BALLAND Alexis
FERRY Alexandre
3ETI

Projet TSI

I. Introduction

Pour notre jeu 3D minimaliste, nous nous sommes basés sur le concept d'un jeu connu, déjà existant, *Crossy Road.* Le but de notre jeu est simple, on contrôle le stegosaurus et il suffit de traverser la route avec de nombreuses voitures qui traversent afin de passer au niveau supérieur. Cependant, le niveau de difficulté va également augmenter, ce qui entraîne l'augmentation de la vitesse de la voiture qui traverse la route. Si a un moment l'une des voitures rentre en contact avec le stegosaurus qu'on contrôle, on revient au premier niveau de jeu. Et pour rendre le jeu un peu plus compliqué, pour chaque niveau, il est nécessaire de traverser la route avant que le chronomètre atteigne 0.

II. Fichiers Shader

Shader.frag : La fonction de ce fragment shader est de calculer la couleur finale de chaque pixel d'un objet en tenant compte de l'éclairage et des textures appliquées. Le fichier combine les effets d'illumination ambiante, diffuse et spéculaire pour donner une rendu réaliste. En utilisant les coordonnées de texture, la normale du fragment et les propriétés de la lumière, le shader détermine la couleur finale du pixel, ce qui permet d'afficher des objets 3D avec des textures détaillées et un éclairage dynamique.

Shader.vert : La fonction de ce vertex shader est de calculer la position et les propriétés des sommets d'un objet en appliquant des transformations de modèle, de vue et de projection. Le fichier transforme les coordonnées des sommets en espace écran, gère les normales pour l'éclairage et transmet les couleurs et coordonnées de texture aux fragments shaders. Cela permet de préparer les données nécessaires pour afficher correctement les objets 3D avec les effets d'éclairage et de texture dans une scène graphique.

Gui.frag : La fonction de ce fragment shader est d'afficher les caractères à partir d'une texture contenant une grille de caractères ASCII. Il utilise les coordonnées de texture et d'un code ASCII pour extraire le caractère approprié de la texture et le rendre à l'écran. En ajustant les coordonnées de texture, il positionne correctement chaque caractère dans la grille. Si la couleur resultant est presque noir, le fragment est ignore. Cela permet de rendre des textes en utilisant une seule texture, facilitant l'affichage de texte dans des applications graphiques.

Gui.vert : Ce fichier est une vertex shader pour les objets 2D. Il transforme les coordonnées des sommets d'un objet pour le positionner correctement sur l'écran et génère les coordonnées de texture associées. Il ajuste la taille et la position de l'objet avec les uniformes 'size' et 'start', puis calcule la position finale dans l'espace écran. Cela permet d'afficher des objets 2D avec des textures appliquées à la bonne échelle et position.

III. Affichage d'un objet 3D

Pour charger et afficher des objets 3D avec OpenGL et des shaders en python, nous avons :

Utiliser une bibliothèque pour charger un fichier OBJ qui contient le modèle 3D de l'objet et on normalise le maillage pour s'assurer que ses dimensions sont dans une plage standard, facilitant la manipulation et l'affichage. On applique par la suite une matrice de mise à l'échelle pour redimensionner l'objet selon les besoins.

```
m = Mesh.load_obj('stegosaurus.obj')
m.normalize()
m.apply_matrix(pyrr.matrix44.create_from_scale([1.5, 1.5, 1.5, 1]))
```

Ensuite, on crée un objet de transformation pour définir la position, la rotation et l'échelle de l'objet dans l'espace 3D et on charge une texture pour l'appliquer à l'objet.

```
tr = Transformation3D()
tr.translation.y = -np.amin(m.vertices, axis=0)[1]
tr.translation.z = -3
tr.rotation_center.z = 0.5
texture = glutils.load_texture('stegosaurus.jpg')
```

Finalement, on crée l'objet 3D avec le maillage, le nombre de triangles, l'identifiant du programme shader, la texture, et la transformation et on l'ajoute a la vue.

```
o = Object3D(m.load_to_gpu(), m.get_nb_triangles(), program3d_id, texture, tr)
viewer.add_object(o)
```

Pour afficher une autre fois le même objet, on n'a besoin pas de recharger le maillage mais on doit réinitialiser une transformation 3D en définissant des vecteurs de position différentes pour l'objet et on peut également changer la texture (comme on l'a fait pour les différents skin du stegosaurus).

```
tr = Transformation3D()
tr.translation.y = -np.amin(m.vertices, axis=0)[1]
tr.translation.z = 5
tr.translation.x = 0
rotation_angle = np.radians(180)
tr.rotation_euler[2] += rotation_angle
texture = glutils.load_texture('stego_vert.png')
o = Object3D(m.load_to_gpu(), m.get_nb_triangles(), program3d_id, texture, tr)
viewer.add_object(o)
```

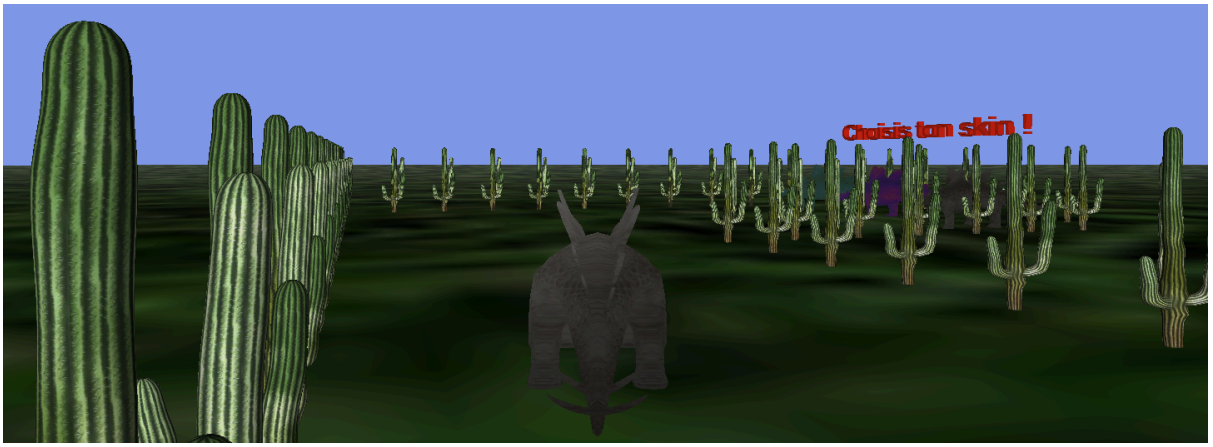
IV. Affichage d'un texte

Nous commençons par initialiser la géométrie nécessaire en configurant un vao avec la méthode Text.initialize_geometry(), qui définit les sommets et indices des triangles pour les caractères. Ensuite, nous chargeons une texture contenant les caractères de la police à partir du fichier 'fontB.jpg' à l'aide de glutils.load_texture. Nous créons une instance de la classe Text, en précisant le texte à afficher, sa position, son échelle, le VAO, le programme

shader et la texture. Nous ajoutons ensuite cet objet texte à la scène. Pour le rendu, la méthode draw de l'objet texte désactive le test de profondeur, ajuste la taille de chaque caractère, passe les informations nécessaires aux shaders via des variables uniformes, puis dessine chaque caractère un par un en utilisant des triangles.

V. Création de la map avec les cactus

Afin de dessiner notre surface de jeu avec les cactus, nous avons créé un fichier cactus.py dans lequel nous avons créé la classe Cactus. Cette dernière nous permet de créer et manipuler notre objet catus dans l'environnement 3D. On y retrouve le chargement du modèle 3D, le chargement dans le GPU, la gestion des texture etc.
Ensuite, dans le main.py, nous avons créé la fonction create_cactus qui va nous permettre de positionner nos cactus. Nous avons créé des variables pour la longueur et la largeur de nos segments de cactus et avons utilisé notre fonction pour créer une surface de jeu satisfaisante. Nous avons finalement ajouté tous les cactus à notre liste de cactus et les avons ajoutés à notre liste d'objets. Nous retrouvons alors bien nos cactus correctement alignés :



Il nous faut maintenant gérer la collision entre ces derniers et notre stégosaure. Pour cela, nous calculons la future position de notre stégosaure si la flèche avant (ou arrière) est appuyée :

```
future_position = self.objs[0].transformation.translation
if glfw.KEY_UP in self.touch and self.touch[glfw.KEY_UP] > 0:
    future_position = self.objs[0].transformation.translation + pyrr.matrix33.apply_to_vector(pyrr.matrix33.create_from_eulers(self.objs
                                                                pyrr.Vector3([0, 0, 0.08]))

elif glfw.KEY_DOWN in self.touch and self.touch[glfw.KEY_DOWN] > 0:
    future_position = self.objs[0].transformation.translation - pyrr.matrix33.apply_to_vector(pyrr.matrix33.create_from_eulers(self.objs
```

Nous avons ensuite créé une fonction qui renvoie si oui ou non (True ou False) notre future position est valable. Cette fonction parcourt notre liste de cactus. La position est valable si la norme de la différence entre notre future position et la position de chaque cactus est supérieure à 0,5. Si ce n'est pas le cas, c'est-à-dire si un des cactus est à moins de "0,5" de notre stégosaure, alors la fonction renvoie False :

```
def future_position_valable(self, future_pos):
    for obj in self.objs[6:57]:
            d = future_pos - obj.transformation.translation
            distance = pyrr.vector.squared_length(d)

            if distance < 0.5 :
                return False

    return True
```

Dans notre méthode run, nous utilisons cette fonction. Si elle renvoie True, alors la position du stégosaure devient la future position. Nous n'utilisons donc plus les méthodes update_key_arriere et update_key_avant pour nous déplacer.

VI. Création et déplacement des voitures

Tout comme pour les cactus, nous avons créé un fichier voiture.py dans lequel nous créons la classe voiture qui est similaire à la classe Cactus, cependant nous avons aussi créé dans ce fichier la classe VoiturePolice et Voiture Course qui érite de Voiture mais qui utilise directement l'objet voiture_course ou police. Ensuite, dans main.py, nous utilisons random afin d'initialiser aléatoirement une voiture de course ou une voiture de police afin de diversifier l'aspect des voitures. Si randint renvoie 1, nous actualisons une voiture de police, sinon une voiture de course. Nous plaçons les 7 premières voitures sur x = -20 et les 7 autres sur x = -23 afin d'avoir une route de 2 voies :

De plus, on initialise chaque voiture avec un espace entre elles afin de ne pas avoir un gros pack de voitures. Nous avons choisi d'ignorer les collisions entre les voitures afin de ne pas avoir de bouchons, ce qui pourrait rendre le jeu trop simple.

Pour le déplacement des voitures, nous créons une fonction mouvement dans la méthode run. Dans celle-ci, nous parcourons notre liste de voitures. Nous divisons notre traitement pour chaque ligne. Pour chaque ligne, nous créons une initial_position, position à laquelle nous voulons faire spawn nos voitures en boucle. Ensuite, nous appliquons à chaque voiture une translation sur l'axe z que l'on multiplie par le niveau, afin d'augmenter la vitesse des voitures lorsque le niveau augmente et par un randint entre 1 et 3 afin d'avoir différentes vitesses pour les voitures. Enfin, nous avons créé une target_position : lorsque les voitures arrivent à cette position, elles seront téléportées à l'initial_position. Avec ce système, nous avons donc des voitures qui avancent en boucle. Nous mettons une target_position à 20, que l'on utilisera sur l'axe des z. Nous soustrayons donc 20 à la position en z de chaque voiture. Si cette valeur est est inférieure à 0,01, nous positionnons notre voiture à initial_position :

```
    if (target_position - obj.transformation.translation[2]) < 0.01:
        obj.transformation.translation = initial_position
```

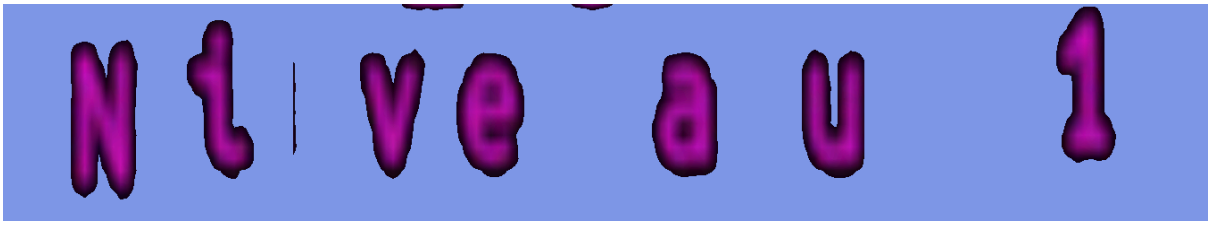
Il ne nous reste maintenant plus qu'à gérer les collisions entre les voitures et le stégosaure. Pour cela, nous parcourons notre liste de voitures. Si la norme de la distance entre la position d'une des voitures et la position de notre stégosaure est inférieure à 1, nous faisons quelque chose dans le if :

```
for obj in self.objs[57:71] :
    d2 = moving_obj.transformation.translation - obj.transformation.translation
    d = pyrr.vector.squared_length(d2)
    if d < 1 :
```

Nous expliquerons ce qu'il se passe dans ce cas dans la partie Gestion des niveaux.

VII. Gestion des niveaux

Premièrement, nous ajoutons via main.py 3 textes à notre jeu en utilisant programGUI_id. Le premier affiche le niveau actuel :



Le deuxième affiche un décompteur :

Et le dernier affiche le niveau record que l'on a atteint :

Ensuite, dans la classe ViewerGL, nous créons différentes méthodes : la méthode update_level_text permet de modifier le texte affichant le niveau actuel. La méthode update_best_score permet de modifier le texte affichant notre niveau record. La méthode update_decompteur permet de faire fonctionner le décompteur. Elle récupère le temps actuel et le soutrait en boucle à un nouveau actuelle. Lorsque cette différence est supérieure à 1, on enlève un à decompteur.value et on modifie le texte du décompteur en y affichant la nouvelle valeur. Enfin, la méthode reset_game remet le niveau à 1, decompteur_value à 30, rafraîchit le texte du niveau actuel et du décompteur et repositionne le stégosaure à sa position initiale.
Ensuite, dans la méthode run, lorsque la position du stégosaure est inférieure à 25.5, c'est à dire lorsque qu'il a traversé la route, on positionne ce dernier à sa position initiale, on augmente le niveau de 1, on remet le decompteur_value à 30 et on rappelle les méthodes update_decompteur et update_level_text pour les mettre les textes à jour. De plus si self.niveau, le niveau actuel, est supérieur à self.best_niveau, le niveau record, alors on appelle self.update_best_score pour mettre à jour le texte.
Enfin, reset_game est appelé dans 2 cas : si decompteur_value est égale à 0 ou si une collision est détectée avec une des voitures :

```
for obj in self.objs[57:71] :
    d2 = moving_obj.transformation.translation - obj.transformation.translation
    d = pyrr.vector.squared_length(d2)
    if d < 1 :
        self.reset_game(init_stego_pos)

if self.decompteur_value == 0 :
    self.reset_game(init_stego_pos)
```

VIII. Gestion des skin

Nous avons mis en place un système de changement skin (se traduisant par un changement de texture) pour notre stégosaure qu'on contrôle. Dans notre fichier main.py, nous avons initialisé 3 stegosaurus différents avec des textures différentes qui sont nos skins qu'on peut sélectionner. Nous les avons placé côte à côte :

Afin de changer le skin de notre stegosaurus, il suffit de rentrer en contact avec l'un des 3 stegosaurus présents. Dans notre fichier viewerGL, on calcule la distance au carré constamment entre notre personnage et l'un des skins présents. Si cette distance devient inférieur à 2, alors on appelle la méthode change_skin :

```
if pyrr.vector.squared_length(self.objs[0].transformation.translation-self.objs[3].transformation.translation) < 2:
    self.change_skin('stego_radio.png')
if pyrr.vector.squared_length(self.objs[0].transformation.translation-self.objs[1].transformation.translation) < 2:
    self.change_skin('stegosaurus.jpg')
if pyrr.vector.squared_length(self.objs[0].transformation.translation-self.objs[2].transformation.translation) < 2:
    self.change_skin('stego_vert.png')
```

Cette méthode permet de changer la texture de notre stégosaure. Elle lui donne la texture passée en argument.

Nous avons également créé un objet pour l'affichage du message rouge qu'on place bien au-dessus des skins grâce aux translations et rotations choisies.

IX. Conclusion

A travers ce projet, nous avons appris à manipuler openGL pour le rendu d'image 2D et 3D, en exploitant les fragments shaders et vertex shaders qui nous ont permis de contrôler la manière dont les objets sont positionnés.

De plus, ce projet nous a introduit à plusieurs nouvelles bibliothèques en python comme viewerGL, glutils, pyrr…

L’utilisation d’OpenGL était assez floue après l’approche théorique mais l’élaboration de ce jeu nous a permis de mieux comprendre comment tout cela s’organise.