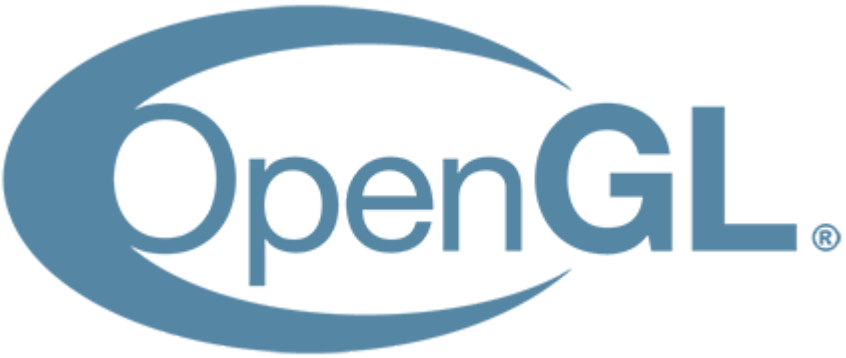
ANCHLING Luc
TEYSSIER Timothée

Rapport Jeu TSI



Sommaire

LIVE AND DISCOVER

Introduction

Le cours de **Traitement et Synthèse d'Image** a donné suite à un projet dans lequel il nous a été demandé de réaliser un **jeu 3D minimaliste**. En utilisant la bibliothèque logiciel **OpenGL**, nous sommes partis sur une idée de jeu 3D que nous allons développer dans la partie **Projet** ci-dessous. Le temps qui nous a été imparti était de 8h en TP. L'ensemble du projet a été réalisé à partir d'un dossier **Github** partagé, ce qui nous a permis tout autant d'avancer séparément qu'ensemble.

Le choix du sujet était libre mais nous devons respecter quelques contraintes :

- Jeu / Monde en 3D
- 2 programmes graphiques minimum
- Gestion de collision et déplacement

Le projet

L'idée du jeu

Nous avions pour idée première de faire un jeu avec un 2 en 1, c'est-à-dire que nous voulions faire une phase 2D (style space invaders) suivie d'une phase 3D… Mais la subtilité résidait dans le fait que le jeu 2D était en fait un jeu 3D pris avec le point de vue adéquat pour voir de la 2D, pour ne pas à avoir à faire 2 jeux séparément mais une première phase avec un angle de caméra adéquat et une seconde avec une simple transition de caméra. Ce projet a vite été lâché car trop ambitieux.

Nous sommes donc partis sur l'exploitation unique de la phase 3D de notre projet initial. Cette phase aurait dû s'apparenter à un labyrinthe duquel il aurait fallu sortir tout en échappant aux différents mobs présents sur la map. Ce jeu serait un jeu à la première personne.

Le jeu est composé d'un labyrinthe, de différents monstres (2 types) et d'un arbre. Tous sont générés aléatoirement. L'arbre en question représente la ligne d'arrivée du jeu. Le joueur possède 3 vies, lesquelles seront susceptibles de descendre lorsqu'un monstre sera touché.

Explications sur le gameplay

Dès lors que l'on lance le programme :

- Le déplacement se fait via les touches **zqsd**
- La rotation est assuré par la **souris** (comme un fps)
- Un appui sur **espace** permet réaliser un saut
- Un appui sur la touche **Q** permet de quitter
- La touche **c** permet de courir
- L'appui sur la touche **F3** permet de basculer entre l'affichage ou non des coordonnées

Des explications plus techniques sur chacune des parties sont à suivre dans les parties éponymes qui suivent.

Gestion des déplacements

La vue First Person

La vue à la première personne recherchée est celle présente sur l'ensemble des jeux du style FPS sur PC, à savoir le contrôle des déplacements du personnage à l'aide du clavier et le déplacement de la vue du personnage en utilisant la souris.

Déplacement du personnage

L'idée étant la réalisation d'un jeu à la première personne, les déplacements du personnage sont donc gérés par des touches du claviers (lesquelles sont **z,q,s & d**).

La prise en compte de l'appui sur ces touches est gérée par des fonctions fournies par la bibliothèque **Glut**. Nous avons utilisé **glutKeyboardFunc** pour l'appui sur une touche classique du clavier et **glutKeyboardUpFunc** pour le relachement de la touche du clavier.

Pour fluidifier le déplacement du personnage, il a fallu utiliser des booléens dont on va changer leur état suivant l'appui ou non sur la touche concernée.

Exemple pour l'appui :

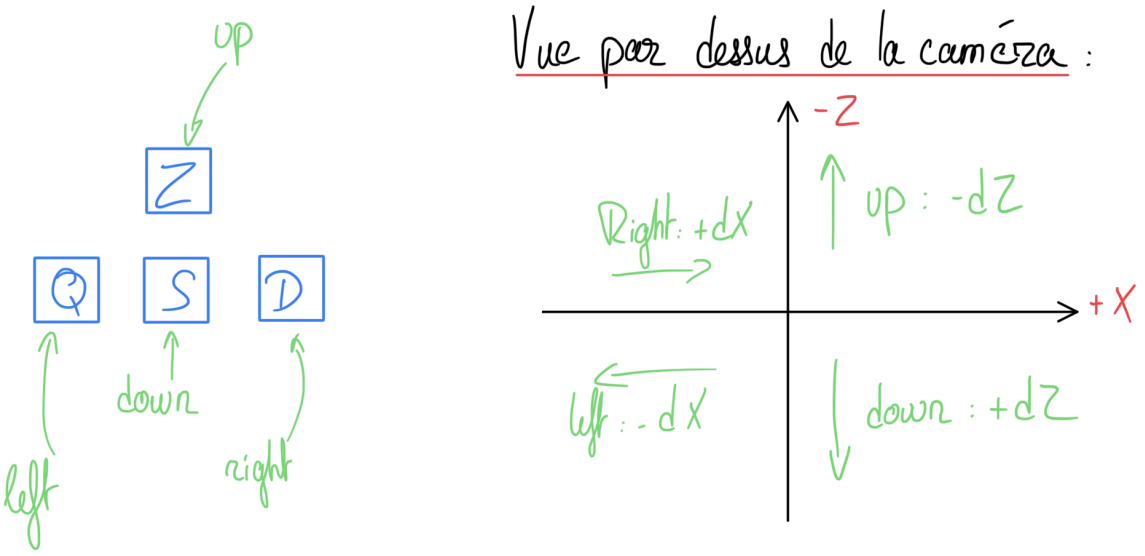
```
case 'z':
  up=true;  //rotation avec la touche du haut
  break;
```

Exemple pour le relâchement :

```
case 'z':
  up = false;
  break;
```

Ces booléens vont alors être utilisés en tant que conditions par la fonction permettant le déplacement.

Pour pouvoir avancer, on applique une transformation adéquate à la matrice de translation de la caméra, ces transformations sont explicitées dans le schéma ci-après :



Si l'on se réduit seulement aux translations décrites précédemment, l'on ne peut pas réaliser l'ensemble de la vue FPS recherchée, en effet, il faut prendre compte de l'orientation de la caméra pour que la caméra se déplace correctement.

Exemple pour l'application d'une translation dans la fonction déplacer :

```
if (up) {
  if(dodge)cam.tr.translation += rotation * vec3(0, 0, 3*dL);
  else cam.tr.translation += rotation * vec3(0, 0, -dL);
}
```

* la partie avec le dodge sera décrite dans la partie **Esquive**.

Pour prendre en compte l'orientation de la caméra, on va récupérer la matrice de rotation de la caméra (modifiée via l'utilisation de la souris – voir partie suivante) que l'on va appliquer à la matrice de translation de la caméra (voir l'exemple ci-dessus).

On ne prend en compte seulement la composante en y puisqu'elle est la seule à être utile du fait que l'on se déplace uniquement suivant les axes **X** & **Z**.

Exemple pour la prise en compte de l'orientation :

```
// Récupération de la matrice de rotation de la caméra
 mat4 rotation_x = matrice_rotation(cam.tr.rotation_euler.x, 1.0f,
0.0f, 0.0f);
 mat4 rotation_y = matrice_rotation(-cam.tr.rotation_euler.y,
0.0f, 1.0f, 0.0f);
 mat4 rotation_z = matrice_rotation(cam.tr.rotation_euler.z, 0.0f,
0.0f, 1.0f);
 mat4 rotation =  rotation_y;
```

Déplacement de la caméra

Concernant le déplacement de la caméra via l'utilisation de la souris, tout comme la gestion des touches du clavier, nous avons utilisé une fonction de la bibliothèque **Glut**, laquelle est **glutPassiveMotionFunc**. Cette fonction nous permet de récupérer en temps réel les coordonnées de la souris.

Pour déplacer la caméra de manière cohérente avec le déplacement de la souris, on a calculé un delta entre la position actuelle de la souris et la suivante et appliquer ce delta aux angles d'euler adéquats.

Exemple pour l'angle d'euleur suivant y :

```
cam.tr.rotation_euler.y -= 0.001f * d_angle * 2*M_PI*float(HEIGHT
/ 2 - tempX);
```

On va faire en sorte de garder le curseur au centre de la fenêtre pour que nos déplacements de caméra soient toujours cohérents avec ceux de la souris. Pour se faire, l'on a utilisé la ligne suivante :

```
glutWarpPointer(WIDTH / 2, HEIGHT / 2);
```

Pour que la vue soit toujours cohérente suivant la translation effectuée, il a fallu actualiser le centre de rotation de la caméra suivant la translation que l'on a fait subir à la caméra, cela s'est fait via la ligne suivante :

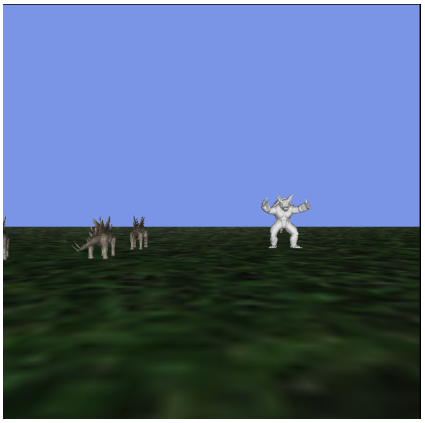
```
cam.tr.rotation_center = cam.tr.translation;
```

Le Saut

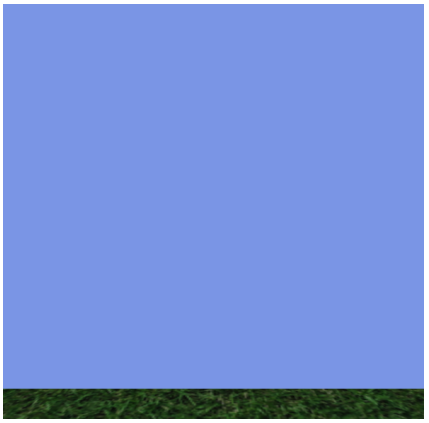
Le saut est géré par la touche **'_'(espace)**. Dans le jeu comme nous sommes en vue FPS, nous devions juste modifier la hauteur, c'est-à-dire les coordonnées en **y.**

Avant le saut nous sommes à une ordonnée de 1.0f choisie arbitrairement car par manque de temps nous n'avons pas pu réfléchir aux collisions entre la caméra et le sol.

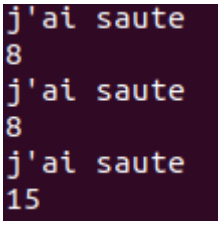
Avant saut :



Après saut :



Nous avons codé une fonction **sauter()** qui permet cette action, cependant si vous restez appuyé sur la touche **espace** en continu, cela vous fait planer à la hauteur maximale qui a été codé soit **8.0f** dans notre code. Cela nous permettra de nous sortir de certaines situations que l'on verra dans le chapitre sur les collisions. De plus, si vous n'appuyez pas assez longtemps sur la touche espace, le saut n'ira pas forcément jusqu'à la hauteur maximale, un exemple ci-dessous : la hauteur maximale est de **15.0f** pour cet exemple. Cependant, on n'atteint que **8.0f** avec un appui bref sur la touche.



L'esquive (pour courir)

L'esquive est programmée sur la touche c. En effet lorsque cette touche est enfoncée, le déplacement en translation en x et en z sera alors plus important qu'un déplacement sans esquive.

```
if (down) {
  if(dodge)cam.tr.translation += rotation * vec3(0, 0, 3*dL);
  else cam.tr.translation += rotation * vec3(0, 0, dL);
}
```

Comme nous pouvons l'observer sur la capture d'écran ci-dessous, on observe que si le booléen dodge vaut true, cela veut dire que l'on veut esquiver donc la translation ici en **z** est plus importante.

On avait pensé à implémenter ce booléen **dodge** pour esquiver des mobs qui peuvent nous foncer dessus et que nous ne pouvons pas sauter par dessus. Cependant, par manque de temps nous n'avons pas eu le temps de coder la fonction permettant de bouger les mobs, donc cela nous permet simplement de nous déplacer plus vite.

Gestion des modèles

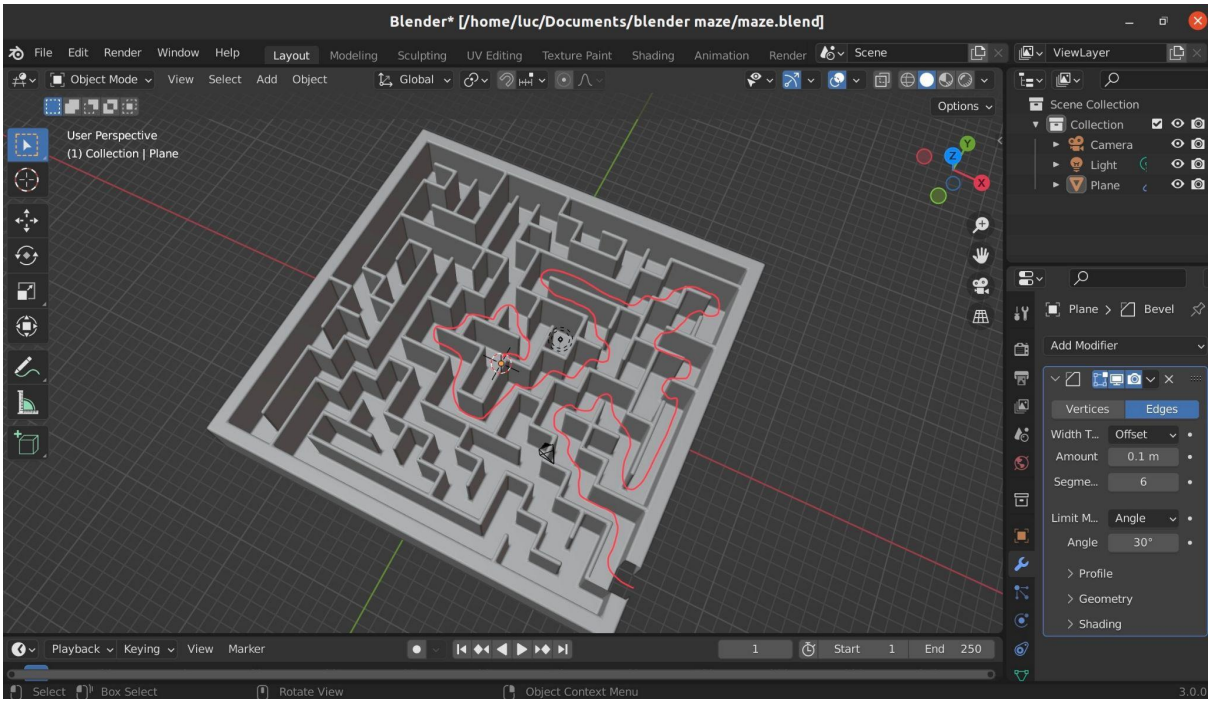
Les Monstres

Les deux modèles de monstres sont ceux qui étaient déjà présents dans l'archive que l'on avait à disposition. Les modifications que l'on a apportées se trouvent dans le positionnement des **stégosaures**, en effet, on a utilisé la méthode de la copie de l'objet pour placer d'autres **stégosaures** avec une position et une orientation aléatoires via l'utilisation de la fonction **init_model_4**.

Un exemple du placement des monstres est le suivant :

Le Labyrinthe

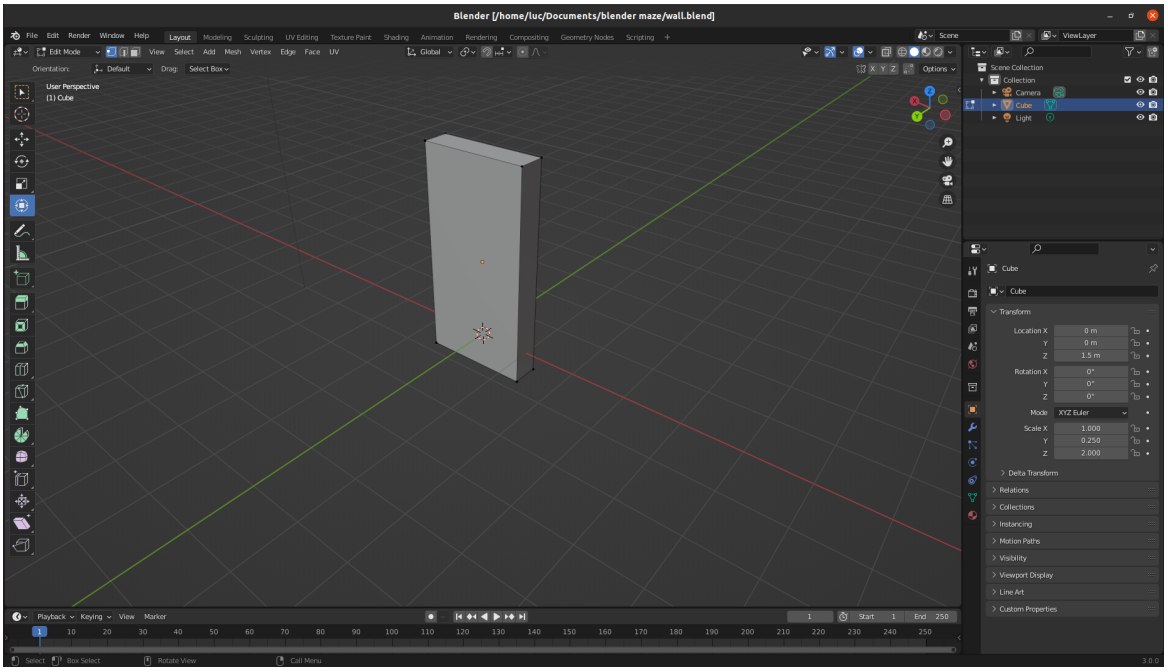
Le modèle de base du **labyrinthe** a été réalisé à partir d'un **add-on blender** (*mesh-maze*) récupéré sur **github**, il est montré ci-dessous



Puis, ce modèle a été exporté au format **.obj** pour pouvoir l'utiliser dans notre jeu via la fonction d'initialisation **init_model_5**. Une fois importé, le jeu ressemble à cela :

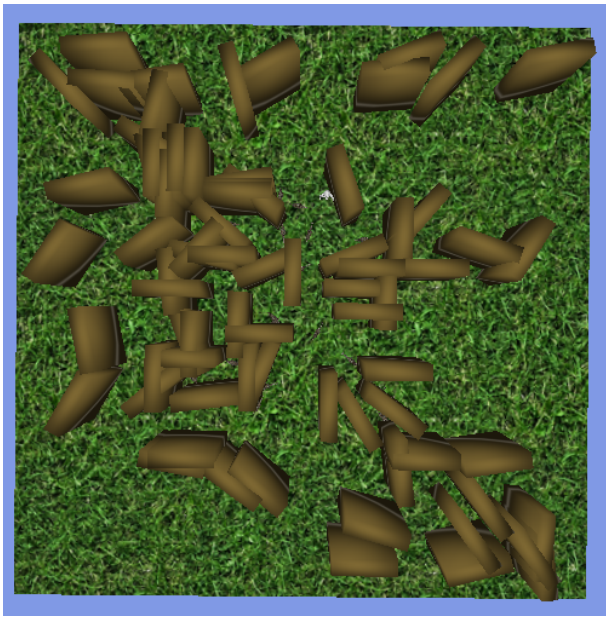
Le problème avec ce modèle de labyrinthe est qu'il possède des formes convexes qui ne sont pas gérables avec **OpenGL**.

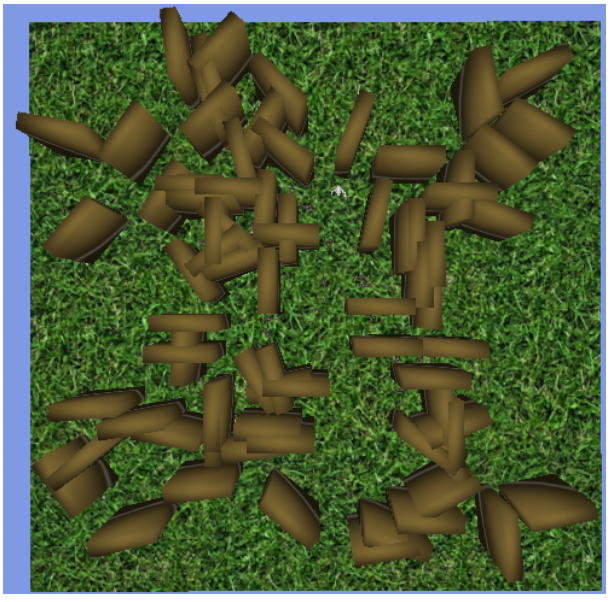
L’on a dû se rabattre sur l’utilisation d’un **modèle simple** avec lequel le gabarit utilisé pour les collisions pourrait être de forme cubique. Ce modèle est le suivant :



Pour le positionnement de ces murs, via l’utilisation de **init_model_6**, l’on a décidé de les placer de **manière aléatoire** (position et orientation) tout en faisant attention à ce que le personnage ne spawn dedans (en gérant le rayon de positionnement).

<u>Différents résultats de labyrinthe :</u>



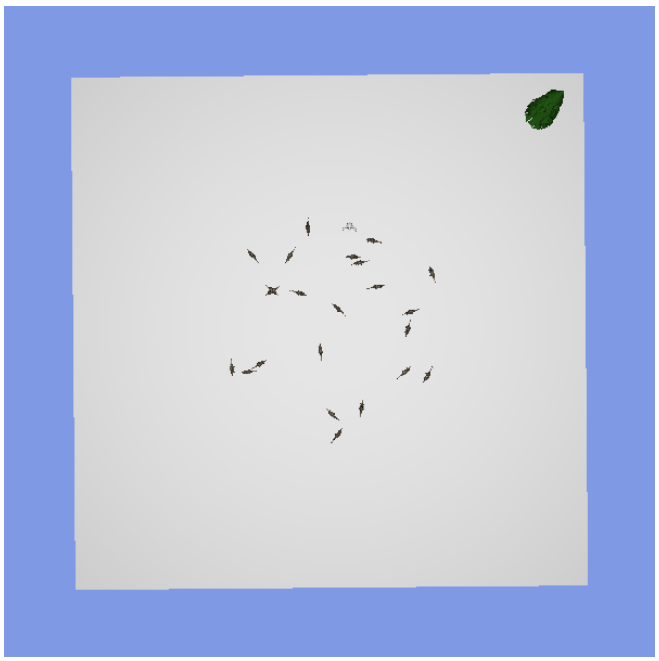


L’arbre

Pour former une **ligne d’arrivée** à notre jeu, on a décidé de placer un arbre de manière aléatoire sur le terrain, il a été formé à partir d’un modèle que l’on a importé avec la fonction **init_model_7**. Sa position sera toujours dans l’un des quatre coins de la carte.

Le modèle de l’arbre en question vient avec sa texture, comme on le voit ci-après :

Exemples de positionnement de l’arbre :



Gestion des collisions

La gestion des collisions est un élément très important dans notre projet, en effet nous avons trois différents types de collisions :

- la collision entre le personnage et un ennemi
- la collision entre le personnage et les murs de notre labyrinthe
- la collision entre le personnage et le sapin de fin

En premier temps, pour gérer la collision entre le personnage et un ennemi, nous devons créer une sphère autour de notre personnage et une sphère autour de notre ennemi comme ci-dessous :

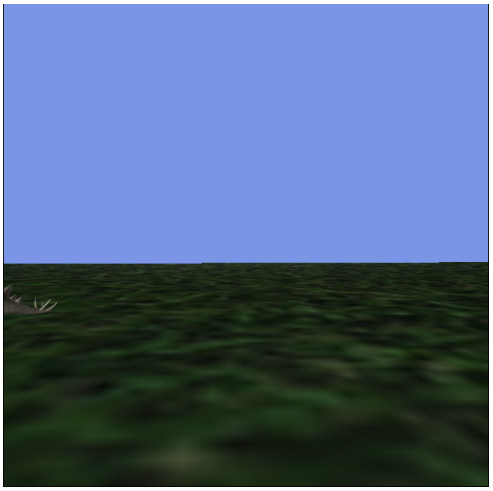
```
295   struct Sphere        You,
296   {
297     float x,y,z;
298     float rayon;
299   };
```

```
Sphere camera;
camera.x= cam.tr.translation.x;
camera.y= cam.tr.translation.y;
camera.z= cam.tr.translation.z;
camera.rayon= 0.5f;
```

Avec ses “hitbox”, on utilise la fonction Collision(Sphere 1,Sphere 2) pour calculer notre si une des sphères est comprise dans l’autre ce qui entraînera ainsi une collision et donc une vie en moins, l’ennemi ne sera alors plus visible et il sera déplacé en dehors de la map pour ne compter la collision qu’une seule fois.

avant collision:

après collision



En deuxième type de collision, il y a la collision entre le personnage et les murs du labyrinthe, nous allons cette fois-ci utiliser des hitbox de type cube pour mieux gérer les collisions avec les blocs de murs.

```
struct Cube
{
  float x,y,z;
  float w,h,d;
};
```

```
Cube cameraAABB;      You, 3 days a
cameraAABB.x=cam.tr.translation.x;
cameraAABB.y=cam.tr.translation.y;
cameraAABB.z=cam.tr.translation.z;
cameraAABB.w=10*dL;
cameraAABB.h=50*dL;
cameraAABB.d=10*dL;
```

On utilise ensuite la fonction **CollisionCube()** pour gérer la collision grâce au booléen isCollision. Cependant à certains moments, la hitbox des blocs de murs ne sont pas parfaitement configurés, donc si nous passons à côté de l'extrémité du mur, la collision peut ne pas être détectée. Le mur nous empêche normalement de le traverser grâce au booléen isCollision.

Si la collision avec le mur est détectée le isCollision passe à true ce qui empêche normalement de traverser le mur cependant on peut se retrouver bloquer à l'intérieur du mur, si cela arrive nous pouvons utiliser le saut en continu plus les déplacements pour se sortir de la situation.

En dernière collision nous avons la collision entre le personnage et l'arbre de fin du jeu. On retrouve la même structure sphérique utilisée pour la première gestion de collision vue précédemment. Lorsque le personnage entre en contact avec le sapin, le jeu s'arrête.

Fin du jeu

Gestion des Vies

Le joueur possède **3 vies** lorsqu'il lance le jeu. Ces vies sont supprimées une à une si le joueur vient à toucher un des monstres présents sur le terrain. Dès lors que le nombre de vies arrive à zéro, le jeu s'interrompt.

Le nombre de vie passe bien de 3 à 2 lorsque l'on touche un monstre.

La Ligne d'arrivée

Une autre façon de finir le jeu est de franchir la **ligne d'arrivée** représentée par un arbre placé **aléatoirement** dans **l'un des quatre coins du terrain**. Lorsque ce dernier est atteint, tout comme lorsque le nombre de vie atteint zéro, le jeu s'interrompt.

Conclusion

Nous avons bien répondu aux quelques attentes du cahier des charges. En effet, nous avons bien développé un jeu 3D en faisant attention à la translation et la rotation des objets. Nous avons travaillé sur le déplacement de la caméra, puisque la vue obtenue est une vue à la première personne.

N'ayant pas eu assez de temps pour finir le projet comme on l'aurait souhaité, il reste pas mal de pistes et de perspectives d'amélioration que le jeu aurait pu avoir. Lesquelles peuvent être les suivantes :

- Gestion collision avec le mur à optimiser
- Ajout d'un score
- Améliorer les textures
- Améliorer la fonction de saut (pour ajouter du réalisme)
- Ajouter un menu (pour pouvoir recommencer en fin de partie…)
- Optimiser le code

Bibliographie

Exemple de structure en C++ :
https://docs.microsoft.com/fr-fr/cpp/cpp/struct-cpp?view=msvc-170

Aide pour la gestion des collisions :
https://pub.phyks.me/sdz/sdz/eorie-des-collisions.html#AABB3D

Remerciements

Nous voudrions remercier Mr Dupont et l'équipe d'encadrants qui nous ont aidé tout au long de notre projet.