LANGUILLE Antoine Groupe A

MICHELI Sébastien

**Voxel Rush**

Rapport de projet de TSI

# Introduction :

Ce projet a pour but de nous faire manipuler les fonctionnalités d’OpenGL à travers la réalisation d’un jeu vidéo. Nous avons décidé de nous inspirer du jeu Geometry Dash : le but du jeu est d’aller au bout d’un niveau en sautant pour éviter les obstacles. Votre personnage (un cube pouvant sauter) se déplace tout seul et il faut le faire sauter pour passer au-dessus des différents obstacles sur votre route à travers différents niveaux.

Voici donc un rendu du jeu à l’écran :

Une image contenant dôme, bâtiment

Description générée automatiquement

Le cube bleu de gauche avec une spirale est votre personnage.

Le petit rond noir est un tremplin, passer dessus vous permettra de faire un saut plus haut que celui de base de votre personnage.

Le triangle rouge est un pic. Si vous le touchez vous mourrez et vous retourner au début du niveau.

Le cube de droite est une plateforme sur laquelle vous pouvez sauter. Attention, si vous arrivez de face en rentrant dans la plateforme, vous mourrez également. Il faut donc bien faire attention à arriver au-dessus de celle-ci.

# Prérequis :

Pour lancer le jeu, il suffit d’exécuter le programme *main* avec les modules python pyrr, openGL et glfw installés.

Le développement ayant été effectué sous Windows par manque de machine virtuelle, certaines fonctions des shaders de fragment sont différentes entre Windows et Linux, un switch initial a donc été fait pour séléceionner la version adaptée sur Windows ou sur Linux. Si vous êtes sous Linux, entrez « y » dans le terminal pour ouvrir la fenêtre du jeu, sinon sous Windows entrez « n ».

# Commandes :

- ECHAPPE: Retour au menu arrière / affichage du menu de pause / quitter le jeu

- ESPACE : Séléctionner menu / sélection niveau / sauter

- Flèche > : sélectionne le niveau suivant

- Flèche < : sélectionne le niveau précédent

- H visualiser les hitbox

# Structure du code :

Le code est composé de différents objets et fonctions réparties en plusisurs fichiers permettant chacun d’effectuer une partie précise du jeu. Nous allons donc vous expliquer les parties importantes de chacun d’entre eux.

Fichier *main* :

Initialise les différents objets du programme, y compris la fenêtre et les différentes valeurs partagés, il lance la boucle d’affichage.

Fichier *background* :

On définit le fond d’écran de la zone jouable. On se sert pour cela de 3 rectangles ayant la même texture et placés de telle sorte qu’ils bougent en accord avec la caméra.

Dans ce jeu, la caméra et le fond bougent, les obstacles restent fixes.

Une image contenant dôme, bâtiment

Description générée automatiquement

Cette zone se décompose en trois parties :

- Les parties 1 et 3 qui sont faces à la caméra, faisant office de mur.

- La partie 2 qui relie ces deux murs et sert de sol pour notre jeu. C’est sur cette partie que le joueur et les obstacles seront situés.

Fichier *cpe3d* :

Ce module regroupe les fonctionnalités de base permettant de créer, modifier et de manipuler diverses Objets3D, elle se compose de plusieurs objets distincts :

\_ Transformation3D gère l’offset, la translation, ainsi que la rotation des vertex des maillages en utilisant les angles d’Euler, grâce à la librairie pyrr. Ces paramètres de transformation seront envoyées au vertex shader pour pouvoir modifier la position, orientation... des objets affichés. À chaque fois que le mot « transformation » est utilisée, cet objet est référencé de manière sous-jacente.

\_ Object est une classe visant à être héritée par Objet3D et Text, elle sert principalement à définir des méthodes générales à tous les objets, tels que la méthode \_\_repr\_\_ pour le débug ainsi que la méthode draw() permettant d’afficher les Objets3D grâce à OpenGL (maillage, texture, type de tracé etc).

\_ Object3D est la classe principale de notre jeu vidéo. Héritant d’Object, elle constitue la classe qui sera héritée dans le cas où l’on voudra crée un objet 3D personnalisé en terme de forme, de texture, de coordonnées, transformation etc

Elle regroupe également, en plus des fonctions de définition d’affichage (maillage, texture etc), des fonctions utilitaires permettant à partir de la boite de collision bounding\_box (constituée d’un point min et d’un point max) les coordonnées de tous les 8 points du cube (pour la détection de collisions par exemple).

La fonction draw() est surchargée, de telle sorte de fournir au vertex shader, les transformations à effectuer provenant de Transformation3D (offset, translation, rotation, centre de rotation) pour le calcul de la position final des vertex.

\_ Camera est un objet, au même titre que le Texte, le Player ou les Objets3D, qui se distingue ici surtout par une matrice de projection dont les paramètres sont réglés dans le code, il sera utilisé dans la boucle d’afffichage run() de viewerGL.py.

\_ Text est la classe permettant d’afficher un texte précisé en paramètre lors de sa création, incluant un caractère spécial (ici ’&’) pouvant être remplacé par une valeur enregistrée (similaire au .format des str de python), principalement utilisé pour l’affichage des titres VOXEL RUSH ou le nombre de tentatives par exemple. Une option de sélection de couleur est également implémentée, permettant d’ajouter et de change la couleur du texte.

Fichier *hitbox*:

Ce module comporte une unique classe, qui est en fait un objet3D représentant un Cube dont les dimensions sont réglables, en vue d’être affichée pour permettre de visualiser les zones de collisions des objets.

Elle sont affichables grâce à la touche H du clavier, permettant de les afficher ou de le cacher.

IMAGE

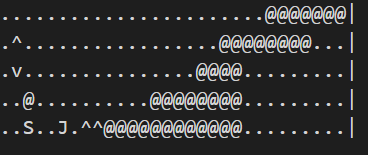
Fichier *glutils* :

Ce programme comporte des fonctions utiles pour initialiser et compiler les shaders, ainsi que pour charger des images sur le GPU. L’import de texture a surtout été utilisé lors de la création des objets 3D.

Fichier *level* :

La classe Level contenue dans ce fichier gère la lecture des fichiers level, leur chargement, la création des objets. C’est également cet objet qui détermine, à partir de la position du joueur, quels obstacles doit être affiché, de sorte de limiter la charge de calcul et de rendu visuel.

La première étape est de charger le niveau que le joueur a choisi. Chaque niveau correspond à un fichier texte dont les symboles écrit à l’intérieur vont être associé à un obstacle. Voici à quoi ressemble une partie du fichier texte du niveau de test :

IMAGE LEVEL

Chaque ligne représente la position (selon x,y) où sera placer un objet. Ils sont ensuite affichés dans l’ordre des symboles. Chaque symbole va correspondre à quelque chose à afficher :

- . : on ne place pas d’obstacle.

- J : on place un tremplin (J pour Jump).

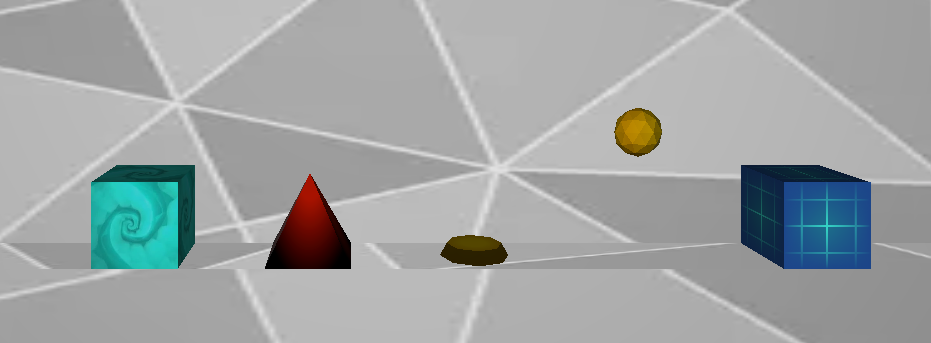
- @ : on place un Cube.

- <, v ou ^ : on place un pic orienté selon le sens du symbole.

- | : fin du niveau.

- S : position de départ du personnage.

- D : Double saut, permet un double saut en l’air (Double jump)



Le fichier est lu et chaque obstacles est crée et placé à l’endroit correspondant à sa position dans le fichier texte, de sorte que le caractère S corresponde à la position (0,0,0) de l’espace.

Fichier menus :

La classe Menus contenue dans ce fichier sert à initialiser et stocker les objets Text ainsi que leur couleur en fonction de l’état du jeu (niveau, écran titre, écran de sélection)

Fichier *mesh* :

La classe Mesh, fournie par le professeur est une classe utilitaire, au même titre que glutils, permettant de lire un maillage depuis un fichier obj, de le transformer en vertex, triangles et coordonnées de texture.

Elle permet aussi, une fois ce maillage chargé, normalisé ou modifié à l’aide d’une matrice pyrr, d’envoyer les informations sur les vertex bruts à la carte graphique, pour un stockage sur la VRAM.

Fichier *obstacles* :

On définit une classe pour chaque type d’obstacles. Les classes définies sont les Cubes, Pics, Jump, et Double Jump, héritant toutes d’Objet3D pour pouvoir être représentées à l’écran avec un modèle et une texture unique, obtenus grâce au logiciel libre Blender.

Pour éviter une surcharge mémoire du GPU, les points de chaque modèle ne sont chargés qu’une seule fois sur le GPU, ils sont réutilisés grâce à leur vao, en modifiant alors l’orientation et la translation.

Chaque objet possède une boite de collision (AABB) (visible grâce aux hitbox) qui lui est propre et peut être ajustée pour changer le comportement du mécanisme de détection des collisions.

Fichier *player* :

Cette partie est dédiée aux fonctionnalités rattachées au joueur, à son déplacement et à la gestion des collisions.

Le joueur est semblable à un cube (pour respecter l’inspiration principale de notre jeu vidéo) ayant la seule capacité de sauter, le déplacement horizontal étant fixé et imposé.

Le déplacement est géré grâce aux équations du mouvement (accélération, vitesse et position), la position étant représentée par la composante translation de la transformation associée à l’objet 3D représentant le joueur. Les intervalle de temps utilisée est l’intervalle de temps entre la frame présente et la frame précédente.

Outre le déplacement, les collisions sont également gérées dans cette partie :

Le principe de base est l’intersection des AABB définies pour chaque objet :

La joueur récupère les objets proches de lui (pour ne pas scanner tout le niveau), obtiens leur coordonnées de leurs 8 points et vérifie pour chaque point du Player (aussi un cube à 8 sommets), si le sommet en question intersecte une AABB d’un autre objet.

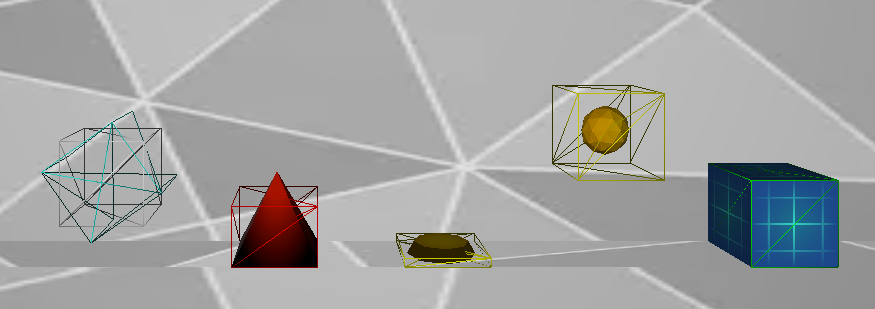
Si un sommet intersecte en effet un obstacle, la situation est évalulée au cas par cas

Dans le cas d’une collision avec un Jump, une composante verticale est simplement affectée au Player pour le faire sauter, dans le cas d’un Double Jump, c’est la même chose si l’on appuie sur ESPACE à proximité de cet obstacle.

Pour le Spike, n’importe quelle intersection signifie la mort du joueur.

Le cas le plus compliqué est le cube, en effet, si le Player arrive « de face » le Player doit mourir mais si celui-ci arrive depuis le « haut » il faut que le Player se « rattrape » sur le bord du Cube, sans mourir s’il y a un Spike en dessous.

Les comportements désirés ont principalement été obtenus en mofidiant les zone de collision des différents objets : Le Player fait moins d’un bloc de haut, pour ne pas mourir s’il passe exactement sous un Cube ou Spike, le Cube a eu une « zone de tolérance » ajoutée pour permettre le « rattrapage » sur un de ses bords...



Fichier *viewerGL* :

Cet objet est la base de tout jeux vidéo 3D, il contient l’initialisation de la fenêtre et du contexte OpenGL ainsi que la boucle d’affichage contenant : La gestion des touches, la gestion du temps, l’actualisation des positions inter-objets ainsi que l’affichage de tel ou tel objet grâce à leur attribut visible.

Les objets à afficher en fonction de l’état du jeu (menu, niveau, écran titre etc) sont regroupés dans une grande liste qui est parcourue séquentiellement, chaque élément étant affiché avec sa hitbox.

# Conclusion :

Récap de ce qu’on a fait.