TSI Projet Compte-Rendu

DO Frédéric
&
TRUONG Guillaume

23 Juin 2023

Introduction

Notre projet est de réaliser le jeu PAC-MAN en 3D et à la première personne en python avec OpenGL. Pour cela, nous avons utilisés le projet d'origine et y avons fait des modifications.
Voici les règles :

1. - Il s'agit d'un Pac-man en 3D à la première personne, le but est comme dans le jeu original, c'est-à-dire de ramasser toute les points (boules blanches) dans le labyrinthe
2. - Les mouvements se feront avec les touches ZQSD ainsi que des mouvements de caméra grâce à la souris
3. - Si vous prenez une des 2 sorties du labyrinthe, vous serez téléporté à l'autre sortie comme dans le jeu originel
4. - Lorsque vous marchez sur une boule blanche, elle disparaît et vous augmentez votre score qui est affiché avec l'appellation "score"
5. - L'affichage "Points restants" vous indique le nombre de boules blanches restantes dans le labyrithe avant de gagner la partie.
6. - Appuyer sur la touche "SPACE" vous fera voir la carte avec une vue du dessus du labyrinthe pour avoir une meilleure vision mais dans cet état, vous ne pourrez plus bouger
7. - Appuyer sur la touche "LEFT_CONTROL" vous fera repasser en vue première personne, il est possible de se déplacer uniquement dans cette état -
8. - 4 ennemies se déplacent aléatoirement dans le labyrinthe. Ils changent de direction toutes les 7 secondes ou lorsqu'ils touchent un mur
9. - Toucher un ennemi fait perdre une vie (3 vie au départ) et téléporte le joueur au centre du labyrinthe. Si le joueur n'a plus de vie il perd la partie.

Étapes de programmation

Création de la carte

Pour créer notre monde, nous avons d'abord pris le sol "grass" d'origine et modifier la texture. Ensuite, nous avons créer notre carte sous forme de matrice nommée *laby_matrice* dans notre code **labyrinth.py** en se basant sur la carte originale de PAC-MAN. (*Figure 1*)

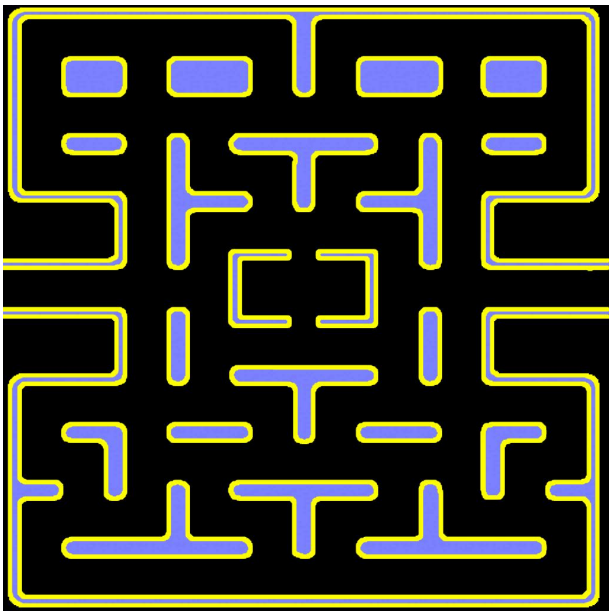

Figure 1: Carte originelle de PAC-MAN

A l'aide de cette matrice composé de *1* pour les murs, *2* pour les points, *3* pour les pommes et *0* pour le vide, on peut construire notre monde en remplaçant la valeur par les objets correspondants. Sachant qu'un mur est représenté par un cube et les points et les pommes par des sphères blanches et rouge. On ajoute alors tout les objets dans une liste pour pouvoir bien les manipuler et les afficher.

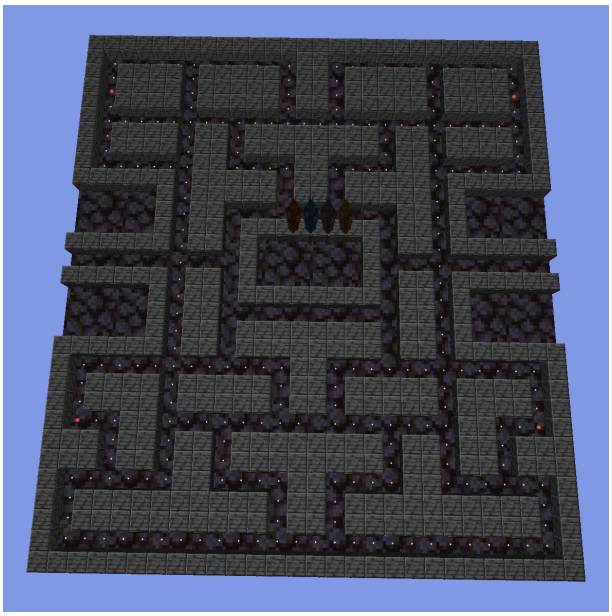

Figure 2: Carte du jeu

Dans notre monde, une unité de distance (taille d'un bloc) est de 2. On a alors converti la position de la matrice avec la position réelle en multipliant les index de la matrice par 2. On également créer les fantômes (stégosaures) au centre de la carte.
A noter que les ennemies sont censés être dans la cage mais par souci d'efficacité, nous les avons mis en dehors
Notre personnage quant à lui, apparaît dans la case 14/23 de la matrice (zone où il manque une boule blanche).

Fonctionnalités du joueur

Notre personnage peut se déplacer dans toutes les directions possibles et nous avons la possibilité de tourner la caméra. De plus, la caméra se bloque verticalement pour empêcher de retourner la caméra.
Nous pouvons changer de point de vue en appuyant sur la touche *SPACE* (*Figure 3*), ce qui place la caméra au-dessus du labyrinthe pour avoir une vue d'ensemble de la carte. Cependant, dans cet état nous avons bloqué la possibilité de se déplacer avec une variable *fps_view* dans **viewerGL.py** car ce jeu est d'abord d'un point de vue première personne, ce changement de point de vue sert uniquement à faciliter la jouabilité.
En appuyant sur *LEFT_CONTROL* (*Figure 4*), nous pouvons revenir en vue première personne dans le labyrinthe et ainsi bouger de nouveau. Tout le principe est d'alterner entre les 2 points de vues pour être le plus efficace.
Notre personnage n'est qu'une caméra et n'a pas d'objet associé.

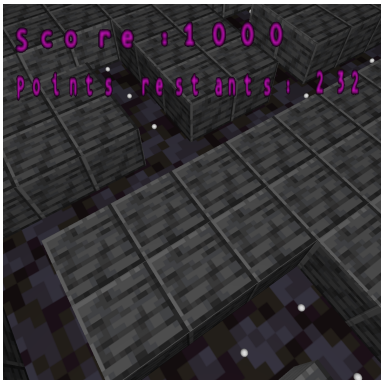

Figure 3: POV du dessus

Figure 4: POV FPS

Position et collision

Pour pouvoir implémenter les collisions avec les objets, nous avons utilisé la matrice *laby_matrice* pour repérer tout la position de tout nos objets ainsi que notre joueur. On a alors crée une fonction CONT_TO_DISC() qui traduit la position réelle des joueurs et des ennemies en sa position dans la matrice *laby_matrice*. Grâce à cela, nous pouvons alors déterminer lorsque le joueur entre dans le mur. Pour cela, à chaque appel de la fonction UPDATE_KEY(), nous faisons d'abord un test de la position du joueur si on se déplace, puis on le traduit dans la matrice avec la fonction CONT_TO_DISC() et vérifie si la valeur dans la matrice vaut 1. Si non, on peut se déplacer dans la direction voulut si oui, le programme n'effectue pas la demande de déplacement. (*Figure 5*) De plus, comme dans

```
if glfw.KEY_W in self.touch and self.touch[glfw.KEY_W] > 0: #avancer
    test_collision_x0 = self.cam.transformation.translation + \
        pyrr.matrix33.apply_to_vector(pyrr.matrix33.create_from_eulers(angle), pyrr.Vector3([0, 0, -0.5]))
    test_collision_x1 = self.cam.transformation.translation + \
        pyrr.matrix33.apply_to_vector(pyrr.matrix33.create_from_eulers(angle), pyrr.Vector3([0, 0, -0.25]))
    position_matrice_x, position_matrice_z = cont_to_disc(test_collision_x0[0], test_collision_x0[2])
    if not(laby_matrice[position_matrice_z][position_matrice_x] == 1):  #si il n'y a pas de collision
        self.cam.transformation.rotation_center = test_collision_x1
        self.cam.transformation.translation = test_collision_x1

if glfw.KEY_S in self.touch and self.touch[glfw.KEY_S] > 0: #reculer
    test_collision_x0 = self.cam.transformation.translation + \
        pyrr.matrix33.apply_to_vector(pyrr.matrix33.create_from_eulers(angle), pyrr.Vector3([0, 0, 0.5]))
    test_collision_x1 = self.cam.transformation.translation + \
        pyrr.matrix33.apply_to_vector(pyrr.matrix33.create_from_eulers(angle), pyrr.Vector3([0, 0, 0.25]))
    position_matrice_x, position_matrice_z = cont_to_disc(test_collision_x0[0], test_collision_x0[2])
    if not(laby_matrice[position_matrice_z][position_matrice_x] == 1):  #si il n'y a pas de collision
        self.cam.transformation.rotation_center = test_collision_x1
        self.cam.transformation.translation = test_collision_x1
```

Figure 5: Code des collisions

le jeu original, en essayant de sortir d'une des 2 sorties du labyrinthe, on se retrouve téléporter à l'autre sortie, notre carte est ainsi connecté. Pour cela, on a simplement changer la position de notre joueur lorsqu'il arrive à une de ces 2 zones de sorties. (*Figure 6*)

```
if x_pos_mat == 0 and z_pos_mat == 14: #sortie gauche du labyrinthe
    self.cam.transformation.rotation_center.x = 26
    self.cam.transformation.rotation_center.z = -18
    self.cam.transformation.translation.x = 26
    self.cam.transformation.translation.z = -18

if x_pos_mat == 27 and z_pos_mat == 14: #sortie droite du labyrinthe
    self.cam.transformation.rotation_center.x = -26
    self.cam.transformation.rotation_center.z = -18
    self.cam.transformation.translation.x = -26
    self.cam.transformation.translation.z = -18
```

Figure 6: Code de la téléportation

Pour gérer les collisions avec les points et les pommes, nous avons initialisé la position de chaque objets de la liste et créer une liste parallèle avec d'un côté les points et de l'autre les pommes. Ensuite par le même procédé, le programme regarde si le joueur est sur un point ou une pomme dans la matrice et si oui, on POP l'objet correspondant de la liste objet et la liste des points ou des pommes selon l'objet. Ainsi, on conserve le même indexage car POP enlève l'élément de la liste et change la position des objets dans la liste. Une fois l'objet enlevé de la liste des objets, le programme ne l'affiche plus.

A noter que toucher un ennemie fait perdre une vie et ramène à la position initiale

Affichage des informations

Pour l'affichage, nous avons décider d'afficher le score avec les points obtenus, les points restants et les vies .(*Figure 7*) Pour faire cela, nous avons crées une

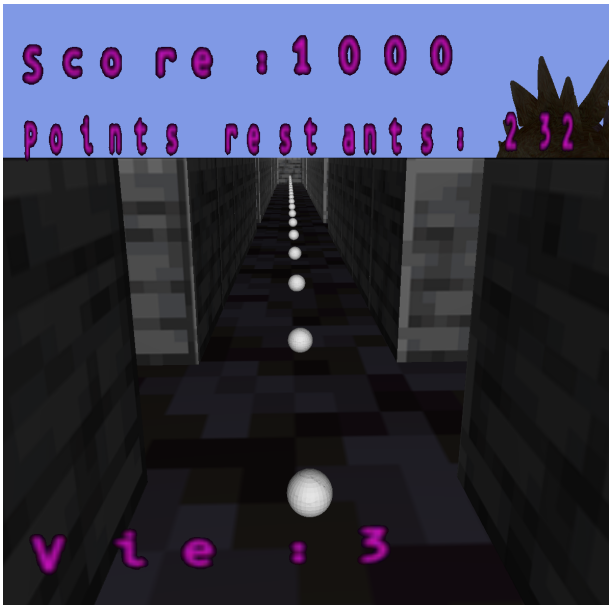

Figure 7: Affichage des informations

liste à part pour stocker les textes à afficher, on modifie alors la valeur de ces textes pour mettre à jour l'affichage.
De plus, nous avons ajouté un message de victoire (*Figure 8*) et de défaite (*Figure 9*) lorsqu'on ramasse tout les points ou que le nombre de vie tombe à zéro. Cette vérification se fait avec un simple IF.
Le score ajoute 100 au score tout les points récupérer, il a suffit alors d'incrémenter de le score à chaque collision avec un point. Pour les points restants, *len(self.Liste_points)* renvoi le nombre de points restants. Et pour les vies, il suffit de décrémenter.

Figure 8: Écran de victoire

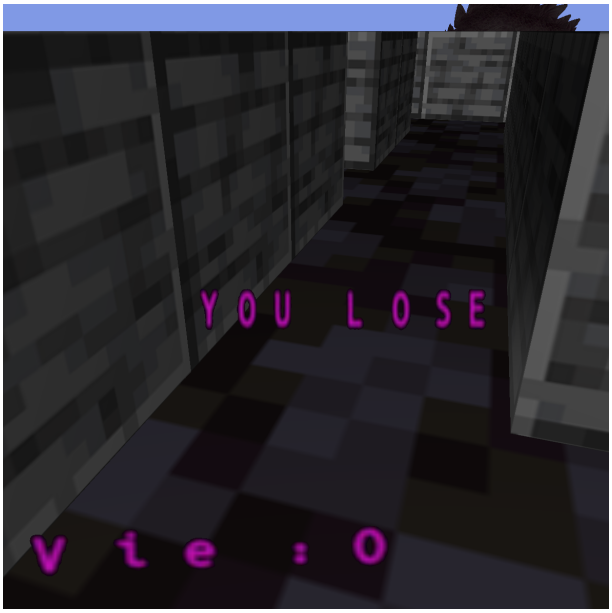

Figure 9: Écran de défaite

Pour aller plus loin

Pour améliorer notre jeu, nous avions prévu de bien configurer la musique mais surtout implémenter la mécanique d'invincibilité du joueur à la prise d'une pomme. Nous aurions pu également améliorer le déplacement des ennemies mais par manque de temps, nous n'avons pas pu ajouter ces fonctionnalités.