15 MAI 2020

Figure 1 Machine Micro5[1]

SIMULATEUR MACHINE-OUTIL M5

RAPPORT DE PROJET INFOGRAPHIE 19 / 20

JEANNET NINO
FAGA DAVIDE &
LAIPE KEVIN
INF3-DLMB

SUPERVISÉ PAR LE PROF. BENOIT LE CALLENNEC

[1] *http://projets.he-arc.ch/micro5/*

Table des matières

1 Introduction

L'intelligence artificielle prend de plus en plus d'ampleur dans le domaine de l'industrie de l'usinage de pièce. Le but étant d'optimiser et d'améliorer au maximum les performances des machines pour augmenter le rendement. L'intelligence artificielle permet, en quelques secondes, d'automatiser, de prédire et d'évaluer des situations pour lesquelles un cerveau humain mettrait plusieurs heures.

Une excellente manière de vérifier qu'un programme fonctionne correctement avant de le déployer en production est d'avoir un environnement de simulation permettant de tester et de valider le programme. C'est ce que nous allons réaliser dans ce projet.

Le but du projet est donc de créer un simulateur de la machine-outil Micro 5 développée par la He-ARC. Il existe déjà, à la He-Arc, un simulateur de machine-outil avec parcours optimisé en 2D. L'objectif est de réaliser un simulateur similaire avec une intelligence artificielle permettant d'optimiser le parcours de la fraise, mais cette fois dans un environnement à 3 dimensions afin de rendre la simulation plus réaliste. Le réalisme est un point important du projet, la simulation doit au maximum représenter un usinage réel.

Le projet est donc séparé en deux parties, une d'infographie en créant l'environnement de la simulation et l'autre d'intelligence artificielle en implémentant du Machine Learning afin d'optimiser le parcours de la fraise.

Le point important de ce projet en infographie est le choix des technologies utilisées afin d'obtenir les meilleures performances possibles. Dans ce rapport, plusieurs approches seront comparées en décrivant leurs avantages comme leurs limites.

Le rapport commence par décrire les premières approches naïves implémentées en montrant leurs limites puis une partie pour chaque étudiant expliquant la démarche utilisée et les résultats obtenus pour terminer sur une synthèse globale du projet.

2 Répartitions des tâches

Le projet, pour la partie infographie, est séparé en 3 parties, une partie pour chaque étudiant : modélisation géométrique, rendu graphique et interface graphique & animation.

Voici les descriptions détaillées des parties :

Modélisation géométrique

Cette partie consiste à générer un voxel space avec une taille paramétrable. Dans un premier temps ce voxel space sera créé avec des cubes pour modéliser un voxel. Le but est de montrer les limites de cette approche et d'ouvrir la porte a d'autres technologies comme DOTS ou les Octree.

Les deux objectifs exploratoires de cette partie sont donc l'implémentation et documentation de DOTS et des Octree.

La modélisation géométrique est réalisée par Kevin Laipe.

Rendu graphique

Le but de cette partie est de rendre la simulation réaliste. Le premier objectif est d'afficher des textures différentes de matières pour chaque partie de la pièce si celle-ci est usinée ou non.

Le premier objectif exploratoire est de rendre le rendu encore plus réaliste en utilisant les surfaces implicites afin que le rendu soit moins « cubique ».

Finalement, le second objectif exploratoire est d'ajouter un effet d'usinage sur la pièce lorsque la fraise tourne.

Le rendu graphique est réalisé par Nino Jeannet.

UI & Animation

Cette dernière partie consiste à :

- Ajouter une fraise dans la simulation manipulable sur 3 axes pour usiner la pièce.
- Une interface graphique qui permet de paramétrer la simulation

La partie exploratoire est de faire l'état de l'art sur la comparaison de GIT et Unity Collab.

Cette partie est réalisée par Davide Faga.

La partie intelligence artificielle est commune aux trois étudiants.

3 Approche naïve

3.1 Approche naïve V1

La première approche que nous avons utilisée pour ce projet est de simplement avoir un voxel space composé de voxels. Ces voxels sont eux-mêmes composés de 6 panels (pour représenter un cube). Cette première approche a rapidement pu être mise en place et être fonctionnelle.

Pour la partie rendue, à la création des voxels, on vérifie leur position dans le voxel space. Si un voxel est au bord du voxel space, on va modifier la texture du panel extérieur afin d'afficher une texture de base. Toutes les autres textures représentent la texture usinée. Ainsi, tout est calculé à la création et il n'y a plus besoin de mettre à jour la partie graphique durant la simulation. Actuellement, les textures font 1024x1024 pour une taille de 0.7Mo.

3.1.1 Limites

Cette méthode est très coûteuse, car il y'a énormément d'objets. Exemple : pour un voxel space de 20x20x20 on a 20x20x20x6 objets => 48'000 objets

3.1.2 Tests

Avec cette technique et un voxel space de 10x10x10, on obtient des performances très mauvaises :

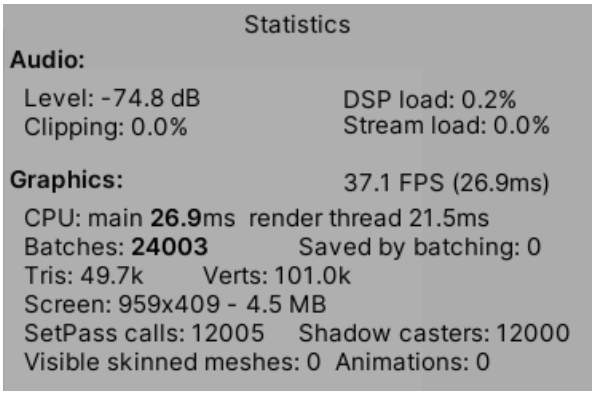


Figure 2 Performances de la simulation V1 I

En enlevant :

- Les ombres sur la lumière
- les Cast Shadows, Receive Shadows, Dynamic Occlusion et les Reflection Probes sur chaque panel d'un voxel

On obtient une amélioration des performances (FPS) de ~40% :

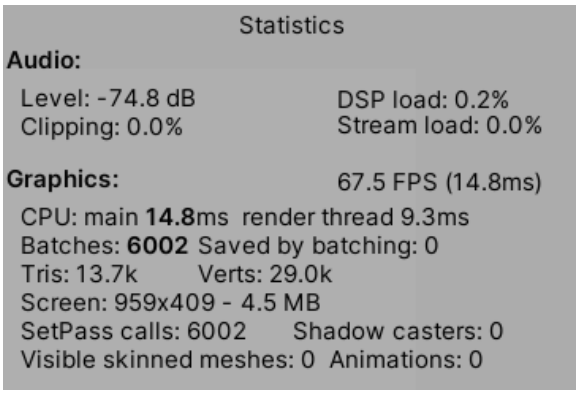


Figure 3 Performances de la simulation V1 II

En réduisant la taille des textures de 1024x1024 à 64x64 on n'augmente presque pas les performances.

On obtient cependant d'excellents résultats en n'utilisant pas le prefabs d'un voxel contenant 6 panels, mais un simple voxels (cube).

3.1.3 Optimisation

Nous avons réussi à améliorer les performances en gardant notre cube fait de 6 panels et en appliquant un matériel de base (contenant la texture usinée) et en changeant seulement les textures des panels sur les bords

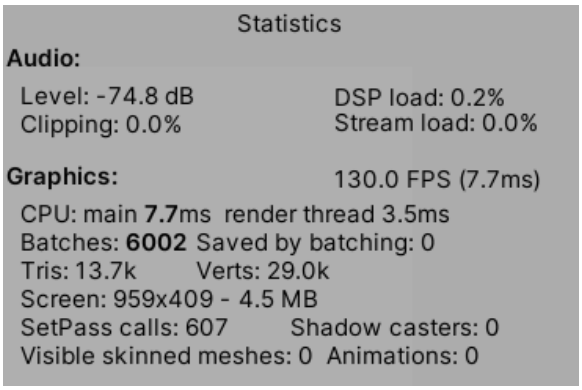


Figure 4 Performances de la simulation V1 optimisées

Comme suggéré dans l'article « Maximizing Your Unity Game's Performance »[10] au point 11,nous avons set le paramètre "Enable GPU instancing" du shader à true. Cette technique doit diminuer le draw calls si nous avons beaucoup d'objets utilisant le même matériel.

Nous nous attendions donc à un gain de FPS mais nous sommes passés de 130 à 100 FPS…

3.1.4 Conclusion approche naïve V1

Nous sommes ensuite arrivés à court d'idées et nous avons donc comparé les performances d'une scène entièrement vide avec notre scène et avons observé le profiler.

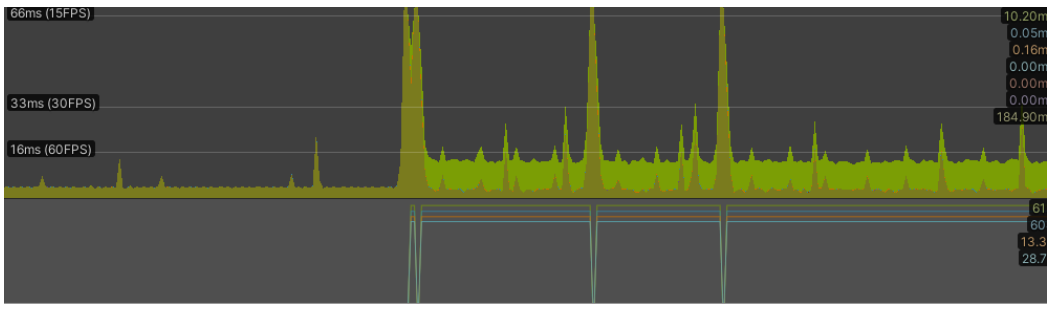


Figure 5 Profiler des ressources de la simulation

La partie gauche correspond à la scène vide. On voit que seul "Others" prend des ressources. Il y a ensuite 3 pics qui correspondent aux relancements de notre scène (3 fois). La partie "Others" ne prend pas sensiblement plus de place qu'avant, ce qui nous fait dire que nous ne pouvons pas chasser plus de FPS.

3.2 Approche naïve V2

La seconde approche à laquelle nous avons pensé consiste à créer 4 prefabs de voxels différents.

- Un cube simple avec une texture usinée. Représente un voxel au centre de la pièce.
- Un cube simple avec une texture usinée et un quad sur une face avec une texture non-usinée. Représente un voxel sur un côté du voxel space.
- Un cube simple avec une texture usinée et deux quads sur deux faces avec une texture non-usinée. Représente un voxel sur une arête du voxel space.
- Un cube simple avec une texture usinée et trois quads sur trois faces avec une texture non-usinée. Représente un voxel sur un angle du voxel space.
- Avec ces 4 prefabs, il faudrait à la création, afficher le bon cube avec la bonne rotation en fonction de la position au sein du voxel space.

3.2.1 Conclusion approche naïve V2

Avantages :

- Cela permettrait de minimiser grandement le nombre de GO dans la scène comparé à la version V1 avec les 6 quads pour un cube.
- Cela permettrait aussi d'avoir tout l'aspect graphique rendu au démarrage de la simulation.

Désavantages :

- Il y'a 26 possibilités de cube avec des positions / rotations différentes(6 angles, 8 faces, 12 arêtes), ce qui entraîne beaucoup de tests (if…else) à la création (pas très propre).
- Cette méthode, bien qu'intéressante ne s'adapterait pas très bien aux deux axes exploratoires que sont Dots et le Marching Cube. Pour le Marching cube, le fait d'avoir 16 cubes différents ne servirait à rien, car les voxels ne seraient pas forcément toujours des cubes (quand on usine).
- Malgré une amélioration des performances comparée à la version V1, cela reste très loin des performances attendues.

3.3 Approche naïve V3

L'idée est de changer le mesh filter des voxels en fonction de leur position dans le voxel space. Imaginons la pièce non-usinée (un cube de x de côté):

- Les huit voxels aux coins ont un mesh filter composé de trois quads (jaunes)
- Les voxels sur les arrêtes ont un mesh filter composé de deux quads adjacents (verts)
- Les voxels se trouvant sur une face, mais pas sur un bord ont un mesh filter composé d'un seul quad (rouges)
- Les voxels qui ne se trouvent ni sur une face, ni sur une arrête ni sur un coin n'ont pas de mesh filter (ils ne sont de toute façon pas visibles comme cachés derrière les autres voxels

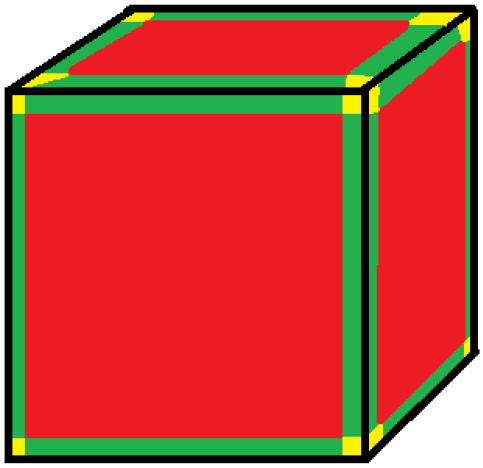


Figure 6 Emplacement des différents types de voxels

Lorsqu'un voxel est usiné, ce voxel est supprimé et les mesh filters des voxels voisins sont modifiés en conséquence.

3.3.1 Conclusion approche naïve V3

Cette approche nous semblerait bonne d'un point de vue des performances mais un peu complexe à mettre en place et encore plus difficile à gérer lorsque des voxels seront usinés.

3.4 Conclusion approches naïves

Avec ces différentes approches, bien qu'intéressantes, on constate que les performances laissent à désirer et que ce ne sont pas des solutions à retenir pour notre implémentation finale. D'aborder ce sujet avec ces approches naïves et intuitives nous a permis de nous rendre compte des limites que possède une carte graphique lorsqu'un programme n'est pas optimisé. Nous avons observé des chutes de performances assez drastiques lorsque nous augmentions le nombre d'objets à rendre dans la scène. Nous avons retenu qu'il est primordial de réfléchir pleinement à l'architecture et aux technologies utilisées dans un projet pour éviter de se retrouver dans des impasses dues aux performances comme nous l'avons vécu.

4 Rendu graphique par Nino Jeannet

4.1 Introduction

Cette partie du projet à pour but de rendre la simulation réaliste, c'est à dire qui se rapproche le plus possible de la réalité en implémentant différentes caractéristiques non pas techniques, mais visuelles qui vont permettre une impression de réalisme pour l'utilisateur.

Une pièce non-usinée aura une texture mate un peu rougeâtre (pour la rouille) alors qu'une fois usinée celle-ci sera brillante avec un aspect métallique. Un autre aspect qui améliorera le rendu est le fait d'utiliser le principe de surface implicite pour passer d'un rendu cubique à un rendu beaucoup plus lisse. Ce processus est implémenté avec l'algorithme du Marching Cube.

Ce chapitre va d'abord présenter le Marching Cube et son fonctionnement puis présenter les programmes similaires qui existent déjà et spécifiquement celui sur lequel nous nous sommes basés pour réaliser l'implémentation finale.

La seconde partie de ce chapitre présente la solution choisie avec son implémentation, l'interprétation des résultats, les problèmes et les améliorations possibles pour terminer sur une synthèse globale du projet.

4.2 Marching Cube

Le Marching Cube est un algorithme qui permet de créer une forme polygonale à partir d'un simple voxel space en se basant sur une isosurface ("courbe de niveau 3D").

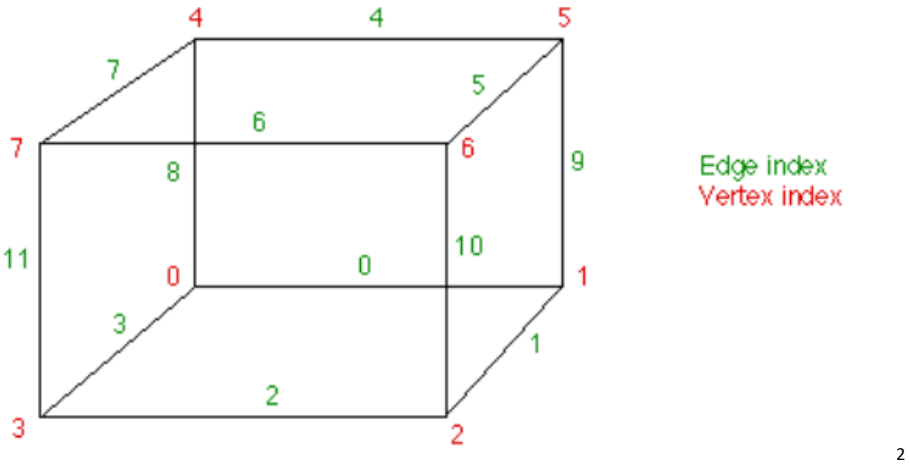
Appliqué à l'infographie, cet algorithme permet de créer des surfaces bien plus détaillées et bien plus réalistes qu'en créant des surfaces à partir de voxels simples (cube). Le résultat donne un rendu beaucoup moins cubique avec la possibilité d'avoir des surfaces beaucoup plus arrondies.

Ce chapitre présente le fonctionnement de base de l'algorithme.

4.2.1 Fonctionnement de l'algorithme

L'algorithme va, pour chaque voxel contenu dans le voxel space, calculer la partie du voxel à afficher.

On utilise la convention suivante pour nommer les arêtes et vertex :



[2]

Figure 7 Schéma de la convention de nommage des arêtes et vertex [1]

[2] http://paulbourke.net/geometry/polygonise/

La première étape consiste à regarder quel vertex du voxel est en dessus ou en dessous de l'isosurface afin de déterminer quelle forme recréer dans le voxel. On crée un index avec ce résultat. Il existe 256 possibilités différentes, mais plusieurs d'entre elles se répètent, on peut donc isoler 16 possibilités différentes :

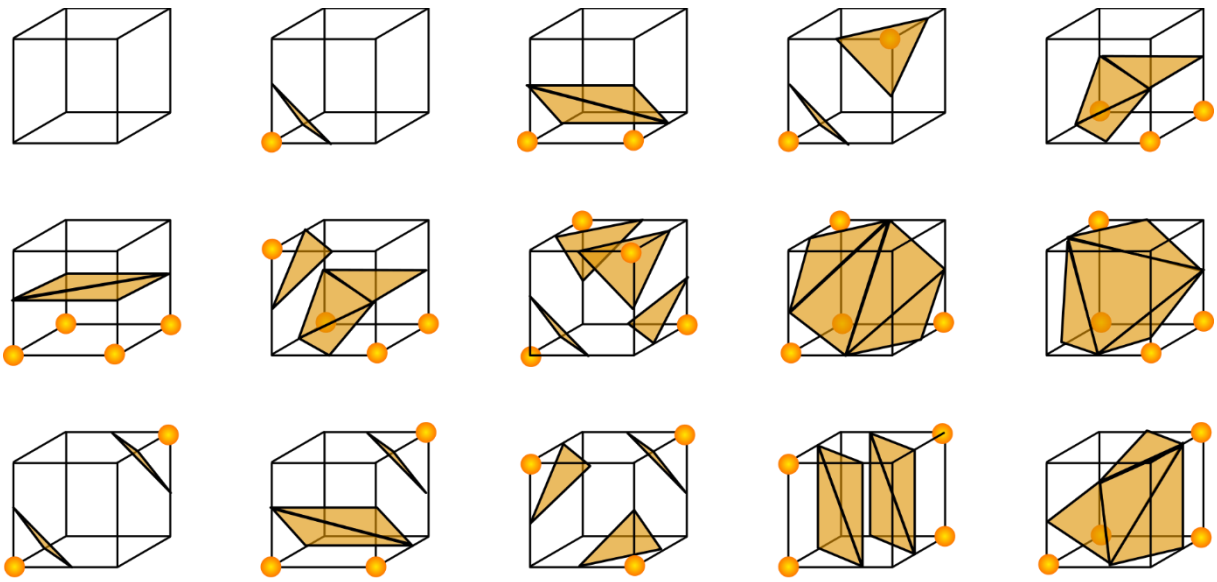

[3]

Figure 8 Les 15 configurations du Marching Cube

Pour faciliter les choses, nous utilisons une "lookup table" appelé "edgeTable" qui retourne les arêtes qui seront coupées par la surface en fournissant l'état des vertex du voxel (dedans/en dehors de la surface) qui est notre index calculé précédemment.

Maintenant que nous avons les arêtes qui nous intéressent, nous pouvons calculer, pour chaque arêtes le point d'intersection entre la surface et l'arête en effectuant une interpolation linéaire.

$$P = P_1 + (\text{isovalue} - V_1)\,(P_2 - P_1) / (V_2 - V_1)$$

[4]

Figure 9 Calcul du point d'intersection avec une interpolation linéaire

P1 et P2 sont les vertex de l'arête et V1 et V2 les densités à ces points.

Ensuite, nous allons de nouveau utiliser une "lookup table" qui référence cette fois-ci les différents index des arêtes pour former les différents triangles de la surface. Pour récupérer ces valeurs, on utilise à nouveau le cubindex utilisé précédemment.

Finalement, nous obtenons une liste de vertex et une liste d'indice de vertex (qui représente nos triangles) ce qui nous permet de mettre à jour un mesh avec les nouvelles valeurs et d'obtenir la nouvelle surface calculée.

[3] https://fr.wikipedia.org/wiki/Marching_cubes#/media/Fichier:MarchingCubes.svg
[4] *http://paulbourke.net/geometry/polygonise/*

4.2.2 Implémentation

Afin de comprendre correctement l'algorithme, celui-ci a été implémenté totalement dans un programme de test. Même si cette implémentation ne fait pas partie de la solution finale, cette étape à été nécessaire à la bonne compréhension de cet algorithme qui n'est pas trivial à prendre en main.

4.3 Travaux existants

Nous avons effectué plusieurs recherches sur des travaux déjà existants afin de voir ce qu'ils se faisaient actuellement. Nous avons trouvé beaucoup de projets de génération de terrain à l'aide du Marching Cube. Parmi ces projets, il y'en a un qui a particulièrement retenu notre attention. Ce projet est un générateur de terrain avec un Marching Cube avec la possibilité d'éditer en direct la forme du terrain (creuser / construire). Cette dernière fonctionnalité nous intéresse beaucoup, car le fait de modifier une forme à la volée est un objectif de notre projet. En étudiant un peu plus le projet, nous avons remarqué que celui-ci utilise plusieurs principes de DOTS (Burst et Job), technologie qui doit être implémentée dans la partie modélisation géométrique. Nous avons donc décidé de récupérer ce projet, de le comprendre et de l'adapter pour notre propre projet.

4.3.1 Marching-Cubes-Terrain by Eldemarkki

Les fonctionnalités de ce projet sont les suivantes :

- Génération de terrain à partir d'une heightmap (texture) en utilisant le Marching cube.
- Déplacement de la caméra dans la scène.
- Déformation du terrain en utilisant le raycasting.
    - possibilité de creuser dans le terrain
    - possibilité d'ajouter du terrain
- possibilité de spécifier la hauteur du terrain

Les performances de ce programme sont impressionnantes avec possibilité de créer un voxels pace de taille 300x300x300 avec des performances correctes (> 20'000'000 de voxels).

Ceci est principalement dû au fait que ce projet utilise le Unity Job System et le Burst Compiler qui font tous deux parties de l'implémentation de DOTS. Ces 2 technologies sont traitées dans le chapitre « Modélisation géométrique ».

Ce projet a été trouvé via une vidéo de démonstration[5] et un lien menant au repository Github[6]

[5] https://www.youtube.com/watch?v=Tge_kwjj2So
[6] https://github.com/Eldemarkki/Marching-Cubes-Terrain

4.3.1.1 Architecture du projet

Un objet **World** est composé de plusieurs Chunks. Un world va créer une certains nombre de chunks en fonctions de la taille de la texture d'entrée.

Un **Chunk** est un gameobject représentant une partie du terrain. Il possède une taille et va s'occuper de mettre à jour sa partie du terrain en utilisant le Marching cube. Le Marching Cube va, à partir d'un ensemble de densités, construire un ensemble de triangles et de vertex pour mettre à jour la forme d'un Chunk.

Terrain Deformer est un script attaché à la caméra utilisateur. Celui-ci va, au clic de souris, lancer un rayon (raycasting) dans la direction de la caméra et récupérer le point d'intersection entre le rayon et le voxel space s'il y'en a un. Le script va ensuite mettre à jour les densités dans la zone touchée afin de creuser dans la matière.

A Chaque update() chaque chunk vérifie si ses densités ont été modifiées et si c'est le cas, il recalcule sont mesh avec le Marching Cube et les nouvelles densités.

4.3.1.2 Génération du terrain

Au lancement du programme, la classe **HeightmapTerrainSettings** va créer la heightmap à partir de la texture fournie. Pour construire cette heightmap, le programme va parcourir la texture en récupérant pour chaque pixel son niveau de gris. Si le niveau de gris vaut 0 (noir) alors il n'y aura pas de terrain. Au contraire si le niveau de gris est 255 (blanc) alors il y'aura du terrain le plus haut possible. La hauteur du point pour chaque pixel dépend donc de la valeur en niveau de gris sur la texture.

Exemple :

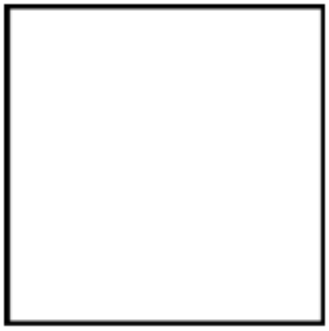


Figure 10 heigthmap d'un cube

Cette image fait 128x128 pixels. Une bande de deux pixels noirs (0) est autour d'un cube blanc (255). Cet exemple va donner un cube de 126x126

4.4 Solution proposée

La solution choisie est la récupération du projet de déformation de terrain présenté au chapitre précédent en l'adaptant, supprimant le code inutile et en ajoutant nos nouvelles fonctionnalités.

4.5 Lien avec le livre WebGL par la pratique

Le chapitre Modélisation d'une sphère. Il présente différentes approches de modélisation d'une sphère notamment celle du Marching Cube. Ce chapitre fait le lien avec le fait que l'approximation de courbe avec le Marching Cube est un « excellent ratio qualité sur nombre de polygones » et effectivement dans ce projet la solution la plus efficace est effectivement le Marching Cube.

4.6 Relations avec les trois piliers de l'infographie

4.6.1 Modélisation géométrique

La relation avec ce pilier est assez triviale, car notre pièce à usiner est composée de plusieurs Mesh correspondant chacune à une partie de la pièce. Ces Mesh définissent la géométrie de la pièce dans un espace 3 dimensions.

4.6.2 Rendu graphique

Ce pilier représente la majeure partie de ce chapitre. La gestion des textures (quand afficher quoi) ainsi que celle de la lumière et des ombres jouent sur le rendu graphique et pour ceci le shader créé dans cette solution l'illustre très bien.

4.6.3 Animation

Les mouvements de la caméra, de la fraise ou encore le fraisage lui-même (déformations de la pièce) utilise l'animation pour rendre la simulation plus interactive et vivante.

4.7 Implémentation

4.7.1 Création d'un voxel space

La première étape est de créer un voxel space afin d'avoir une structure à afficher. Pour ceci, il y'a un paramètre dans l'objet Simulation World de la scène qui permet de spécifier la taille du voxel space (Cube size).

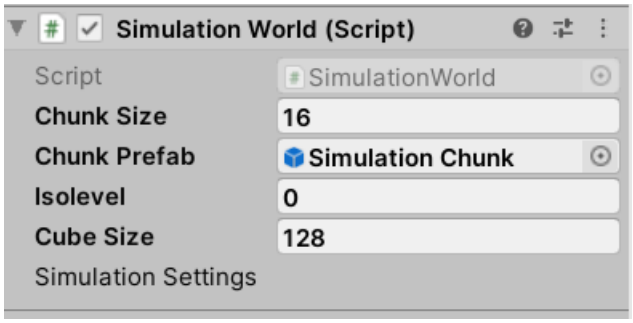


Figure 11 Paramètres de la simulation

Ce paramètre va permettre de créer la heightmap avec laquelle le cube sera créé. La heightmap est un tableau 2 dimensions indiquant la hauteur du cube pour chaque position X Y. La heightmap est très basique puisque tous les points sont à la même hauteur (car c'est un cube). Cependant, afin que l'on voie les côtés du cube, il est nécessaire de laisser une marge vide tout autour du cube. Dès lors, il suffit de mettre toutes les hauteurs à 1 sauf sur tous les bords mettre la hauteur à 0. Cette hauteur correspond au pourcentage de hauteur par rapport à la hauteur totale (128 dans l'exemple). Donc une hauteur de 1 correspond à une hauteur de 128 voxels et une hauteur de 0 à une hauteur de 0 voxels. Il suffit donc de modifier le paramètre Cube Size pour adapter la taille du voxel space.

4.7.2 Texture non-usinée / usinée

Cette partie consiste à afficher une texture différente en fonction de l'état de la pièce. La surface de la pièce à une certaine texture non-usinée et une fois qu'une partie de la pièce est usinée (lorsque l'on voit l'intérieur de la pièce), celle-ci doit avoir une texture différente afin de pouvoir visualiser la différence entre usinée / non-usinée.

4.7.2.1 Approche avec un Vertex – Fragment Shader

La première approche était de créer un shader classique de récupérer la position de chaque vertex et si celui-ci était sur les bords de la pièce alors on affichait la texture non-usinée sinon on affichait la texture usinée. Cette méthode fonctionne cependant il n'y a aucun relief ni surface sur la pièce, elle est uniforme, on ne remarque donc pas lorsqu'on usine la pièce. Cette solution n'est donc pas viable.

4.7.2.2 Approche avec un Surface Shader

La deuxième approche est d'utiliser le même principe que l'approche précédente, mais cette fois-ci avec un Surface Shader qui permet de spécifier plusieurs propriétés du matériau notamment ses réactions avec la lumière comme l'albédo, l'effet métallique ou encore la régularité (smoothness) de la matière. Cela permet de bien voir la surface de la texture ainsi que ces déformations.

C'est cette solution qui a été choisie avec un Surface Shader customisé permettant de choisir nos deux textures avec leurs différents paramètres d'affichage afin de permettre l'affichage d'une texture plutôt mat à l'extérieur de la pièce et une texture avec un effet beaucoup plus métallique, lisse et brillant à l'intérieur de la pièce. Voici la configuration que j'ai choisie.

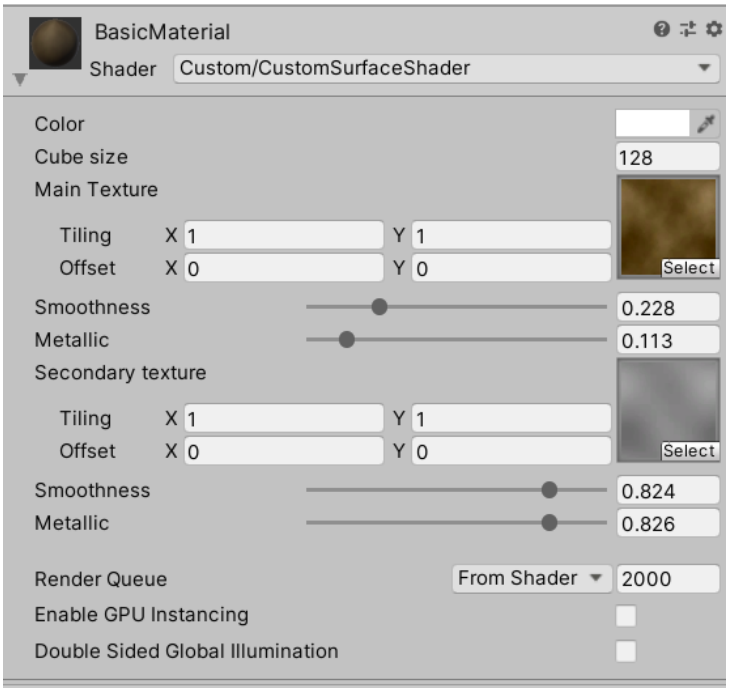


Figure 12 Paramètres du Surface Shader

Le paramètre Cube Size est modifié à la volée pour chaque Chunk à son initialisation pour ne pas devoir modifier la taille du cube à deux endroits lorsque l'on fait des modifications. Voici les deux lignes de codes qui permettent de modifier un paramètre du Shader d'un Chunk :

```
Renderer renderer = GetComponent<Renderer>();
renderer.material.SetFloat("_CubeSize",cubeSize);
```

4.7.3 Collision entre la fraise et la pièce

Une simple sphère a été utilisée pour représenter la fraise et sera par la suite modifiée dans le chapitre « UI & Animation par Davide Faga ». Cette fraise temporaire est utile pour faire une démonstration sans le rendu des autres parties de ce projet.

La méthode CheckSphere de la classe Physics permet de détecter une collision avec les Collider de chaque Chunk de notre simulation. Cette méthode prend en paramètre la position de la sphère et la taille du rayon. En appelant cette méthode à chaque Update() cela permet de retourner un booléen True s'il y'a n'importe quelle collision avec cette sphère. La position de la sphère est constamment mise à jour avec la position actuelle de la fraise. Dès qu'il y'a une collision, la pièce est éditée et déformée à la position de la collision.

4.8 Résultats

Finalement, la simulation fonctionne avec de bonne performance. Il est possible d'usiner la pièce en déplaçant la fraise et aussi de déplacer la caméra.

Voici à quoi ressemble la simulation avec un voxel space de 128x128x128 (> 2'000'000 de voxels). Avec un voxel space de cette taille les performances sont très bonnes avec une moyenne de 150 fps pour 300'000 triangles et 1 million de vertex.

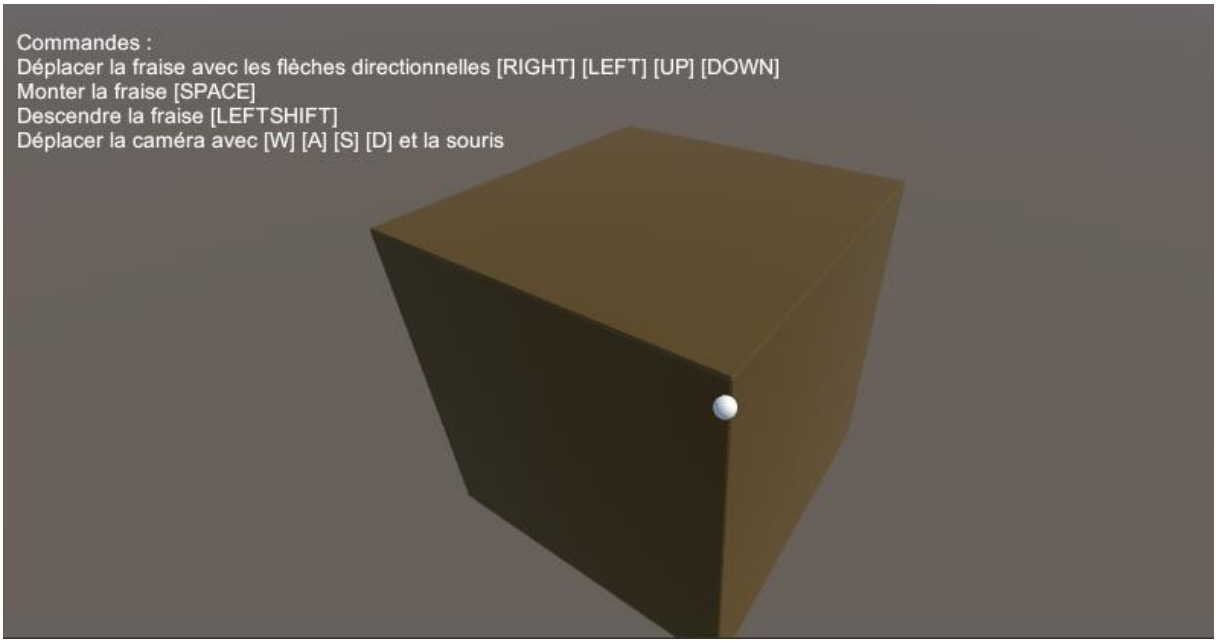


Figure 13 La simulation avec 2'000'000 de voxels

Avec le Shader customisé on remarque très clairement la partie usinée de celle non-usinée (voir ci-dessous).

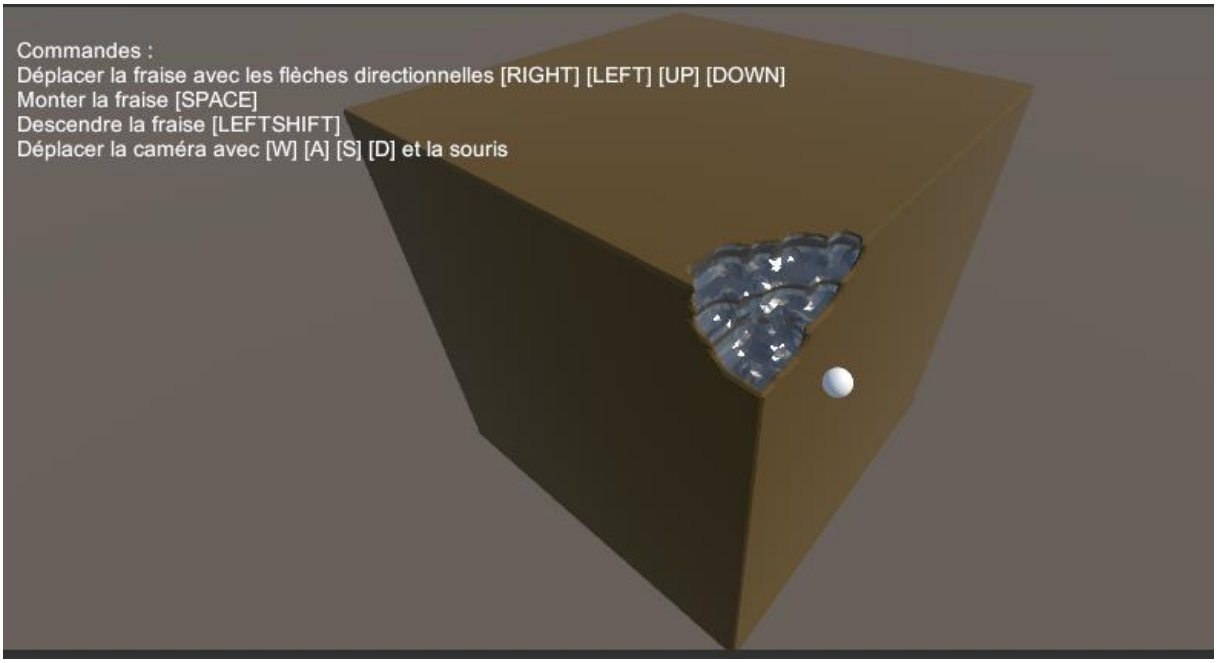


Figure 14 Simulation avec la pièce partiellement usinée

En rapprochant la caméra, on aperçoit encore mieux l'usinage.

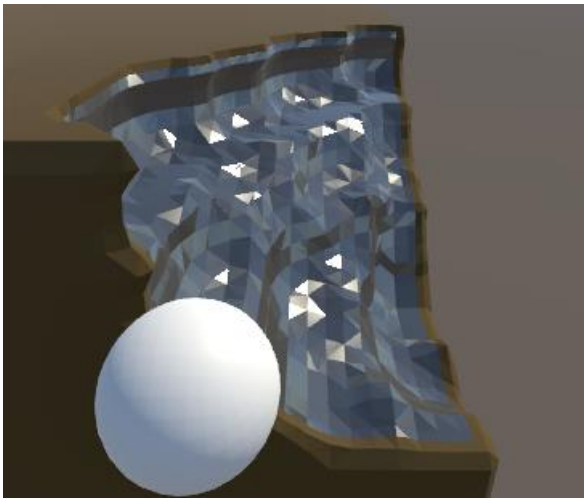


Figure 16 Simulation avec caméra rapprochée sur l'usinage

Figure 15 iImulation avec caméra depuis le dessus

Comme expliqué précédemment il est possible de changer la taille de la pièce en la rapetisissant ou en l'agrandissant. Voici un exemple avec un voxel space de 64x64x64 avec des performances de ~300 fps de moyenne.

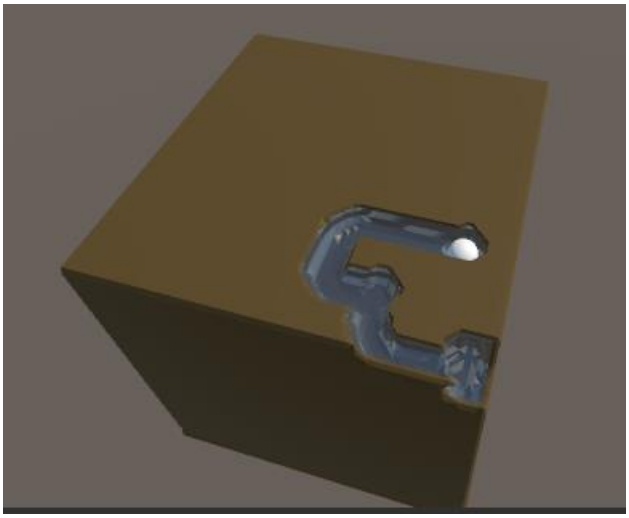


Figure 17 Simulation avec un cube de taille 64

Un dernier exemple avec un voxel space de taille 256x256x256 soit plus de 16 millions de voxels avec cette fois-ci des performances plus faibles avec environ 40 fps de moyenne ce qui reste tout à fait correct compte tenu du nombre de triangle (1.3 million) et de près de 4 millions de vertex.

4.9 Problèmes

Plusieurs problèmes existent dans ce programme, voici les principaux :

Difficilement extensible

Actuellement, cette simulation ne fonctionne qu'avec des pièces non-usinées carrées de base, le programme n'est pas du tout prévu pour supporter des pièces de formes différentes et est difficilement adaptable pour de telles fonctionnalités.

Le dessous de la pièce

Avec cette implémentation, il est impossible d'usiner la pièce en commençant par le bas de la pièce car il n'y a pas de matière (voir exemple ci-dessous) car le programme fonctionne avec un heightmap et donc 1 seul niveau d'affichage au démarrage.

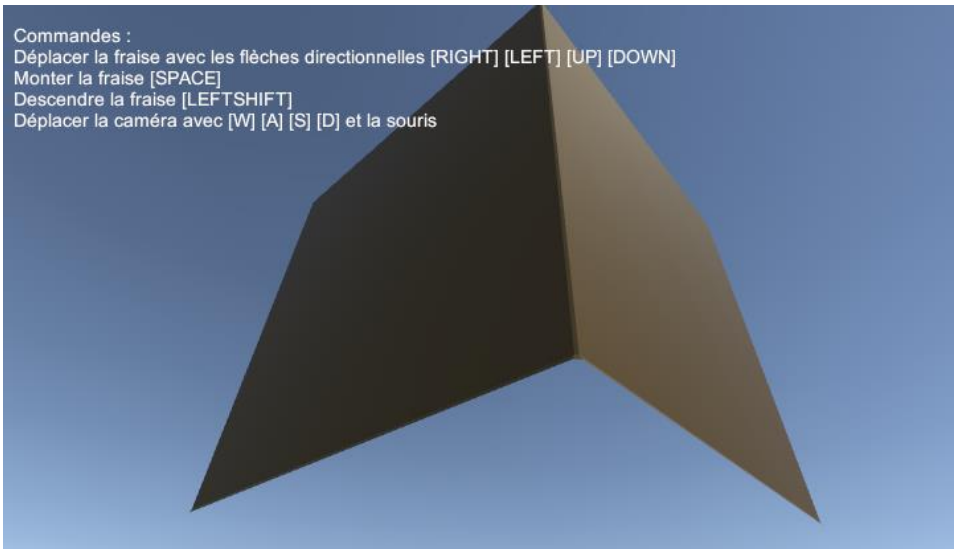


Figure 18 Simulation vue du dessous de la pièce

Réalisme de la simulation

On peut déplacer la fraise avec les touches directionnelles, mais la fraise ne fonctionne pas avec des forces comme avec une vraie machine. Les paramètres spécifiques de la machine n'ont pas été pris en compte pour cette simulation ce qui enlève du réalisme.

4.10 Améliorations possibles

4.10.1 Les Shared Vertices

Dans l'implémentation actuelle, chaque triangle de la pièce est composé de trois vertex uniques. Il y'a donc 3 fois plus de vertex que de triangles. Chaque vertex a donc sa propre normale et lorsque, dans le shader, nous allons calculer la couleur de chaque vertex pour remplir nos triangles nous allons réutiliser cette normale. L'impact sur la couleur finale de la lumière directionnelle qui arrive sur chaque vertex dépend de sa normale et comme chaque triangle a ses propres vertex on s'aperçoit que les triangles adjacents ont des couleurs et réflexions très différentes (Voir figure ci-dessous). Cet effet est dû au fait qu'on a plusieurs vertex aux mêmes positions avec des normales différentes.

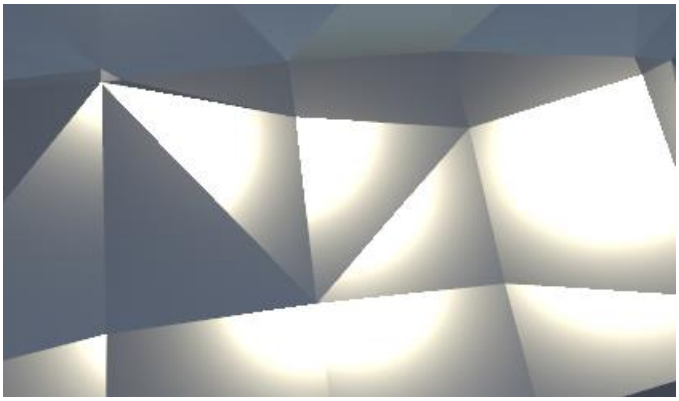


Figure 19 Exemple de Unique Vertices

Une amélioration serait de fusionner les vertex aux mêmes positions afin d'avoir moins de vertex et des normales fusionnées qui donneraient un rendu beaucoup plus lisse et propre. On appelle ça des Shared Vertices, des vertex qui sont utilisés par plusieurs triangles.

Avec la manière dont ce programme créé les mesh, il faudrait **au lancement de la simulation**, fusionner les vertex similaires et recalculer les normales.

4.10.2 Position de la fraise et de la caméra

La position de la caméra et celle de la fraise sont adaptées pour un cube de 128. Il faudrait recalculer de manière proportionnelle en fonction de la taille du cube les positions de ces deux éléments afin qu'ils soient correctement placés, peu importe la taille du cube.

5 UI & animation de la fraise, gestion de projet

5.1 Introduction

Une fois le rendu graphique du cube visuellement réaliste, c'est au tour de la fraise d'être implémentée. Avec les différents types de fraises, plusieurs formes d'usinage sont possibles pour la création d'une rainure, d'un alésage, de trous lamés, etc. Plusieurs fraises sont présentées, ainsi qu'une explication de leurs constructions. Une remarque sur les résultats attendus du rognage est de mise pour rendre attentif sur l'importance de la structure d'une fraise. Une partie explique le fonctionnement du GUI. Finalement une mise en lumière du concept de Unity Collaborate pour en tirer un bilan.

5.2 Fraise

Une fraiseuse est une machine-outil qui a pour but d'usiner des pièces mécaniques en enlevant de la matière à partir d'un bloc à l'aide d'une partie coupante dite « la fraise ». Une fraise est composée de dents et est mise en rotation afin de rogner la matière. Elle est attachée à la fraiseuse par un « arbre ». Il existe différentes fraiseuses pour répondre aux multiples besoins de l'industrie. Trois fraises sont implémentées dans ce projet. Il s'agit d'une fraise simple, d'une fraise en forme de « T » inversé et d'une fraise à rayon convexe.

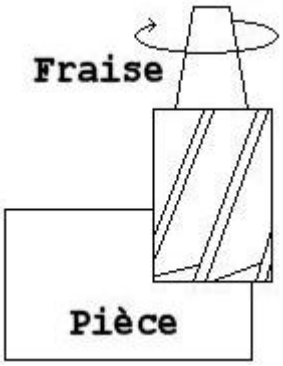


Figure 20 Fraise 2 tailles

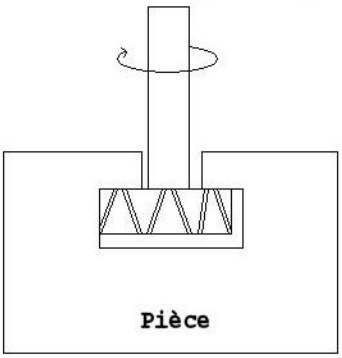


Figure 21 Fraise T inversé

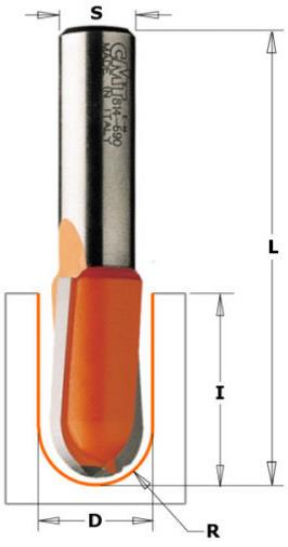


Figure 22 Fraise convexe

5.2.1 Approche

L'algorithme proposé par le projet de base d'Eldemarkki pour creuser et construire le terrain est basé sur un rayon qui creuse le terrain de façon sphérique.

(Rappel du point 4.7.3) La classe *Phisics* de Unity utilisée dans le projet de base pour le rayon, permet aussi de détecter des collisions avec différents types de *colliders* via les méthodes, Check*Box, CheckCapsule, CheckSphere,...* Ces méthodes apportent une panoplie de possibilités de mise en forme de fraiseuses, afin de lier forme et détection de collision. En modifiant un peu l'algorithme d'édition de terrain, il est possible de fournir un effet de rognage cubique, rectangulaire, ou de demi-sphère.

5.2.2 Création d'une fraise

L'idée pour une fraise simple est de créer un *gameobject* composé d'une forme 3D pour le visuel, de lui ajouter un *collider* pour détecter les collisions, une texture de mèche de perceuse et les deux scripts essentiels qui sont *DrillMouvement* et *MaterialDeforme*. Le premier script gère l'effet de rotation et de déplacement, il est identique pour toutes les fraises, mais peut être personnalisé. Quant au second script, son but est de détecter les collisions, d'éditer le terrain, de modifier la vitesse et la distance de rognage. Cela suffit pour donner rapidement un effet proche du réalisme d'une fraise en action.

Les fraises plus complexes comme celle en forme de « T » inversé doivent être construites en plusieurs parties pour pouvoir faire des rainures. Au départ, une approche naïve était de modifier l'algorithme afin qu'il fournisse un effet d'usinage pour la forme voulue en un seul script. Il s'est avéré plus facile de personnaliser un script pour chacune des parties de la fraise. Ainsi, une fraise en « T » est composée de trois parties, l'arbre, la partie haute et la partie basse. Seules les deux dernières sont équipées de *colliders* et de scripts *MaterialDeforme*.

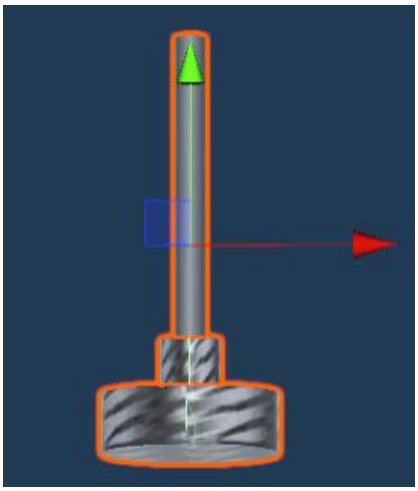


Figure 23 Fraise en forme de « T » inversé

Lorsque la fraise est contruite en plusieurs parties, seul le *gameobject* contenant les parties est munit du script *DrillMouvement*, ainsi toutes les positions enfants se deplacent ensemble.

La fraise à « rayon convexe » est également construite en plusieurs parties. Il y a différentes manières de la créer, en voici deux. L'approche naïve, la plus simple, est représentée par un cylindre et à son bout une sphère. Chaque forme a son *collider* et ses scripts respectifs. Une seconde approche, plus complexe, est de coller cinq sphères les unes sur les autres pour en faire un block avec, pour chaque sphères leurs scripts *MaterialDeform*, le tout emballé dans un cylindre sans *collider*, ni script pour donner un visuel d'un seul morceau. Visuellement, ces deux manières sont identiques, mais le résultat sur l'usinage est différent. Une explication se trouve au prochain chapitre.

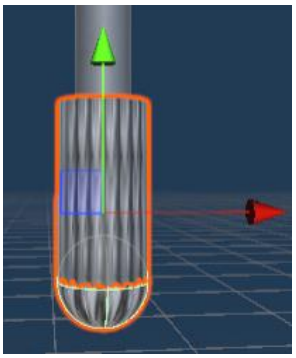


Figure 23 approche naïve

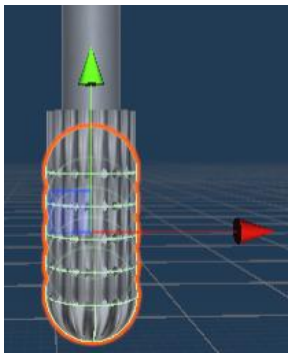


Figure 24 approche complexe

5.2.3 Les limites et défauts comment parer

Revenons à la fraise simple. Si une fraise est composée d'un cylindre lors de l'usinage, le résultat attendu en regardant par le haut est une demi-sphère en direction de l'usinage et le résultat par le côté un rectangle. Vu que l'édition de terrain est possible que par une forme sphérique ou cubique, une parade est mise en place.

Figure 25 Résultat approche naïve vue dessus

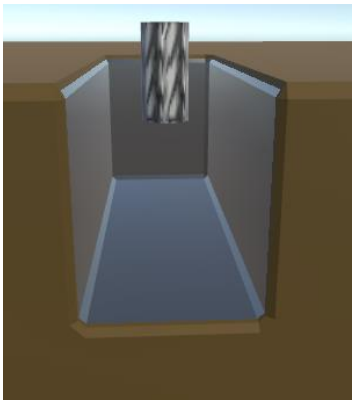


Figure 26 Résultat approche naïve vue de côté

La parade consiste à empiler des demi-sphères en block avec les parties arrondies dans le sens du centre du cylindre. Cela permet d'avoir le centre plat des sphères en direction des extrèmités du cylindre, ainsi l'effet arrondi esconté apparait. Unity ne propose pas de demi-sphères, c'est pourquoi c'est via l'algorithme de *MaterialDeform* des sphères qu'il suffit d'ajuster l'axe des Y en s'arrêtant à la moitier.

Figure 27 Résultat approche complexe

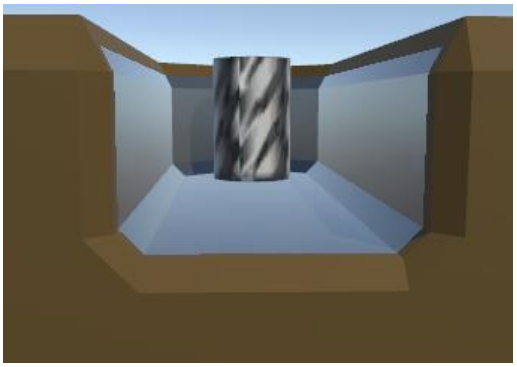


Figure 28 Résultat approche complexe

Même si une fraise est construite avec minutie en multiple morceaux, une autre limite va apparaitre. A cette échelle, certaines parties sont limitées par le Marching Cube affichant des effets de bord comme des asymétries.

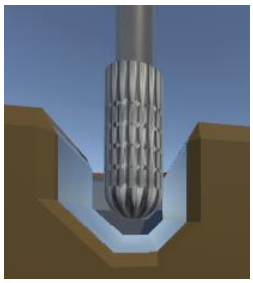


Figure 29 Effet de bord

Figure 30 Effet de bord

Modifier l'échelle de la fraise, avec un coefficient de 50 ou 100, permet d'amoindrir visuellement cet effet, mais détruit les performances de l'édition du terrain qui est composé de trois boucles imbriquées. Il y a donc un algorithme de complexité cubique en O(n^3) avec n=[1 ; 5] qui passe à n=[50 ; 500], les pertes sont abyssales.

5.3 UI

5.3.1 Approche

La scène du GUI est indépendante et n'est pas en lien direct avec la scène de simulation. Tout le projet aurait pu être dans une unique et même scène. En premier lieu, c'est un choix pour simplifier le travail en groupe. Ensuite cela permet de garder une scène purement de simulation qui est flexible pour le développement par ses multiples variables publiques, auxquelles il suffit d'ajouter une scène GUI qui n'est pas essentielle au projet et qui a pour mission de simplifier le paramétrage. Une fois ces paramètres choisis, il suffit de presser le bouton START pour démarrer la simulation. La scène GUI est détruite pour laisser place à celle de la simulation.

L'utilisation correcte pour un bon fonctionnement est d'ajouter les deux scènes dans le projet, puis de ne charger que la scène du GUI. Cependant, dans le *Build Settings,* il faut mettre les scènes dans le bon ordre. Ainsi, une cohérence entre le choix des paramètres et le lancement de la simulation est respectée.

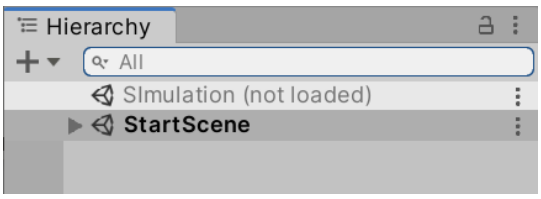


Figure 31 Charger la scène du GUI

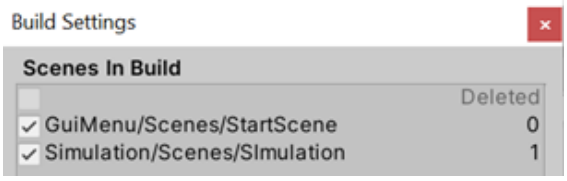


Figure 32 Ordre du Build Settings

Un design sobre est de mise. Le fond est gris et le texte de la même couleur que le logo de notre filière, framboise écrasée. Trois boutons se trouvent à la page d'accueil, dont le bouton OPTION qui ouvre un nouveau menu.

Figure 33 Menu principal

Figure 34 Menu option

Le menu principal possède un script (*scriptMainMenu)* afin de gérer les boutons START et EXIT. Le menu des options a un script (*scriptSimulatorSaveSettings*) permettant de sauvegarder les paramètres.

5.3.2 PlayerPrefs

Pour sauvegarder des paramètres de manière indéfinie, Unity met à disposition la classe *PlayerPrefs*. Cette dernière permet de sauvegarder les trois types de données de base : *int, float, string*. Pour les autres types comme *vector3*, ils peuvent être sauvegardés en les fractionnant en trois composantes *int*.

La structure de la sauvegarde d'une donnée est simple, il s'agit d'un dictionnaire clé-valeur. Ce projet se limite à sauvegarder trois informations : le type de fraise, la taille du cube et la taille des *chunks*.

Pour charger ces données, il suffit, au moment voulu, d'y accéder via les méthodes statiques fournies par la classe. Le lieu de stockage dépend de l'OS, sur Windows c'est dans le registre que ces données sont sauvegardées.

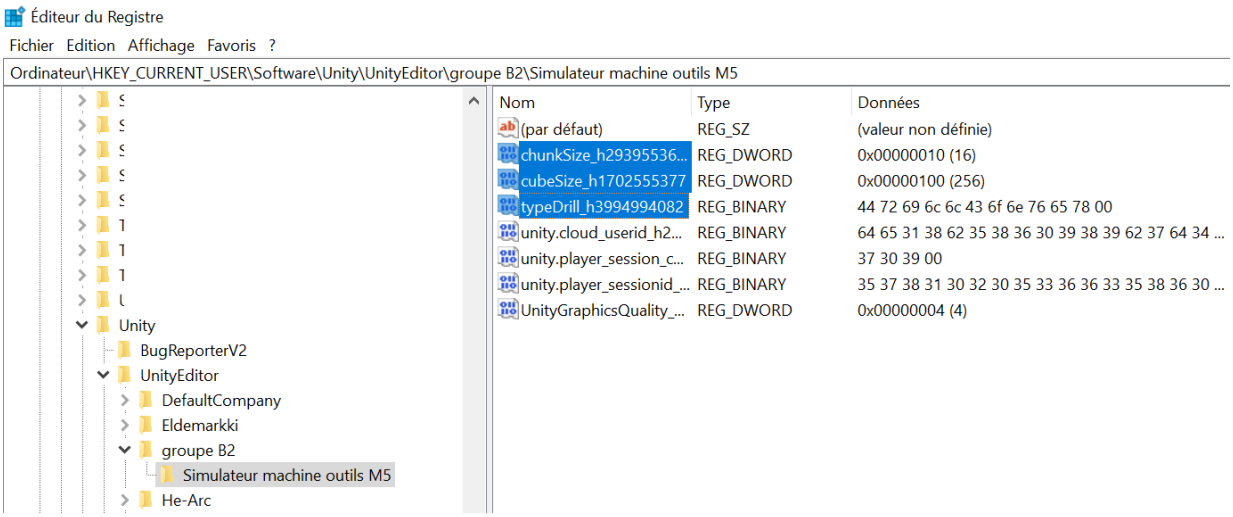


Figure 34 Lieu de sauvegarde du PlayerPrefs dans un système Windows

5.4 Gestion de projet

Le travail en groupe s'avère être un défi lorsqu'il n'est pas structuré à l'avance. Nous avons au départ partagé notre travail de façon naturelle par sujets, mais aucune règle préalable n'avait été définie quant à la hiérarchie des dossiers ou des chemins d'accès. Dès les premiers *merge* pour la démo, il a été nécessaire de nous concerter pour mettre en place une stratégie, car certains *prefabs* étaient introuvables.

Tout d'abord, fractionner chaque partie du plus général, au plus spécialisé. Pour chaque *gameobject* un dossier, avec ses sous dossiers minimum, a été créé : un dossier pour ses scripts, un dossier pour ses *gameobjets* enfants, un dossier pour ses textures, un dossier pour ses *prefabs.* Ensuite la nomenclature, il suffit d'être explicite pour qu'aucune confusion humaine ne s'ajoute à la complexité d'un travail de groupe. Ceci résout quasiment tous les problèmes de conflit.

Le projet est sur un GIT, permettant ainsi de travailler par branche sans empiéter sur le travail d'un camarade. Dans sa globalité, une fois la stratégie mise en place, aucun problème n'est à signaler.

5.4.1 Unity Collaborate

Unity Collaborate permet aux équipes de développement de sauvegarder, de partager et de fusionner leurs projets. Ses fonctionnalités sont faciles à utiliser et sont intégrées à Unity. Il suffit de l'activer avec le bouton Collab en haut à droite. Il faut préalablement créer un compte sur Unity. Dès la première modification du projet, il est possible de faire un commit. En pressant sur « publish now » un push sur le cloud est effectué, un champ est à disposition pour enrichir le commit d'un message explicatif sur les modifications apportées.

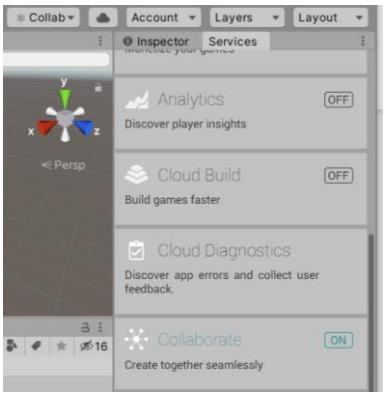


Figure 35 Service Collaborate

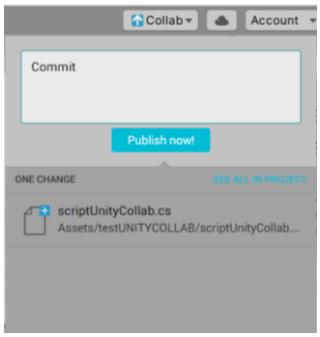


Figure 36 Publish now

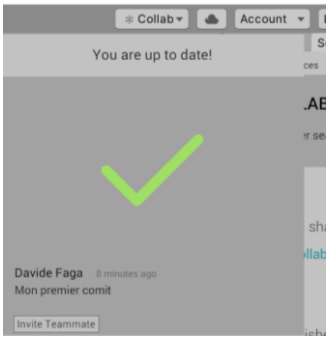


Figure 37 Le projet est sauvé sur le cloud

Une fois le projet sauvé un historique est à dispostion ainsi qu'un bouton pour ajouter des membres.

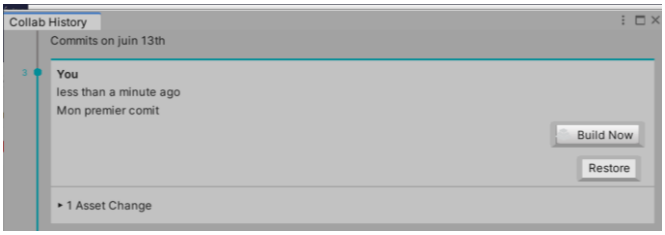


Figure 38 L'historique de sauvegarde

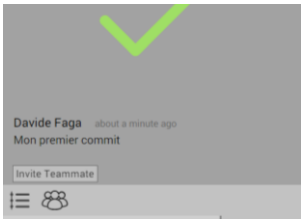


Figure 39 L'historique et le bouton d'ajout de membre

Pour ajouter un membre, il faut d'abord l'inviter via le « Dashboard » du site de Unity en entrant son adresse électronique qu'il utilise pour compte sur le site. Ensuite on attribue un rôle afin de limiter ses actions. Le nouveau membre est informé par mail qu'il est associé à une nouvelle organisation, qui lui donne accès aux projets de ladite organisation.

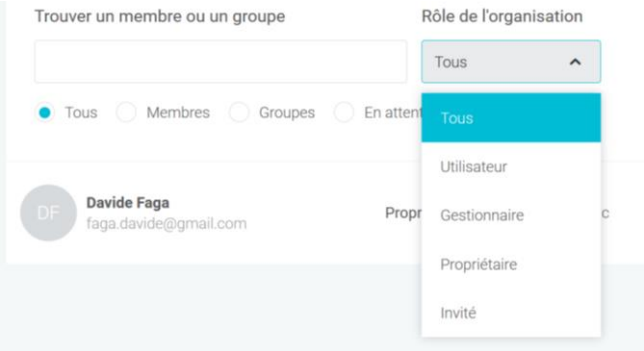


Figure 40 Rôles possibles pour un membre

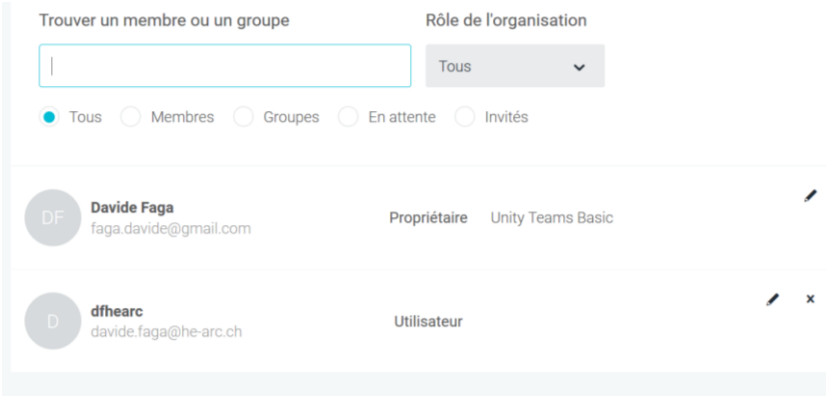


Figure 41 La liste des membres

Le nouveau membre peut désormais modifier dans son onglet « Service » sous la rubrique « collaborate » l'organisation, pour se lier au projet de son choix.

5.4.2 Bilan

Tout d'abord les points positifs de Unity Collaborate. C'est simple d'utilisation et rapide, il n'y a pas besoin d'installation supplémentaire car il fait partie intégrante de Unity. Le logiciel est très pratique pour faire des sauvegardes ou pour partager un projet entre plusieurs personnes ou machines, via un système de cloud performant. Pour faire des sauvegardes dans l'éventualité d'un crash disque ou que l'on travaille sur différente machine, c'est le top. La version gratuite propose 1GB de cloud ce qui est correcte pour des petits projets comme notre simulateur de machine outils qui fait moins de 50MB.

Les points négatifs ne sont pas en grand nombre et peuvent nuancer proportionnellement à la grandeur du projet. Le plus important : il n'y a pas de versioning possible. Aucune approche de type branche comme Git n'est proposée. Ainsi, lorsque plusieurs développeurs travaillent sur une même scène, durant un hackathon par exemple, une confusion pourrait écraser une partie du travail et ralentir le projet, aucune sécurité à ce niveau-là. Concernant le stockage, il n'y a aucun contrôle possible du lieu géographique de stockage, ni une liberté à posséder son propre serveur d'entreprise ou privé. Dans une époque de grands débats sur les politiques de confidentialité, ce point est à relever. La version gratuite se limite à 3 collaborateurs, à 1GB pour le stockage et à 90 jours pour l'historique de sauvegarde. Le fait de payer un stockage supplémentaire et une date étendue pour un historique de sauvegarde peut être compréhensible en raison des ressources demandées, mais limiter à trois personnes pour une collaboration reste maigre pour un logiciel d'une telle envergure.

6 Conclusion

Avant de faire une synthèse du projet, il est important de savoir que ce projet devait être réalisé par trois étudiants avec chacun une partie du projet différente. Malheureusement, dû au COVID-19, Kevin Laipe a dû aller servir à l'armée durant la quasi-totalité du projet et de même pour Davide Faga qui a été appelé par la protection civile.

En tenant compte de cet imprévu, on remarque tout de même que le projet final est fonctionnel et affiche de très bonnes performances. Effectivement, si on compare les performances de la version naïve avec la version finale, on passe de 130 fps avec 1000 voxels à 150 fps avec plus de 2'000'000 de voxels. En plus de ça, le rendu graphique est beaucoup plus agréable et réaliste. Ces améliorations sont dues notamment à l'utilisation de DOTS pour les performances et du Marching Cube pour le rendu. On s'aperçoit donc à quel point, on peut augmenter les performances d'un programme en utilisant les technologies adaptées à une situation.

Le résultat est donc très satisfaisant, il manque cependant encore quelques améliorations comme avoir une fraise plus réaliste, une interface graphique permettant de paramétrer la simulation et de la démarrer. Pour aller plus loin dans le rendu, il faudrait utiliser des Shared Vertices afin d'avoir un rendu bien plus lisse.

Vu les circonstances, la partie intelligence artificielle n'a pas pu être implémentée, cependant le programme, dans sa forme actuelle, est prêt à recevoir de l'IA pour automatiser l'usinage. Pour implémenter du Machine Learning, il existe le projet ML-Agents qui permet de mettre en place un système d'entraînement directement dans Unity afin d'entraîner son intelligence artificielle.

7 Références

[1] Gobron, Stéphane, et Mario Gutierrez. WebGL par la pratique. PPUR, 2015.

[2] Guerne, Jonathan. Digital Twin & AI: création de parcours outil optimisé via des algorithmes de « reinforcement learning », thèse de master. 2019. HE-ARC

[3] Bourke Paul, Polygonising a scalar field. 1994, http://paulbourke.net/geometry/polygonise/

[4] Frappat, Maxime, et Jonathan Antoine. Unity3D: développer en C# des applications en 2D-3D multiplateformes : (iOS, Android, Windows...). Éditions ENI, 2016. https://www.editions-eni.fr/livre/unity3d-developper-en-c-des-applications-2d-3d-multiplateformes-ios-android-windows-9782746099043. Accédé via la plateforme “eni-training.com”.

[5] Bourry, Xavier. WebGL: guide de développement d’applications web 3D. Éditions ENI, 2013. https://www.editions-eni.fr/livre/webgl-guide-de-developpement-d-applications-web-3d-9782746078178. Accédé via la plateforme “eni-training.com”.

[6] Lorensen, William E., Cline, Harvey E., "Marching cubes: A high resolution 3D surface construction algorithm". 1987. ACM SIGGRAPH Computer Graphics. 21 (4): 163–169. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.545.613

[7] Abi-Chahla, Fedy. Next-Gen 3D Rendering Technology: Voxel Ray Casting. 2009. https://www.tomshardware.com/reviews/voxel-ray-casting,2423.html

[8] Unity Official User Manual, version 2019.3. https://docs.unity3d.com/Manual/index.html

[9] Flick, Jasper. Catlike Coding, Unity Tutorials. https://catlikecoding.com/

[10] Gonzalez, Jonathan. Maximizing Your Unity Game’s Performance. 2017. https://cgcookie.com/articles/maximizing-your-unity-games-performance