CRUVEILLER Pierre - FILIOT Jules

GRASS FIELD INVADERS - ARMADILLOS' SQUAD FIGHTS BACK

A space invader remake

4-ETI Groupe B

2020/2021

1. INTRODUCTION
2. STRUCTURE
    a. Le joueur
    b. Les ennemis
    c. Les balles
    d. Le sol
    e. Le texte
3. INTERACTIONS ENTRE MODÈLES
    a. Le joueur
    b. Les ennemis
    c. Les balles
    d. Le texte
4. LE JEU
    a. Régénération des ennemis
    b. Fin de partie
5. CONCLUSION

1. INTRODUCTION :

L'objectif de ce projet est de créer un jeu simple qui utilise les principes et techniques vues en TP de découverte de l'API OpenGL.

Suite au TP nous avions donc une base de programme incluant les différentes fonctions permettant d'afficher différents éléments graphiques en 3D incluant un sol, un monstre (Armadillo) ainsi qu'un Stégosaure. Le code mis à notre disposition comprenait également la gestion de la caméra première personne à partir du clavier ainsi que la gestion du déplacement "touche par touche" d'un modèle 3D.

Ayant déjà codé un remake de space invader en 2D en python nous avons décidé de coder un remake de space invader sur OpenGL. L'avantage de cette démarche est que nous avions déjà en tête les fonctionnalités à coder et la structure du projet en Python donc seulement l'adaptation en OpenGL à réaliser avec les éventuelles ajouts ou suppressions de fonctionnalités adaptées à OpenGL.

Notre démarche a donc été dans un premier temps de poser sur papier toutes les fonctionnalités à implémenter et donc les fonctions à réaliser qui en découlent. Nous nous sommes ensuite réparti le travail et nous avons enfin commencé à coder à proprement parler.

Avant de commencer l'explication de notre code nous souhaitions faire une rapide explication du principe d'un Space Invaders. Le but du jeu est donc de tuer une horde d'ennemis qui se situe face au joueur et qui avance de plus en plus vers le joueur tout en tirant. Le joueur contrôle à l'origine un vaisseau spatial qui tire des missiles, si le joueur touche un ennemi il meurt, sinon si c'est lui qui est touché il perd la partie. Si les ennemis avancent suffisamment pour toucher le joueur, ce dernier perd également la partie.
Notre version de Space Invaders ne s'arrêtera jamais tant que le joueur ne meurt pas, le but du jeu est donc d'avoir le score le plus grand possible.

2. STRUCTURE :
    a. Le joueur

Pour notre jeu, il fallait que le joueur incarne un objet qui puisse se déplacer sur les axes x et y pour éviter les tirs ennemis et leur tirer dessus. Notre choix pour l'objet modélisant le joueur s'est porté sur le Stégosaure fourni avec le projet, nous avons simplement modifié la couleur de sa texture pour qu'il soit bleu :

Le projet implémentait déjà des fonctions permettant de déplacer le Stégosaure sur les axes x et z, donc il nous a suffit de remplacer l'axe z de déplacement par l'axe y. Cependant, le déplacement initialement codé était lent et saccadé, ce qui n'était pas satisfaisant pour avoir un gameplay agréable.

À l'origine, le déplacement vers le haut était codé comme suit :

```
static void special_callback(int key, int,int)
{
  float dL=0.03f;
  switch (key)
  {
    case GLUT_KEY_UP:
      model_dinosaure.transformation_model.translation.y += dL;
      break;
```

Le problème de cette méthode est que son principe est de déplacer d'un cran le joueur lorsqu'on clique sur la flèche du haut, et la gestion du clic enfoncé relevait de de la gestion qui en était faite par l'ordinateur. Ainsi lorsque l'on cliquait et que l'on restait appuyé, le joueur se déplaçait d'abord d'un cran, puis après une petite pause il se déplaçait continuellement mais de manière saccadée.

Pour régler ce problème nous avons déployé la fonction specialUp_callback qui réagit au relâchement d'une touche spéciale initialement enfoncée, et nous avons construit des variables globales pour chaque déplacement qui sont mises à 1 au clic sur une flèche et remises à 0 lorsqu'on relâche la touche :

```
static void special_callback(int key, int,int)
{
  float dL=0.03f;
  switch (key)
  {
    case GLUT_KEY_UP:
      deplacementHaut=1;
      break;
    case GLUT_KEY_DOWN:
      deplacementBas=1;
      break;
    case GLUT_KEY_LEFT:
      deplacementGauche=1;
      break;
    case GLUT_KEY_RIGHT:
      deplacementDroite=1;
      break;
  }
}
```

```
static void specialUp_callback(int key, int,int)
{
  float dL=0.03f;
  switch (key)
  {
    case GLUT_KEY_UP:
      deplacementHaut=0;
      break;
    case GLUT_KEY_DOWN:
      deplacementBas=0;
      break;
    case GLUT_KEY_LEFT:
      deplacementGauche=0;
      break;
    case GLUT_KEY_RIGHT:
      deplacementDroite=0;
      break;
  }
}
```

En parallèle, nous avons codé une fonction gestionDeplacement qui s'occupait de gérer les différents déplacements du joueur en fonction des valeurs de chaque constante de déplacement. De plus, nous avons construit une limite pour empêcher le joueur de descendre sous le sol.

```
void gestionDeplacement(){

  if(deplacementHaut==1 ){
    model_dinosaure.transformation_model.translation.y += 0.03;
  }
  if(deplacementBas==1&& model_dinosaure.transformation_model.translation.y>0.0){
    model_dinosaure.transformation_model.translation.y -= 0.03;
  }

  if(deplacementGauche==1){
    model_dinosaure.transformation_model.translation.x -= 0.03;
  }
  if(deplacementDroite==1){
    model_dinosaure.transformation_model.translation.x += 0.03;
  }
}
```

En mettant tout ceci en place nous avons réussi à avoir un Stégosaure contrôlé par le joueur avec des déplacements très satisfaisants.

b. <u>Les ennemis</u>

Pour notre jeu, il fallait construire une horde d'ennemis qui se déplacent de gauche à droite et se rapprochent petit à petit du joueur tout en tirant des balles. Pour cela nous avons choisi le modèle d'Armadillo fourni dans le projet et nous avons créé 8 ennemis. Nous avons initialisé tous les ennemis au même point car cela nous permet de faciliter la comparaison de coordonnées des différents acteurs du jeu.

Pour ordonner la horde nous avons donc appliqué différentes transitions spécifiques à chaque ennemis :

```
void move_enemy_squad() {
  if (init_parameter == 1) {  //si c'est le premier appel de cette fonction on positionne les armadillos à leur coordonnées d'origine

    //translation sur z pour chaque armadillo : on leur fixe une position d'origine pour faire deux rangées de quatres
    model_armadillo.transformation_model.translation.z -= 10.0f;
    model_armadillo2.transformation_model.translation.z -= 10.0f;
    model_armadillo3.transformation_model.translation.z -= 10.0f;
    model_armadillo4.transformation_model.translation.z -= 10.0f;
    model_armadillo5.transformation_model.translation.z -= 10.0f;
    model_armadillo6.transformation_model.translation.z -= 10.0f;
    model_armadillo7.transformation_model.translation.z -= 10.0f;
    model_armadillo8.transformation_model.translation.z -= 10.0f;
    //positionnement de chaque armadillo pour qu'ils soient tous alignés et qu'ils forment deux rangée l'une au dessus de l'autre
    model_armadillo2.transformation_model.translation.x += 1.5f;
    model_armadillo3.transformation_model.translation.x += 3.0f;
    model_armadillo4.transformation_model.translation.x += -1.5f;
    model_armadillo6.transformation_model.translation.x += 1.5f;
    model_armadillo7.transformation_model.translation.x += 3.0f;
    model_armadillo8.transformation_model.translation.x += -1.5f;
    model_armadillo5.transformation_model.translation.y += 2.0f;
    model_armadillo6.transformation_model.translation.y += 2.0f;
    model_armadillo7.transformation_model.translation.y += 2.0f;
    model_armadillo8.transformation_model.translation.y += 2.0f;
    //application de la même rotation à chaque armadillo pour qu'il soit de face au joueur
    model_armadillo.transformation_model.rotation = matrice_rotation(move_rotation, 0.0f, 1.0f, 0.0f);
    model_armadillo2.transformation_model.rotation = matrice_rotation(move_rotation, 0.0f,1.0f,0.0f);
    model_armadillo3.transformation_model.rotation = matrice_rotation(move_rotation, 0.0f, 1.0f, 0.0f);
    model_armadillo4.transformation_model.rotation = matrice_rotation(move_rotation, 0.0f, 1.0f, 0.0f);
    model_armadillo5.transformation_model.rotation = matrice_rotation(move_rotation, 0.0f, 1.0f, 0.0f);
    model_armadillo6.transformation_model.rotation = matrice_rotation(move_rotation, 0.0f,1.0f,0.0f);
    model_armadillo7.transformation_model.rotation = matrice_rotation(move_rotation, 0.0f, 1.0f, 0.0f);
    model_armadillo8.transformation_model.rotation = matrice_rotation(move_rotation, 0.0f, 1.0f, 0.0f);
    //Décalage vers la gauche
    model_armadillo.transformation_model.translation.x += -2.5f;
    model_armadillo2.transformation_model.translation.x += -2.5f;
    model_armadillo3.transformation_model.translation.x += -2.5f;
    model_armadillo4.transformation_model.translation.x += -2.5f;
    model_armadillo5.transformation_model.translation.x += -2.5f;
    model_armadillo6.transformation_model.translation.x += -2.5f;
    model_armadillo7.transformation_model.translation.x += -2.5f;
    model_armadillo8.transformation_model.translation.x += -2.5f;
```

Ce qui nous donne :

Une fois la horde créée et ordonnée, nous avons codé une fonction de déplacement qui la fait se déplacer de gauche à droite et se rapprocher du joueur. Nous avons utilisé une variable move_side qui définit le pas de déplacement latéral et un compteur qui permet de savoir à quelle moment rapprocher la horde et remplacer move_side par son opposé pour changer le sens de déplacement latéral.

c. Les balles

Notre jeu met en œuvre des tirs du joueurs et des ennemis, il a donc fallu modéliser ces dernières. Pour cela, nous avons simplement utilisé le modèle du Stégosaure que nous avons créé de plus petite taille et nous avons modifié sa texture pour qu'il apparaisse rose :

Le joueur dispose de 3 balles qu'il peut tirer, mais il doit attendre qu'une balle ait fini sa course pour tirer la même balle à nouveau.
4 ennemis disposent d'une balle qu'ils tirent à chaque fois qu'ils le peuvent, c'est-à-dire à chaque fois qu'elles terminent leur course.

Nous avons donc créé 7 balles différentes que nous avons initialisé au même point de coordonnées, encore une fois pour faciliter les calculs à venir sur les comparaisons de coordonnées entre les différents acteurs.

Après avoir été initialisée chaque balle est déplacée au niveau de son tireur grâce à la fonction gestionTirs qui sera développée dans la partie sur les interactions entre modèles.

d. Le sol

Nous voulions pour le jeu un effet de sol qui se déplace à l'infini pour donner au joueur une impression de mouvement continuel.
Pour cela nous avons créé un deuxième sol en se basant sur celui fourni avec le projet, puis nous l'avons translaté et collé au bord du premier sol. Ensuite, la manipulation consiste à déplacer les deux sols en même temps sur l'axe z en direction de la caméra, et lorsqu'un sol sort du cadre on le recolle derrière celui qui est encore dans le cadre, ainsi les deux sols se déplacent continuellement et s'alternent.

e. Le texte

Pour notre jeu, il était nécessaire d'avoir un affichage de texte dynamique qui montre au joueur son nombre de balles restantes et son score actuel. Pour cela, nous avons créé un deuxième programme graphique dédié uniquement à l'affichage du texte.

```
gui_program_id = glhelper::create_program_from_file(
    "shaders/shadertext.vert",
    "shaders/shadertext.frag"); CHECK_GL_ERROR();
glUseProgram(gui_program_id);
```

Ce programme fonctionne grâce à une texture spécifique qui contient chaque chiffre et chaque lettre de l'alphabet. Grâce à la fonction draw_texts fournie, on peut écrire le texte voulu à l'écran, et il nous suffit de créer une fonction qui change le contenu de la variable qui contient le texte à afficher pour avoir les textes que l'on veut de manière situationnelle. Nous développerons plus sur cette fonction dans la partie sur les interactions entre modèles.

Par exemple, lorsqu'on lance le programme :

Puis, lorsque l'on joue :

3. INTERACTIONS ENTRE MODÈLES
    a. Le joueur

La seule interaction que nous avons codé pour le joueur est le tir de balles. Cette interaction se fait donc entre le modèle Stégosaure et les trois modèles représentants les balles du joueur.

Lorsque le joueur appuie sur la touche espace il a la possibilité de tirer, son stock de munitions s'élève à trois, il ne peut donc pas tirer plus de trois balles à la fois d'où les trois modèles de balle. La touche espace va donc appliquer le code suivant :

```
392
393     case ' ':
394       if(tir1==0){
395         tir1=1;
396         }
397       else if (tir2==0){
398         tir2=1;
399       }
400       else if (tir3==0){
401         tir3=1;
402       }
403       break;
404
```

Ici nous avons trois variables globales, tir1, tir2, tir3, qui représentent la "disponibilité" de chaque balle, donc de chaque modèle qui représente une balle. Cette variable va donc traduire la réponse à la question suivante : "Puis-je tirer cette balle ?". Ensuite le principe est de coder un ordre de priorité de tir des balles, si la balle 1 a été tirée on l'a rend disponible sinon on test la même chose avec la balle 2 et on break dès qu'une balle est rendu disponible.

La sélection de la balle qui va être tirée est faite au moment de l'appuie sur la touche espace, ces variables globales sont ensuites utilisées dans la fonction suivante qui est appelée à chaque affichage :

```
2048  void gestionTirs(){
2049
2050    if(tir1==1){
2051    model_balle1.transformation_model.translation.z -= 0.06;
2052    draw_model(model_balle1);
2053    if (model_balle1.transformation_model.translation.z<=-10.0){
2054      model_balle1.transformation_model.translation.z = 0.0;
2055      tir1=0;
2056      }
2057    }
2058    else {
2059        model_balle1.transformation_model.translation.x=model_dinosaure.transformation_model.translation.x;
2060        model_balle1.transformation_model.translation.y=model_dinosaure.transformation_model.translation.y;
2061
2062    }
2063    if (tir2==1){
2064    model_balle2.transformation_model.translation.z -= 0.06;
2065    draw_model(model_balle2);
2066    if (model_balle2.transformation_model.translation.z<=-10.0){
2067      model_balle2.transformation_model.translation.z = 0.0;
2068      tir2=0;
2069      }
2070    }
2071    else {
2072        model_balle2.transformation_model.translation.x=model_dinosaure.transformation_model.translation.x;
```

La capture d'écran présente les deux premiers tests pour les deux premiers tirs mais le test reste le même pour le dernier tir.

Nous allons tout d'abord nous focaliser sur les deux premiers if concernants le tir1. Ici la fonction gestionTirs() va se contenter de tester si la balle 1 est disponible, si c'est le cas cette balle est affichée à l'écran avec draw_model() puis on l'a déplace. Si la balle atteint une certaine distance fixée on n'affiche plus la balle, on rend la possibilité de la tirer impossible (tir1 = 0) et on l'a replace à son origine.
Maintenant si on regarde ce que réalise le else, c'est à dire ce que l'on fait si la balle ne peut pas être tirée, on remarque que l'on copie simplement la position du joueur (Le stégosaure). L'objectif de copier la position du joueur est que la balle soit toujours fixée à lui afin qu'elle soit tirée à partir de la bonne position dès qu'on le souhaite.

b. Les ennemis

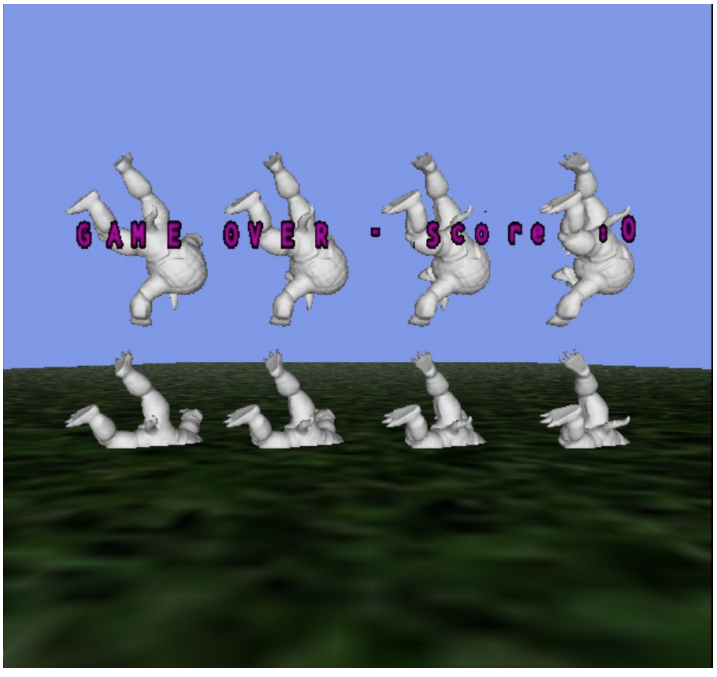
L'interaction que nous avons codée sur les ennemis est l'une des plus simples. En effet nous avons seulement besoin que ces derniers puissent tuer le joueur s'ils arrivent à son niveau.

Pour se faire nous avons simplement besoin de connaître la position de notre joueur car il ne peut pas s'approcher des ennemis mais seulement se déplacer de haut en bas (axe y) et de droite à gauche (axe x). Obtenir la position du joueur est simple car comme expliqué précédemment nous faisons apparaître tous les modèles au même endroit et donc nous pouvons facilement connaître les positions de chaque modèle en récupérant les translations de chacun.

On voit donc dans la capture d'écran ci-dessous le code qui traduit le fait que dès qu'un des ennemis arrive à la position en z correspondant à la position de notre joueur, ce dernier perd la partie.

```
2308
2309  void player_collision_detection() {
2310    /*Cette fonction test la collision entre les enemies et le joueur : si les enemies touchent le joueur on lance le game over*/
2311    if (model_armadillo.transformation_model.translation.z >= -1.8f) {
2312      game_over();
2313      model_armadillo.transformation_model.rotation = matrice_rotation(move_rotation, 3.0f, 1.0f, 0.0f);
2314      model_armadillo2.transformation_model.rotation = matrice_rotation(move_rotation, 3.0f, 1.0f, 0.0f);
2315      model_armadillo3.transformation_model.rotation = matrice_rotation(move_rotation, 3.0f, 1.0f, 0.0f);
2316      model_armadillo4.transformation_model.rotation = matrice_rotation(move_rotation, 3.0f, 1.0f, 0.0f);
2317      model_armadillo5.transformation_model.rotation = matrice_rotation(move_rotation, 3.0f, 1.0f, 0.0f);
2318      model_armadillo6.transformation_model.rotation = matrice_rotation(move_rotation, 3.0f, 1.0f, 0.0f);
2319      model_armadillo7.transformation_model.rotation = matrice_rotation(move_rotation, 3.0f, 1.0f, 0.0f);
2320      model_armadillo8.transformation_model.rotation = matrice_rotation(move_rotation, 3.0f, 1.0f, 0.0f);
2321      move_rotation += 0.5f;
2322    }
2323  }
2324
```

Le code ci-dessus lance bien la fonction game_over() que nous détaillerons plus tard et qui met fin à la partie. En plus de la fonction game_over() nous avons ajouté une fonctionnalité inutile mais divertissante qui consiste à appliquer une matrice de rotation à tous les ennemis en changeant un des paramètres de manière linéaire, ce qui a pour effet de faire tourner les ennemis sans arrêt comme on peut le voir sur la capture d'écran ci-dessous. L'effet de rotation des ennemis se déclenche au même moment que la fonction game_over() qui met fin à la partie.



c. <u>Les balles</u>

Les balles sont les modèles qui comportent le plus d'interactions. En effet nous avons dû coder les collisions entre les balles ennemis et le joueur, entre les balles

du joueur et les ennemis ainsi que les balles qui suivent les coordonnées des ennemis comme pour celles du joueur.

Tout d'abord pour coder les interactions entre les balles des ennemis et le joueur nous souhaitons que la partie soit terminée dès que le joueur se fait toucher par une balle. Nous avons donc besoin de comparer les coordonnées de la balle testée ainsi que celles du joueur et de lancer la fonction game_over() si les coordonnées sont identiques. Nous devons cependant modéliser le principe de "hitbox", c'est-à-dire de créer une boîte autour du modèle qui dès qu'elle est traversée traduit le fait que le joueur a été touché. Nous avons le code suivant (la capture ne comporte que le test pour une seule balle car il est identique pour les autres) :

```
2526  void enemy_bullet_collision() {
2527    float taille_hitbox = 0.08f;
2528    if (model_balleEnnemie1.transformation_model.translation.z >= -0.8f) {
2529      if (model_dinosaure.transformation_model.translation.x-taille_hitbox <= model_balleEnnemie1.transformation_model.translation.x
2530      && model_balleEnnemie1.transformation_model.translation.x <= model_dinosaure.transformation_model.translation.x+taille_hitbox) {
2531        if (model_dinosaure.transformation_model.translation.y-taille_hitbox <= model_balleEnnemie1.transformation_model.translation.y
2532        && model_balleEnnemie1.transformation_model.translation.y <= model_dinosaure.transformation_model.translation.y+taille_hitbox){
2533          //Joueur touché = fin de partie
2534          game_over();
2535        }
2536      }
2537    if (model_balleEnnemie2.transformation_model.translation.z >= -0.8f) {
```

Ici on va simplement tester grâce aux trois if si une balle ennemie entre dans la hitbox du joueur et si c'est le cas on lance la fonction game_over().

On test avec le premier if si la balle se trouve au niveau du joueur sur la coordonnée z, ici nous n'avons pas besoin de modéliser de hit box car l'axe z correspond à la balle qui traverse le plan frontal au joueur, c'est-à-dire le plan que l'on peut créer en posant la paume de notre main sur l'écran d'ordinateur.
Si ce test est validé on test la coordonnée suivante qui est x, si la translation totale en x de la balle est plus grande que la translation totale en x du joueur moins la taille de la Hitbox et que la translation totale en x de la balle est plus grande que la translation totale du joueur en x plus la taille de la Hitbox on peut alors tester la coordonnée suivante. Ce test modélise donc la collision de la balle avec le joueur sur sa largeur.
Enfin le dernier test suit le même principe que le test précédent mais suivant la coordonnée y du joueur qui correspond à la hauteur du joueur.
Si tous ces tests sont passés on lance la fonction game_over().

On réalise évidemment les tests décrits précédemment pour toutes les balles en même temps.
La collision entre les balles du joueur et les ennemis (Armadillos) suit le même principe que la collision entre les balles ennemis et le joueur. On a le code suivant :

```
2341
2342  void bullet_collision_detection() {
2343    float taille_hitbox = 1.0f;
2344    if (model_balle1.transformation_model.translation.z <= model_armadillo.transformation_model.translation.z+0.5 &&
2345    model_balle1.transformation_model.translation.z >= model_armadillo.transformation_model.translation.z-0.5 ){
2346      if (model_armadillo.transformation_model.translation.x-taille_hitbox <= model_balle1.transformation_model.translation.x &&
2347      model_balle1.transformation_model.translation.x <= model_armadillo.transformation_model.translation.x+taille_hitbox
2348      && test_draw_1 == 1){
2349        if (model_armadillo.transformation_model.translation.y-taille_hitbox <= model_balle1.transformation_model.translation.y &&
2350        model_balle1.transformation_model.translation.y <= model_armadillo.transformation_model.translation.y+taille_hitbox){
2351          //Armadillo1 touché
2352          test_draw_1 = 0;
2353          score += 175;
2354        }
2355      }
2356      if (model_armadillo2.transformation_model.translation.x-taille_hitbox <= model_balle1.transformation_model.translation.x && mod
2362      }
```

De la même manière que précédemment on test donc si la balle tirée par le joueur entre en collision avec la hitbox d'un ennemi pour chaque coordonnée. On ajoute également un test pour vérifier si l'ennemi est affiché à l'écran, car la balle peut entrer théoriquement en collision avec un ennemi déjà mort car les modèles sont encore existants même s'ils ne s'affichent plus à l'écran.
Si un ennemi est touché, on ne l'affiche plus à l'écran grâce à une variable globale qui gère si l'ennemi doit ou non être affiché.
La difficulté de cette fonction est qu'il faut réaliser ces tests pour tous les ennemis pour chaque balle qui est tirée par le joueur, ce qui fait beaucoup de tests en même temps.

Enfin nous devons coder le fait que les balles qui sont tirées par les ennemis sont "collées" à eux et bougent donc en même temps qu'eux. Nous avons donc choisi quatre ennemis qui allaient tirer et nous leur avons fixé une balle. On a le code suivant :

```
2137  void gestionTirsEnnemis(){
2138    if(tirEnnemi1==1 && test_draw_1 == 1){
2139    model_balleEnnemie1.transformation_model.translation.z += 0.09;
2140    draw_model(model_balleEnnemie1);
2141      if (model_balleEnnemie1.transformation_model.translation.z>=model_dinosaure.transformation_model.translation.z+3){
2142        model_balleEnnemie1.transformation_model.translation.z = model_armadillo.transformation_model.translation.z;
2143        tirEnnemi1=0;
2144        }
2145    }
2146    else {
2147        model_balleEnnemie1.transformation_model.translation.x=model_armadillo.transformation_model.translation.x;
2148        model_balleEnnemie1.transformation_model.translation.y=model_armadillo.transformation_model.translation.y;
2149        tirEnnemi1=1;
2150    }
```

Chaque ennemi a donc une balle qui suit ses coordonnées tout comme pour les balles qui suivent les coordonnées du joueur. On choisit d'afficher seulement quand elle est tirée, elle est invisible sinon.
Nous testons donc dans un premier temps si l'ennemi doit tirer et s'il est bien affiché (car comme dit précédemment il n'est que rendu invisible lorsqu'il est tué). Si ces tests sont validés on affiche la balle et on la déplace puis on test si la balle est en fin de course, si c'est le cas on l'a rend invisible et on l'a replace aux coordonnées de l'ennemi auquel elle est affectée.

En revanche si la l'ennemi n'est pas en train de tirer on demande simplement à la balle de suivre les coordonnées de l'ennemi puis on lui ré-ordonne de tirer(dernier else sur la capture d'écran ci-dessus).

d. Le texte

Les interactions avec le texte sont très simples, nous voulons ajouter 175 au score du joueur à chaque fois qu'il tue un ennemi.

Pour ce faire nous avons une variable globale score qui est constamment affichée et nous la mettons à jour à chaque fois que l'on tue un ennemi. Cette mise à jour se fait dans la fonction bullet_collision_detection() présentée précédemment, on a :

```
model_balle1.transformatio
  //Armadillo1 touché
  test_draw_1 = 0;
  score += 175;
}
```

Si l'ennemi est touché lorsque l'on teste la collision entre la balle et l'ennemi, on ajoute simplement 175 à la variable score.

4. LE JEU

Maintenant que nous avons présenté les différentes fonctions qui régissent les règles du jeu et que nous savons comment les interactions sont réalisées entre tous les modèles, il reste simplement à implémenter les notions de fin de partie et de régénération des ennemis.

a. Régénération des ennemis

Lorsque le joueur a tué tous les ennemis nous devons régénérer une horde d'ennemis pour que le jeu continue et cela sans cesse jusqu'à ce que le joueur meurt. Tout ceci s'effectue dans la fonction suivante :

Cette fonction est appelée à chaque affichage des éléments du jeu, on a donc besoin de tester si aucun des ennemis n'est affiché à l'écran. Si le test est réussi on fait deux choses :

- On passe les variables globales qui gèrent si les ennemis doivent être affichés ou non à 1 pour indiquer que l'on doit maintenant les afficher de nouveau à l'écran,
- On passe à 1 la variable globale qui gère si les ennemis doivent être repositionnés à leur position d'origine. Une fois cette variable passée à 1 elle impacte la fonction move_enemy_squad() qui gère les déplacements des ennemis et qui va s'occuper de remettre les ennemis à leur position d'origine.

C'est tout ce que fait cette fonction, étant appelé sans cesse il est très important qu'elle ne se déclenche qu'au moment où tous les ennemis ont été tués.

b. Fin de partie

La fin de partie se déclenche lorsque le joueur entre en collision avec une balle ennemi, ou s'il entre en collision avec la horde des ennemis car il n'a pas eu le temps de tous les tuer avant que la horde arrive à son niveau. La fin de partie doit simplement stopper l'affichage du joueur à l'écran, empêcher les ennemis de tirer ainsi qu'afficher "Game Over" sur l'écran.
On a alors le code suivant :

```
2565  void game_over() {
2566    test_draw_player = 0;
2567    tirEnnemi1 = 0;
2568    tirEnnemi2 = 0;
2569    tirEnnemi3 = 0;
2570    tirEnnemi4 = 0;
2571    //afficher game over
2572  }
```

Une fois appelée la fonction va donc passer les variables globales qui gèrent les tirs des ennemis à 0 pour les empêcher de tirer ainsi que la variable globale qui régit si le joueur doit être affiché ou non. On remarque aussi que l'affichage de "Game Over" à l'écran n'est pas directement effectué par cette fonction cependant elle le déclenche indirectement.

On remarque en effet dans la fonction de gestion du texte qui modifie la variable contenant le texte à afficher que lorsque la variable test_draw_player est à zéro on afficher "Game Over". Cette variable est bien la variable qui régit si le joueur doit être affiché et elle est bien passée à zéro par la fonction game_over() et uniquement par cette fonction.

5. CONCLUSION

L'objectif de ce projet était de créer un jeu simple utilisant les principes et techniques vues au cours du module de Synthèse et durant le TP de découverte de l'API OpenGL.
Ce TP nous a donné le bagage de compétences nécessaires pour aborder un projet de conception du jeu de notre choix. Pour ce jeu, nous nous sommes servi de l'ébauche de projet qui contenait déjà un monstre, un sol et un Stégosaure ainsi que différentes fonctions de déplacement du stégosaure et de la caméra.

Nous nous sommes alors attelés à concevoir un jeu de type Space Invaders en 3D. Nous avons codé un déplacement fluide pour le joueur, ainsi que sa capacité à tirer des balles, mais aussi une horde de monstres qui se déplacent en direction du joueur et lui tirent dessus.
À cela nous avons ajouté l'affichage de texte dynamique grâce à l'utilisation d'un deuxième programme graphique.

Ce projet nous a permis de vraiment approfondir nos connaissances car nous avons été confrontés à de réels problèmes d'implémentation et à la complexité d'utilisation

d’OpenGL. Ainsi, le fait d’avoir pris du temps pour s’impliquer dans la résolution des problèmes qui sont survenus nous a permis d’appréhender de manière plus aisée l’API OpenGL qui nous paraissait relativement difficile d’utilisation au départ.