Projet INF8702

Adrien Logut (1815142), Chathura Namalgamuwa (1815118)

Introduction

Le sujet retenu pour ce projet d'infographie est l'implémtentation d'un comportement réaliste de propagation d'onde dans un milieu aquatique. L'objectif du projet est de permettre l'intéraction avec une surface d'eau. Celà passe par les effets de lumières qui donne un air réaliste à l'eau, ainsi que l'ajout de perturbations (correspondant à des gouttes d'eau qui tombent sur la surface) et leur comportement au cours du temps.

1 Le viewer

Les différents travaux pratiques nous donnaient déjà un viewer fonctionnel. N'ayant pas fait le cours d'infographie de base, on voulait vraiment avoir un viewer qui soit codé par nos soins de A à Z. Comprendre comment construire notre piepline graphique ainsi que la gestion de la caméra. Pour celà, nous avons codé en C++ en tentant au maximum de factoriser et encapsuler notre code. Celà passe par l'utilisation de l'héritage. Ainsi par exemple, un objet de la scène peut être bougé (*class Moveable*) comme notre caméra ou tout autre objet de la scène, avec comme attribut principal leur position dans le monde. Cependant, la caméra n'est pas rendue à l'écran. On a donc une classe *Renderable* qui définit un objet qui va être rendu à l'écran. Ainsi notre scène contient une camera et une liste d'objets à rendre.

La camera peut bouger en gardant un point de focus fixe. Elle tourne alors sur une sphère et permet de regarder la scène sous différents points de vues. Si l'on sélectionne un point en dehors du cube, on peut la faire bouger (si la souris est dans le cube, la caméra reste fixe). Elle peut aussi zoomer/dezoomer avec la molette de la souris.

Au final, ce viewer doit encore être amélioré mais est une bonne base et pourra être réutilisé pour n'importe quel projet de 3D.

La scène est actuellement composée d'une caméra que l'on peut déplacer, un cube texturé représentant une piscine et enfin une surface d'eau.

2 Propagation d'onde

2.1 Equation d'onde

Pour faire notre simulation d'eau et de propagation d'onde, il est important de voir les équations physiques qui régissent notre simulation.

Une onde qui ne subit aucune perte d'énergie et qui se propage en 2D est régie par cette équation :

$$\nabla^2 \overrightarrow{E} = \frac{1}{c^2} \frac{\partial^2 \overrightarrow{E}}{\partial t^2}$$

où $\nabla^2 = \Delta$ représente l'opérateur laplacien et c la célérité de l'onde.

2.2 Résolution numérique

Bien entendu cette équation est dans le domaine continue. Elle doit donc être discrétisée pour pouvoir l'utiliser. Pour cela, on utilise la formule du laplacien discret. Si l'on prend comme pas de mesure dx et dy, le calcul du laplacien est :

$$\nabla \overrightarrow{E}(x,y,t) = \overrightarrow{E}(x+dx,y,t) + \overrightarrow{E}(x-dx,y,t) + \overrightarrow{E}(x,y+dy,t) + \overrightarrow{E}(x,y-dy,t) - 4\overrightarrow{E}(x,y,t)$$

De manière intuitive, si les déplacements autour de notre point sont positifs(resp. négatifs) (les hauteurs sont positives (resp. négatives)), notre point va tendre vers cette valeur et donc monter (resp. descendre).

Pour notre simulation, on définit aussi un pas de temps dt. La méthode de propagation est donc incrémentale. On calcule les valeurs de déplacement à $t + \delta t$ à partir des valeurs en t.

Les équations finales sont donc :

$$\begin{cases} \frac{\partial \overrightarrow{E}}{\partial t}(x,y,t+dt) = 0.995 * (\frac{\partial \overrightarrow{E}}{\partial t}(x,y,t) + \nabla \overrightarrow{E}(x,y,t) \frac{c^2 dt}{dxdy}) \\ \overrightarrow{E}(x,y,t+dt) = \overrightarrow{E}(x,y,t) + \frac{\partial \overrightarrow{E}}{\partial t}(x,y,t+dt)dt \end{cases}$$

Le coefficient 0.995 correspond à la perte d'énergie de l'onde, cela assure que l'onde ne va pas se propager indéfiniment mais va s'atténuer au cours du temps. Plus l'on réduit cette valeur, moins l'onde se propage loin.

3 Implémentation en OpenGL

3.1 Représentation de la surface de l'eau

Tout d'abord, il a fallu s'assurer de pouvoir représenter une surface en OpenGL. La scène comporte donc une grille de quad qui est positionnée proche du haut du cube, occupant toute la surface du cube.

Dans le projet, on peut changer la granularité de notre grille (le nombre de subdivisions) pour augmenter la résolution de notre modélisation.

Le but est ensuite de savoir de combien déplacer chaque sommet de la grille. Pour cela on implémente une *heightmap* qui est une texture et l'on fait du *Displacement Mapping* à l'aide d'une texture

3.2 Displacement Mapping

Notre texture de déplacement est une texture qui stocke les valeurs de déplacement sur le channel rouge (Première composante de RGBA). Les autres channels seront utilisés plus tard pour mettre à jour notre texture.

Cette texture *mappe* donc chaque vertex en lui associant un déplacement positif ou négatif. Pour ne pas s'encombrer de conversion, la texture stocke des nombres flottants sur 32 bits (chaque sommet a donc 4 composantes : RGBA qui sont chacune représentée par un flottant).

Sachant qu'a chaque pixel de notre texture, un vertex lui est associé, la texture fait la taille de la grille. Ainsi si la grille est 50x50, la texture est aussi 50x50 pixels.

Pour appliquer le déplacement à chaque sommet, le nuanceur de sommet de notre grille prends en variable uniforme la *heightmap* et sample la valeur correspondant au vertex et applique cette valeur sur ce sommet.

3.3 Mise à jour de la texture

La lecture de la texture étant faite, il faut maintenant appliquer nos équations précédemment établies pour mettre à jour notre texture.

Pour cela, on va écrire dans une nouvelle texture les nouvelles valeurs puis *swapper* les deux textures pour que la nouvelle texture devienne la texture que l'on va lire.

Pour écrire dans une texture, on utilise la technique qui a été vue dans le TP4, qui est d'utiliser un Framebuffer auquel on associe une texture dans laquelle écrire. Ainsi à chaque frame de notre simulation, l'on met à jour la texture puis l'on rend notre grille.

Le rendu de notre texture n'a pas besoin de ce faire sur une grille. Il suffit de rendre notre texture sur un simple quad. La texture dans laquelle on écrit doit bien entendu être de même dimension que la texture que l'on lit. On calcule donc une nouvelle couleur qui va être écrite dans notre texture.

Un nuanceur a part est donc nécessaire pour écrire dans cette texture. Le nuanceur de sommets est trivial, il ne transforme que les positions du quad (comprises entre -1 et 1 en x et en y) en coordonnées normalisées entre 0 et 1 (dans la variable texCoords).

La partie intéressante est donc dans le nuanceur de fragments. Pour chaque pixel de notre texture, on sample les valeurs dans le channel rouge (la valeur de déplacement on le rappelle) des 4 pixels environnants ($+dx$, $-dx$, $+dy$ et $-dy$) ainsi que la valeur de notre pixel courant pour appliquer la formule du laplacien.

Comme on a besoin aussi de garder les valeurs de vitesses $\frac{\partial \vec{E}}{\partial t}$, on utilise un autre des 4 channels, le channel vert.

La suite suit exactement les formules explicitées précédemment. Il est juste à noter que le quotient $\frac{c^2}{dxdy}$ a été réduit à la simple valeur c dans les équations (qui peut être modulée à souhait dans la classe *WaterGrid*). Numériquement cela ne change rien au problème.

Enfin, les deux derniers channels permettent de garder la valeur de la normale en chaque vertex. Une normale est normalement définie par trois composantes. Cependant deux valeurs ici nous suffisent car l'on garde un vecteur normal unitaire. La 3ème valeur se déduit donc des 2 autres en respectant la norme unitaire.

3.4 Calcul des normales

Le calcul des normales se base sur le différentiel de hauteur entre le pixel de référence, et celui qui se trouve à $+dx$ (resp. $+dy$) (pour la composante en x (resp.y)).

Ces deux vecteurs sont donc :

$$\begin{cases} \overrightarrow{v_x} = [0, dx, \overrightarrow{E}(x+dx, y, t) - \overrightarrow{E}(x, y, t)] \\ \overrightarrow{v_y} = [dy, 0, \overrightarrow{E}(x, y+dy, t) - \overrightarrow{E}(x, y, t)] \end{cases}$$

Pour obtenir la normale, on effectue alors le produit vectoriel $\overrightarrow{v_x} \times \overrightarrow{v_y}$, que l'on normalise.

Enfin, on stocke les valeurs pour x et z sur les 2 channels qui nous reste : bleu et alpha.

Ces valeurs sont *clamp* en dessous de 0.5 car les valeurs étant trop petites, on remarquait des artéfacts sur notre grille. En coupant les valeurs inférieures à 0.5 les artéfacts ont disparus et les normales restaient cohérentes.

3.5 Ajout de perturbation

Pour l'ajout de perturbation à la surface de notre grille, il est possible d'appuyer sur la touche P qui génère une perturbation au centre, ou bien de cliquer sur grille à l'endroit voulu.

Le nuanceur qui s'occupe de faire ça est très similaire à celui de mise à jour de la texture, car il va à l'endroit de la perturbation changer le channel rouge (déplacement) de la valeur voulue.

Pour ce qui est de la souris, obtenir le point sur la surface n'est pas trivial. Pour cela il faut transformer les coordonnées de notre pointeur de souris du référentiel de la fenêtre vers le référentiel du monde. Il faut donc faire la manipulation inverse que celle que l'on effectue pour afficher notre scène à l'écran.

Lorsque l'on clique sur la souris, les coordonnées sont récupérées et sont comprises entre $(0, 0)$ et $(largeur, hauteur)$. Il faut de plus récupérer la valeur en z qui est stocké dans un *depth buffer* interne à OpenGL. On récupère alors la valeur en z du pixel sur lequel on clique.

On obtient alors le vecteur position $\overrightarrow{p} = (x, y, z, w)$ avec $w = 1$

Ensuite, on doit normaliser notre coordonnées (obtenir les *normalized device coordinates* ou *NDC*). Les formules sont disponibles sur le wiki d'OpenGL et sont :

$$\begin{cases} x_{ndc} = 2\frac{x}{largeur} - 1 \\ y_{ndc} = 1 - 2\frac{y}{hauteur} \\ z_{ndc} = \frac{2}{far - near}(z - \frac{far + near}{2}) \end{cases}$$

avec $far = 1$ et $near = 0$ les valeurs par défaut de clipping d'OpenGL. On remarque que y est inversé par rapport à x. C'est parce que le sens de l'axe y dans glfw est l'inverse de celui d'OpenGL. Il faut donc faire cette transformation en plus.

A partir de là, il faut exécuter la transformation inverse. Comme dans le processus initial, il faut multiplier par la matrice de projection P et de vue V, on effectue ici l'inverse.

$$\overrightarrow{p}_{world} = V^{-1} P^{-1} \overrightarrow{p}_{ndc}$$

Il ne faut pas oublier que lors de la transformation initiale, toutes les coordonnées sont divisées par w pour obtenir les *NDC*, la position finale dans notre monde est donc :

$$\overrightarrow{p}_{final} = \frac{1}{w} \overrightarrow{p}_{world}$$

avec w la 4ème composante de notre vecteur $\overrightarrow{p}_{world}$

Au final, dans le nuanceur pour la perturbation, on vérifie que le point cliqué est sur la grille (c'est à dire à la même hauteur z que la grille et sur la surface en x et y).

4 Gestion de la lumière et de la vue

Maintenant que tout le *displacement* est implémenté, il faut donner à l'eau l'apparence de l'eau. Cela passe avant tout par teinter légèrement la couleur par une couleur bleuâtre ainsi que de jouer sur la réflexion et réfraction.

4.1 Réfraction

Sans l'eau, notre vue voit le cube texturé. Avec l'eau, il faut que ce que l'on voit à travers l'eau soit "décalé" (qui est l'effet optique de la réfraction). Le calcul du rayon réfracté est plutôt facile, étant donné que GLSL nous donne une fonction *refract*. Il suffit donc d'avoir comme rayon incident notre vecteur vue et la normale associée au fragment que l'on process.

Le plus compliqué est de savoir où se rayon va taper sur notre cube. Car on veut pour le fragment que l'on process sur notre grille qu'il affiche la texture du cube.

Pour cela, on fait une simplification de lancer de rayon sur GPU, car l'on connait beaucoup de variables (qu'il aurait été plus compliqué de faire un lancer de rayon complet sur GPU).

Ainsi pour chaque fragment, on calcule l'intersection entre le rayon refacté et le plan inférieur du cube (le fond de la piscine). Si le point d'intersection est sur la face inférieure, on sample la texture du cube à cet endroit là et l'on a la couleur de notre pixel sur notre grille.

De même si le point d'intersection n'est pas sur la face inférieure mais avec des coordonnées en x et/ou y plus grandes que la largeur du cube, cela veut dire que le rayon a tapé une des faces latérales du cube. Lorsque l'on sait quelle face on tape, on relance l'algorithme d'intersection avec la face voulue.

Il reste tout de même un problème lorsque et x et y sont plus grandes que la largeur du cube. Quelle face prendre dans ce cas là ?

Il faut calculer une *frontière* qui va séparer les cas. Cette frontière est calculée cette fois ci on calculant l'intersection entre le plan inférieur et le vecteur d'origine le vertex de la grille et d'arrivée un point sur l'arête ambigüe. Pour donner un exemple, si x et y sont tous les deux trop grands pour être sur la face inférieure, l'arête que l'on va utiliser est celle qui se situe en $(\frac{c}{2}, \frac{c}{2})$, c étant la longueur du côté du cube.

Ce point d'intersection nous donne la frontière qui est la droite passant par ce point et le coin inférieur de l'arête ambigüe (dans le cas précédent, le coin est aux coordonnées $(\frac{c}{2}, \frac{c}{2}, -\frac{c}{2})$)

Suivant que le point d'intersection calculé initialement se trouve d'un côté ou de l'autre de la frontière, on peut déterminer la face que le rayon intersecte.

4.2 Normales

Précédemment on a calculé les normales pour chaque vertex de notre grille. Cela permet entre autre de donner le bon effet à notre eau qui est de brouiller la vision à l'endroit où il y a une

perturbation (comme dans la réalité).

5 Simulation finale

Il est possible de faire varier les valeurs de simulations si l'on modifie les valeurs de perturbation et de célérité dans le fichier *WaterGrid.h*. Il est possible aussi de changer la résolution de la texture (et donc la granularité de notre grille) ainsi que le nombre de FPS maximum dans les constantes en haut du fichier *main.cpp*.

Actuellement les performances sont très bonnes et avec une grille de 256x256 on atteint aisément les 60fps. A titre de comparaison, toutes les équations et simulations étaient fait sur le CPU (pour tester la véracité de nos équations), et n'atteignait le 60fps que si la grille faisait 50x50.

A noter que la granularité peut encore être augmentée sans perte visible de FPS. Le temps d'initialisation sera par contre plus long (initialiser tous les buffers de vertex + texture) et la simulation sera plus lente à se propager (car dx et dy sont plus petits).

Au final la simulation donne un comportement proche de la réalité sur les fonctionnalités implémentées.

6 Conclusion

Ce projet d'infographie a été l'occasion de regrouper toutes les connaissances emmagasinées tout au long de la session, ainsi que des notions de physique/mathématiques vues antérieurement et fournir quelque chose de concret qui fonctionne bien.

La satisfaction d'avoir pu tout coder de A à Z aussi est très appréciable et pourra être réutilisée pour de futurs autres projets.

La mise en place a cependant été longue, et on avait peur de ne pas être capable de fournir le résultat sur GPU. Etant donné que cette partie nous a pris énormément de temps, les autres fonctionnalités prévues (comme les caustiques ou l'interaction avec d'autres objets) n'a pu être faites, mais on tentera tout du moins de continuer le projet après la date de rendu pour obtenir un projet abouti comme on l'entendait au début.

En conclusion, bien que tout n'est pas présent, la simulation a vraiment belle allure et nous conforte sur nos connaissances acquises tout au long de cette session.