Master GIG — Géométrie et Informatique Graphique
Rapport

Développement d'un Simulateur d'inondation de Musées en réalité virtuelle

Lucas Crisci et Edouard Pigot

Aix-Marseille Université

Résumé
Ce TER a pour but de développer un simulateur d'inondation de musées en réalité virtuelle. Un premier scénario a déjà été développé et mis en oeuvre dans le cadre d'un musée générique (but pédagogique). Le sujet proposé concerne la modélisation 3D d'un musée spécifique (avec une partie de ses objets) et la simulation de son inondation suivant différents scénarios (pluie, infiltrations, etc).
Le travail mis en place antérieurement utilise un Asset afin de modéliser l'eau sous la forme d'un plan qui peut monter ou descendre. Ce fonctionnement est assez simpliste et ne permet pas de simuler des scénarios comme des inondations liées à la pluie ou encore de représenter des courants. Il a donc été nécessaire de travailler sur une modélisation assez réaliste mais également peu coûteuse en ressources.

Mots clé : Simulation, inondation, marching cubes, voxels, modélisation géométrique, maillages, Unity.

1. Introduction

1.1. Contexte du problème

En 2007, la Commission européenne a adopté une directive visant à mettre en place des méthodes de travail permettant de réduire les conséquences d'éventuelles inondations sur certains territoires. Dans ce cadre, le ministère de la Transition Écologique et Solidaire ainsi que le Ministère de la Culture soutiennent plusieurs projets visant à sensibiliser chacun d'entre nous à la sauvegarde du patrimoine culturel.

Le projet Musées Résilients aux Inondations (MRI) en fait partie et vise à évaluer l'exposition et la vulnérabilité de certains musées français de l'arc méditerranéen face aux risques d'inondation. Le second objectif du MRI est d'offrir des outils théoriques, méthodologiques, pédagogiques et pratiques aux équipes muséales permettant la mise à jour de leurs Plans de Sauvegarde des Biens Culturels.

1.2. Problématique

L'objectif principal de ce projet est de permettre de représenter plusieurs scénarios d'inondation différents tels qu'une infiltration par le toit (due à la pluie par exemple) ou bien une infiltration par le sol (en montant par le sol). Il faut donc réaliser un modèle de simulation de fluide convainquant.

L'outil créé au final pourra être intégré dans un navigateur ou être exécuté sur des machines possédant des performances moyennes. Il est donc essentiel que les algorithmes mis en place soient suffisamment optimisés pour que leur exécution dans ces conditions soit possible sans perte de performances.

1.3. Organisation du TER

Ce projet a été commencé l'an passé. Il a d'abord fallu que nous récupérions et analysions ce qui avait été fait. Une fois que Monsieur Mercantini nous a envoyé l'archive du projet, nous avons pu commencer à travailler dessus. Afin de contrôler notre avancement et de répondre à nos questions, notre encadrant a organisé chaque semaine une réunion via Skype ce qui a simplifié nos échanges. Nous avons également réalisé des comptes rendus pour apporter plus de précisions que celles données à l'oral durant les réunions.

1.3.1. Projet de base

Le projet commencé l'année dernière utilisait différents Assets de Unity pour remplir certaines contraintes.

Un modèle 3D de bâtiment comprenant un rez-de chaussée, un étage et plusieurs salles d'exposition a été acheté dans la bibliothèque de modèle de Unity. Il a ensuite été complété avec un sous-sol afin de permettre un plus grand nombre de scénarios différents. L'Asset "Gallery - Showroom Environment", dont est issu le bâtiment principal, contenait également différentes oeuvres archéologique ou oeuvres d'art qui ont permis de meubler le musée.

Un autre élément important du projet de l'an passé est l'asset "Dynamic Water Physics". Il permet essentiellement de représenter un plan d'eau avec un aspect visuel réaliste. Cet outil offre également la possibilité de faire flotter des

objets. Dynamic Water Physics ne permet cependant pas de simuler des courants ou la pression ce qui est problématique pour remplir l'objectif du projet, mais le principale problème de cet asset est sa simplicité. En effet, on ne peut que faire apparaître un plan d'eau et le faire monter ou descendre. Il n'est pas possible de simuler un écoulement. Le seul scénario qui peut être représenté est une inondation qui monte du sous-sol de manière uniforme dans toutes les pièces.

1.3.2. Solutions existantes

Unity dispose de nombreux assets permettant la simulation de fluides. Nous pouvons citer entre autres NVIDIA FleX [Ref2] et Obi Fluid [Ref3]. Il existe également de nombreux algorithmes personnalisés qui tentent de décrire la physique des fluides grâce à des particules. On peut facilement trouver des vidéos [Ref6] ou des forums [Ref4] sur ce sujet. Il est également possible de simuler un fluide en utilisant la technique des voxels [Ref5].

Pour générer des modèles 3D de manière efficace à partir d'une grille de calculs, la méthode des Marching Cubes est souvent employée. Certaines vidéos expliquent très bien le fonctionnement de cet algorithme [Ref7], tandis que d'autres donnent des informations très utiles pour se lancer dans le codage de ce type de solution [Ref11].

Notre projet doit représenter le flottement d'objets dans l'eau et ce sujet pousse également à la réflexion chez certains utilisateurs de Unity. Au cours de nos recherches, nous avons trouvé une conférence [Ref1], mais également un forum [Ref13], qui recherchent le meilleurs moyen de représenter un système de flottaison réaliste.

Il existe de nombreux calculs et algorithmes permettant de représenter l'évolution de volumes d'eau de manière réaliste. Ce sujet est traité dans des documents de recherche [Ref10] [Ref12]. On peut aussi trouver assez facilement des cours sur la physique des fluides. Certains sont assez détaillés [Ref14], tandis que d'autres sont plus schématiques [Ref15].

Pour finir, différents moyens d'optimisation ont déjà été réalisés. Nous avons trouvé certaines vidéos proposant des approches pour optimiser un système de simulation de fluides par particules en 2D [Ref8]. La technique des marching cubes avec une simulation par voxels peut également faire l'objet d'améliorations. Le tutoriel que nous avons trouvé au cours de nos recherches se base sur des Marching Squares (équivalent 2D des marching cubes) [Ref9].

Il existe donc un nombre conséquent de projets et de documents sur le même thème que le travail que nous avons dû réaliser. Cependant aucun ne remplissait l'ensemble des contraintes qui nous sont fixées.

1.3.3. Cahier des charges

Afin de répondre à la problématique, 2 points doivent être respectés.

Tout d'abord, un algorithme permettant une représentation réaliste de la physique de l'eau doit être développé. Il faut qu'il puisse représenter différents scénarios d'inondations dues à la pluie et des infiltrations par le sol. Il est également nécessaire que des objets puissent interagir avec l'eau (flottaison, déplacement, destruction). Le modèle physique doit donc également modéliser des forces ainsi qu'un système de pression. Ces éléments permettent la création de courant ainsi que la représentation de l'effet coup de bélier (effet de surpression pouvant causer la destruction d'objets).

Le second point essentiel est l'optimisation du projet. En effet, un système de simulation de fluides en temps réel peut-être très coûteux en ressources. Or, le projet doit pouvoir être exécuté depuis un navigateur ou sur une machine aux performances moyennes. Il est essentiel que les algorithmes et les modèles 3D conçus tiennent compte de cette contrainte.

Pour finir, il est nécessaire de retoucher le modèle 3D du musée déjà présent dans le projet. En effet, il ne possède pas d'enveloppe externe. Il nous a été demandé d'en réaliser une qui correspond au style de l'intérieur du bâtiment.

1.3.4. Répartition des tâches

La première phase du projet consistait en une période de recherches sur le modèle de simulation à employer, que nous avons mené tous les deux. Une fois cette étape terminée, nous avons séparé le projet en 3 tâches principales :

— la simulation de la physique de l'eau,
— la création d'un système d'optimisation,
— la retouche du modèle 3D du musée.

La première a été attribuée à Lucas ainsi que l'intégration avec la deuxième tâche. L'optimisation ainsi que la troisième tâche ont été effectuées par Edouard.

2. Les pistes étudiées

2.1. Modèle de simulation

Nous avons, dans un premier temps, cherché les différentes techniques de simulation de fluide (eau, fumée, objets gélifiés, etc.). L'une des techniques les plus utilisées est la simulation par particules. Il s'agit de simuler des particules (un point dans l'espace qui possède des caractéristiques physiques tel qu'un poids, un rayon, une densité, etc.) d'eau qui interagissent entre elles et avec l'environnement. Le déplacement de chaque particule est calculé grâce aux équations de Navier-Stokes. Ces équations décrivent le comportement de fluides newtoniens (fluides dont la déformation est proportionnelle à la force subie). Nous avons trouvé un asset très complet qui permet de mettre en place ce genre de simulation. Il s'agit du package NVIDIA Flex. Après avoir créé une scène simple avec une boîte pour contenir les particules, nous avons lancé la simulation. Plusieurs problèmes sont apparus immédiatement.

— pour avoir une simulation dans de petites zones ouvertes, cette solution est parfaite. Malheureusement pour simuler de plus grands espaces avec une précision suffisante, il faut augmenter considérablement le nombre de particules maximales et leur durée de vie (elles disparaissent au bout de quelques secondes en général).
— l'augmentation du nombre de particules (environ 10 000) demande plus de ressources matérielles. Sur une de nos machines (i7 6700 HQ (4 coeurs / 8 threads), GTX 1060 6go, 12go DDR4 ; ordinateur portable de

2016 à 1500 euros) la simulation tournait à 7 images par secondes en temps réel.
— la réduction de la durée de vie des particules permet de gagner un peu en performance, mais empêche de simuler le remplissage d'un volume. Il y a une alternative à ce problème que nous expliquerons plus bas.
— pour le remplissage de volumes, un problème au niveau des arrêtes des modèles 3D survient également. En effet, les particules ont tendance (sans doute à cause d'un problème de précision) a passé à travers les modèles 3D, comme une fuite.

Voyant ces différents problèmes (notamment celui des performances), nous avons poursuivi nos recherches sur les méthodes de simulation.

Une méthode prometteuse à nos yeux est la simulation par voxel. Un voxel est un pixel volumétrique, pour faire simple : un point dans un espace 3D. En créant une grille en 3 dimensions de ces points, on change notre point de vue pour la simulation. Au lieu d'avoir un point de vue de l'eau (particules), on a un point de vue de l'espace de simulation. Cette méthode utilise chaque voxel dans l'espace pour calculer les déplacements du fluide avec 2 données importantes : la densité et le flux. La densité permet de savoir si le voxel correspond à de l'eau ou de l'air, et le flux permet de simuler les déplacements du fluide (là aussi, il est possible d'utiliser les équations de Navier-Stokes).

Les avantages principaux des voxels par rapport aux particules sont :

— on ne définit pas un nombre de particules, mais un nombre de points dans l'espace. Ces points sont répartis uniformément (dans un premier temps, voire dans les optimisations possibles) dans l'espace de simulation. Cela signifie que, peu importe le moment dans la simulation, cette donnée ne change pas et n'a pas besoin de changer (contrairement aux particules où l'on peut arriver à bout du nombre de particules disponibles et donc devoir attendre la mort des anciennes ou augmenter leur nombre).
— il n'y a pas de fuite aux bords des modèles, car la simulation est restreinte dans l'espace de voxels.
— de très nombreuses optimisations sont possibles pour augmenter si besoin les performances. Nous détaillerons celle que l'on met en place actuellement et celles qui sont envisageables plus bas.

Après la partie simulation, vient la partie représentation graphique. Pour la méthode par particules, chacune d'entre-elles est remplacée par une metaball qui va fusionner avec ses voisines (voir photo). Dans le package NVIDIA Flex, une option permet d'activer ce rendu.

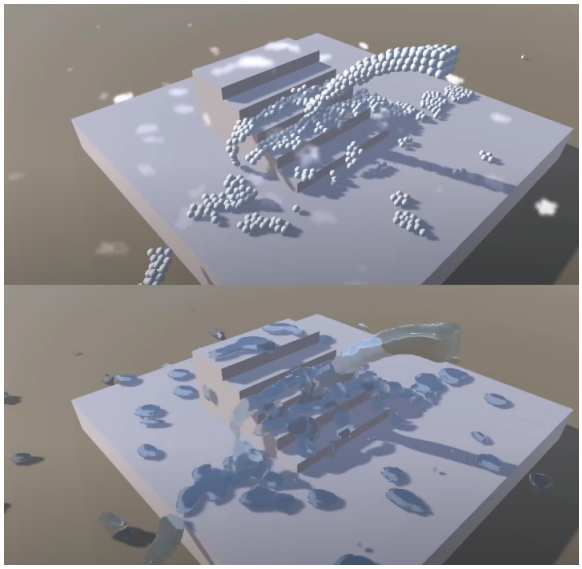

Figure 1: *NVIDIA Flex pour les liquides.*
Les particules en haut et le rendu par metaballs en bas

Pour la méthode par voxel, il faut utiliser les marching cubes. Cette technique parcourt tous les points de l'espace de simulation et va construire un modèle 3D de manière dynamique. L'algorithme parcourt les points 8 par 8 et, en fonction de leur valeurs (eau ou air, soit 1 ou 0), il va construire le maillage pour ce cube. En parcourant tous les cubes, il va alors pouvoir construire bloc par bloc la surface de l'eau.

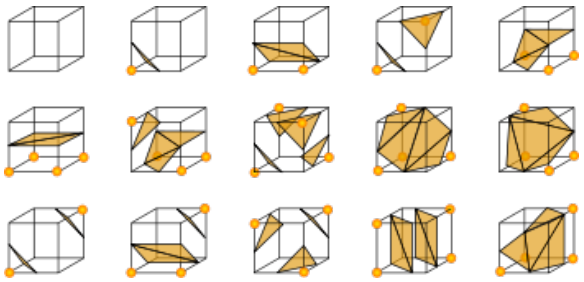

Figure 2: *Les différentes combinaisons de voxels.*
En les assemblant, on obtient la surface de l'eau.

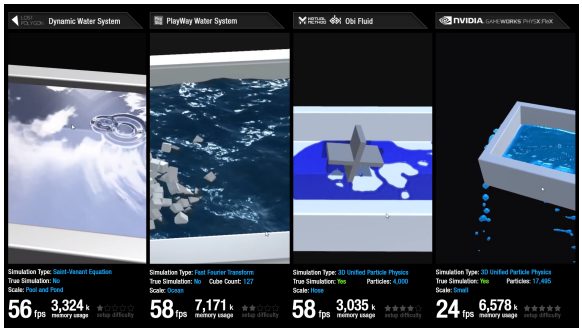

Figure 3: *Les différentes solutions de simulation présentent sur l'asset store de Unity simulé sur une machine très haut de gamme (Xeon E5-2640 v4 (40 coeurs / 80 threads), GTX 1080 8go, 64go RAM).*
On observe les fuites sur la solution NVIDIA Flex.
Source : https ://vimeo.com/221217941

2.2. Méthodes de calculs

Afin de faire évoluer un fluide dans un espace de simulation, il est nécessaire de calculer son évolution. Il existe plusieurs approches pour la simulation de fluides, mais 2 ont attirées notre attention.

L'approche Lagrangienne est très utilisée dans les simulations par particules. Son principe consiste à suivre les différentes particules dans leur déplacement et les faire évoluer en fonction des forces s'exerçant dessus. Différentes équations sont utilisées pour faire varier ces forces en fonction du temps et des interactions directes avec d'autres particules.

Avec une approche Eulérienne, en revanche, on se concentre sur un point fixe de l'espace ou un groupe de points que l'on va faire évoluer en fonction des changements effectués au sein de ce groupe.

L'approche Lagrangienne permet une plus grande liberté, mais est complexe à transcrire dans un fonctionnement par voxel, puisque ceux-ci ne vont pas se déplacer. Une approche Eulérienne semble ainsi plus cohérente. En effet, nos voxel ne se déplaçant pas, on peut leur attribuer des éléments comme un volume et une force. Il est ensuite possible d'en observer un groupe et de faire évoluer les propriétés des voxels, qui en font partie, en fonction de la valeur de ses voisins dans le même groupe. Cette méthode peut cependant être coûteuse en calcul, car elle est régie par des équations complexes. Nous avons donc poursuivi nos recherches et la technique des Height-fields a attiré notre attention. Elle consiste à représenter uniquement la surface d'un fluide à l'aide d'une fonction 2D, ce qui allège énormément les calculs. En effet, plus besoin d'avoir un algorithme parcourant chaque point de l'espace, un parcours par colonne suffit.

3. Travail effectué

3.1. Physique de l'eau

Une fois le fonctionnement par voxel choisi, il nous a fallu définir une méthode de simulation afin de représenter notre liquide. Cette méthode doit permettre de représenter la chute de l'eau ainsi que son mouillage statique (l'étalement de l'eau sur une surface). Le flottement ou non flottement d'objet doit également être représenté. Pour finir, notre méthode doit pouvoir simuler des forces dues aux mouvements de l'eau ou aux chutes d'objets, ainsi que la pression permettant de simuler l'effet coup de bélier.

3.1.1. Les modèles étudiés

Nous avons, dans un premier temps, opté pour une méthode proche de la technique des Height-fields. Chaque point se voit attribuer 2 valeurs : la hauteur du terrain/sol et la hauteur totale (surface de l'eau + sol). Afin de représenter le mouillage statique, l'algorithme parcourt les points de la grille un à un. Pour chacun d'eux, on récupère les niveaux d'eau et de terrain de tous les points entourant le point actuel. L'algorithme additionne ces valeurs avant d'en faire la moyenne. Le résultat obtenu représente la hauteur totale que doit atteindre l'eau dans la case actuelle (donc hauteur de l'eau + hauteur du terrain). Il ne reste plus qu'à y soustraire la hauteur du terrain pour obtenir le nouveau niveau de l'eau dans cette case. Résultat, le niveau de l'eau s'équilibre sur une même hauteur au fur et à mesure des étapes d'exécution.

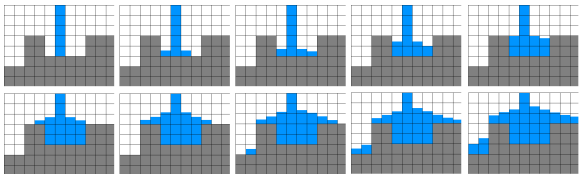

Figure 4: *Représentation du niveau de l'eau à chaque étape de l'exécution.*

Cette méthode est très peu coûteuse en ressources grâce au passage sur une grille 2D, mais c'est également son point faible. Effectivement, le désavantage majeur de cette technique de simulation est le fait que chaque colonne ne peut avoir qu'une seule valeur pour ses 2 attributs (hauteurs sol et eau). Cela implique qu'il ne peut pas y avoir de cavité, où l'eau serait sur un autre niveau (de l'eau à l'étage par exemple). Ainsi, plusieurs scénarios sont impossibles à simuler (escalier, inondation depuis un étage, inondation partielle d'un sous-sol et du rez-de-chaussée juste au-dessus) avec une seule grille.

Pour ne plus avoir cette limitation, il a fallu abandonner le système de grille 2D et faire une simulation au niveau de chaque voxel au lieu de chaque colonne.

La première idée que nous avons exploité correspondait à une technique Lagrangienne. Cette technique est essentiellement réservée à un fonctionnement de l'eau par particules avec des coordonnées linéaires. Or, notre simulation repose sur un système de voxel. Suivre une approche Eulérienne nous a semblé plus cohérent avec les coordonnées fixes de nos voxels. Cette méthode peut être lourde à calculer dans un espace en 3 dimensions. Nous avons donc choisi de mélanger l'approche Eulérienne avec la technique des Height-fields.

Chaque point de notre espace de simulation possède une valeur entre 0 et 1, représentant le volume d'eau sur ce point. Afin d'obtenir un rendu réaliste, 2 calculs sont appliqués en commençant par l'action de la gravité. Durant cette étape, on vérifie pour chaque point de la grille, s'il peut "tomber", ce qui est possible seulement si les conditions suivantes sont vérifiées :

— Le point contient de l'eau (volume > 0)
— Le point en dessous n'est pas du sol
— Le point en dessous n'est pas plein (volume < 1)

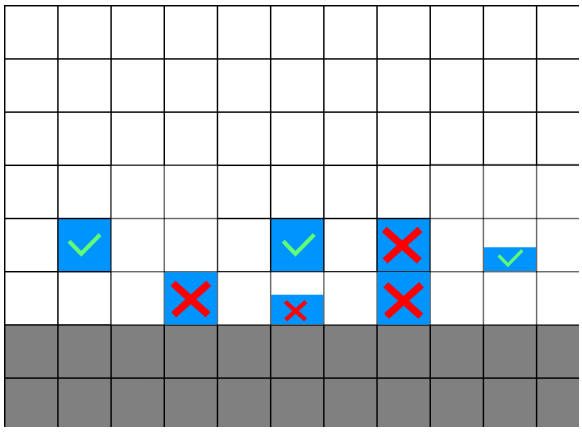

Figure 5: *De l'eau peut s'écouler vers le bas, sauf si elle est marquée d'un X.*

Chaque case peut avoir un volume maximal de 1. Si une case a un volume supérieur à celui manquant dans la case en dessous, seul le volume nécessaire pour remplir la case en dessous sera déplacé. Par exemple, si la case actuelle a un volume de 1 et que la case en dessous a un volume de 0.3, seulement 70 % du volume va descendre. La valeur de la case actuelle va passer de 1 à 0.3 et la case d'en dessous passera de 0.3 à 1. En revanche, si le volume d'eau dans la case actuelle est insuffisante ou juste suffisante pour remplir la case en dessous, tout le volume sera déplacé. Par exemple, si le point actuel a un volume de 0.4 et que celui d'en dessous a un volume de 0.3, le point actuel va se vider et le point en dessous aura un volume de 0.7.

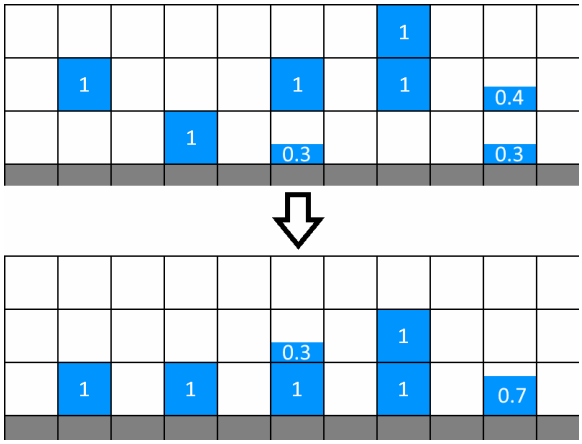

Figure 6: *Passage d'une étape de chute d'eau.*

Tout au long du traitement de ces points, certains sont gardés en mémoire afin de subir un second traitement : le mouillage statique. Comme expliqué précédemment, il peut arriver qu'une case ne soit pas complètement vide après les calculs de chute d'eau. Cela peut également arriver quand le point en dessous est un point de terrain (sol). L'eau restante va se diffuser dans les points autour en faisant une moyenne des volumes.

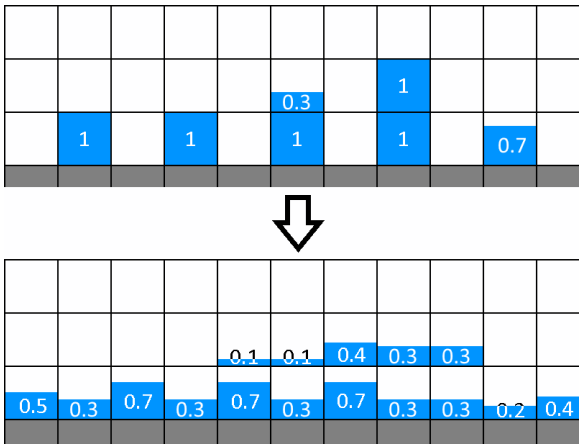

Figure 7: *Passage d'une étape d'étalement.*

Il est possible que certains points "flottent", mais ce phénomène est ensuite lissé par l'algorithme des marching cubes. Le cas ci-dessus donnera le résultat suivant :

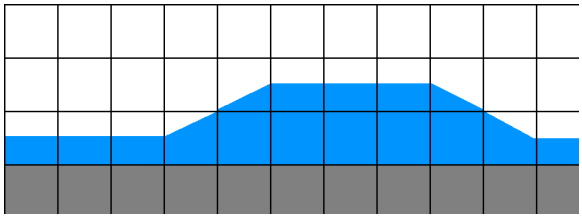

Figure 8: *Lissage via l'algorithme des marching cubes.*

Les calculs de chute et de mouillage sont fait tour à tour jusqu'à obtenir un résultat homogène. Afin de simuler une inondation due à de la pluie par exemple, il suffit d'ajouter de l'eau dans certains points du sommet de la grille et l'algorithme se chargera du reste.

3.1.2. Flottement

Un des objectifs du projet est de visualiser les réactions des oeuvres d'art d'un musée au cours d'une inondation. Dans Unity, on peut attribuer à un objet le composant Rigidbody ce qui permet de régler sa masse, si la gravité s'applique à lui ou non, etc. De plus, afin que les divers objets du musée puissent interagir avec notre eau, il faut qu'ils soient mis en relation avec la grille de calcul. Nous avons donc créé un script permettant de faire flotter des objets de manière réaliste.

En plus des éléments proposés par Rigidbody, nous pouvons à présent ajouter un coefficient de flottement, un coefficient de tangage (qui définit le "balancement" de l'objet) ainsi qu'un centre de flottabilité (centre géométrique de la partie immergée de l'objet). L'algorithme vérifie régulièrement l'état de la grille de simulation aux coordonnées du centre de flottabilité de l'objet. Si l'objet est dans l'eau (volume > 0), on cherche où se trouve la surface en parcourant les points de la grille situés dans la même colonne que le centre de flottabilité afin de récupérer la hauteur de la surface de l'eau. Une fois le niveau d'eau "waterLevel" et le coefficient de flottaison "floatHeight" obtenus, on peut calculer le facteur de la force de flottement "forceFactor" :

$$forceFactor = 1f - ((y - waterLevel)/floatHeight) \quad (1)$$

La poussée d'Archimède correspond à la force exercée par un fluide sur tout objet plongé dedans. Cette force est essentielle afin de représenter le flottement d'un objet. Pour cela, il faut également prendre en compte d'autres éléments que dans la formule précédente et ainsi trouver la vitesse d'élévation "uplift". Nous ajoutons au calcul la force de gravité (obtenue grâce à la fonction "Physics.gravity" de Unity) la vitesse verticale de l'objet et son coefficient de tangage "bounceDamp" :

$$uplift = -gravity * (forceFactor - velocity.y * bounceDamp) \quad (2)$$

3.1.3. Forces et pression

Afin de simuler une dynamique de l'eau, les principales forces qui seront présentes dans notre eau sont :

— Les forces créées lors de la diffusion de l'eau
— Les perturbations causées par la chute d'objets dans l'eau
— La pression

Nous avons ajouté une grille de forces à notre grille de calcul. Chaque point de la grille de calcul correspond un point de la grille de forces, pour lequel un vecteur 3D initialisé à null, représente la force. Lors de la diffusion de l'eau, une force est créée en fonction du volume d'eau déplacé et non l'inverse, comme cela serait le cas avec la technique Lagrangienne. Pour la chute d'objet, l'ajout de force se fait au moment où un impact est détecté. Des forces sont ajoutées dans la grille de calculs autour du point de collision en fonction de la masse et de la vitesse de l'objet. Comme c'est le cas dans la réalité, les forces se diffusent ensuite dans l'eau avec des pertes légères tout au long de la diffusion. Afin que la propagation des forces soit crédible, la transmission ne se fait pas de manière uniforme autour d'un point, mais dépend de la direction de la force. Pour chaque point parcouru, on observe les 6 points autour (sur les 4 côtés, en haut et en dessous). L'algorithme vérifie, pour chacun de ces points, la direction de sa force. Celle-ci peut être orientée vers le point en cours de traitement ou non. Et c'est cette direction, qui va définir la proportion de la force qui sera transmise. Par exemple, si le point à gauche du point traité a une force orientée vers la droite (donc vers le point actuel), une grande proportion de la force du point de droite va être transmise au point actuel. En revanche, si le point de gauche a une force orientée vers le haut, seule une petite partie de sa force sera transmise. Cette méthode de diffusion permet de créer des courants réalistes sans réaliser un trop grand nombre de calculs.

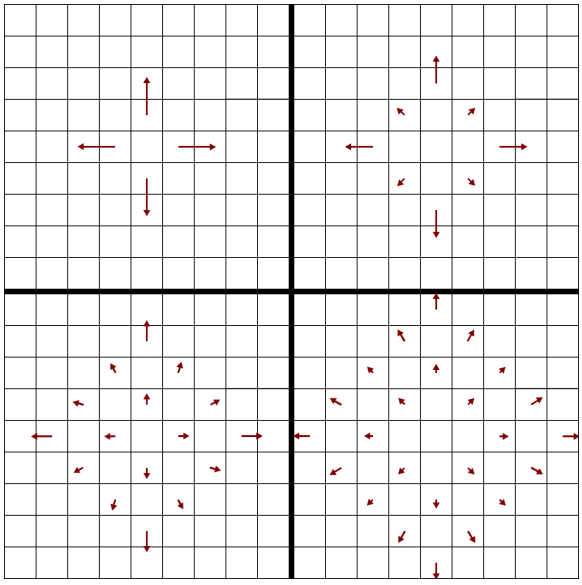

Figure 9: *Les forces se répandent dans la grille de calculs.*

Les forces ne peuvent se répandre dans une case que si elle contient de l'eau (volume > 0). L'algorithme de flottaison de chaque objet observe régulièrement l'état de la grille de force, afin de les appliquer à l'objet et donner ainsi une impression de courant réaliste. Une des contraintes essentielles de notre TER est sa fluidité. En effet, déplacer de l'eau en fonction des forces qui s'appliquent sur elle nous permettrait essentiellement de représenter des éclaboussures ou vaguelettes. Cela diminuerait les performances de notre algorithme sans être très intéressant.

La pression, quand à elle, est gérée de manière assez simple. Une grille regroupe la pression en chaque voxel de l'espace de simulation. Nous avons également créé un type d'objet, assez semblable aux objets flottants, cités précédemment. En plus des différents coefficients et des paramètres de

Rigidbody, il est possible d'ajouter un coefficient de résistance à la pression. L'algorithme d'objets cassants récupère les valeurs de la pression et des forces des différents voxels autour de l'objet. Si la somme de ces éléments devient supérieure à la valeur du coefficient de résistance de l'objet, celui-ci va devenir un objet flottant comme les autres et subir les déplacements causés par la pression trop importante, qui s'exerce sur lui. C'est ce qui représente l'effet coup de bélier.

3.2. Modélisation du musée générique

La simulation de l'inondation se passe dans un musée. L'approche du projet original était de faire, dans un premier temps, la simulation dans un musée générique type avec principalement un rez-de-chaussée et un étage servant d'espaces d'exposition et un sous-sol servant d'espace d'entreposage de matériels et d'oeuvres d'art. Afin de permettre une plus grande variété de scénarios de simulation, nous avons repris le modèle du musée générique et nous l'avons amélioré. Le modèle 3D du musée est composé de plusieurs maillages qui correspondent à la coque extérieure, les murs intérieurs, les rambardes et escaliers, et les verrières du toit. Nous avons supprimé la coque extérieure pour étendre les murs intérieurs, et ainsi créé un trou représentant une entrée. Les verrières n'ont à la base q'une seule face qui est dirigée vers l'intérieur.Nous avons donc dupliqué le modèle et créé une face vers l'extérieur. Pour finir, nous avons créé une place extérieure. Ces modifications permettent d'augmenter la possibilité de scénarios, notamment avoir une inondation qui viendrait de l'extérieur du musée comme lors d'une crue.

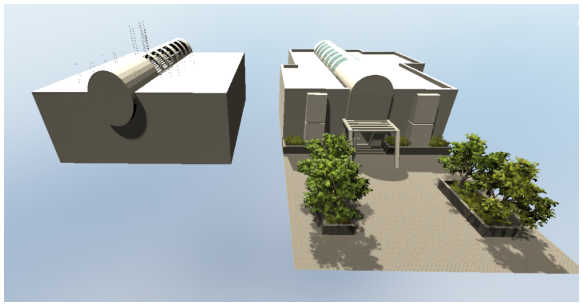

Figure 10: *Le musée générique d'avant à gauche, et à droite, la refonte du modèle.*

3.3. Maillage

Grâce aux voxels, nous avons les données qui correspondent à notre fluide à certains points de l'espace. Pour construire un maillage à partir de ces points, il nous faut utiliser un algorithme que nous allons voir. Tout d'abord, nous devons stocker nos voxels. Pour se faire, on utilise une liste qui permet de parcourir tous les voxels de notre grille dans un sens particulier. Nous aurions pu utiliser une liste 3D pour stocker nos voxels, mais nous avons choisi une liste par soucis de lisibilité et de sens de parcours de la grille.

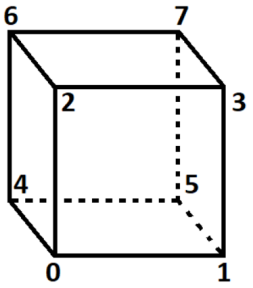

Figure 11: *Sens de stockage des voxels dans une grille 2x2x2.*

Il faudra retenir deux valeurs importantes : la taille de la liste et la résolution de la grille. La taille de la liste correspond au nombre total de voxels dans la grille, tandis que la résolution de la grille donne la largeur, la hauteur, et la profondeur en voxels de la grille. Par exemple, si la résolution est de 3, alors la grille sera de taille 3x3x3 voxels sur X, Y et Z. On obtient la taille de la liste en faisant le calcul 3 x 3 x 3 = 27 et on aura donc des voxels numérotés de 0 à 26. Toutes ces informations permettent à l'algorithme de savoir pour n'importe quel voxel à quelles coordonnées il se trouve dans la grille. On passe d'un identifiant dans la liste à une position en X, Y, Z.

Maintenant, il faut générer un maillage à partir de nos voxels. Pour se faire, on utilise l'algorithme des marching cubes. Le principe de cet algorithme est de parcourir chaque voxel et de former un cube qui aura pour sommet le voxel sur lequel on est et ses voisins aux coordonnées x+1, y+1, z+1, x+1/y+1, x+1/z+1, y+1/z+1 et x+1/y+1/z+1.

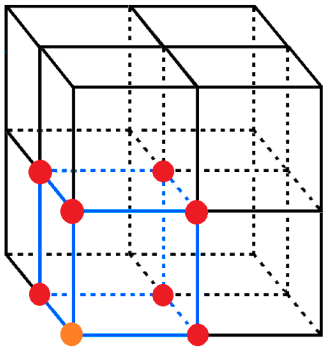

Figure 12: *Le cube formé avec les voisins (rouge) du voxel actuel (orange)*

En fonction des informations que chacun des voxels du cube contient, nous aurons plusieurs cas possibles pour construire le maillage. En 3D, il y a 256 cas différents dont certains sont juste des variations. Au final nous avons les cas suivants :

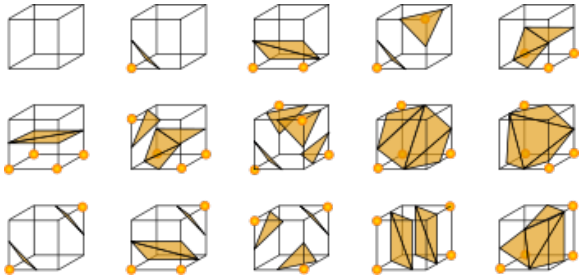

Figure 13: *Représentation des différentes configurations de Marching Cubes.*

Comme on peut le voir sur l'image, la construction du maillage requiert de passer par le centre des arêtes du cube. On appelle cela le primal contouring. Il nous faut donc stocker en plus, dans chaque voxel, les positions qui correspondent aux centres des arêtes qui leurs sont liées, donc en X, Y et Z.

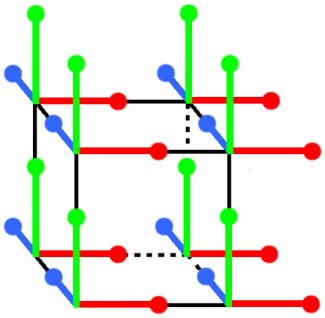

Figure 14: *Les coordonnées en X, Y et Z stockées dans chaque voxel sur une grille 2x2x2*

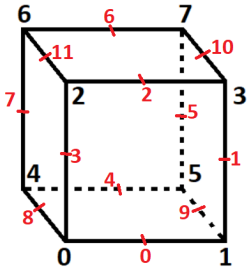

Figure 15: *Les indices des centres des arêtes par rapport au voxel actuel (0)*

Chaque cas de voxel contenant de l'eau ou non va nous donner une configuration. Cette configuration va nous permettre de savoir quels points relier pour créer le maillage. Pour obtenir cette configuration, nous allons assigner un nombre à chaque sommet que l'on code en binaire. Sur l'exemple ci-dessous, le x est remplacé par 1 ou 0 en fonction de la valeur contenue dans le voxel correspondant (eau ou vide) et on les combine avec l'opérateur OU (équivalent d'une addition en décimal) pour obtenir un nombre en décimal.

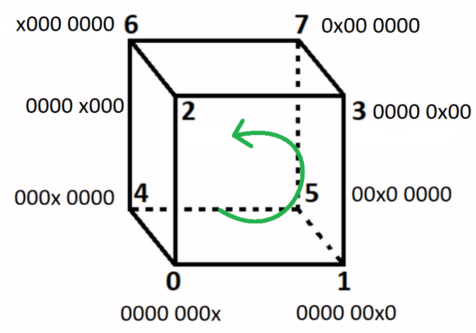

Figure 16: *Attribution de nombres binaires aux coins du cube et sens de lecture.*

Prenons comme exemple les voxels 0 et 2 comme contenant de l'eau et les autres voxels n'en contenant pas. Au final, nous voulons un maillage qui ressemble à cela :

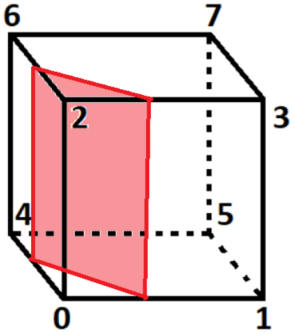

Figure 17: *Face obtenue avec les coins 0 et 2 actifs*

Tout d'abord nous devons trouver le numéro de notre configuration. On applique donc la conversion en binaire puis en décimal que l'on a vu juste avant :

0 - 0000 0001

1 - 0000 0000

2 - 0000 1000

3 - 0000 0000

4 - 0000 0000

5 - 0000 0000

6 - 0000 0000

7 - 0000 0000

OU - 0000 1001

= 9

Cela nous donne la configuration 9. Maintenant, il nous faut un moyen de connaître quels sommets relier pour générer le maillage. Pour cela, on utilise une table de correspondance qui donne les sommets à relier pour former ces triangles en fonction du numéro de la configuration. Chaque ligne de la table nous donne le numéro des sommets à relier. Par exemple, pour la configuration 9, on a :

{0, 2, 11, 8, 0, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}

Sur cet exemple, on aura les triangles 0, 2, 11 et 8, 0, 11. Les -1 permettent de savoir si on est arrivé au bout des triangles à créer. L'algorithme s'arrête quand il arrive sur le -1 le plus à gauche.

Maintenant que l'on connaît les sommets à relier, on peut créer le maillage :

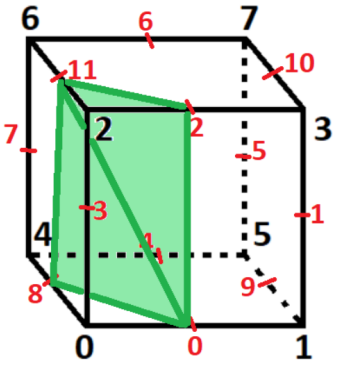

Figure 18: *2 triangles reliant les points 0, 2 et 11 et les points 8, 0 et 11*

Pour obtenir un maillage sur tous les voxels de notre grille, on répète cet algorithme sur tous les voxels que l'on parcourt.

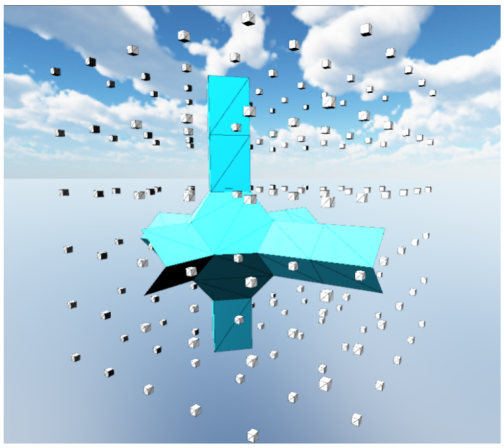

Figure 19: *Résultat de l'algorithme des marching cubes dans une grille 6x6x6 avec une ligne de voxels mise à 1 sur chaque axe.*

3.4. Optimisation

Après avoir intégré la simulation de fluide avec les grilles de voxels et les marching cubes, nous avons très vite constaté que des artefacts apparaissent sur le maillage, lorsque la grille dépassait une taille de 80x80x80 environ.

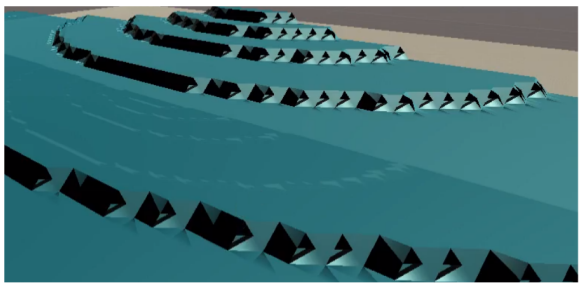

Figure 20: *Des artefacts sont visibles sur le maillage.*

Nous avons rapidement conclu qu'il s'agissait d'un bug engendré par la taille de la liste nécessaire pour stocker les voxels. En effet, pour stocker une grille de 80x80x80, il faut une liste de 512 000 éléments. Nous avons alors décidé de diviser notre grille en plusieurs sous-grilles, appelés chunks. Chacun de ces chunks se comporte comme une grille à part entière et un script les regroupe pour les positionner au bon endroit et leur indiquer leurs chunks voisins. Chaque chunk va utiliser la liste de voxels de ses voisins afin de les cloner pour appliquer l'algorithme des marching cubes sur ses frontières avec le ou les chunks voisins.

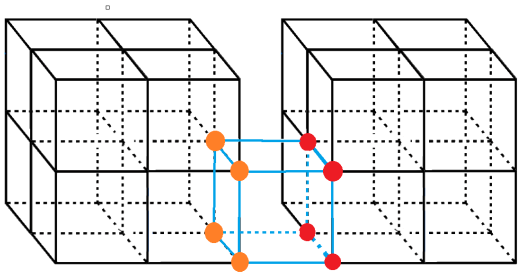

Figure 21: *Le marching cube sur la frontière X avec les voxels du chunk actuel (orange) et les clones des voxels du chunk voisin (rouge).*

Chaque clone de voxel va reprendre les informations du voxel original : son état (eau ou vide) et les positions des milieux des arêtes. Pour la position des clones, on part de la position des voxels de notre chunk actuel (les voxels oranges) et on applique un décalage vers la direction du chunk voisin. Cette opération est la même pour toutes les frontières.

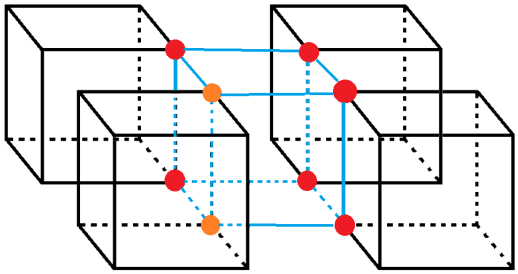

Figure 22: *Le marching cube sur la frontière XZ avec les voxels du chunk actuel (orange) et les clones des voxels des chunks voisins (rouge).*

Pourquoi utiliser des clones ? Cela est dû aux systèmes de coordonnées. L'objet qui sera le maître des chunks est placé dans l'espace 3D grâce aux coordonnées globales. Il va ensuite placer les chunks nécessaires avec des coordonnées locales par rapport à sa position dans le monde. Ensuite, chaque chunk va placer ses voxels avec des coordonnées locales au chunk. Cela donne pour une grille de 2x2x2 chunks de 2x2x2 voxels avec un objet maître en 0, 0, 0 :

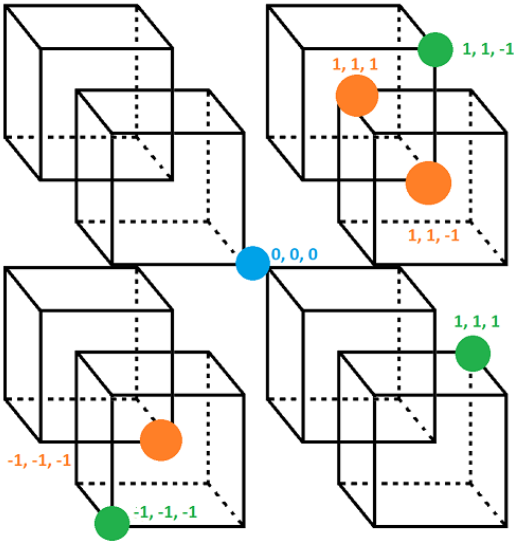

Figure 23: *Les coordonnées de certains éléments : en bleu l'objet maître, en orange les chunks, en vert les voxels .*

Ce système de chunks a un grand avantage pour l'optimisation. Il permet d'ignorer les chunks où il n'y a pas d'eau et donc de ne pas exécuter l'algorithme des marching cubes sur ceux-ci. Il ouvre également des possibilités de multithreading.

Les gains de performance obtenu avec cette implémentation sont déjà assez conséquents.

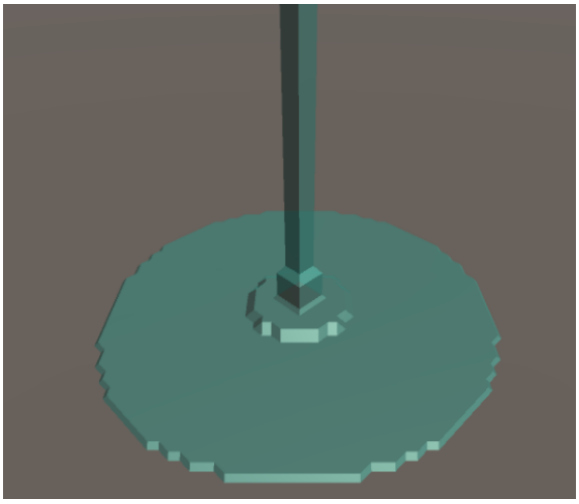

Figure 24: *L'espace de simulation utilisé pour mesurer le gain de performance, il s'agit d'une grille de 64x64x64 soit 262144 voxels.*

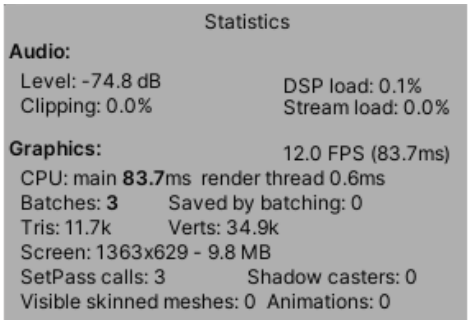

Figure 25: *Performances obtenues avec une seule grille de simulation.*

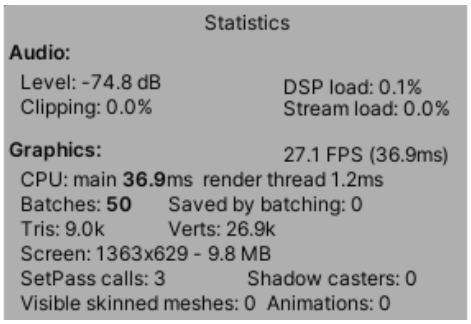

Figure 26: *Performances obtenues avec une grille de 8x8x8 chunks de 8x8x8 voxels (soit 64x64x64 voxels).*

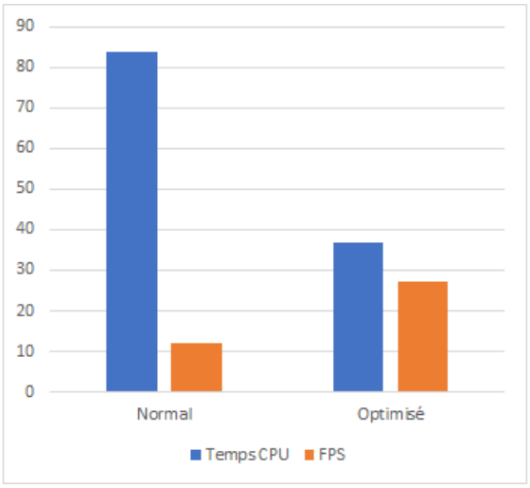

Figure 27: *Différence de performances entre les deux implémentations.*

4. Résultats

4.1. Validation

Le projet n'a pas été entièrement terminé, mais un grand nombre d'éléments ont néanmoins été achevés.

La physique de l'eau est entièrement fonctionnelle. L'eau coule et se répand de manière réaliste tout en créant des courants. Des objets peuvent flotter à sa surface ou, au contraire couler. Lors de la chute d'un objet lourd dans l'eau, celui-ci crée également des perturbations affectant les autres objets. Un effet de surpression (effet coup de bélier) a également été reproduit, et permet d'enfoncer des objet résistants comme des portes par exemple.

Un système d'optimisation consistant en un découpage de la grille de calcul a également été mis en place. Une fois couplé aux calculs de la physique de l'eau, il améliore considérablement les performances du projet et permet de générer des grilles de taille moyenne sans problèmes de performances. Les éléments flottants et cassables n'ont cependant pas encore été ajoutés au système optimisé.

4.2. Problèmes rencontrés

Le problème principal que nous avons rencontré durant ce projet a été la réutilisation du projet pré-existant. En effet, l'asset utilisé pour retranscrire la physique de l'eau était trop simple. Il ne permettait pas de simuler des scénarios réalistes. Or, les contraintes principales de notre cahier des charges était la flexibilité des scénarios et l'optimisation des algorithmes. De plus, aucun asset de Unity ne remplissait ces deux critères et nous avons dû créer une solution complètement personnalisée. Le seul élément que nous avons pu récupérer en vue d'une réutilisation est le modèle 3D de l'intérieur du musée.

Nous avons dû nous intéresser à la physique des fluides et aux calculs qui y sont reliés. De plus, les contraintes imposées ne nous ont pas permis de nous tourner vers la méthode de simulation la plus utilisée, la simulation par particules. La recherche et le développement pour élaborer une nouvelle solution complète était un travail très chargé.

5. Conclusion et travaux futurs

Ce projet nous a passionné et nous avons beaucoup appris, notamment sur les méthodes de simulation des fluides. Nous n'avons malheureusement pas pu terminer à temps tous les éléments nécessaires à la complétion du projet. Cependant, après en avoir parler avec notre encadrant, nous serons autorisés à poursuivre notre travail un mois de plus.

Ce temps supplémentaire nous permettra d'une part, de terminer l'intégration du système optimisé de "chunks", et d'autre part, d'ajouter des éléments au projet. En effet, nous souhaiterions ajouter un algorithme de détection d'obstacles. Cela permettrait d'adapter automatiquement notre grille de simulation sur n'importe quel modèle 3D de bâtiment. Nous aimerions également améliorer notre système d'optimisation afin de permettre la génération de grilles de calculs plus grandes et plus précises.

Références

[Ref1] *Unite 2015 - A Little Math for Your Big Ideas* Vidéo de présentation de calcul d'interaction entre un objet et un liquide. https ://www.youtube.com/watch ?v=OOeA0pJ8Y2s

[Ref2] *NVIDIA FleX for Unity* Asset Nvidia FleX de Unity destiné à la simulation de fluides par particules. https ://assetstore.unity.com/packages/tools/physics/nvidia-flex-for-unity-1-0-beta-120425

[Ref3] *Obi Fluid* Asset Obi Fluid de Unity destiné à la simulation de fluides par particules. https ://assetstore.unity.com/packages/tools/physics/obi-fluid-63067 ?aid=1011l34eQ&utm_source=aff

[Ref4] *Floodgate - Massive fluid simulations* Forum Unity traitant de la simulation de fluides par particules. https ://forum.unity.com/threads/beta-floodgate-massive-fluid-simulations.653635/

[Ref5] *LiquidVoxels in New Cubiquity Shader* Vidéo de démonstration de simulation de fluides par voxels. https ://www.youtube.com/watch ?time_continue=30&v=pkA7WdcPDqE&feature=emb_logo

[Ref6] *Fluid Simulation / Smoothed Particle Hydrodynamics in Unity* Vidéo de démonstration et d'explication d'une simulation de fluide par particules. https ://www.youtube.com/watch ?v=NJBz8rMJ0ZU

[Ref7] *Coding Adventure : Marching Cubes* Vidéo d'explication du concept des marching cubes. https ://www.youtube.com/watch ?v=M3iI2l0ltbE

[Ref8] *Hybrid Water Simulation* Démonstration d'une simulation de fluide par particules optimisée en 2D. https ://www.youtube.com/watch ?v=v0rjaYNs8zY

[Ref9] *Marching Squares, partitioning space* Tutoriel sur le fonctionnement d'un algorithme de marching squares. https ://catlikecoding.com/unity/tutorials/marching-squares/

[Ref10] *Large-scale Water Simulation in Games* Rapport de recherche sur la simulation de l'eau dans les jeux vidéos https ://tutcris.tut.fi/portal/files/4312220/kellomaki_1354.pdf

[Ref11] *How to Make 7 Days to Die in Unity - 01 - Marching Cubes* Tutoriel de réalisation d ?un terrain de jeu vidéo avec un algorithme de Marching cubes. https ://www.youtube.com/watch ?v=dTdn3CC64sc

[Ref12] *Real-Time Fluid Dynamics for Games* Rapport de recherches sur la simulation de fluides en temps réel. https ://pdfs.semanticscholar.org/847f/819a4ea14bd789aca8bc88e85e906cfc657c.pdf

[Ref13] *floating a object on water* Forum traitant du flottement d'objets sur Unity. https ://forum.unity.com/threads/floating-a-object-on-water.31671/

[Ref14] *mecanique fluide cours Approche Eulérienne Lagrangienne* Cours de mécanique des fluides sur les approches Lagrangienne et Eulérienne. https ://www.youtube.com/watch ?v=2E-SR4sz7cQ

[Ref15] *Lagrangian vs. Eulerian (In Simple Terms)* Synthétisation des approches Lagrangienne et Eulérienne. https ://www.youtube.com/watch ?v=zUaD-GMARrA