Ben Salah Mariem
Taider Silia

Groupe A
09/01/2022

SNOW SURFER

I. Introduction

L'objectif de ce projet est de créer un jeu minimaliste en employant les connaissances de bases sur openGL et C++ acquises en cours ainsi que démontrer une certaine maîtrise des techniques mises en jeu. Notre jeu consiste en un bonhomme de neige qui découvre son environnement et les différents éléments de décor qui le constituent tout en évitant les obstacles (monstres) sur son parcours.

Le jeu dispose de certaines fonctionnalités qu'on détaillera par la suite, la plus marquante étant celle qui fait transformer le personnage du bonhomme à chaque fois qu'il rencontre un objet spécifique, par exemple : il se transforme en père noël s'il rencontre un chapeau rouge.
Le joueur déplace le bonhomme là où il regarde, le faire avancer ainsi que reculer avec le clavier, plus précisément en utilisant les touches :

- A : pour tourner à gauche
- E : pour tourner à droite
- Q : pour reculer
- D : pour avancer

S'il veut donc tourner à droite (ou à gauche), il n'a qu'à faire tourner le bonhomme avec la touche "E" (ou "A"), et ensuite avancer avec la touche "D".

Il faut noter que la caméra est située derrière notre personnage principal, en suivant le sens où il se dirige (tourne avec lui et avance avec lui), on le voit donc toujours de dos.

Notre travail s'est déroulé de la manière suivante : nous avons d'abord commencé par créer les objets dont nous avions besoin (sapin, bonhomme, monstre, éléments de décoration) et fixer leurs positions, ainsi que celle de la caméra.

Nous nous sommes intéressées dans un second temps aux déplacements des différents objets constituant notre monde, principalement le bonhomme, la caméra qui le suit, et les monstres.

En dernier, nous nous sommes focalisées sur la gestion des différentes interactions entre les objets, notamment la collision et ce qui en résulte comme action (perte du jeu si collision avec un monstre par exemple).

Dans ce compte-rendu, nous expliquerons ce que nous avons réalisé et les méthodes que nous avons utilisées.

II. Création et importation de nouveaux objets

Pour commencer, nous avons démarré du squelette du code qui nous a été proposé et ensuite nous nous sommes inspirées des codes exécutés pendant le TP synthèse.

Pour stocker les objets différents, nous avons donc créé une liste d'objets d'une taille initialisée à nb_obj = 200.

A noter que tous les objets utilisés ont bien été traités par nous sur blender (redimensionnement, déplacement, rotation, nettoyage du maillage.).

1. Le bonhomme

Nous avons choisi un bonhomme de neige comme personnage principal de notre jeu s'inspirant de la période des fêtes en décembre (exporté en .obj depuis le logiciel blender).

Ce bonhomme est le premier élément de notre liste d'objets (obj[0]), cet élément a été créé et géré dans la fonction init_snowman().

2. Le sol

Pour que le sol (obj[1]) soit en cohérence avec le thème, nous avons choisi un sol blanc avec une texture de neige (snow.tga). Nous avons aussi changé les coordonnées géométriques des sommets à -200 et 200 au lieu de -25 et 25 afin d'avoir un terrain de jeu plus large et que notre bonhomme puisse bien circuler.

3. Les sapins

L'objectif est de créer plusieurs sapins répartis aléatoirement sur la scène.

Pour cela, nous avons procédé de la manière suivante dans la fonction **[void init()]**:

- Nous avons d'abord créé le sapin principal (obj[3]) et défini sa position initiale.
- Nous avons ensuite défini une matrice qu'on a nommée "sapins" de taille nb_lignes*nb_colonnes (constantes déclarées comme variables globales), contenant des 0 et des 1, selon notre choix.
- Le programme rentre par la suite dans 2 boucles "for" qui s'enchaînent dans le but de parcourir notre matrice element par element: si la case actuelle (sapins[i][j]) contient un "1", on crée obj d'indice n_objets (variable globale initialisée au début du code contenant le nombre d'objets principaux, et donc 10 dans notre cas), on incrémente la variable n_objets (pour que le prochain sapin soit d'indice n_objet + 1) et on continue à parcourir; si la case actuelle contient un "0", on passe et on continue.
- Si on est dans le cas où on crée un sapin, il faut bien-entendu le positionner quelque part pour que les sapins ne se superposent pas ; on a donc initialisé deux variables (d_x et d_z), selon lesquelles on effectue une translation du nouveau sapin créé. Après chaque création, les deux variables correspondant respectivement au décalage horizontal et vertical sont incrémentées de sorte que le prochain sapin créé soit décalé.
- Nous nous retrouvons donc à la fin avec des sapins identiques distribués aléatoirement sur le plateau.

4. Les armadillos (monstres/ obstacles)

Les armadillos représentent l'élément perturbateur de notre jeu, ce sont eux que le bonhomme doit éviter dans sa quête de recherche des objets pour gagner la partie.

Tout comme les sapins, les armadillos ont été créés à travers une matrice contenant des "0" et des "1" (procédé détaillé dans la description des sapins).

5. *Les boules de décor*

Pour décorer le ciel, nous avons importé des boules de décoration de noël, qu'on a réparties aléatoirement dans le ciel (en effectuant une première translation sur le y d'une valeur assez importante).

Encore une fois, les boules ont été initialisées et créées en utilisant la même méthode que celle des sapins et des armadillos.

6. *Le skate et le chapeau*

Le skate et le chapeau sont deux objets cachés dans la forêt, que le bonhomme doit retrouver sans se faire tuer par des monstres.

S'il les trouve et les récupère (notion de collision), il se transforme en un autre personnage.

Tout comme le bonhomme, les deux objets sont simplement initialisés par les fonctions **[void init_skate()]** et **[void_init_hat()]**.

7. *Le bonhomme sur skate et le père noël*

Si le bonhomme rencontre l'un des éléments cités ci-dessus (skate ou chapeau), il se "transforme" en un autre personnage, en l'occurrence en un autre bonhomme de neige qui se transporte sur un skate, ou en un père noël rouge.

Ces deux personnages sont chargés et initialisés au début de l'exécution du programme comme tous les autres objets dans leurs fonctions **[void init_snowman_skate()]** et **[void_init_Santa()]**, mais ont la particularité d'avoir leur attribut "visible" mis à "false (obj[n].visible = true par défaut), pour pas qu'on les voit pas au début de la partie.

III. Les déplacements

Les déplacements ont principalement été effectués en utilisant des matrices de translation et des matrices de rotation pour définir les sens des mouvements.

1. *Le bonhomme*

Vu que le déplacement du bonhomme s'effectue en appuyant sur des touches du clavier, la gestion de ses mouvements a été effectuée dans la fonction **[static void keyboard_callback(unsigned char key, int, int)]**.

a. *En appuyant sur un bouton pour tourner à droite ou à gauche :*

On effectue une rotation sur l'axe des "y" (pour que le bonhomme tourne sur lui-même) avec la ligne suivante :

obj[0].tr.rotation_euler.y += dtheta;

b. *En appuyant sur un bouton pour avancer ou reculer :*

Si les conditions de collisions (qu'on détaillera plus tard) sont satisfaites, on effectue une translation sur l'axe des x avec la ligne suivante :

obj[0].tr.translation += matrice_rotation(obj[0].tr.rotation_euler.y, 0.0f, 1.0f, 0.0f) * vec3(dL, 0.0f, 0.0f);

dL étant une quantité de déplacement définie au préalable.

Dans cette ligne de code, on ajoute au vecteur de translation courant du bonhomme une quantité qui permet de lui faire effectuer une translation selon son angle de rotation actuel (enregistré dans son vecteur de rotation_euler sur l'axe y), et donc éventuellement d'avancer uniquement là où il regarde.

2. *La caméra*

On voulait que la caméra ait toujours une vue de derrière sur notre objet et qu'elle le suit dans son mouvement, nous avons donc modifié la vue de celle-ci pour la centrer sur celle du bonhomme en ajoutant le code suivant dans la fonction keyboard_callback() :

```
cam.tr.translation = obj[0].tr.translation - matrice_rotation(obj[0].tr.rotation_euler.y,0.0f,1.0f,0.0f)*vec3(7.,0.0f,0.0f) + vec3(0.0f,1.0f,0.0f);
cam.tr.rotation_euler.y = M_PI/2. - obj[0].tr.rotation_euler.y;
cam.tr.rotation_center = cam.tr.translation;
```

3. *Les armadillos*

Le déplacement des armadillos s'effectue dans la fonction **[static void display_callback()].**

Nous avons généré 2 floats aléatoires avec la fonction rand (), dl et dz correspondant respectivement aux translations sur l'axe des x et l'axe des z.

On effectue ensuite une translation de ces deux quantités là sur leurs axes respectifs en parcourant dans une boucle "for" tous les armadillos dans le tableau d'objets.

Pour créer une illusion de mouvement encore plus aléatoire, on distingue deux cas :

- (i%2) : translation de dl et dz.
- else: translation de -dl et -dz.

4. *Les boules de décor*

Même principe que celui des armadillos.

IV. Les collisions

Trois types de collisions sont à prendre en compte dans notre jeu.

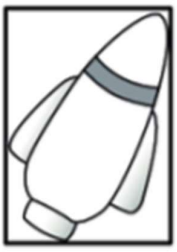
Tout d'abord nous devons vérifier que le bonhomme ne peut pas traverser les objets de décor (sapins, maison) et qu'à chaque fois qu'il rencontre un objet qui ne bouge pas, le bonhomme doit rester fixe (ne peut ni avancer ni reculer selon le sens d'où il vient), mais pas pour autant bloqué ! Il doit pouvoir avancer s'il tourne et veut se diriger ailleurs.

La deuxième collision à gérer est celle entre le bonhomme et les objets qu'il cherche (skate,chapeau), en effet comme expliqué précédemment à chaque fois que notre bonhomme rencontre l'un de ces éléments, il se transforme en un autre personnage.

La dernière collision est celle entre le bonhomme et les monstres, leur rencontre engendre la mort du bonhomme et la fin du jeu (perte de la partie).

La gestion des collisions a été faite avec la méthode AABB (Axis Aligned Bounding Box) qui consiste à faire le contour des objets en prenant leurs extrémités et par la suite entourer les objets avec un rectangle virtuel.

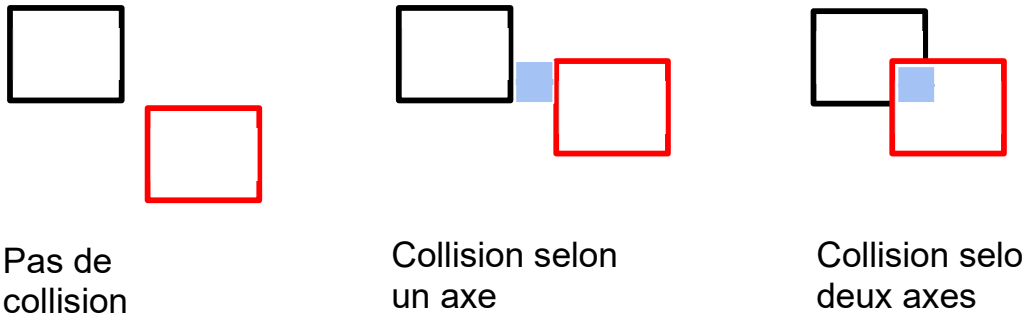
On doit donc créer des fonctions de collisions de type “boolean” (qui testent si une collision a eu lieu, et donc renvoient TRUE ou FALSE), et agir en conséquence avec des boucles “if” qui font appel à ces fonctions là en condition.



AABB

Détails de la méthode :

La collision n’est détectée que si jamais il y a un chevauchement sur les deux axes



1. *Le bonhomme et les éléments de décor*

b. *Collision bonhomme et sapin*

On a créé une fonction **[bool collision_sapin(int objet1, int objet2)]** qui prend en arguments deux entiers, correspondant aux indices des objets dont on veut tester la collision.

A l’intérieur de la fonction, on a créé deux booléens collisionX et collisionZ qu’on a initialisés à False. La fonction retourne True si une collision selon X ou Z a eu lieu.

L’appel de la fonction [**bool collision_sapin(int objet1, int objet2)]** a été faite dans la fonction **[static void keyboard_callback(unsigned char key, int, int)].** En effet, quand on appuie sur la touche “D”, on avance.

Ensuite, on rentre dans une boucle “if” qui parcourt le nombre de sapins existants et lorsque le programme détecte une collision avec un sapin (si la fonction renvoie true) , on fait reculer notre objet avec la même quantité de translation dont il a avancé, et vu que ceci s’effectue avant le redisplay, on ne voit pas notre bonhomme avancer puis reculer, mais plutôt fixé à sa position, sans être bloqué.

c. *Collision bonhomme et maison*

Par le même principe, on gère la collision bonhomme et maison dans la fonction **[bool collision_home(int objet1, int objet2)]**.

2. *Le bonhomme et les armadillos*

Pour gérer la collision entre les armadillos et le bonhomme, on a créé une autre fonction collision_armadillo qui a le même fonctionnement que celle du sapin mais nous avons juste changé les dimensionnements du rectangle.

3. Le bonhomme et les objets (skate + chapeau)

La collision entre le bonhomme et ces objets-là est un peu spéciale vu que ce qui en résulte est la "transformation" de l'objet (il est remplacé par un autre).

La vérification des collisions est effectuée exactement de la même manière que celles d'avant à travers les fonctions booléennes **[bool collision_skate(int objet1, int objet2)]** et **[bool collision_hat(int objet1, int objet2)]**.

La différence est donc au niveau de ce qui se passe après : en effet, en appuyant dans sur une touche pour avancer ou reculer, une boucle "if" se situant dans la fonction keyboard_callback teste la collision, s'il y en a une elle remplace l'objet courant (obj[0]) par l'objet souhaité (obj[9] pour le père noël et obj[5] pour le bonhomme sur skate) en échangeant leur VAO et leur NOMBRE DE TRIANGLES DU MAILLAGE avec les lignes suivantes:

obj[0].vao = obj[5].vao;

obj[0].nb_triangle = obj[5].nb_triangle;

V. Echec de la partie

Pour perdre le jeu, il suffisait d'entrer en collision avec un armadillo. En effet, on a créé une variable globale booléenne qu(on a nommée LOSE et on l'a initialisée à false et quand on appelle la fonction collision_armadillo dans la fonction display_callabck() on met cette variable à True. Enfin, on fait un test : si Lose est égale à true on affiche le text game over.

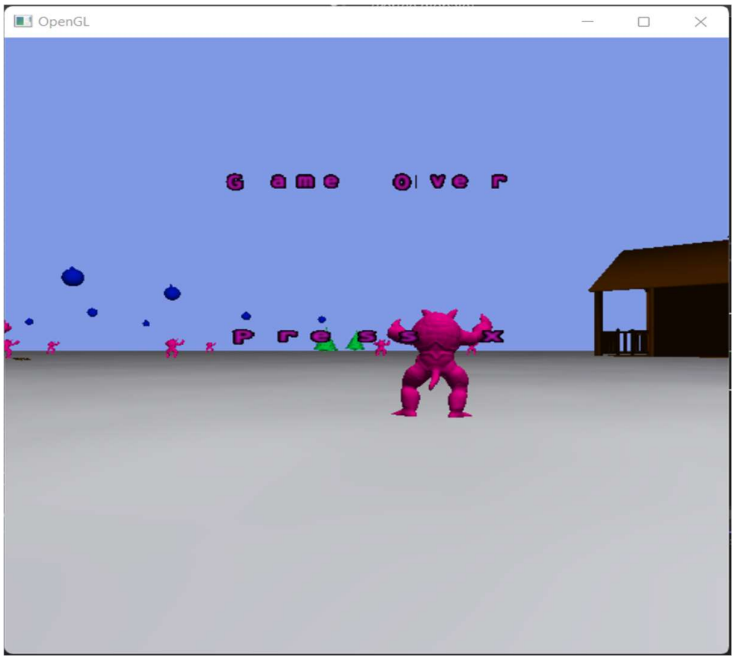


Figure 1 : Affichage en cas de perte

VI. Réussite de la partie

Pour gagner le jeu il faut réussir à collecter les deux objets qui existent, pour cela on a créé trois variables booléens col_skate, col_hat et WIN. Ces trois variables ont toutes été intialisées à false. Dans l'appel de chacune des fonctions **[bool collision_skate(int objet1, int objet2)]** et **[bool collision_hat(int objet1, int objet2)]**.on met col_skate à true et col_hat à true et à la fin on fait un test, si col_skate et col_hat sont à true, on met WIN à true et on affiche le texte Bravo.

Figure 2 : Affichage en cas de gain

VII. Difficultés

- Faire marcher le projet sur différents environnements de développement
- Gestion des collisions (plusieurs méthodes)
    - La diversité des méthodes AABB - OBB - Convex Hull rendait difficile le choix de la méthode adéquate.
- Gestion et visualisation des positions (sur x y et z)
    - Au début, on avait du mal à comprendre la direction des axes sur openGL, les déplacements ainsi que les rotations sur x, y et z
- Import des objets avec des couleurs sur openGL :
    - En effet, on a réussi a coloré les objets sur blender et de mettre un peu de texture mais on a trouvé qu'il fallait les exporter en .fbx et on a pas su comment gérer cette extension sur openGL.
- Fluidité des actions (ex: pas de possibilité d'animation quand changement du personnage) :
    - En effet, lors de la rencontre du bonhomme avec un skate ou un chapeau, celui-ci se transforme en un autre personnage, on voulait que cette transformation soit fluide et non pas brusque mais c'était très compliqué à notre niveau.

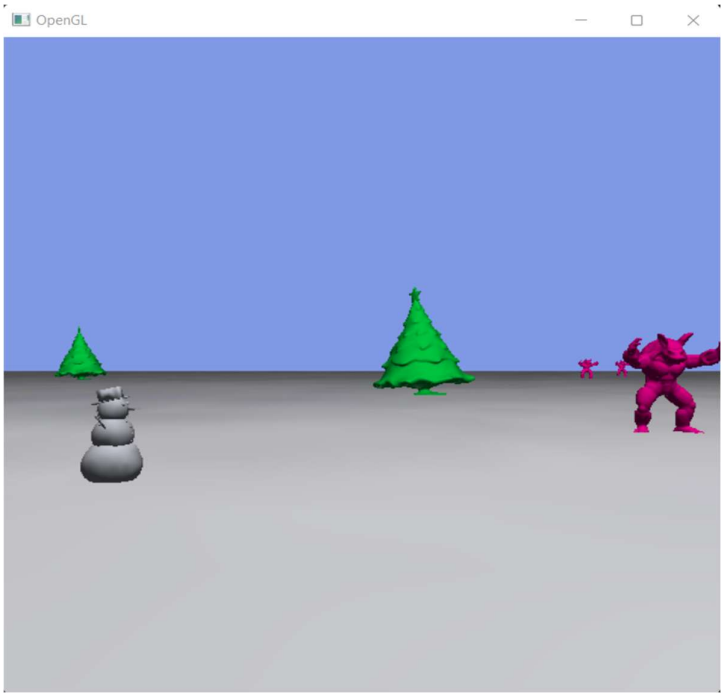


Figure 3 : Affichage du début

Figure 4 : Affichage quand on appuie sur une touche

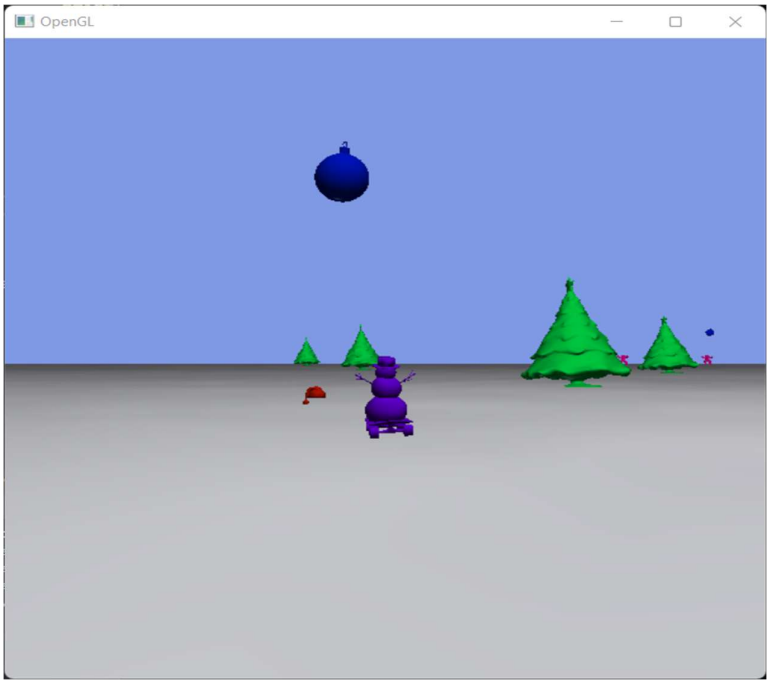


Figure 5 : Affichage avant la collision avec le chapeau et après la collision avec le skate

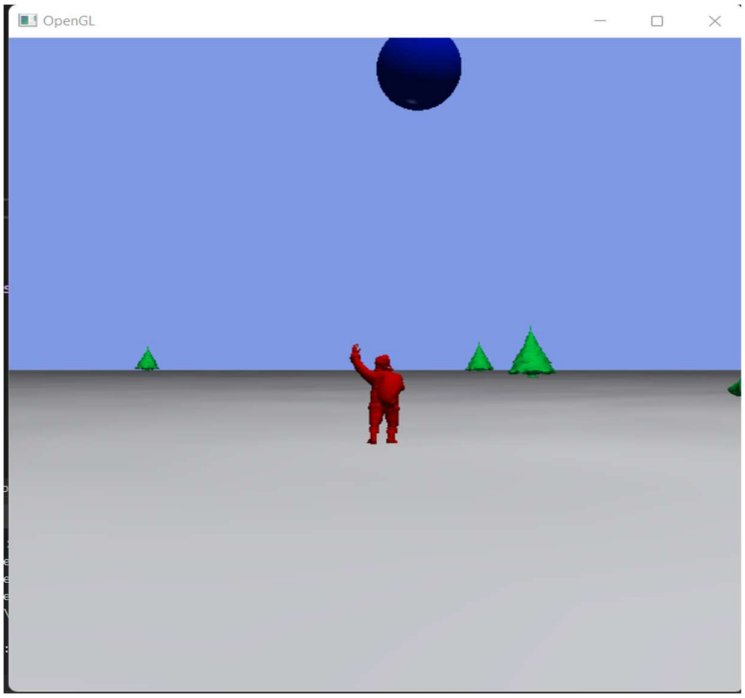


Figure 6 : Affichage après la collision avec le chapeau

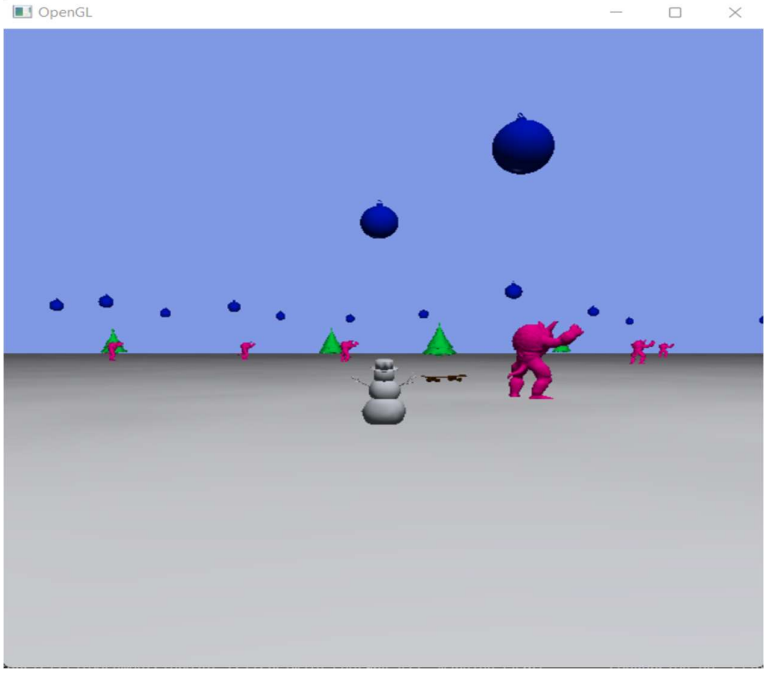


Figure 7 : Affichage avant la collision avec le skate

Figure 8 : Affichage après la collision avec le skate