du maillage physique : il ne serait pas assez détaillé (il formerait de gros triangles qui ne respecteraient pas du tout les matériaux de l'objet source). La solution à ce problème mis en place dans Start Wars consiste à intégrer des fragments haute résolution conçus par les artistes dans le modèle existant. Comme l'illustre la figure 23, ces morceaux sont visuellement associés entre eux pour former une surface, ils sont également attachés individuellement à un tétraèdre correspondant qui contient leur centroïde (points rouges, figure 24). Ainsi au moment de la fissuration, il est également testé si le centroïde de l'éclat appartient au tétraèdre le contenant. Si cette condition n'est pas vérifiée le fragment est dissocié du tétraèdre afin d'être associé a son voisin le plus proche (figure 24, en haut à droite).

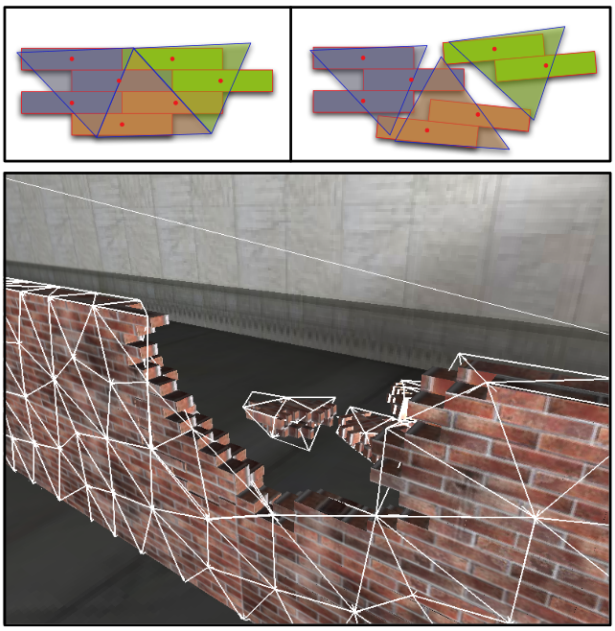


Illustration 24: Capture représentant la géométrie discrète utilisé pour les calculs de la fracture (en blanc) et le modèle visuel haute résolution (le mûr de brique)(Parker et O'Brien, 2009).

La notion d'optimisation est importante dans l'optique d'obtenir des performances temps réelles. Pour calculer la simulation efficacement O'Brian et al ont mis en place un système de multithreading. Le multithreading est un procédé qui permet à un cœur de processeur d'exécuter plusieurs tâches simultanément (pour les machines en ayant la capacité). Dans Star Wars, un système se base sur les îlots[23] générés pour dispatcher les calculs sur les différents threads. Le calcul de chaque îlot est effectué sur des threads différents, faisant ainsi gagner la méthode en stabilité.

22 Retopologie : redéfinition du maillage d'un mesh
23 Ilot : ensemble de fragments isolé

Le contrôle artistique accordé par cette technique est un point important à relever ; en permettant aux artistes de définir directement les matériaux intérieurs des objets détruits, O'Brian et al [15] donnent un contrôle du pipeline visuel et artistique complet aux artistes : ils modélisent la totalité de l'objet. Dans Star Wars, les FEM n'agissent donc que sur le comportement physique des objets conçus. Cela soulève une remarque : la création d'objets étant entièrement faite par les artistes, elle rallonge le pipeline de production. Ne serait-il pas plus efficace de créer procéduralement l'intérieur des objets afin d'avoir une densité de maillage adaptative ? Et si oui dans quelle mesure est-il possible d'obtenir un maillage cohérent en fonction de paramètres définis par un artiste ?

Cette méthode a certains avantages, son utilisation permet d'obtenir des comportements physiquement réalistes en temps réel : la propagation de la fracture va suivre les forces appliquées par l'acteur extérieur. Cependant avec un nombre de détails élevé, la quantité de polygones à stocker n'en sera que plus élevée, dans le cas d'un mur de brique, toutes les briques sont stockées., ce qui n'est pas optimal en terme de vitesse de rendu.

1.3.2 Approche non physiquement réaliste

Aujourd'hui majoritairement utilisée dans l'univers du jeu vidéo, la méthode RBS est présente nativement dans la majorité des moteurs tel l'Unreal Engine ou encore Unity. Cependant certaines implémentations diffèrent sur de nombreux points et sont extrêmement intéressantes en terme d'optimisation et de pipeline. C'est notamment le cas de Rainbow Six Siege qui met en œuvre une destruction procédurale en temps réel dans son moteur REALBLAST.

Le concept est le suivant: à la base, les objets sont séparés en plusieurs sous objets en fonction de leurs matériaux physiques, par exemple un mur en béton et son armature en acier seraient deux objets séparés. Cette étape s'effectue lors de la conception du jeu avec certains effets supplémentaires comme les particules ou encore les *decals*[24]. Ils sont ensuite instanciés en suivant un pipeline classique de RBS en temps réel (voir la partie 1.2.1.3). Durant le jeu, les objets plan (murs barricades et trappes) se détruisent procéduralement en suivant la procédure suivante (voir aussi illustration 25) :

1. Génération du schéma de découpe en 2D en fonction des paramètres du matériau physique de l'objet.
2. Intersection entre les polygones du schéma et les polygones de la surface de l'objet (A l'aide d'algorithmes tel que *Weiler – Atherton* [20]).
3. Triangulation de la surface restante sur l'objet ciblé, au moyen d'un *Ear Clipping* [4].
4. Création du modèle 3D à partir de la surface généré par extrusion de cette dernière.
5. Ajout d'effets visuels (Particules, *Decals*, Géométrie supplémentaire).

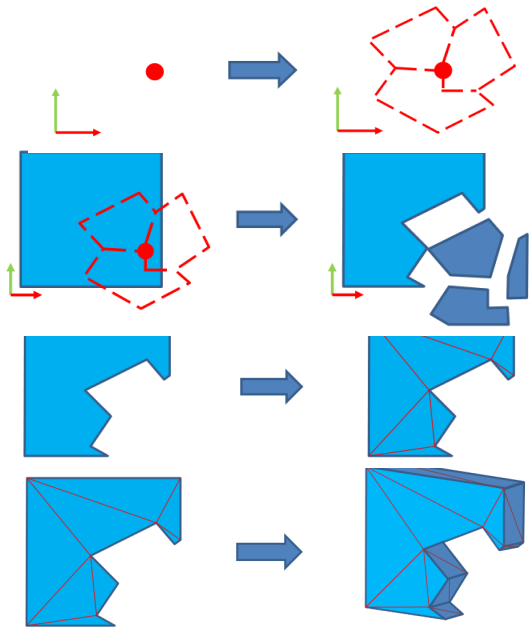


Illustration 25: Procédure de fracture dans Rainbow Six : Siege (Ubisoft, 2016).

Par un souci d'optimisation, les fragments ne sont pas générés, seuls les trous laissés par les impacts sont calculés. Les effets de particules sont là pour faire paraître au joueur qu'il y a des fragments en instanciant des objets de fragments aléatoires. L'image 26 illustre la démolition finale telle qu'elle est dans le jeu avec les particules (débris, fumée).

24 Decals : textures projetées sur un objet en 3D

Illustration 26: Visuel de l'effet final de la fracture procédurale, Rainbow Six : Siege (Ubisoft, 2016).

L'utilisation du procédural dans la destruction donne au yeux du joueur une crédibilité supplémentaire : cette dernière ne sera jamais la même et dépend du point d'impact. C'est un point clef nécessaire pour garantir l'immersion dans le jeu (ou l'expérience), il repousse les limites du vraisemblable. De plus cet algorithme de démolition donne aux artistes un contrôle poussé:

- Les matériaux physiques paramétrables permettent de modeler le comportement de la création du schéma de découpe.
- Les effets visuels post-destruction permettent d'affiner la simulation afin de les rendre plus vraisemblables.

Dans les solutions et méthodes plus classiques, il en est une qui est aujourd'hui mise en place dans beaucoup de jeux : PhysX, une plateforme de simulation physique pensée et créée pour le jeux vidéo par Nvidia. Elle est intégrée nativement dans les plus célèbres moteurs de jeux tel que l'Unreal Engine ou encore Unity. Parmi les fonctionnalités proposées par la librairie, il y a notamment tout un module de destruction ainsi qu'un éditeur dédié à cela : PhysX Lab.

PhysX fonctionne de la manière dont nous avons décrit les RBS, dans un premier temps, l'artiste pré-fracture l'objet aux moyen de PhysX Lab, l'objet est ensuite exporté au format apx afin d'être importé dans le moteur de jeux pour être intégré. L'étape d'intégration consiste principalement à réaliser les matériaux visuels pour *l'asset* destructible, aussi bien pour les surfaces internes (les fragments) qu'externes. Une fois l'intégration réalisée vient la simulation dans le jeu telle que nous l'avons décrite dans la partie précédente.

Le réel avantage de PhysX en dehors de son optimisation réside dans physX Lab qui permet d'unifier le pipeline classique RBS en regroupant l'étape de pré-fracture et de contrainte. Dans l'éditeur il est possible de briser le mesh voulu et de contrôler les différents paramètres physiques des fragments. PhysX donne accès à trois méthodes de fracture :

- Slice: découpe le long d'un axe.
- Cutout : découpe à partir d'une image donnée en entrée, a l'avantage d'être extrêmement personnalisable.
- Voronoï : découpe en suivant les cellules de Voronoï.

La pré-fracture par niveau est une fonctionnalité clef de la librairie, elle permet d'établir des niveaux de profondeur de fragmentation : les fragments peuvent se rebriser *x* fois, *x* étant le niveau de profondeur de fracture établi.

Tout ces outils permettent aux artistes de contrôler les effets de destruction intuitivement et efficacement, c'est à ce jour le pipeline le plus complet en terme de destruction pour le temps réel. Cependant, selon le type de jeu, le joueur peut être amené à vite atteindre la limite principale de cette technique : la redondance.

2 Développement et expérimentation d'une fracture d'objet solide

Dans la partie précédente, l'étude des différentes méthodes existantes pour créer une fracture d'objet solide a été faite en débutant par les techniques génériques pour aboutir à celles utilisées aujourd'hui dans le temps réel. Dans cette partie, j'expliquerai la réalisation du prototype d'outils de destruction que j'ai développé afin de créer la destruction d'un objet solide en temps réel. Dans un premier temps, les choix de pipeline effectués seront expliqués en fonctions des problématiques posées. Les deux parties suivantes traitent des détails de l'implémentation et de l'expérimentation de l'outil.

2.1 Pipeline de destruction choisi

Après avoir fait l'état des différentes méthodes existantes, le choix du pipeline est devenu primordial. La contrainte du temps fût un élément crucial dans mes décisions, il a été nécessaire d'établir une *roadmap* du prototype afin de procéder étape par étape. La méthode sélectionnée pour le prototype sera introduite dans la première partie puis dans un second temps l'étude du contrôle artistique et des possibilités de *game design* fournis par cette procédure seront étudiés. L'objectif final étant d'avoir un prototype fonctionnel j'ai opté pour l'utilisation d'outils tiers fournissant certaines fonctionnalités très précises pour mon projet, il en sera discuté dans le troisième point.

2.1.1 Procédure dans les grandes lignes

Compte tenu du temps disponible (environ un mois) pour développer l'outil, il est tout de suite apparu évident d'éliminer la simulation MEF comme méthode sur laquelle se baser. Cela prendrait trop de temps à mettre en place mais surtout à optimiser pour avoir des performances temps réel. De ce fait je me suis orienté vers la méthode RBS avec laquelle il est plus sûr de parvenir à un résultat concluant.

Mon outil se base sur une partie de la méthode mise en place par Muller et al [13] qui s'appuie sur une décomposition en Voronoï de l'espace pour générer la forme de la fracture (aussi appelée *pattern* de fracture) .

Afin de mieux comprendre la procédure lorsque nous rentrerons dans les détails, regardons tout d'abord le pipeline dans son ensemble avec le schéma 27. La première phase consiste à créer le pattern[25] de fracture, il s'agit d'un objet définissant la forme qu'aura la fracture. Cette phase est composée de deux étapes majeures :

1) Dans un premier temps le volume est échantillonné , un ensemble de points est extrait.
2) Dans un second temps, le diagramme de Voronoï est généré à partir de ces points.

De multiples patterns peuvent être créés afin de rendre différentes visuellement les fractures.

La seconde phase s'effectue en temps réel, le diagramme généré permet de fracturer les différents *meshes* (visuels et physiques) en suivant ses cellules. Une fois les fragments calculés, ils sont instanciés ; la phase finale peut alors commencer : le moteur physique simule les diverses contraintes exercées sur eux (gravité, impulsions, etc.) afin de les animer.

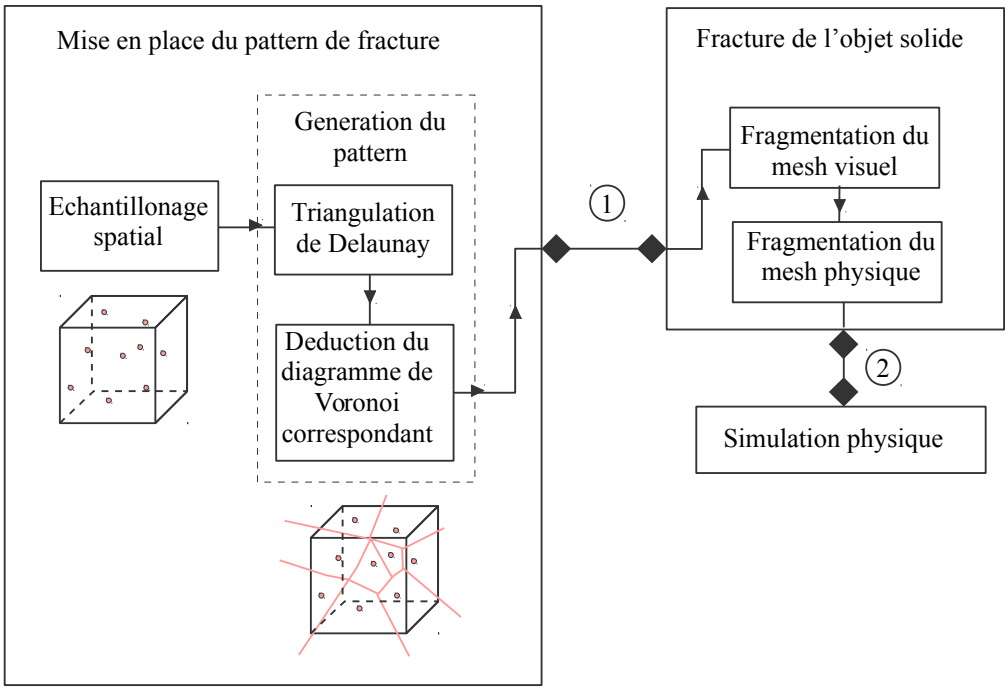


Illustration 27: Représentation schématique du pipeline de fracture conçus

Cette méthode a l'avantage d'être procédurale, l'objet peut être fragmenté à l'infini ! Cependant, les simulations purement procédurales ont peu de valeur si elles ne peuvent pas être contrôlées.

25 Pattern : Schéma de fracture.

2.1.2 Contrôle artistique et intégration dans le game design

Le contrôle artistique est un point critique dans les simulations procédurales et plus encore dans le jeu vidéo où il est nécessaire d'avoir un degré de contrôle élevé pour créer les interactions du joueur et élaborer le gameplay. Comment donner un haut contrôle sur la destruction à l'artiste et au *game designer* ? Nous répondrons à ces questions dans ce chapitre.

Une destruction peut être décomposée en plusieurs aspects artistiques :

- L'aspect sonore,
- L'aspect visuel :
  - Forme de la fracture,
  - Matériaux,
  - Effets,
  - La chorégraphie.

Il est critique de garantir le contrôle de ces différents points afin que l'effet soit réussi. La sonorisation peut être découpée en deux catégories, les sons d'impact et les sons post-impact produits par les collisions entre fragments. Les sons d'impact peuvent être mis en place à l'aide d'un *callback* lancé depuis la méthode appelée lors du début de la procédure de fracture en revanche les sons post impact sont plus complexes car physiques. Par chance l'Unreal Engine propose le plugin natif *Phya* répondant exactement à ce besoin : il permet de faire un rendu sonores des réponses physiques pendant le jeu.

L'échantillonnage spatial peut être réalisé en ajoutant des points manuellement, l'artiste peut ainsi créer un pattern de fracture; de cette manière le contrôle artistique est total sur la forme.

Lorsqu'un objet est amené à se détruire, le matériau intérieur définissant visuellement la matière de l'objet doit pouvoir être facilement défini par l'artiste. Dans le cas où l'objet aurait éventuellement une texture, il est vital qu'après la fracture les fragments soient visuellement cohérents et conservent les coordonnées UV de la texture ainsi que les différents matériaux assignés.

Le pipeline énuméré en 2.1.2 n'y fait pas référence mais l'étape de la simulation des *rigids body* doit donner la possibilité à l'artiste d'influencer les mouvements des fragments afin de contrôler la chorégraphie de la fracture. Prenons l'exemple d'un mur de pierre détruit par une voiture: si nous laissions la simulation physique classique agir, le destruction ne serait pas spectaculaire et le *feedback*[26] du joueur en serait amoindri. Afin de pallier à ce problème, l'outil de destruction doit permettre à l'artiste d'influencer directement les paramètres physiques de la simulation afin de l'exagérer et de la rendre spectaculaire. Le *game design* peut également être directement impacté par cette simulation dans la mesure où la destruction d'une entité est

26 Feedback : retour sur un évènement, il peut être visuel, haptique ou encore sonore.

impliquée dans le gameplay (La nécessité de détruire un mur pour progresser dans le niveau par exemple).

L'aspect *game design* dans la fracture est surtout d'ordre comportemental : son comportement influence directement le gameplay. Le comportement touche à la chorégraphie et donc à la physique appliquée aux fragments lors de la simulation. Le game designer doit donc avoir la main mise sur ces deux éléments pour intégrer l'effet au jeu.

2.1.3 Choix des technologies et algorithmes

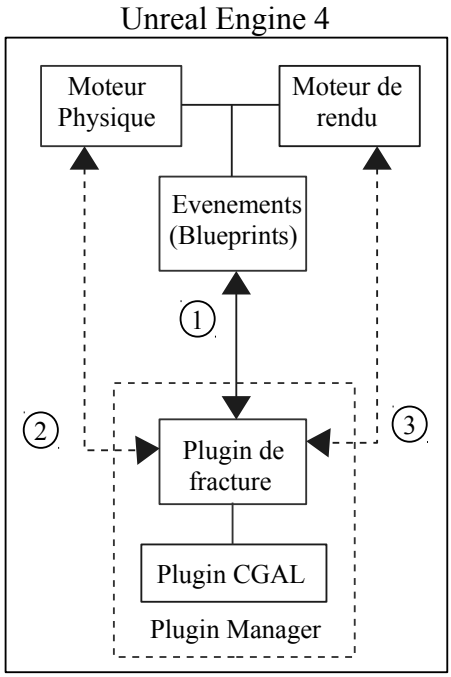
Le temps imparti pour la réalisation du prototype étant court, il n'était pas question de réinventer la roue. Le choix de concevoir l'outil pour un moteur de jeu a certains avantages :

- L'environnement de développement souple et très rapide à prendre en main , facilement extensible via un système de plugin modulaire.
- Le moteur physique intégré gère nativement la gestion des *rigids bodies*.
- La communauté très active permet d'avoir tout un panel d'utilisateurs.

J'ai donc commencé par étudier les différentes technologies de destructions disponibles dans les moteurs de jeu les plus répandus. Au final mon choix s'est porté sur l'Unreal Engine 4 : les outils de destruction qu'il inclut ne permettent que de réaliser des destructions pré-fracturées et donc peu interactives. Sa force réside dans son langage de programmation nodal appelé *blueprints* qui permet de mettre en place rapidement les différentes interactions du jeu . Étant *open source*, le moteur est également facilement modifiable en cas de besoin. Le système de *plugin* permet d'avoir une architecture modulaire, j'ai donc opté pour réaliser la fracture temps réel sous la forme d'un *plugin*.

La génération du diagramme de Voronoï se déroule en deux étapes : la triangulation 3D de Delaunay à partir des points donnés par l'artiste puis la déduction du diagramme dual. Au départ j'ai retenu la méthode de Hugo Ledoux [9] qui se base sur une insertion par *flip* pour générer la triangulation en prenant en compte les délais donnés, l'utilisation de CGAL s'est alors imposée d'elle même. En complément, CGAL implémente une méthode très proche. CGAL est une librairie d'algorithmes géométriques opensource maintenus par de nombreux chercheurs dans le monde. Cette dernière contient une implémentation de triangulation 3D par insertion me permettant de rendre l'éditeur de pattern de fracture interactif.

Aux prémices du développement, la première étape a consisté à rendre CGAL directement accessible en C++ dans Unreal en l'intégrant sous forme de plugin afin de pouvoir directement l'utiliser dans mon outil de fracture.



1: Interfacage de la fracture dynamique via les blueprints. 2: Calculs des meshs physiques des nouveaux fragments et reinjection de ces derniers dans le contexte PhysX. 3: Instantiations des entités visuelles des fragment sous forme de mesh procedurale.

Illustration 28: Schéma représentant l'architecture de l'outil de fracture au sein d'Unreal

Le schéma 28 résume l'architecture de l'outil dans l'environnement d'Unreal, le plugin permet de paramétrer la fracture depuis les *blueprints*. Ce dernier est dépendant du plugin CGAL car il y fait directement appel lors de la création du pattern. L'utilisation de CGAL rend le prototypage plus rapide mais sa complexité d'interfaçage avec le moteur est l'un de ses principaux inconvénients: elle nécessite certaines DLL contenant ses dépendances pour fonctionner. Cette architecture n'est pas viable pour le multiplate-forme. Une intégration des diagrammes de Voronoï native au plugin de fracture serait optimale pour la *release*.

L'artiste ou le game designer doit appeler la fonction M_fractureProceduralMesh (1) afin de lancer la destruction. En arrière plan, le plugin de fracture récupère les informations visuelles (2) et physiques (3) de l'objet ciblé et calcule les fragments pour ensuite les instancier (1 et 2).

2.2 La génération du pattern

La génération du pattern de fracture occupe une place extrêmement importante dans le pipeline de destruction que j'ai élaboré, elle permet de créer les fondations de la destruction. Cette fonctionnalité donne la possibilité de créer la fracture avec un contrôle total sur la forme sans pour autant être fastidieuse car les points de contrôle représentent chacun un fragment. Cette étape est mise en place sous forme d'un éditeur afin de rendre le pattern facilement modifiable avec un processus de création intuitif. Dans un premier temps l'algorithme au cœur de la création du pattern sera expliqué puis l'interface et les interactions possibles pour créer le pattern seront mises en lumière dans une seconde section.

2.2.1 Génération des cellules

J'ai développé la création des cellules du pattern en deux volets principaux: la triangulations des points fournis et l'extraction du diagramme de Voronoï. Une troisième partie facultative consiste à nettoyer les cellules extraites afin d'optimiser les traitements effectués lors de la fracture. Les cellules peuvent également être générées en temps réel dans le cas où l'artiste souhaiterait donner un caractère aléatoire à la destruction mais il est recommandé de réaliser cela lors de la production du jeu dans un souci de performance. La génération pouvant être lente avec beaucoup de points de contrôles.

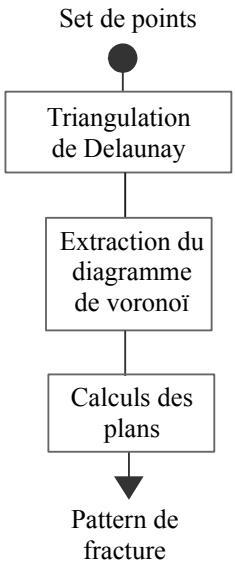


Illustration 29: Principales étape de la génération du pattern en ordre séquentiel

Afin de mieux comprendre la mise en place des cellules dans les détails, le schéma 29 visualise la création du pattern étape par étape. Il est important de noter que mon outil se base sur des points de contrôle définis par l'artiste ou générés aléatoirement. Cet ensemble de points *P* est directement utilisé comme source pour mettre en place la triangulation de Delaunay. Il existe de nombreux paradigmes pour calculer la triangulation (appelée aussi *tetrahedralization*) de Delaunay d'un ensemble de points en 2D ou en 3D :*Divide and conquer, Sweep and plane , incremental insertion flip based*. La spécificité du pipeline utilisé dans l'outil de fracture pour mettre en place le pattern réside dans sont interactivité : l'utilisateur peut insérer , retirer ou déplacer des points de contrôles tout en visualisant en temps réel l'évolution du diagramme. L'insertion incrémentale par flip est la méthode la plus adaptée pour supporter cette interactivité, lors de l'insertion d'un point elle permet de ne pas reconstruire la totalité de la triangulation. Ce gain d'optimisation permet à l'artiste d'avoir une expérience fluide dans l'éditeur de point.

Le calcul de la triangulation de Delaunay *DT* pour un ensemble de points *P* s'effectue en différentes étapes, dans un premier temps l'initialisation du diagramme puis l'insertion de points. L'initialisation consiste à instancier un grand tétraèdre contenant tous les futurs points insérés, permettant ainsi d'éviter de tester si un nouveau point inséré est en dehors d'un tétraèdre existant. L'algorithme ne tombe donc jamais en dehors de l'enveloppe convexe.

L'insertion est la principale étape de la formation de la triangulation, à chaque insertion de point, la triangulation va s'adapter. Comme l'explique Hugo Ledoux [9] deux tests sont nécessaires pour générer la triangulation:

- **L'orientation :** détermine si un point est sur, dessous ou dessus un plan défini par a, b et c.
- **La sphère circonscrite** : un point *P* de *DT* ne doit pas être à l'intérieur de la sphère circonscrite de l'un des tétraèdres de *DT*.

Lors de l'insertion, une première phase consiste à localiser dans quel tétraèdre de *DT* se situe le point *p*. La routine mise en place dans mon outil au travers de CGAL se base sur l'algorithme de marche développé par Sylvain Pion et al [19], elle consiste à retracer la position du point *p* en partant d'un des points existants dans la triangulation. En testant à chaque itération de quel côté du plan défini par *w*,*u* et *v* le point *p* se situe. Le schéma ci-contre illustre ce principe avec le test d'orientation sur *p* qui se fait à chaque itération.

Une fois que le tétraèdre *t* correspondant au point *p* est localisé, une suite d'opérations topologiques (voir illustration 30) est effectuée sur les tétraèdres adjacents à *p* appartenant à *DT*, ce sont les flips. Ces opérations permettent d'insérer le point dans *DT* tout en conservant le prédicat de la sphère circonscrite énoncé un peu plus haut.

Dans un premier temps le point *p* est inséré dans le tétraèdre *t* avec un *Flip14*, illustré dans les schémas ci-contre, générant ainsi quatre tétraèdres (*p* se situant au centre). Ces quatre tétraèdres sont insérés dans une pile.

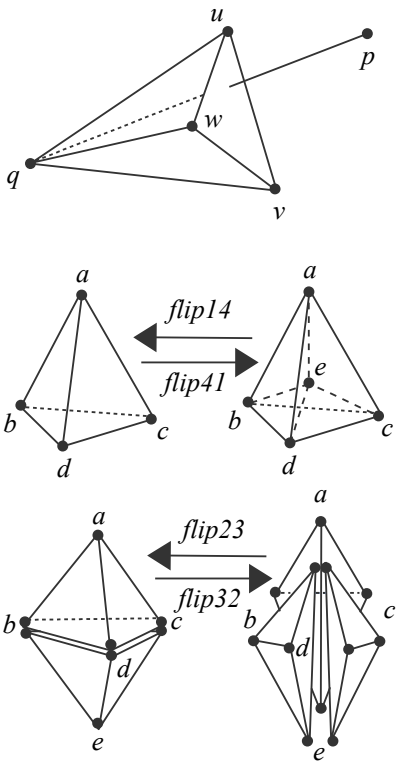


Illustration 30: Opérations géométriques utilisées dans la construction de la triangulation de Delaunay

Vient ensuite la phase itérative qui ne se termine que lorsque la pile est vide. A chaque itération le premier élément T_A (constitué des points *p, a , b, c*) est dépilé et testé avec son tétraèdre adjacent T_B(*a*, *b*, *c*, *d*). Le test consiste à vérifier si le point *d* de T_B est inscrit dans la sphère circonscrite de T_A. Si le point *d* est à l'intérieur alors un flip sur T_B est nécessaire. Il existe différents flips en fonction des configurations possibles entre T_A et T_B:

1. Si une seule face de T_B est visible, un *flip23* est effectué.
2. Si deux faces de T_B sont visibles, que l'union de T_A et de T_B n'est pas convexe et qu'il existe un troisième tétraèdre T_C(formé de *a,b,p,d*) appartenant à *DT* qui partage *ab* avec T_A et T_B alors un *flip32* est effectué.
3. Si deux faces de T_B sont visibles, que l'union de T_A et de T_B n'est pas convexe mais qu'il n'existe pas de tétraèdre tel que dans le Cas 2, alors un *flip44* est effectué.

A la fin de chacun des différents flips, les nouveaux tétraèdres formés sont ajoutés à la pile pour être testés avec leur voisins. Lorsque la pile ne contient plus d'élément, la triangulation de Delaunay est formée, on passe alors à l'extraction du diagramme de Voronoï.

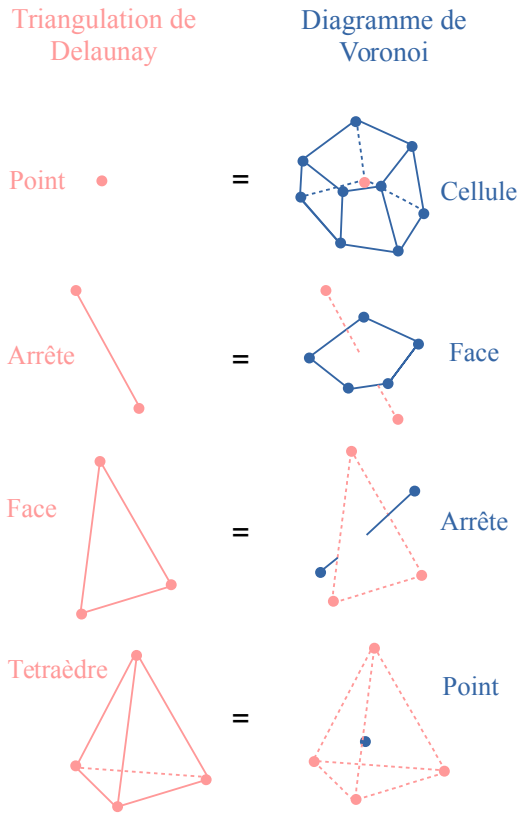


Illustration 31: Table de correspondant entre la triangulation de Delaunay et le diagramme de Voronoï

En mathématiques, pour un ensemble de points donnés, la triangulation de Delaunay forme un graphe dit dual du diagramme de Voronoï pour le même ensemble de points. Il est donc possible de passer de l'un à l'autre via des propriétés de correspondance. Dans le cas de l'outil de fracture il s'agit de trouver le diagramme correspondant à la triangulation. La table de correspondance illustrée par le schéma 31 montre les relations permettant d'extraire les différentes informations. Dans la triangulation un point correspond à une cellule de Voronoï, une arête à une face, une face à une arête et un tétraèdre à un vertex de Voronoï. CGAL permet de trouver les structure de face, d'arête et de point de dual à la triangulation. Cependant il a été nécessaire de mettre en place l'algorithme récursif permettant de construire les cellules à partir des arêtes et points extraits ainsi qu'une structure pour les stocker. Le concept est le suivant: pour chacun des points de Delaunay tout les tétraèdres adjacents sont trouvés et leurs points duaux de Voronoï sont stockés dans la cellule.

Lors de l'extraction des cellules, un plan de la forme $ax+by+cz=0$ est calculé pour chacune des faces de Voronoï puis stocké, ces plans vont être directement utilisés pour effectuer la découpe des fragments (Plus de détails seront donnés dans la prochaine section).

Le diagramme de Voronoi étant construit, il est directement affiché à l'utilisateur. A chaque fois que ce dernier modifiera les points de contrôles, le diagramme sera modifié en conséquence en suivant les étapes définies au dessus.

2.2.2 Interface, construction à l'aide de points de contrôles dans l'éditeur

J'ai développé l'éditeur de pattern afin de faciliter le travail de l'artiste, il permet de créer la forme de la fracture. Dans l'Unreal Engine 4 pour créer un pattern de fracture il faut créer un blueprint héritant de la classe *FPattern* incluse dans mon plugin. Il suffit ensuite d'ouvrir ce *blueprint* pour éditer le pattern.

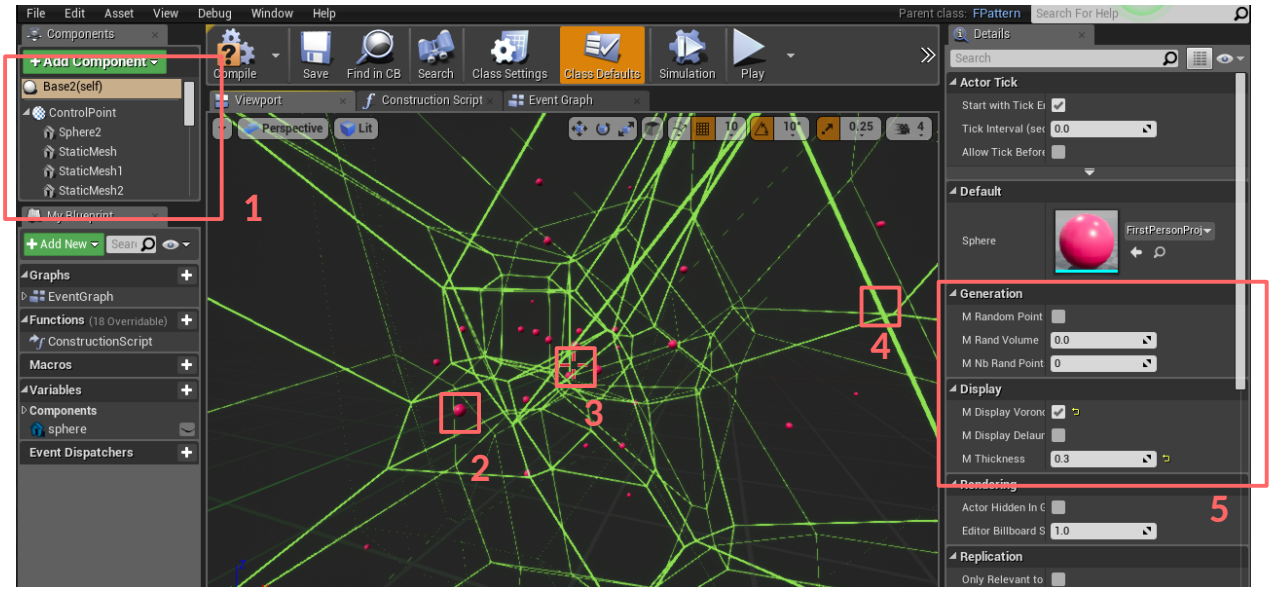


Illustration 32: Capture d'écran de l'éditeur de Pattern dans Unreal

La capture d'écran ci-dessus (illustration 32) illustre les diverses fonctionnalités que j'ai développées lors de la conception de l'éditeur de pattern. *L'Outliner* (Encadrement 1) liste tous les points de contrôle du pattern mais permet également d'en ajouter, dupliquer ou encore supprimer. L'encadré 2 montre un point de contrôle; sa taille, apparence et position peuvent être facilement modifié dans la vue 3D (aussi appelé le *viewport*) via les *gizmos* de contrôle. Le diagramme de Voronoï quant à lui est dessiné en vert (encadré 4), lorsque l'artiste déplace un point, les cellules sont actualisées fluidement (Voir annexe vidéo). Différentes options de générations aléatoires sont disponibles dans le panneau des propriétés (Encadré 5 et agrandissement ci-contre) : l'artiste peut influencer le volume d'échantillonnage aléatoire ainsi que le nombre de points souhaités. Il est possible d'afficher ou non la triangulation (voir illustration 33) et les diagrammes ainsi que de modifier la largeur du trait dessinant ces derniers pour plus de lisibilité.

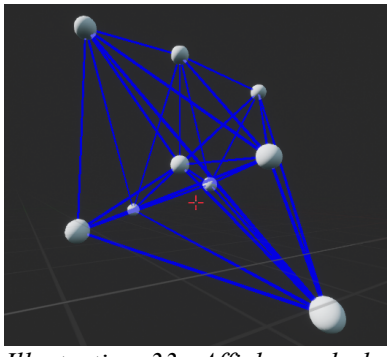


Illustration 33: Affichage de la triangulation dans l'éditeur.

Une croix est marquée à l'origine du repère du *viewport* (encadré 3), il s'agit de l'épicentre de la fracture. Lorsqu'un objet est détruit, ce centre est aligné avec le point de contact à l'origine de la fracture. Cette méthode de création de pattern assure une maîtrise artistique optimale sur la forme de la brisure. Une fois la création du pattern par l'artiste achevée vient l'intégration de l'effet destruction au jeu par le game designer. Dans la prochaine section le fonctionnement la fragmentation en temps réel dans mon outil ainsi que son interfaçage via le système de *blueprint* seront expliqués.

2.3 Fragmentation

La fragmentation est l'étape clé du pipeline de fracture que j'ai développé, l'objet ciblé par le joueur va se briser en fragments en temps réel. Cette phase peut être divisée en plusieurs aspects : le calcul des fragments, les effets supplémentaires qu'il est possible d'ajouter et l'intégration de ces fonctionnalités dans le moteur de jeu. Toutes les étapes décrites ci-dessous se déroulent en temps réel. Cette notion est très importante et représente l'une des problématiques majeures à laquelle j'ai fait face durant la conception de l'outil.

2.3.1 Fracture

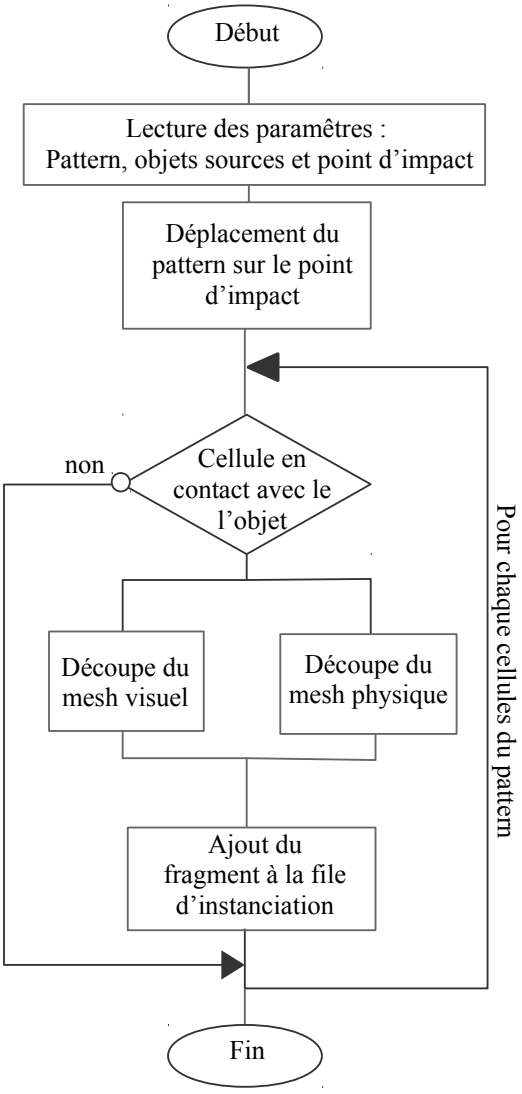


Illustration 34: Procédure de fragmentation mise en place dans mon plugin de fracture.

L'étape du calcul des fragments suit l'algorithme illustré par le schéma 34. Dans un premier temps le **pattern de fracture est aligné** avec le point d'impact, certaines rotations peuvent être mises en place afin de donner un caractère aléatoire à la fracture. Cette opération permet de fracturer l'objet en fonction du point d'impact et ainsi de changer la densité des fragments en fonction de la distance de l'impact.

Une fois le schéma de fracture positionné correctement, un **test d'intersection** a lieu. Un peu à l'image de la broad-phase définie en 1.2.1.3, il permet de ne calculer que les fragments nécessaires. L'intersection de la bounding boxe de l'objet est testée avec chaque cellule du pattern (qui est aussi convexe) en calculant la distance entre le centre bounding box et chaque plan de la cellule. Si la distance est inférieure au diamètre de la bounding box, alors le test est positif, le fragment correspondant à l'intersection de cette cellule et de l'objet est ensuite calculé. Dans la première version du plugin ce test n'était pas effectué, de nombreux objets vides surchargeaient la scène, impactant directement les performances.

La découpe des meshes visuels et physiques est ensuite effectuée en fonction des cellules sélectionnées pour créer un fragment.

2.3.1.1 Traitement du mesh visuel

La **découpe du mesh** visuel est assez simple. Pour former un fragment l'objet source est découpé par chacun des plans de la cellule correspondante du pattern de fracture, cela revient à calculer individuellement l'intersection de chaque cellule avec l'objet à détruire. L'illustration ci-dessous montre le processus pour un fragment: sur la figure de gauche, on peut observer en gris le maillage de la cellule du pattern, en rouge le fragment correspondant. L'objet est découpé selon chacun des plans de la cellule, en bleu sont figurées les nouvelles faces créées à chaque itération. Chaque découpe visible ci-dessous (figure 35) s'effectue en deux étapes, la découpe du mesh et le remplissage du trou laissé par la découpe.

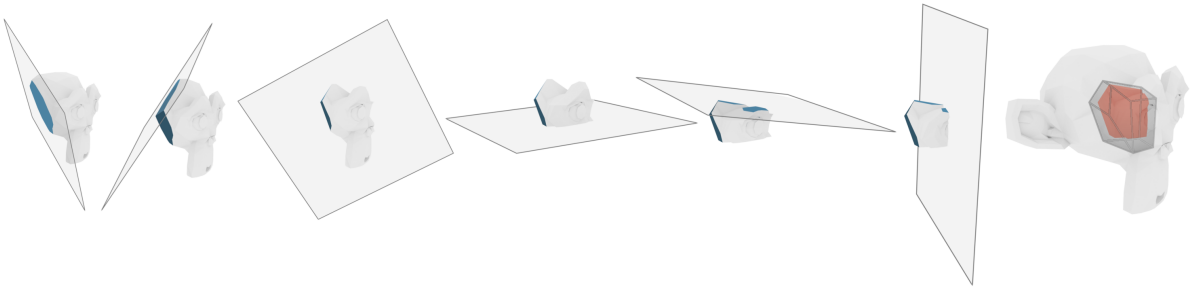


Illustration 35: Découpe d'un fragment du mesh correspondant à la cellules dessinée en wireframe à droite

La phase de découpe de l'objet par un plan se décompose en plusieurs étapes :

1. **La Construction du Vertex Buffer**[27] met en place les *vertex* du fragment en testant la distance de chacun des points du mesh source par rapport au plan de découpe donné en entrée (voir illustration 36.1). Si la distance trouvé est supérieure à 0, le point est ajouté au Vertex Buffer utilisé pour le nouveau fragment.
2. La **Construction de l'Index Buffer**[28] met en place les triangles du fragment en testant si les triangles de l'objet source ont « survécu » à la création du vertex buffer et peuvent être transférés dans le nouvel index buffer du fragment, c'est-à-dire si leurs points sont toujours présents dans le nouveau vertex buffer(voir illustration 36.2). Si oui, le triangle est ajouté au nouvel index. Dans le cas contraire :

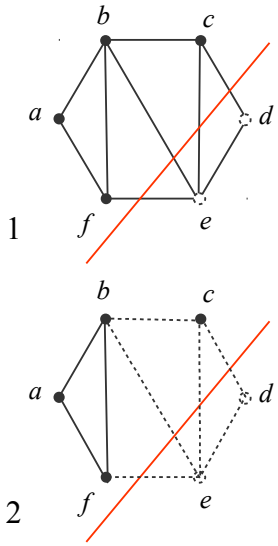


Illustration 36: Construction du vertex buffer (1), les points d et e ne survivent pas. Construction de l'index buffer (2), les triangles bef, bce et cde ne survivent pas.

27 Vertex Buffer : structure stockant les coordonnées de points.

- Si aucun point n'est présent, le triangle n'est pas ajouté au nouvel index buffer.
- Si un ou deux points sont présents on ajoute ces points à l'index buffer et le triangle est clippé au plan en créant un (cgh sur la figure 37.3) ou deux nouveaux triangles en suivant ce processus pour les points restant du triangle de base :
  - Si le point est valide, il est ajouté directement à l'index buffer.
  - Si le point n'est pas valide, un nouveau point est ajouté, ses coordonnées résultent d'une interpolation linéaire entre la position d'un des points valides et de sa propre position. Ce nouveau point est ensuite ajouté à l'index buffer. Sur la figure 10.3, les points e_1, e_2 résultent de l'interpolation de e sur fe, be et ce ; d_1 résulte de l'interpolation de d sur cd.

2. Le **remplissage des trous** laissés par la construction de l'index buffer permet de former de nouveaux polygones visualisant l'intérieur de l'objet. Les vertex présents sur les bordures du plan de coupe sont tout d'abord projetés en 2D permettant ainsi de générer facilement un polygone les incluant tous. Dans le cas de la figure 37.*1b* ce sont les point c_1, c_2,d_1 et d_2 qui forment le trou. Les coordonnés UV des points sont générés de manière planaire en fonction de leurs coordonnées x et y. Le polygone formé est ensuite reprojeté en 3D pour être triangulé au moyen de la méthode des oreilles (figure 37.2b) :

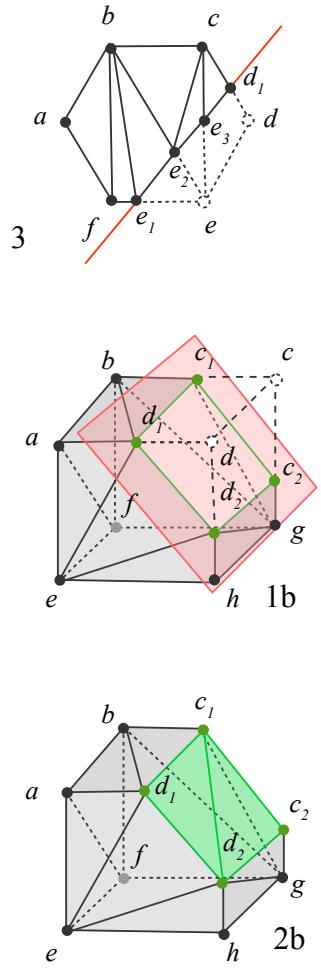


Illustration 37: Opérations géométriques (suite) effectuées lors la découpe du polygone abcdefg en 2D (3) puis sur le cube abcgefgh en 3D (1b,2b).

   1. En suivant la liste des points composant le polygone, on prend les trois premiers pour former un triangle (d_1, d_2 et c_1 par exemple) .
   2. Le convexité du triangle est vérifiée en effectuant un produit en croix des vecteurs d_1d_2 et d_1c_1: si le déterminant est positif, l'algorithme poursuit à l'étape 3.

28 Index Buffer : structure stockant les index de points, permettant ainsi de créer des primitives.

3. Tout les points ne formant pas ce triangle font l'objet d'un test permettant de déterminer s'ils sont localisés dans le triangle. Deux cas sont alors possibles : des points sont dedans, on retourne à l'étape 0 en se déplaçant de un parmi le tableau de points ; si aucun point n'est dedans, on peut alors passer à l'étape 4.
4. Le triangle est ajouté à l'index buffer du fragment et les points le constituant sont retirés de la liste de points du polygone en cours de triangulation. On repasse ensuite à l'étape 1 jusqu'à ce qu'il n'y ait plus de points sur le polygone.

Lorsque le maillage visuel est calculé, il n'est pas encore apte à être instancié dans la simulation car il n'a pas encore d'enveloppe définissant son comportement physique .

2.3.1.2 Traitement du mesh physique

Comme expliqué dans la partie 1.2.1.1, dans les moteurs de jeux modernes, les objets complexes ont un mesh physique très simplifié par rapport au visuel. Comme le montre l'illustration 38 avec Suzanne, les objets concaves sont décomposés en un ensemble de convexes. C'est aussi le cas des fragments produits lors de la fracture. Dans Unreal, ces convexes sont générés à partir d'un set de plans. Les outils déjà en place sont performants, de ce fait le plugin se base dessus pour la génération du mesh de collision. Pour se faire, les plans contenus dans la cellule du pattern sont ajoutés à ceux du mesh visuel existant et le convexe est recalculé en prenant en compte les nouveaux élément. La génération d'un convexe à partir d'un ensemble de plan de déroule en suivant cette procédure pour chacun des plans dans le set :

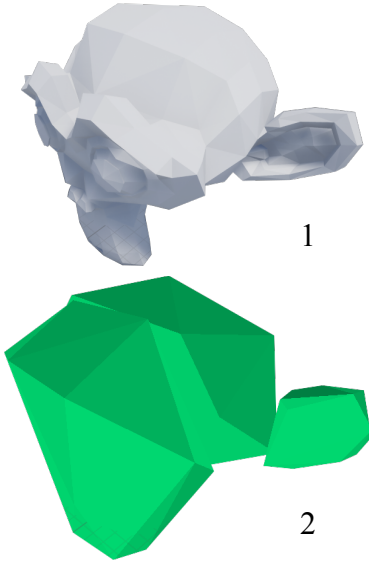


Illustration 38: Comparaison entre un mesh visuel (1) et ses convexe physiques correspondants (2)

1. Un polygone à quatre points est formé, ses points sont les sommets du plan.
2. Ce nouveau polygone est ensuite intersecté avec chacun des plan du set :
   - Un test de distance avec le plan est effectué avec chacun des points, ils sont libellés en fonction de leur côté du plan.
   - Les segments constitués des points labellisé sont coupés en effectuant leur intersection avec le plan, créant ainsi un polygone clippé.

Lorsque les convexes de collisions ont étés calculés, l'objet est ajouté à la pile d'instanciation.

2.3.1.3 Optimisation

Une fois chaque découpe effectuée le fragment est formé et ajouté à la liste des éclats en attente d'instanciation. L'utilité de cette pile a volontairement été cachée jusqu'à maintenant afin d'être développée ici. Au premier abord il serait en effet plus logique d'instancier le fragment juste après l'avoir calculé, c'est ce qui ce déroulait dans la première version du plugin de fracture. Cependant, l'inconvénient est apparu lors des premiers tests avec des objets constitués de nombreux points : le temps de calcul du rendu des images augmentait au-dessus de la limite acceptable des 20 millisecondes. Deux facteurs sont à l'origine de cela :

- Le nombre élevé de triangles à parcourir pour découper l'objet et calculer les fragments augmentent indéniablement le temps de calcul des images.
- Le fait que toute la procédure se déroule sur le thread principal du moteur bloque le calcul du rendu.

Le second facteur est le plus rapide à traiter dans l'immédiat grâce au multithreading (Je ferai des remarques sur l'amélioration du premier facteur dans la partie 3) . Le but est de détourner tous les calculs géométriques sur un autre thread afin de bloquer le thread de rendu. Le point critique lorsque l'on parle d'un algorithme « mutli-threadé » réside dans la gestion des ressources partagées : comment assurer le partage asynchrone des données ?

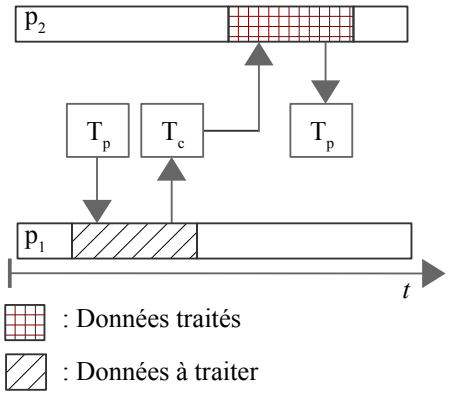


Illustration 39: Fonctionnement du système de pile, T_p représente le thread principal et T_c le thread de calcul.

La technique employée par O'Brien et al [15] pour Star Wars : Forces Unleashed vue en 1.3.1 a directement inspiré celle que j'ai développée dans le plugin, basé sur un système de pile: deux piles sont mises en places, l'une contenant les données pour la production, l'autre contenant les données produites. De cette manière, le thread principal du jeu va nourrir la première pile et se servir dans la seconde tandis que les thread de calcul géométriques se serviront dans la première et produiront dans la deuxième. On dit que le thread du jeu est **producteur** de la pile 1 et **consommateur** de la pile 2 tandis que c'est l'inverse pour les threads de calcul.

Le schéma 39 illustre le comportement des différents threads lorsque le joueur va déclencher une fracture. Dans une premier temps, le thread du jeu va pacquer les données nécessaires à la fracture et placer un pointeur dans la pile 1. Le thread de calcul surveillant en permanence cette pile en quête de nouveaux fragments à calculer va directement récupérer les données de la pile 1 pour les traiter. Lorsqu'il a terminé, il ajoute ses nouvelles données (Contenant les nouveaux fragments) dans la pile d'instanciation que le thread principal du jeu

surveille constamment. Le thread principal va donc immédiatement dépiler les données relatives aux nouveaux fragments afin de les instancier.

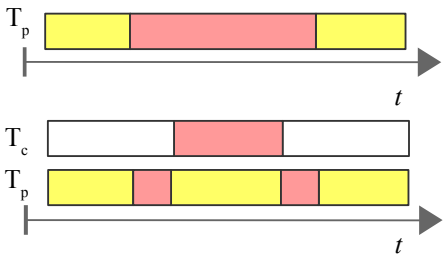


Illustration 40: Différences de charge de calcul géométriques (en rouge) entre l'outil mono-threadé (haut) et multi-threadé (bas); en jaune les tâches habituelles du thread principal.

En procédant ainsi, le thread principal n'est pas bloqué lors du calcul des fragments, le joueur ne ressent donc aucun *input lag* : son immersion n'est pas impactée. Le schéma 40 illustre ce gain, en haut tous les calculs géométriques se font sur T_p tandis qu'en bas une grande partie des calculs géométriques sont déportés sur T_c laissant ainsi T_p continuer ses calculs relatifs au jeu. Le graphe 41 illustre les différents temps de rendu/image du thread principal. On constate qu'avec le multithreading ce temps reste stable (env *20 ms*) tandis qu'en mono-thread il grimpe très vite dans des valeurs bien au dessus de l'acceptable (> *30 ms*) .

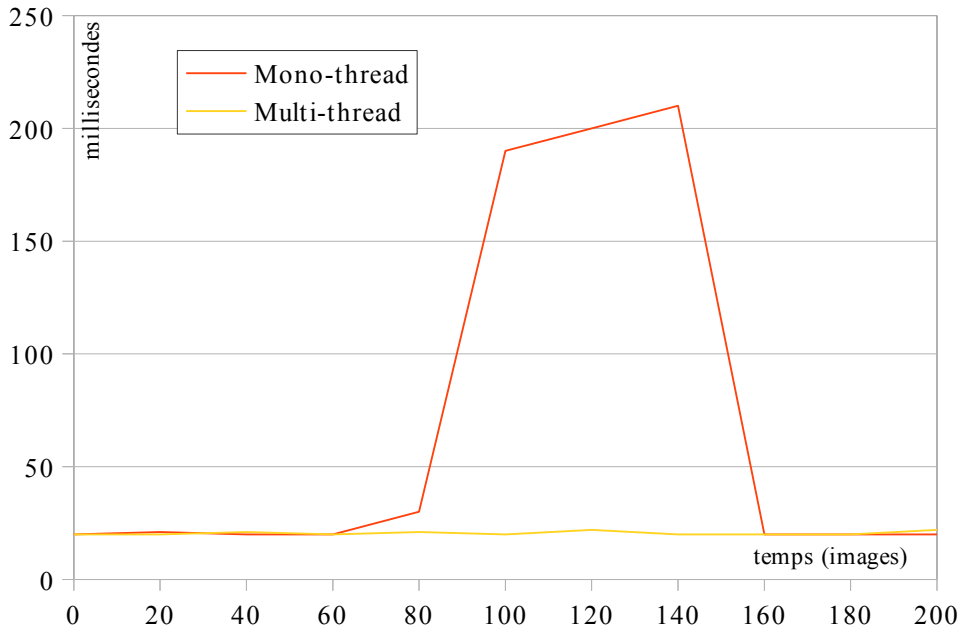


Illustration 41: Variation du temps de rendu des images dans le cadre de la fracture de l'objet présenté en 1.2.1 (illustration 12) sur un i7 4770 avec une gtx geforce 760. En rouge les performance de l'algorithme mono-thread, en vert multi-thread.

Bien que cette méthode paraisse avoir de nombreux avantages, elle ne permet de résoudre qu'en partie le problème. Bien que tout les calculs relatifs aux fragments soient déportés sur un autre thread, le gain est limité :l'accès aux données du mesh stockés sur le GPU depuis le processeur coûte cher en terme de performance. La dernière partie fera état des possibilités de passer le maximum de calcul sur la carte graphique .

La fracture ayant été calculée, l'instancier brut nuirait à sa vraisemblance, il est nécessaire d'y ajouter certains effets visuels supplémentaires.

2.3.2 Effets supplémentaires

L'architecture du plugin de fracture donne la possibilité d'ajouter aisément différents effets visuels, sonores ou encore haptiques via les blueprints. Lorsque l'événement de la fracture est appelé, un *callback* peut être mis en place afin d'instancier les différents effets. La raison d'être de ces effets provient du besoin de vraisemblance du joueur. Afin d'être crédible une fracture a besoin de plus qu'une suite de triangles définissant des fragments, elle nécessite de la fumée, des sons d'impacts ou encore des explosions.

Durant la mise en place du plugin, j'ai tenté d'ajouter différents effets de particules afin d'exagérer la fracture et de la rendre spectaculaire. L'illustration 42 montre trois destructions d'un

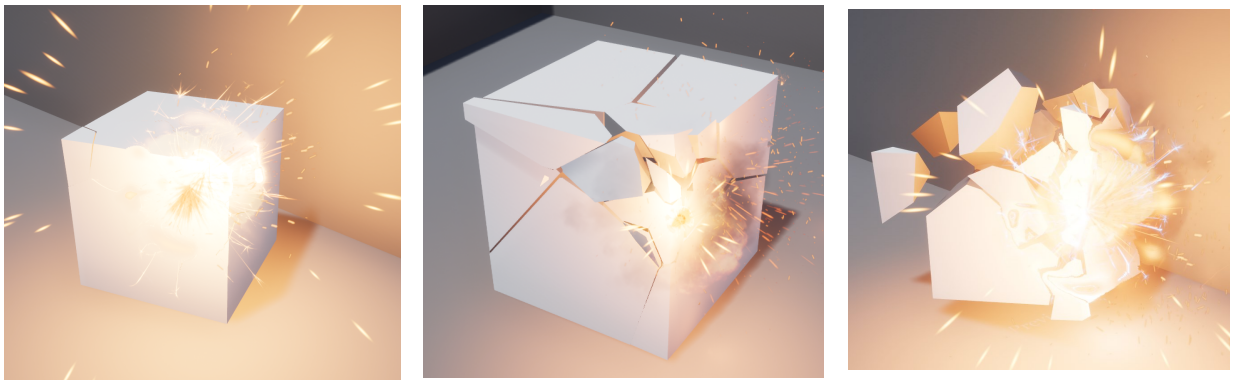


Illustration 42: Fracture avec le plugin, ajout d'un système de particules au moment de l'impact

cube avec l'ajout d'effets de particules simulant l'explosion du projectile heurtant l'objet fracturé, les vidéos correspondantes se trouvent sur le DVD en annexe. Les effets sonores ont également une part extrêmement importante pour le *feedback* du joueur, ils peuvent être déclenchés au même moment que les effets visuels via le système de *blueprints* d'Unreal.

2.3.3 Interface, accès avec les blueprints

Mon système expliqué en 2.4.1 et 2.4.2 ne pourrait pas être fonctionnel sans interface pour permettre à l'artiste ou au game designer de le paramétrer. L'Unreal Engine 4 bénéficie d'un système de programmation nodale idéal pour faire le pont entre l'outil que j'ai développé en C++ et l'utilisateur : le blueprint. J'ai donc exposé les différentes fonctionnalités de mon plugin directement par les blueprints. La capture ci-contre montre la fonction principale permettant de lancer la destruction d'un objet. Elle prend en paramètre l'objet à fracturer, le point d'impact, le pattern de fracture appliqué et le matériau définissant l'apparence intérieure de l'objet. Cette fonction va envoyer les données au thread de calcul, une deuxième procédure (Voir figure 43) permet ensuite de récupérer un tableau de référence aux fragments instanciés et de définir leurs comportements individuellement ou par objet. FractureProceduralMesh et Fwatch sont les deux seules fonctions nécessaires à la création de la destruction procédurale, rendant ainsi sa mise en place très simple.

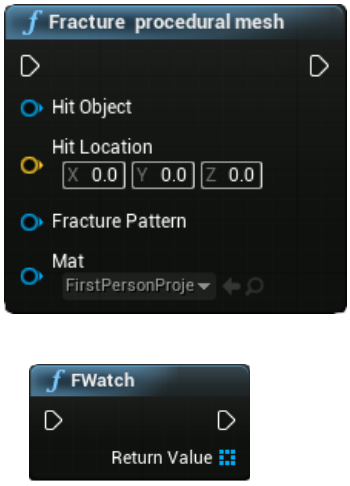


Illustration 43: Fonctions blueprints d'interfaçage de la fracture.

Cette interface résulte d'une réflexion sur le l'équilibre nécessaire pour obtenir un outil simple d'utilisation sans enlever de possibilités à l'artiste. Trop de nœuds aurait rendu le paramétrage trop fastidieux tandis que pas assez retirerait tout le contrôle.

2.4 Expérimentation et application

Illustration 44: Bannière de Finding Paztec

Après le développement du prototype du plugin de fracture, il a été mis à l'épreuve lors de projets divers. L'expérience de Paztec à permis d'adapter l'outil au besoins réels d'une production de jeux vidéo sur de nombreux aspects tel que les performances, l'expérience utilisateur, etc. Cette mise en situation a également permis de pousser l'outil à ses limites et de trouver des pistes d'explorations pour de futurs développements.

Lors de ce projet, notre groupe était composé de Laure Le Sidaner, Samuel Bernou et moi-même. Au commencement, nous savions juste que nous souhaitions faire un jeu vidéo en relation avec nos trois problématiques de mémoire, à savoir les suivantes :

- La création de personnages en se basant sur des formes géométriques (Laure)
- L'autoréférence (Samuel)
- La destruction en temps réel (moi-même)

2.4.1 Presentation du projet

Avant même de savoir ce que nous allions faire comme jeu, nous connaissions nos rôles respectifs, Laure étant en charge de la direction artistique (aussi bien des décors que des personnages) , Samuel en charge de l'animation, du son et du scénario (car en étroite relation avec son sujet de mémoire) et moi-même en charge du développement, du pipeline.

Notre première difficulté fut de trouver un concept de jeu qui se détache un peu de ce qui se fait traditionnellement tout en ne partant pas dans l'extrême opposé, à savoir un jeu ultra expérimental que personne ne comprendrait. Nous avons mis approximativement une semaine à trouver ce que nous voulions faire : un jeu à l'envers. Plutôt que d'incarner des aventuriers "classiques", le joueur incarnera la statuette du niveau final qui se lasse que les joueurs n'arrivent jamais à la libérer et prend la décision de s'émanciper afin de sortir du temple qui la maintient prisonnière.

Ayant le scénario autour duquel le sujet de Samuel s'articule parfaitement, Laure a pu débuter les recherches graphiques. Cependant, une grande question restait sans réponse, en quoi consisterait le gameplay ?

Sur le gameplay, nous voulions au moins intégrer la fracture en temps réel comme un élément principal pour rentrer en phase avec ma problématique de mémoire. Cependant, il était risqué pour moi de ne construire le jeu que sur cette mécanique pour la simple raison que mon plugin de fracture n'était pas encore prêt. Il y aurait donc beaucoup de développement à faire en parallèle de celui du jeu en lui-même.

Au départ, il était question de partir sur un puzzle game dont les énigmes auraient été basées sur la fracture mais après quelques jours de réflexions et beaucoup de casses têtes nous nous sommes rendus compte qu'il était très compliqué de réaliser des niveaux puzzle "non bateau" en se basant sur la destruction. Finalement nous allions tout de même tenter de mettre la fracture au premier plan d'un point de vue gameplay en y ajoutant un système de "blocs" ayant différentes propriétés tel que celle de propulser le joueur en l'air afin d'ajouter de l'agilité, partant ainsi plutôt sur un *platformer*[29].

29 Platformer : jeu de plateformes.

2.4.2 Production

Ayant notre concept et le gameplay définis une semaine après le début de l'intensif, nous pouvions enfin passer à l'étape critique pour le projet: la production du jeu. A partir de là, un planning à été défini avec les *deadlines* des principales étapes de production, à savoir:

- La conception des niveaux,
- La conceptions des personnages,
- L'animation,
- La réalisation des sons,
- Le développement.

Le choix des technologies s'est imposé de lui même, mon plugin de destruction étant conçu pour l'Unreal Engine 4. Blender a été sélectionné pour la modélisation et Unreal pour le moteur de jeu.

Dès le départ, nous avons mis en place un pipeline précis et clair afin de gagner en efficacité. Pour gérer les différentes tâches, un tableau Trello à été mis en place, permettant facilement d'assigner celles-ci à différentes personnes. Au niveau de la gestion des fichiers, j'ai mis en place un repo GIT pour la gestion du projet Unreal et un répertoire Google Drive pour gérer les assets.

Nous avons décidé de séparer la conception en quatre étapes (conseillées par Unreal dans la documentation):

- La conception du level design,
- L'intégration des meshes et assets,
- Le setup des lumières et ajustement colorimétriques,
- Le polissage (ajout de derniers détails et finalisation).

J'ai donc commencé par mettre en place les principales mécaniques de jeu en blueprint (le langage de programmation nodal intégré à unreal) ainsi que créer les niveaux sous forme de cubes basiques tandis que Laure commençait la production des assets et Samuel écrivait les dialogues et concevait le storyboard. Cette étape à été vraiment cruciale car elle nous a permis de prendre du recul sur le jeu et d'enlever tous les éléments superflus afin de garder le noyau même de notre idée de base pour aller droit au but.

Concernant l'intégration de mon système de fracture, j'ai pris la décision de l'intégrer lors de la seconde phase de réalisation du projet tout simplement parce qu'il me restait beaucoup de travail à effectuer dessus mais également pour attendre d'avoir des assets finaux pour le tester dessus .

A la fin de cette première étape de production, nous avions donc un prototype moche mais fonctionnel du jeu contenant toutes les principales fonctionnalités en terme gameplay et de développement (à part la destruction) incluant le contrôleur du joueur, les menus, le système de sauvegarde, les plate-formes de saut ainsi que le level design des différentes salles.

A partir de ce moment là, les assets n'étant pas encore tout à fait prêts, je suis passé sur le développement de l'outil de destruction (détaillé dans la prochaine section). L'intégration des assets nous poussa à nous questionner vis à vis du nombre maximum de polygones à afficher à l'écran. De plus cette problématique impactait aussi le système de destruction que j'avais mis en place, j'en parlerai donc aussi en seconde partie.

Pour chaque salle (ou niveau), après l'intégration des *assets*, je mettais en place le lighting. Comme l'environnement allait être amené à se modifier en temps réel avec la destruction, il n'était pas question d'utiliser le système de lumière statique d'Unreal. J'ai donc recherché des solutions de calcul de lumières dynamiques et retenu la méthode *VXGI* [2] pour sa flexibilité ainsi que le fait qu'il existe une implémentation Unreal faite par NVIDIA. Cela nous a fait gagner beaucoup de temps car nous n'avons pas eu à créer d'UV pour les lightmaps. Cependant j'ai très rapidement dû faire face à de gros soucis d'optimisation: le jeu se mettait à gagner 50ms de rendu par images à certains endroits précis du jeu, le souci venait du radius d'influence trop grand attribué aux lumières.

Après avoir fini l'intégration des assets et du lighting, Samuel avait fini l'animation des cinématiques ainsi que l'enregistrement et la sélection des sons; de ce fait, nous avons débuté la phase de "polissage" à savoir:

- Intégration des sons et cinématiques,
- Ajout des images de menu,
- Ajustement des matériaux,
- Debug.

Au final, malgré le fait que nous soyons partis avec une semaine de retard, notre méthodologie de travail nous a permis de rattraper le temps que nous avions perdu et nous sommes tous les trois parvenus à intégrer notre sujet de mémoire comme élément prépondérant du jeu.

2.4.3 La destruction temps réel dans Finding Paztec

Avant le début du projet intensif, mon prototype de plugin de fracture pour Unreal ne fonctionnait que partiellement car il causait des chutes de FPS du fait qu'il effectuait ses calculs sur le thread principal du jeu. Un des enjeux majeur de cet intensif à été d'optimiser ses performances de manière à ce qu'il ne soit plus bloquant pour le jeu.

Ma principale crainte pour la réalisation de Finding Paztec fut que je n'eus pas assez de temps pour avoir une version stable du module de destruction; avec tout ce qu'il y avait déjà à développer pour le jeu, il était compliqué de travailler à côté pour l'améliorer. Au départ, mes objectifs étaient d'avoir la fracture partielle implémentée dans la version du projet, mais nous avons dû revoir les objectifs à la baisse avec le retard, de ce fait il n'y a que la destruction complète d'objets dans le jeu.

Illustration 45: La déstruction temps réel dans Finding Paztec.

Avoir l'opportunité de travailler sur mon sujet de mémoire durant le projet m'a permis d'approfondir beaucoup de choses concernant le fonctionnement du moteur de jeu en lui-même et surtout sur sa manière de gérer les instances d'objets. Dans un souci d'optimisation, j'ai fait face à un souci majeur lors des premiers essais de destruction dans le projet: les fragments d'objets instanciés n'étaient pas déchargés de la carte graphique lorsqu'ils n'étaient plus affichés tandis que dans le projet où je développais le plugin, ils l'étaient.

Cela était en fait dû à l'utilisation des lumières dynamiques qui généraient des ombres en temps réel, or les fragments étant susceptibles de générer des ombres, ils restaient donc en permanence chargés jusqu'à ce que l'on sorte de la zone d'influence des lumières susceptibles de générer de l'ombres. Deux solutions se sont alors présentées: utiliser une lumière directionnelle qui elle gère

très bien l'occlusion des objets en fonction de leur visibilité même si ces derniers projettent des ombres ou soit désactiver les ombres projetées par les fragments.

Dans le cadre de Finding Paztec j'ai choisi la deuxième option pour la simple est bonne raison que tous les niveaux sont en intérieur, l'utilisation d'une lumière directionnelle n'est tout simplement pas envisageable, de plus les fragments étant illuminés par eux même, l'artifice ne serait pas visible.

La fonctionnalité critique pour le projet est l'ajout du multithreading, sans cela, la destruction en temps réel n'aurait tout simplement pas été envisageable. Du fait que les mesh peuvent parfois atteindre des nombres très élevés de points, rien que le fait de les parcourir afin de les lire peut prendre beaucoup de temps, provoquant inexorablement une importante chute de performance. Aussi, il est important que toute les opérations ne soient pas bloquantes pour le jeu. Pour répondre à ces problématiques j'ai mis en place un système de queue expliqué en 2.3.1.3.

Le projet intensif m'aura donc permis de mettre à l'épreuve le module et de l'adapter aux différentes contraintes du jeu vidéo, me faisant ainsi beaucoup avancer dans son développement. De plus, il m'a également donné différentes pistes d'approfondissement à poursuivre pour la suite tel que la destruction partielle et calcul GPU[30].

30 GPU : Graphics Processing Unit ou carte graphique en français.

3 Travaux futurs

Aussi bien au cours du développement que de l'expérimentation de l'outil, de nombreuses pistes ont vu le jour. Le plugin est fonctionnel cependant il s'agit toujours d'un prototype auquel de nombreuses fonctionnalités peuvent être revues ou ajoutées.

3.1 Fracture partielle

Actuellement, en dehors du cas particulier de Paztec, bien que la taille des fragments varie en fonction de leurs distance au point d'impact, l'outil de fracture détruit l'objet source dans sa totalité. Dans le cas de structures complexes tel que les bâtiments, il est important de mettre en place la fracture partielle. Une destruction partielle consiste à conserver l'intégrité d'une partie de l'objet détruit en fonction de la force et de la localisation de l'impact, nous appellerons cette partie le fragment partiel.

De nombreuse techniques différentes sont envisageables mais la méthode de la sphère circonscrite reste la plus efficace. Différentes modifications sur le pipeline en place dans le plugin sont nécessaires pour la mettre en place :

- Une structure de donnée doit être mise en place pour stocker les nouvelles informations relative à la fracture partielle
- Les fonctions d'interfaçages doivent prendre en compte cette option et la rendre accessible de manière simple et intuitive

Le concept de la sphère circonscrite illustré avec l'image 46 est simple : lors de l'impact sur l'objet détruit, un test supplémentaire est effectué **après** la fragmentation, on vérifie si les fragments sont situés en dehors ou dans une sphère circonscrite *S, S* ayant pour centre le point d'impact (en rouge) et comme diamètre *r* une valeur donnée par le concepteur. Deux cas sont alors envisageables :

- Le fragment testé est dans *S* : il est séparé de l'objet, complètement indépendant lors de la simulation (en vert sur le schéma 46.1).

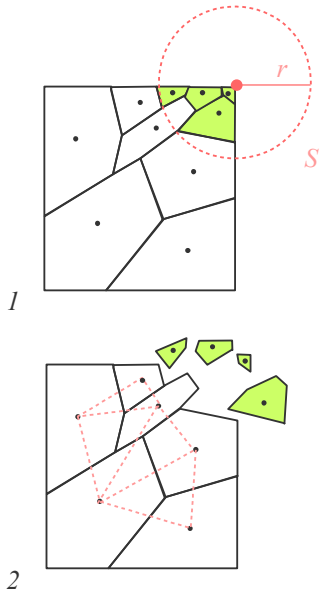


Illustration 46: Représentation de la fracture partielle en 2D sur un carré. En vert les fragments exclus du mesh partiel.

- Le fragment est en dehors de S : ce dernier est ajouté à la structure (voir 46.2 en rouge) stockant la partie partielle de l'objet détruit (en blanc sur le schéma 46).

Effectuer ce test après la création des fragment permet de garder un ensemble de meshs convexes, évitant ainsi les problèmes de collisions.

Un graphe quelconque non orienté est parfaitement approprié pour supporter cette architecture, les nœuds pointeraient vers les fragments et les segments stockeraient les contraintes entre fragments. Dans le cas de la fracture d'un objet solide, ce sont des contraintes rigides (s'apparentant à de la « colle ») qui maintiennent les morceaux entre eux.

Le graphe permet de détruire itérativement le fragment partiel de manière récursive et optimisée. Au moment de l'impact le test de la sphère circonscrite se déroule avant la fragmentation, le graphe est parcouru afin de déterminer les nœuds (ou fragments) inclus dans la sphère (voir 47.1 et 47.2). De cette façon, seuls les nœuds sélectionnés lors du test seront ensuite fragmentés.

La détection d'îlots (voir section 1.3.1) est une nouvelle étape post-fragmentation nécessaire à la formation de multiples fragments partiels. Ce traitement consiste à parcourir les débris en testant chacun pour voir si leur voisins sont connectés ; si la connexion existe, le débris est ajouté à l'îlot. L'exemple de la poutre (voir illustration 47.1b, 47.2b) est le plus parlant pour illustrer ce concept, on peut voir les deux îlots isolés formés (en orange et en bleu). Les îlots formés peuvent être détruits récursivement en suivant la méthode décrite dans les paragraphes précédents.

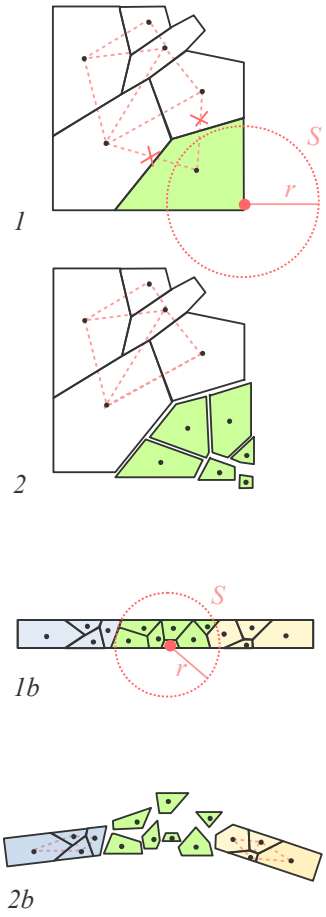


Illustration 47: Étapes de la fracture partielle. Deuxième itération du cube de l'illustration 1(1 et 2). Détection des îlots(1b et 2b).

En terme d'interface, l'utilisateur doit accéder uniquement aux nouveaux paramètres : la possibilité d'activer la fracture partielle et le radius de la sphère circonscrite . Les changements nécessaires dans le plugin actuel figurent dans l'illustration 48 avec l'ajout des paramètres *isPartial* et *radius.*

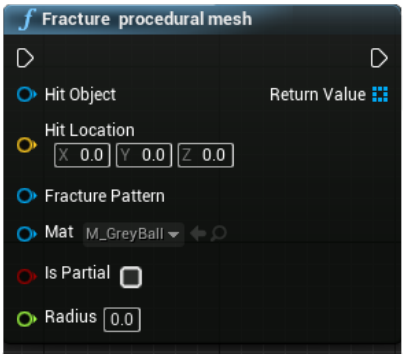


Illustration 48: Fonction blueprint interfaçant la fracture partielle.

La fracture partielle est une fonctionnalité non négligeable du point de vue de la vraisemblance de la

fracture. Elle donne beaucoup plus de crédibilité au comportement de l'effet. Actuellement en cours d'implémentation, elle sera effective dans une prochaine version de mon plugin.

3.2 destruction GPU

Durant la conception de mon *plugin* de fracture, les limites de performance causées par l'exécution des calculs géométriques sur processeur ont motivé ma recherche pour trouver des méthodes alternatives exécutées directement sur la carte graphique. Les deux pistes présentées dans cette partie nécessitent de réaliser une refonte totale de l'architecture de mon module de fracture : la programmation *CPU*[31] et *GPU* diffèrent structurellement énormément. Ces pistes théoriques que j'ai élaborées pourraient différer lors de l'implémentation en fonction des difficultés rencontrés

3.2.1 Fracture d'objet solide basée sur la voxélisation

La déportation des calculs géométriques sur le *GPU* demande une nouvelle architecture et de nouvelles structures de données propres aux cartes graphiques. Actuellement, seul le volume du pattern de fracture est échantillonné (avec les données de l'utilisateur ou aléatoirement), cependant effectuer ce traitement sur l'environnement destructible pourrait avoir de nombreux avantages. Avec la méthode utilisée actuellement, la découpe se fait plan par plan; à chaque découpe, de nombreux calculs ne peuvent se faire que de façon séquentielle. Avoir une décomposition volumétrique de l'objet détruit permettrait d'éviter d'utiliser cette routine lourde en coût de traitement en déduisant directement les fragments de comparaison des volumes de l'objet et du pattern. Les voxels (aussi appelés pixels volumétriques) sont parfaitement désignés pour cette attente.

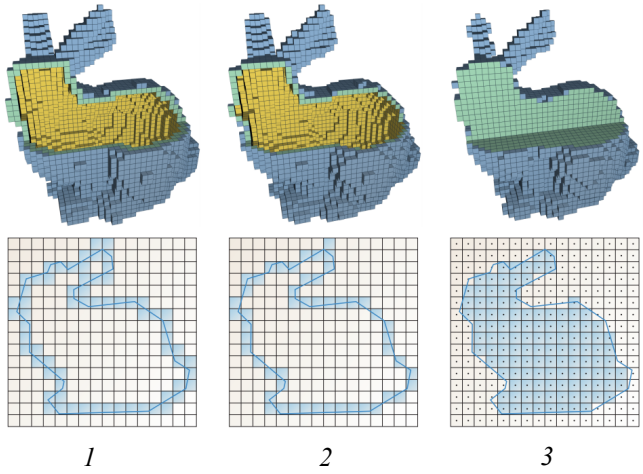


Illustration 49: Différents types de voxelisation, Schwarz and Seidel, "Fast Parallel Surface and Solid Voxelizations on GPUs."

31 CPU : Central Processing Unite ou processeur en français.

Michael Schwarz et al [17] définissent trois principales catégories de voxélisation *GPU* (voir illustration 49), conservatrice (1), séparatrice (2) et solide (3). Étant donné que nous souhaitons connaître le volume de l'objet, la troisième méthode est la plus adaptée. Une voxélisation est dite « solide » lorsque les voxels indiquent s'ils se situent à l'intérieur de l'objet.

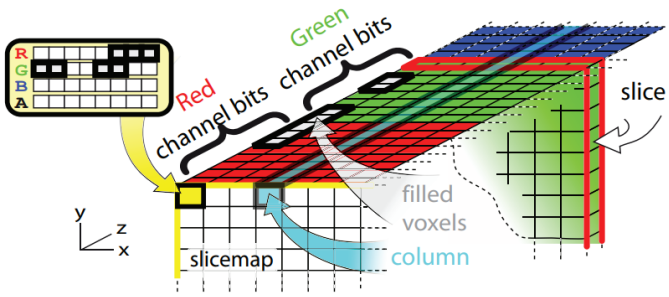


Illustration 50: Représentation de la grille de voxels stockée dans une texture 3D (Eisemann and Décoret, 2008)

Elmar Eisemann et al [3] décrivent une méthode intéressante de voxélisation solide sur *GPU* optimisée pour le temps réel (utilisée dans l'implémentation de VXGI, cf 2.4). Elle se base sur une rastérization[32] conservatrice [21] pour échantillonner le volume . Les Textures2D présentes sur la carte graphique servent de structure pour stocker les informations spatiales de l'objet (les voxels) déduitent de la rastérization. Comme le pointe l'illustration 50, le volume de l'objet est mis en mémoire sous forme d'une grille de bit encodé directement dans la texture. En suivant cette approche, les fragments peuvent être directement formés de l'intersection des volumes respectifs de l'objet et du pattern de fracture.

La complexité de l'algorithme actuellement en place dépend directement du nombre de triangles de l'objet détruit, plus leur nombre est élevé, plus le temps de traitement sera long (il se base sur une représentation en triangle de la scène). La fracture basé sur la voxélisation de la scène s'affranchit de cette complexité en utilisant une représentation voxélisée de la scène, sa complexité dépend de la résolution de la grille de voxels définie par l'utilisateur (et donc adaptative).

Les avantages octroyés par cette méthode sont nombreux :

- La résolution de la fracture serait adaptative, définissable en fonction des configurations des joueurs.
- Le pattern de fracture serait plus personnalisable et pourrait prendre des formes beaucoup plus permissives (autres que diagramme de Voronoï).
- Un gain important de performances générales, le processeur étant déchargé de ses calculs et la puissance de parallélisation de la carte graphique étant pleinement utilisée.

32 Rastérization : transformation d'une scène 3D en 2D afin d'être affichée à l'écran.

- La méthode des CSG (cf 1.2.1.1) pourrait être utilisée pour réaliser la fracture.

3.2.2 Fracture d'objets solides approximée par sphere tracing

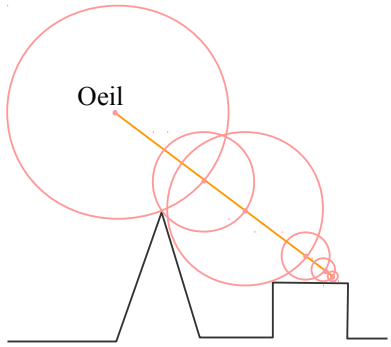


Illustration 51: Représentation du fonctionnement du sphere tracing. A chaques itération, une nouvelle distance est estimée depuis la nouvelle position.

Le *sphere tracing* [7] est une méthode de rendu similaire au *raytracing* qui consiste à calculer itérativement la distance la plus courte entre les rayons lancés et les objets de la scène en avançant à chaque itération jusqu'à un certain seuil de précision (voir illustration 51). Cette dernière permet d'échantillonner des volumes et de rendre des surfaces implicites. C'est Inigo Quilez qui est à l'origine de la démocratisation de cette méthode.

Dans le cadre du plugin de fracture elle serait utilisée afin d'estimer le volume de l'objet et du pattern de fracture. Cette méthode peut être implémenté sur *GPU* en passant par les shaders. Comme le montre l'illustration 52, le principe serait de lancer des rayons aléatoirement en se basant sur la surface de la *bouding box* de l'objet pour définir leurs point de départ. L'intersection entre tous ces rayons et l'objet définiraient une approximation du volume de l'objet, sa définition dépendant directement de la quantité de rayons lancés (Deux résultats à différentes quantités de rayons lancés sont montrés sur l'illustration 7) .

La méthode de stockage sur texture 2D vue en 3.2.1 conviendrait parfaitement pour stocker les données issues de l'échantillonnage. Il serait ainsi possible de définir une résolution de stockage rendant ainsi la méthode plus souple.

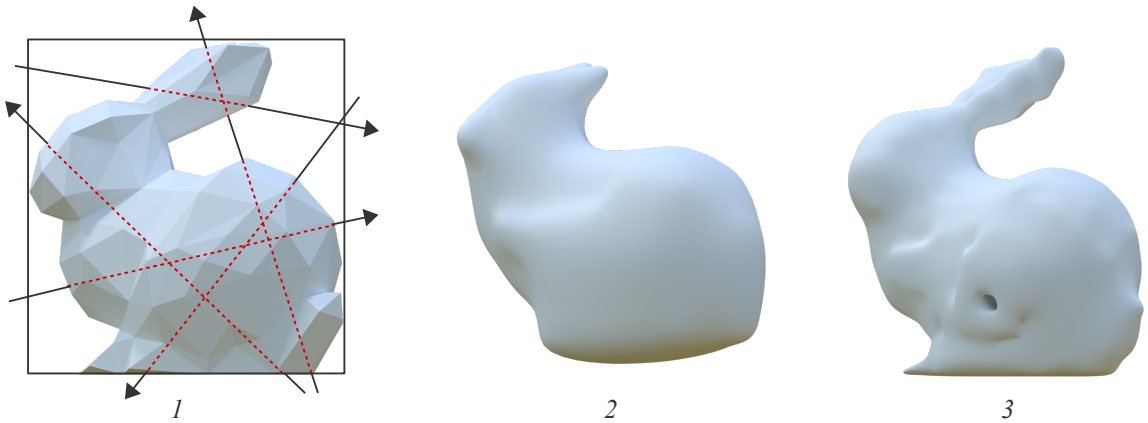


Illustration 52: Représentation en 2D de l'approximation volumétrique de l'objet P par la méthode du sphere tracing. A droite le procédé, à gauche le résultat escompté.1 :Lancé de rayons aléatoire suivant la bounding box de l'objet. 2 et 3 : Deux niveau de détails en fonction de la quantité de rayons.

Le réel avantage amené par l'utilisation du *sphere tracing* comme méthode d'échantillonnage volumétrique réside dans la flexibilité créative qu'elle amène. Le pipeline de création du pattern serait totalement différent dans le sens où il ne serait plus modélisé à la main mais par des fonctions mathématiques définissant le volume du pattern. Ce volume serait ensuite intersecté avec celui de l'objet détruit générant ainsi les fragments. .

Tout comme l'utilisation de la voxélisation expliquée dans la section 3.2.1, la fracture basée sur le s*phere tracing* utilise une représentation volumétrique et non géométrique de la scène. De ce fait sa complexité est indépendante du nombre de triangles de l'objet et rend possible la destruction d'objets très détaillés sans perte de performance.

Cette méthodes ouvrent de nombreuses portes à la destruction procédurale, adaptative et simple dans sa représentation, elle apporterait énormément à mon plugin de fracture. Voici la liste non exhaustive des avantages qu'apporterait cette solution :

- Gains important de performances (Accélération GPU).
- Diversification importantes des patterns de fracture.
- Simplification du pipeline de fracture.
- Utilisation des CSG envisageable (cf 1.2.1.1).

Les pistes présentées donnent matière à poursuivre les travaux que j'ai engagé au cours de la conception de mon outil de fracture temps réel. De nombreuses autres possibilités sont envisageables mais moins pertinentes sur de nombreux aspects. L'ajout des idées présentées à la solution actuellement en place permettrait de découvrir de nouveaux horizons d'interactivité entre le joueur et son environnement en repoussant les limitations techniques des systèmes de destruction actuels.

Conclusion

J'ai présenté, dans ce mémoire, mes travaux sur la destruction d'objets solides temps réel destinée aux expériences interactives (du jeu vidéo aux installations interactives). Mon objectif a été de permettre de réaliser des fractures d'objets solides temps réel avec un large contrôle du pipeline artistique de l'effet. Je présente dans cette conclusion un bilan de mes contributions ainsi que leurs différentes perspectives.

Bilan des contributions

L'étude de l'état de l'art a montré que les méthodes de fracture d'objets solides en temps réel utilisés dans l'industrie du jeu vidéo ne sont pas capables aujourd'hui de gérer l'intégralité du problème de la fracture temps réel en 3D. Les méthodes actuellement en place simplifient le problème en calculant la fracture sur des objets pré-fracturés ou en 2D. Cela a un impact direct sur l'immersion du joueur, sur sa perception et son interactivité avec le monde virtuel l'entourant. Pour permettre une interaction plus complète du joueur avec son environnement, je me suis intéressé aux méthodes de destructions procédurales.

J'ai réalisé un outil de fracture temps réel basé sur des méthodes de décompositions volumétriques procédurales (Voronoï). Ce dernier permet de mettre en place des destructions procédurales temps réel en 3D tout en assurant à l'artiste un haut niveau de contrôle. En testant l'outil lors de divers projets, j'ai pu constaté une meilleure immersion du joueur dans l'interaction avec son environnement.

Perspectives

La destruction d'objets solides en temps réel est un problème qui nécessite encore beaucoup de recherches afin de permettre une destruction à large échelle, artistiquement plus souple et optimisée. Comme proposé dans la partie 3, les différentes problématiques rencontrées lors de la conception de mon outil de fracture pourraient être palliées en optimisant le pipeline de destruction temps réel.

Bibliographie

[1] Cole, Ben. 2011. « Kali: High quality fem destruction in zack snyder's sucker punch ». In ACM SIGGRAPH 2011 Talks, 40. ACM.

[2] Crassin, Cyril, Fabrice Neyret, Miguel Sainz, Simon Green, et Elmar Eisemann. 2011. « Interactive indirect illumination using voxel cone tracing ». In Computer Graphics Forum, 30:1921–1930. Wiley Online Library.

[3] Eisemann, Elmar, et Xavier Décoret. 2008. « Single-pass GPU solid voxelization for real-time applications ». In Proceedings of graphics interface 2008, 73–80. Canadian Information Processing Society.

[4] Eberly, David. "Triangulation by Ear Clipping." Geometric Tools, 2008.

[5] Ericson, Christer. 2005. Real-time collision detection. Morgan Kaufmann series in interactive 3D technology. Amsterdam ; Boston: Elsevier.

[6] Fuchs, Philippe. 1996. Les interfaces de la réalité virtuelle. éditeur AJIIMD.

[7] Fuhrmann, Arnulph, Gerrit Sobotka, et Clemens Gro\s s. 2003. « Distance fields for rapid collision detection in physically based modeling ». In Proceedings of GraphiCon 2003, 58–65.

[8] Hart, John C. s. d.1996 «Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces», Washington State University.

[9] Ledoux, Hugo. 2007. « Computing the 3d Voronoi diagram robustly: An easy explanation ». In Voronoi Diagrams in Science and Engineering, 2007. ISVD'07. 4th International Symposium on, 117–129. IEEE.

[10] Lehericey, François. 2016. "Détection de Collision Par Lancer de Rayon: La Quête de La Performance." Rennes, INSA.

[11] Llamas, Ignacio. 2007. « Real-time voxelization of triangle meshes on the GPU. » In SIGGRAPH Sketches, 18.

[12] Manet, Vincent. 2012. « Méthode des éléments finis ». PNL editor.

[13] Müller, Matthias, Nuttapong Chentanez, et Tae-Yong Kim. 2013. « Real time dynamic fracture with volumetric approximate convex decompositions ». ACM Transactions on Graphics (TOG) 32 (4): 115.

[14] Müller, Matthias, Nuttapong Chentanez, et Miles Macklin. 2016. « Simulating Visual Geometry ». In Proceedings of the 9th International Conference on Motion in Games, 31–38. MIG '16. New York, NY, USA: ACM.

[15] Parker, Eric G., et James F. O'Brien. 2009. « Real-time deformation and fracture in a game environment ». In Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 165–175. ACM.

[16] Sayas, Francisco-Javier. 2008. « A gentle introduction to the Finite Element Method ». University of Delaware.

[17] Schwarz, Michael, et Hans-Peter Seidel. 2010. « Fast parallel surface and solid voxelization on GPUs ». In ACM Transactions on Graphics (TOG), 29:179. ACM.

[18] Su, Jonathan, Craig Schroeder, et Ronald Fedkiw. 2009. « Energy stability and fracture for frame rate rigid body simulations ». In Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 155–164. ACM.

[19] Sylvain Pion, Olivier Devillers, Monique Teillaud.2001.« Walking in a Triangulation ». 2001. In Proceedings of the Seventeenth Annual Symposium on Computational Geometry, 106–114. SCG '01. New York, NY, USA: ACM. doi:10.1145/378583.378643.

[20] Weiler, K. and Atherton, P. (1977) "Hidden surface removal using polygon area sorting", In: Proceedings of the 4th annual conference on Computer graphics and interactive techniques, San Jose, California.

[21] Zhang, Long, Wei Chen, David S. Ebert, et Qunsheng Peng. 2007. « Conservative Voxelization ». The Visual Computer 23 (9-11): 783-92. doi:10.1007/s00371-007-0149-0.

Webographie

[22] « CGAL 4.9 - 3D Triangulations: User Manual ». 2016. http://doc.cgal.org/latest/Triangulation_3/index.html#Chapter_3D_Triangulations.

[23] « Diagramme de Voronoï ». 2017. Wikipédia. https://fr.wikipedia.org/w/index.php?title=Diagramme_de_Vorono%C3%AF&oldid=134363019.

[24] « Framerate et jeux vidéo : combien d'images l'œil perçoit par seconde ? » 2017. . http://hitek.fr/actualite/frame-rate-jeux-videos-oeil_6959.

[25] « GDC Vault - The Art of Destruction in "Rainbow Six: Siege" ». 2017. http://www.gdcvault.com/play/1023003/The-Art-of-Destruction-in.

[26] « Havok Destruction ». 2017. Havok. https://www.havok.com/destruction/.

[27] « Inigo Quilez :: fractals, computer graphics, mathematics, demoscene and more ».2017. http://www.iquilezles.org/www/material/nvscene2008/nvscene2008.htm.

[28] « Pixelux Entertainment ». 2017. http://www.pixelux.com/index.html.

[29] « The Computational Geometry Algorithms Library ». 2016. http://www.cgal.org/index.html.

Filmographie

[30] Bay, Michael. 2014. *Transformers: Age of Extinction*. Action, Adventure, Sci-Fi.

[31] Chaplin, Charles. 1941. *The Great Dictator*. Comedy, Drama, War.

[32] Emmerich, Roland. 2004. *The Day After Tomorrow*. Action, Adventure, Sci-Fi.

[33] Lasseter, John. 1995. *Toy Story*. Animation, Adventure, Comedy.

[34] Lucas, George. 1977. *Star Wars IV : Un nouvel espoir*. Action, Adventure, Fantasy.

[35] Méliès, Georges. N/A. *The India Rubber Head*. Short, Comedy.

[36] Jackson, Peter. 2003. *The Lord of the Rings: The Return of the King*. Adventure, Drama, Fantasy.

[37] Snyder, Zack. 2011. *Sucker Punch*. Action, Fantasy.

[38] Spielberg, Steven. 1993. *Jurassic Park*. Adventure, Sci-Fi, Thriller.

Ludographie

[39] Nishikado, Tomohiro. 1978. *Space Invader.*

[40] Rains, Logg. 1979. Asteroids.

[41] Ubisof. 2015. Tom Clancy's Rainbow Six: Siege.

[42] EA games. 2010. Battlefield: Bad Company 2.

[43] Jumpin' Jack Software. 1995. Ghen War.

[44] THQ. 2009. Red Faction: Guerrilla.

Table des illustrations