Guillaume Pelat
Olivér Facklam

31 mai 2019

RAPPORT DE PROJET INF443

Flying Dragon Hidden Starship

TABLE DES MATIÈRES

INTRODUCTION

Dans le cadre du projet d'INF443, nous avons réalisé un petit jeu vidéo dont l'objectif est d'éviter les obstacles au fur et à mesure de l'avancée du joueur. Ce projet a été l'occasion de mobiliser les acquis pratiques et théoriques de ce cours, depuis le texturage et l'animation procédurale jusqu'à la génération d'arbre par grammaire et la création d'ombres.

Le code a été conçu pour être facilement modifiable par plusieurs personnes à la fois. Ainsi, chaque élément possède ses propres `.hpp` et `.cpp`, comprenant les fonctions de création et d'affichage. Le coeur du projet est situé dans `projet.cpp`, le `main` se contentant de gérer la boucle `glfw` et les évènements clavier.

1 MODÉLISATION

1.1 Terrain

La première étape de notre jeu a été de modéliser un terrain adapté. Pour cela, deux éléments devaient être pris en considération. D'abord, il doit être de largeur fixe : le joueur ne doit pas pouvoir se décaler indéfiniment. Ensuite, sa longueur doit être infinie.

Le premier problème a été réglé rapidement : nous avons enfermé l'espace de jeu dans un canyon, modélisé par une fonction cosh. Un bruit de Perlin 3D et une texture de roche réalisée par la chaîne YouTube *Geoffrey - Apprendre le graphisme* (`https://www.youtube.com/channel/UC8J66rf-wBocdEiy4aBl3ug`) viennent compléter le tableau.

Pour la longueur infinie, nous avons d'abord envisagé de déplacer le terrain en le générant au fur et à mesure, mais cette solution nous a semblé compliquée à mettre en place. Nous nous sommes donc tournés vers un terrain cylindrique : en mettant la caméra suffisamment proche du sol et à une échelle bien choisie, la courbure est minime. La texture d'herbe fournie dans les TP et un autre bruit de Perlin 3D parachèvent la mise en place du mesh.

La forme cylindrique, bien que compliquant un peu le choix des coordonnées des différents éléments, a un autre avantage : elle permet de gérer beaucoup plus simplement l'apparition des obstacles.

1.2 Arbres

Pour modéliser ces obstacles, nous avons décidé de créer des arbres. Afin d'obtenir une géométrie réaliste, ces arbres sont générés procéduralement grâce à des règles de grammaire.

Une `struct element` représente une branche de l'arbre. Cette structure garde en mémoire le type de la branche, son orientation par rapport à son parent et la liste de ses fils, sa longueur et son rayon. Pour le modèle `monopodial`, la génération part d'une seed constituée d'une seule branche de type A. Pour le jeu, nous faisons 5 itérations de génération. À chacune d'entre elles, l'arbre est parcouru et les branches sont remplacées selon les règles de génération suivantes, basées sur *The Algorithmic Beauty of Plants*, Chapitre 2 p. 56 (`http://algorithmicbotany.org/papers/abop/abop-ch2.pdf`) :

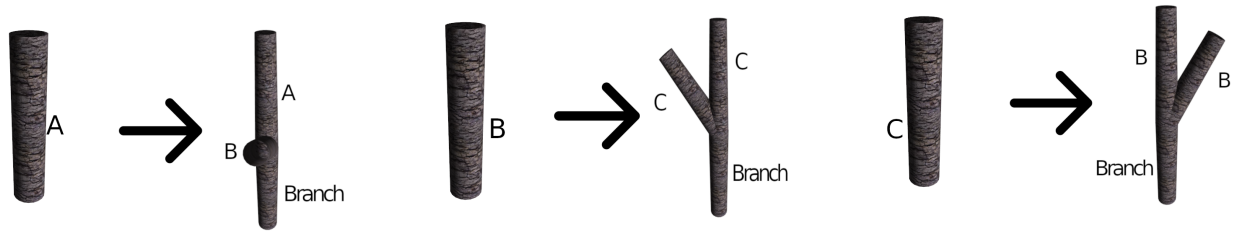

Figure 1 – Règles de génération

Une fois la génération terminée, les éléments sont convertis, texturés avec une image d'écorce et ajoutés dans un `mesh_drawable_hierarchy`.

Un autre modèle de génération, `ternary` a également été implémenté mais ne donnait pas des résultats satisfaisants.

1.3 Skybox

Pour coder la skybox, nous avons codé une classe permettant simplement de créer six carrés autour du monde et de les afficher avec la texture définie à sa création. Nos skyboxes sont composées de six images, qui ont été générées par Colin Lowdes (collines) et Chris Matz (espace) téléchargées sur le site `www.custommapmakers.org/skyboxes.php`.

Elles sont configurées pour tourner avec la caméra, de sorte que l'utilisateur fait toujours face au même côté bien qu'il tourne autour du cylindre.

2 ANIMATION

2.1 Joueur

Le joueur possède deux avatars qui alternent en fonction du niveau. Tous deux ont été téléchargés sur `free3d.com`. Le premier est un vaisseau spatial créé par `ravensky20's` que nous avons choisi de ne pas texturer, et le deuxième est un dragon modélisé par `timrh`. Celui-ci possède trois textures (bleue, verte et jaune) affichées en alternance, et est animé au niveau des ailes et de la tête.

L'avatar se déplace automatiquement et à vitesse constante autour du cylindre. Le joueur peut déterminer le déplacement latéral (coordonnée x_{cible}) souhaité de l'avatar. Afin de donner une impression de fluidité, l'avatar ne se place pas instantanément à la position cible. A la place, sa position x est interpolée linéairement entre sa position précédente et la position cible avec un facteur d'interpolation constant $\alpha = 0.05$: $x = \alpha x_{cible} + (1 - \alpha)x$.

La rotation de l'avatar est alors reliée à son déplacement réel. L'angle de rotation latéral est simplement proportionnel à $x_{i+1} - x_i$. Comme le déplacement de l'avatar est fluide grâce à l'interpolation, sa rotation l'est aussi. De plus, on obtient une certaine cohérence entre la rotation et la translation de l'avatar.

2.2 Collision

La gestion des collisions se fait très simplement. D'une part, on teste la position du joueur sur la largeur du cylindre pour savoir s'il est toujours à l'intérieur ; d'autre part, on calcule la distance entre le joueur et les obstacles, et si elle est trop petite, on suppose qu'il y a collision. Cette distance est arbitraire et fixe ; nous n'avons pas pris en compte le fait que le dragon est moins large lorsqu'il penche.

A noter qu'il existe un moyen de désactiver les collisions en cochant la case `GOD MODE` dans le gui.

2.3 Caméra

La caméra est en un sens le coeur de notre jeu. Elle est fixe selon l'axe x (dans la largeur du terrain) et tourne autour de notre cylindre. Originellement, il était prévue de la fixer derrière l'avatar et de le suivre dans les déplacements, mais cela provoquait une image tremblotante. Aussi avons-nous bloqué ses déplacements.

2.4 Callbacks

Nous avons mis en place des callbacks pour les touches du clavier (un jeu ne pouvant s'envisager sans interaction avec l'utilisateur…). Nous utilisons donc les flèches gauche et droite pour décaler l'avatar du joueur sur l'écran. Après quelques essais infructueux pour gérer le fait de maintenir une touche appuyée, nous avons fini par mettre en place deux booléens qui sont activés lorsqu'on appuie et relâche une touche ; tant que le booléen est actif, on déplace l'avatar.

2.5 Gameplay

Le jeu prenant place sur un cylindre, il nous a paru naturel de faire correspondre un niveau à un tour de terrain. Ainsi, à chaque tour réalisé (matérialisé dans le jeu par un portail), on change l'aspect de la scène. Pour l'instant, seuls deux niveaux différents sont implémentés.

— Les décors et obstacles sont rendus une fois sur deux normalement, et une fois sur deux en wireframe ;
— La skybox est modifiée, alternant entre des collines et l'espace ;
— L'avatar du joueur passe d'un dragon (dont la couleur peut varier) à un vaisseau spatial.

De plus, on génère à chaque niveau des positions différentes pour les obstacles. Afin d'avoir une progression dans la difficulté, 5 obstacles supplémentaires sont créés par niveau et la vitesse de défilement est incrémentée.

3 RENDU

Le rendu se fait en deux passes pour permettre le rendu d'ombres portées. La méthode a été inspirée du tutoriel `https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping`. La boucle de rendu dans `main.cpp` a été modifiée en conséquence.

3.1 Pass 1 : calcul des ombres

On calcule les ombres en générant une `depthMap` qui contient le *L-Buffer* de la scène, c'est-à-dire la profondeur minimale d'un objet pour chaque pixel vu depuis la source de lumière.

Pour ce faire, on réalise un rendu de la scène dont la sortie ne s'affiche pas à l'écran mais est stockée dans une texture de profondeur. On bind un framebuffer contenant notre `depthMap`, puis on render la scène avec un shader spécifique aux ombres et avec les matrices de vue et de perspective de la source de lumière.

La `depthMap` générée sera utilisée comme texture dans le rendu.

3.2 Pass 2 : rendu de la scene

La scène est ensuite rendue avec le shader classique, et avec les matrices de vue et de perspective de la caméra. La `depthMap` est attachée à `GL_TEXTURE1`. Alors, au moment du calcul d'illumination, la distance du fragment par rapport à la lumière est calculée et est comparée à la distance (minimale) stockée dans la texture de profondeur. Dans le cas où on se trouve derrière un autre objet, seule l'illumination ambiante est gardée, les reflets diffus et spéculaire sont annulés.

4 CONCLUSION

Nous avons créé ce petit jeu vidéo avec les méthodes présentées ci-dessus. Le code est relativement générique, notamment pour les obstacles : on travaille directement avec la base class `obstacle` dont hérite `arbre`. Pour changer de modèle d'obstacle, il suffit alors de créer une nouvelle classe et de l'utiliser à la place des arbres. De ce fait, il est relativement simple de l'étoffer pour ajouter différents niveaux, bien que nous n'ayons pas essayé de le faire.

Notre code se situe dans le dossier `code` à la racine de l'archive et se compile de la même façon que les TP. Nous espérons que vous prendrez plaisir à y jouer.