Rapport de 1ère année
de Master Informatique
Année scolaire 2021-2022

Projet de Synthèse d'images

Université de Caen, Normandie
UFR des Sciences

Rédigé par
Raphaëlle LEMAIRE (21802756)
Guillaume LETELLIER (21804030)

Professeur référant
Alexis LECHERVY

Table des matières

1 Introduction

Dans l'UE de synthèse d'images, il nous a été demandé de réaliser un petit jeu jouable à deux sur un même clavier en se basant sur les travaux réalisés en TP. La réalisation du projet a dû être faite dans un espace de temps restreint avec maximum 2 personnes dans l'équipe de développement.

Description du jeu : Nous avons décidé de partir sur un jeu de combat/course spatial. L'objectif principal du jeu est de s'emparer d'une planète faite d'eau. Le joueur 1 est représenté par un dauphin et le joueur 2 par un requin. Les joueurs peuvent gagner de différentes façons :

- soit ils arrivent en premier sur la planète
  - s'ils y sont depuis assez longtemps, la planète pourra repousser la puissance de frappe de l'autre joueur
  - sinon l'autre joueur pourra détruire le vaisseau ennemi
- soit ils se combattent et le dernier vaisseau en vie gagne la nouvelle planète

Pour les combats, le joueur 1 peut lancer des crevettes et le le joueur 2 peut lancer des poulpes. Étant dans l'espace, ils peuvent utiliser l'environnement pour se cacher et surprendre l'ennemi (s'il est considéré caché, le joueur n'apparaîtra plus à l'écran de l'autre joueur) Pour éviter que le jeu dure trop longtemps, un minuteur sera affiché et les joueurs perdront tous les deux au-delà de 3 minutes. Cela sera modélisé par une invasion par une autre armée (représentée par l'ordinateur sous forme de thon).

2 Réalisation du projet

2.1 Gestion des écrans et du fenêtrage

Le fenêtrage et la gestion des écrans ont été basés sur les exemples de code réalisés en travaux pratiques afin de réutiliser l'architecture des exemples. Nous avons créé une classe `ConquererApplication` qui est la classe permettant d'utiliser et de lancer le jeu. Une classe supplémentaire a été créée, `FullScreenApplication` qui permet, comme son nom l'indique, de créer une application prenant entièrement la taille de l'écran principal. Cette classe n'a pas été utilisée car lors du premier lancement de l'application héritant de cette classe, la machine virtuelle a totalement crashée donc nous avons décidé de rester sur un jeu fenêtré. Il est utile de noter que tous les éléments affichés sur les écrans se redimensionnent automatiquement lorsque la fenêtre subit un redimensionnement. Tout le système de rendu est réalisé avec l'aide de la classe `Renderer` qui encapsule toutes les autres instances de rendu afin de n'avoir qu'à déplacer l'instance d'écran en écran afin de visualiser l'environnement.

Écran d'accueil

Lorsque l'on lance le jeu, nous tombons sur l'écran d'accueil (classe `StartStage`) montré à la figure 1.

FIGURE 1 – Écran de lancement du jeu

Comme on peut le voir, c'est une vue assez simple montrant le titre du jeu, l'action à réaliser pour pouvoir y jouer et en arrière-plan, l'environnement du jeu déjà généré. Pour

ce qui est de l'affichage du texte, nous avons repris le code sur l'exemple `rubik` disponible dans `glitter`. Le code a été adapté afin de ne plus contenir des vecteurs (VAO, couleur de texte, couleur d'arrière-plan) mais seulement des dictionnaires associant une clé sous forme de chaîne de caractères et une valeur sous la forme d'une instance de structure (contenant un VAO, une couleur de texte et une couleur d'arrière-plan). Cela permet de très simplement mettre à jour un code en remplaçant l'instance de la structure dans le dictionnaire.

Enfin, quand on appuie sur la touche "Entrée", le jeu se lance en passant à l'écran de jeu.

Écran de jeu

Lorsque nous entrons dans cet écran, nous obtenons un écran semblable à la figure 2 (classe `PlayingStage`).

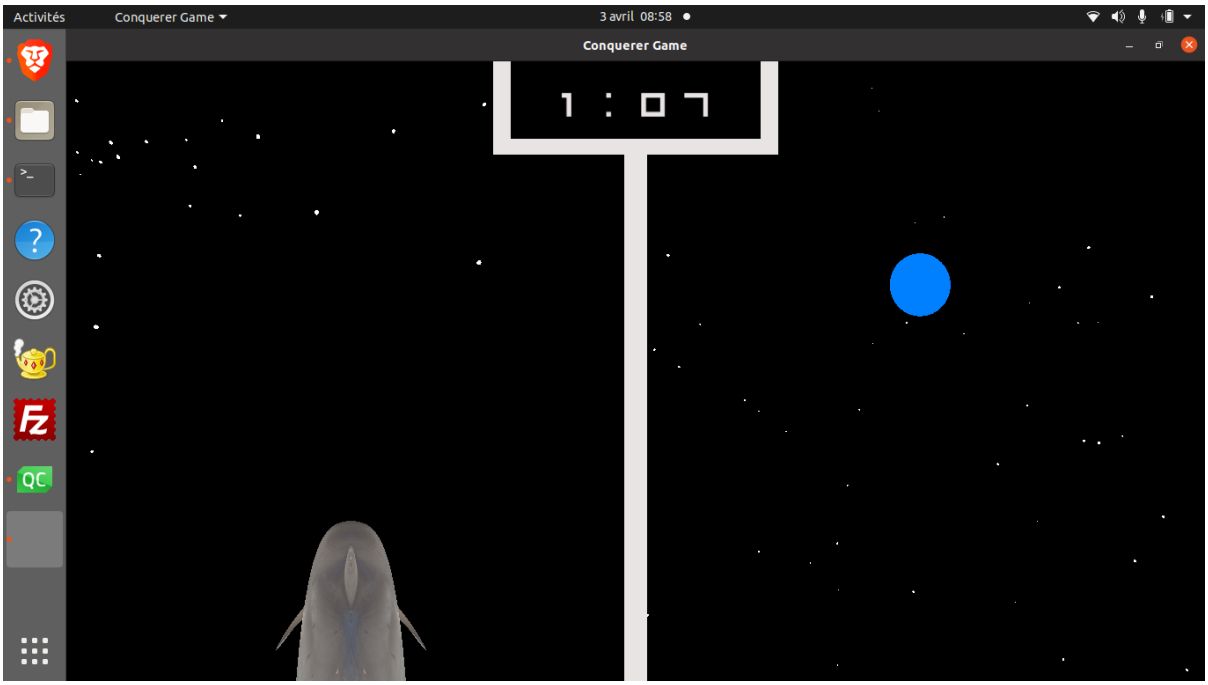

FIGURE 2 – Écran de jeu

C'est à ce moment que le jeu commence et que le chronomètre se lance (voir partie sur la logique du jeu). Comme on peut le voir, chacun des joueurs possèdent leur partie d'écran : la partie gauche pour le joueur 1 et la partie de droit pour le joueur 2. L'écran ne peut pas être quitté à moins de gagner ou perdre, ou encore en appuyant sur "Echap" pour fermer la fenêtre (ou en faisant "Alt + F4"). Le joueur 1 peut jouer en utilisant les touches "Z", "Q", "S", "D" (pour aller vers la haut, vers la gauche, vers le bas et vers la droite) et les flèches directionnelles pour le joueur 2.

L'overlay réalisée dans la classe `GameOverlay` permet d'afficher les rectangles blancs servant de séparateur pour les joueurs et le chronomètre. Concernant le chronomètre,

nous stockons un compteur dans la classe `Renderer` (classe qui permet d'encapsuler tout le système d'affichage du jeu). Il est incrémenté grâce aux `deltaTime`, valeur qui représente le temps passé entre le rendu de deux images. Le compteur est ainsi passé à la méthode `getLeftTime` de la classe `GameLogic` afin de récupérer le temps de jeu restant pour ensuite le passer à l'overlay pour mettre à jour l'affichage du chronomètre.

La vision est faite grâce à deux caméras fixées aux deux joueurs (une par joueur). L'orientation de la caméra est calculée avec une dérivée des angles d'Euler (latitude et longitude) en utilisant des calculs matriciels en modifiant les coordonnées locales de la caméra afin de supprimer les blocages de cardan (gimbal lock) dus aux fonctions sinus cosinus. Une amélioration possible que l'on peut noter est que l'on pourrait utiliser à la place de quaternions afin de simplifier les calculs d'orientation et de rotation des objets et caméras.

Lorsque le jeu se termine, nous arrivons à l'écran de fin.

Écran de fin

La figure 3 représente l'écran de fin que nous obtenons lorsque le jeu se termine (classe `EndStage`).

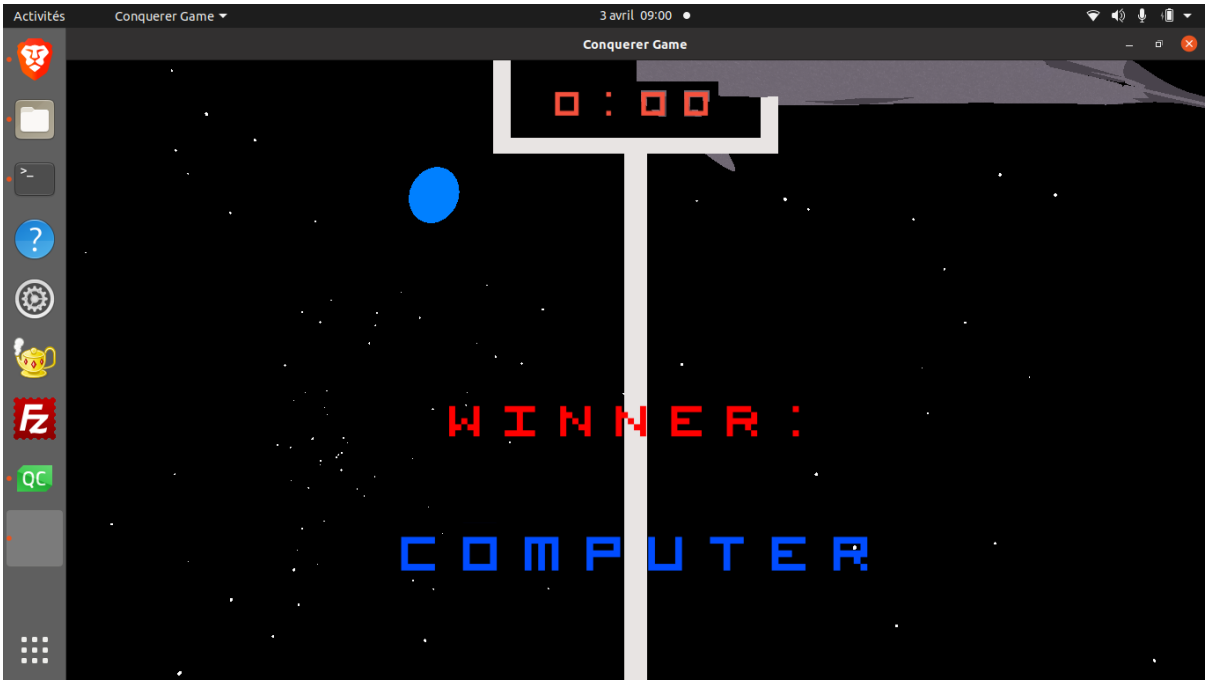

FIGURE 3 – Écran final

Nous affichons l'environnement du jeu en arrière-plan qui n'est plus modifiable (effet d'arrêt complet). De plus, nous affichons le gagnant de la partie.

Lorsque nous appuyons sur la touche "Entrée", le jeu se ferme complètement.

2.2 Création de l'espace

Définition de la taille

La taille de l'espace de jeu est défini lors de son appel dans le `StartStage`, nous l'avons défini à 20 afin que les joueurs puissent se promener dans un espace convenable. Cet espace a été créé en forme de sphère pour donner l'illusion d'un univers infini sans les problèmes d'étoiles amassées dans les coins d'un carré. Ainsi, le 20 définit le rayon de la sphère.

Calcul des étoiles

Le calcul des étoiles se fait à l'aide de deux méthodes, `generateRandomStars` pour leurs créations et `generateRandomSizes` pour la définition de leurs tailles. Ainsi, pour le placement des étoiles, on calcul en fonction de la taille de la sphère et d'angle choisis aléatoirement. C'est comme ça que l'on va avoir des étoiles sur la sphère pour donner un effet 3 dimensions. Le calcul de la taille se fait aussi grâce à une variable calculée aléatoirement pour chaque étoile. Le nombre d'étoiles quant à lui est défini dans la classe `Renderer` et est ici de 250.

2.3 Réalisation et affichage des objets

Quatre classes servent à la réalisation des objets. Chacune d'elles héritent d'une classe abstraite : `AbstractGameObject`. Cela va permettre aux objets d'avoir un fonctionnement très proche. Ils vont ainsi avoir des méthodes en commun, notamment la collision, gestion de la vie et des déplacements, rotations ainsi que zoom.

La planète bleue

L'un des objectifs de notre jeu, autre que la destruction de l'autre joueur, était d'atteindre une planète faite d'eau. Cette dernière est modélisée sous la forme d'une sphère de couleur bleue. Sa création et son affichage sont géré par deux classes, `BasicObject` et `Mesh`. Pour mieux visualiser cette sphère, vous pouvez vous référer à l'image 4.

Les joueurs

En regardant la figure 4, vous avez dû remarquer que l'on pouvait apercevoir un objet, il en va de même pour l'image 1. Cette objet est, à gauche un dauphin tandis qu'à droite, il s'agit d'un requin. Ils représentent les deux personnages jouables.

Ces deux objets sont affichés avec leurs textures grâce à l'appel de méthode présentent dans `RenderObjectConqueror`. Les modèles 3D ont été trouvés sur un site qui depuis a été supprimé. Il persiste quelques problèmes au niveau de nos objets. Non pas sur l'affichage mais sur la gestion des rotations, ainsi, vous pourrez apercevoir, en jouant au jeu, un dauphin rentrer dans la planète bleue à reculons ou même de côté.

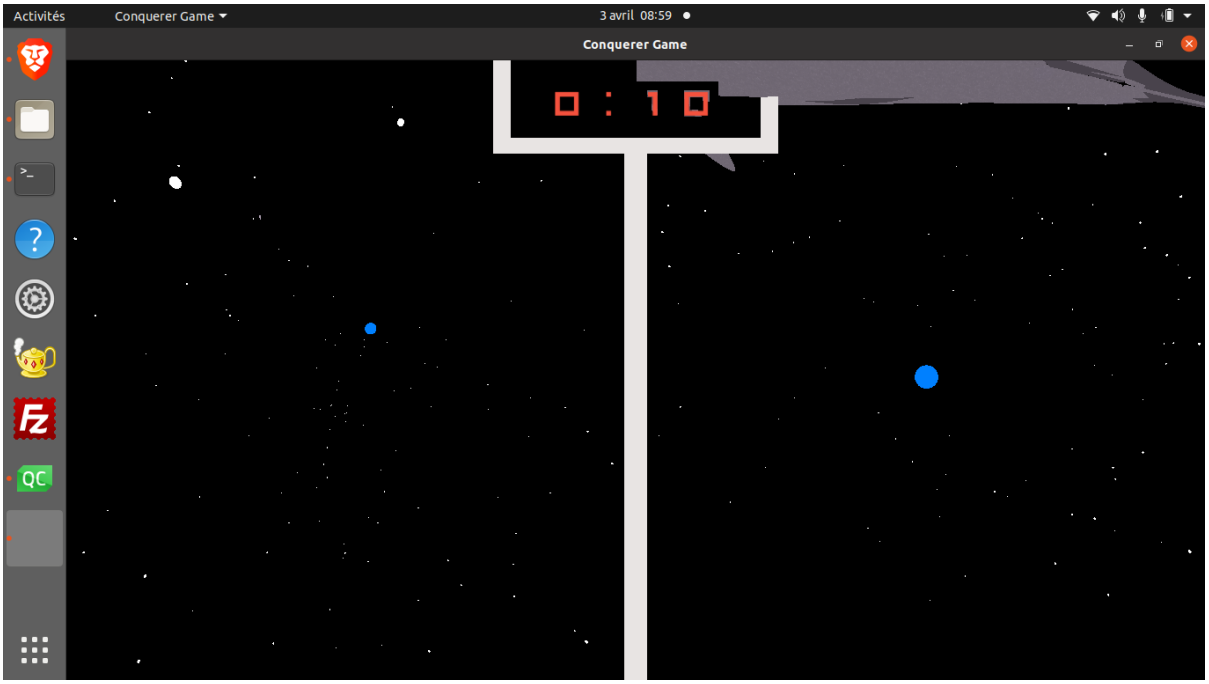

FIGURE 4 – Écran de jeu

Les astéroïdes

Aucun astéroïde n'est visible dans le jeu mais, ce n'est pas pour autant qu'il ne sont pas affichables. Vous pourrez, en visitant la classe `AsteroidObject` vous convaincre de cela. Il existe trois formes d'astéroïdes possibles, les Thons (qui conquerrons la planète bleue si les joueurs ne l'atteignent pas dans les 3 minutes), les Alaska Pollok mais aussi les Cods.

Ces astéroïdes sont tous des poissons. La raison est simple, nous voulions matérialiser une sorte de vengeance des poissons envers leurs prédateurs en les plaçant en obstacles sur le passage du requin et dauphin. Si les joueurs passent trop près des astéroïdes, ils auraient perdu un peu de leur vie.

Les projectiles

Les astéroïdes ne sont pas les seuls à ne pas être affichés à l'écran. Cela vient du fait que, nous n'avons pas encore attribué une touche à la création d'un projectile. Encore une fois, le code derrière l'envoi des projectiles et la baisse de la vie est présent. Nous parlerons de cette partie avec plus de précision avec les collisions et la gestion de la vie. Leurs code de création se trouve dans `ProjectileObject`.

Affichage aléatoire

Dans la classe `Renderer`, vous pourrez trouver une partie permettant le calcul de l'emplacement des objets dans l'espace. Les méthodes sont `calculPositionPlanet` et

`calculPlacementObject`. La première méthode calcul l'emplacement de la planète à une distance qui varie entre 0 et la taille de l'univers -10 du centre du monde. Ce calcul permet au objets d'être placé à une distance d'un minimum de 10 de la sphère. Dans la seconde méthode, on calcul la distance des objets par rapport à la planète, elle sera la même pour les deux.

Une fois le calcul des emplacements effectués, il faudra créer les objets en créant une instance de leur classe et les afficher grâce au `update` de cette même classe. On a donc maintenant les joueurs et la planète placé à bonne distance pour éviter que le jeu ne soit trop simple.

2.4 Gestion de la collision

Dans le fichier `CollisionShapes.hpp`, nous avons créé plusieurs classes qui permettent de gérer des collisions entre différents objets, que ce soit des points, des rectangles et des sphères.

La collision globale

Nous avons différentes méthodes permettant de gérer tous les types de collisions :
- point-point : `isCollidedPointToPoint` ;
- point-rectangle : `isCollidedPointToRectangle` ;
- point-sphère : `isCollidedPointToSphere` ;
- rectangle-rectangle : `isCollidedRectangleToRectangle` ;
- rectangle-sphère : `isCollidedRectangleToSphere` ;
- sphère-sphère : `isCollidedSphereToSphere`.

Pour cela, nous nous sommes appuyés sur ce tutoriel permettant de réaliser toutes les collisions comme nous le souhaitions.

Une méthode permet de simplifier les appels, `isCollided`. Elle prend deux formes qu'elle cast ensuite dans les objets qu'ils sont censé l'être grâce à la fonction `static_cast<T*>`. Elle appelle ensuite la méthode adéquate en fonction des casts réalisés.

Axes d'amélioration

On pourrait utiliser des rectangles OBB au lieu de AABB qui permettent de tourner un rectangle en même temps que l'objet afin que la boîte de collision reste dans la même rotation que l'objet associé. La méthode de calcul change drastiquement car on doit vérifier la collision sur chacun des plans associés à la boîte.

Une autre méthode possible est de créer des arbres de rectangles OBB qui se mettent à la bonne taille en fonction de différentes parties d'un objet (par exemple, une boîte pour le corps du requin, plusieurs pour sa queue et d'autres pour ses nageoires). [1]

1. `https://www.gamedeveloper.com/programming/advanced-collision-detection-techniques`

2.5 Logique du jeu

Comme pour tout jeu, une logique doit être présente afin de limiter les actions possibles des joueurs, définir les règles du jeu, etc.

Mise à jour du système logique

Tout d'abord, nous avons créé une méthode `updateObjects` permettant de mettre à jour tous les objets présents dans l'environnement, comme le déplacement automatique des joueurs et des projectiles vers la direction actuelle. Cette méthode supprime les objets inutiles grâce à la méthode `removeUselessObjects` notamment des projectiles qui iraient trop loin et qui ne pourraient entrer en collision avec un autre objet.

De plus, on appelle une méthode privée nommée `checkCollisions` qui permet de vérifier les collisions entre les différents objets de l'environnement (projectiles, joueurs, planètes, astéroïdes). En effet, on vérifie les collisions entre les objets qui peuvent se déplacer et tous les autres. On vérifie donc :

— collisions joueur-joueur ;
— collisions joueurs-astéroïdes ;
— collisions joueurs-planètes ;
— collisions projectiles-planètes ;
— collisions projectiles-joueurs.

Lorsque des objets n'ont plus de vie, ils sont ajoutés à une liste afin d'être supprimés. Ceci permet donc de libérer de la mémoire et des capacités de calcul.

Cette méthode réalise toutes ces actions (plus la méthode de calcul du joueur en cours de capture de la cible et du gagnant) uniquement lorsque le jeu est lancé et qu'il n'est pas encore terminé.

Système de temps

Le chronomètre se lance lorsque la méthode `launch` est appelée. Effectivement, elle assigne le nombre de secondes écoulées dans la fenêtre GLFW afin de vérifier avec `getLeftTime` que le temps restant est supérieur à 0. De plus, on peut vérifier que le jeu est fini grâce à la méthode `isFinished`. Analoguement, on peut savoir si la planète cible est en cours de capture et si oui, depuis combien de temps avec les méthodes `isTargetCaptureBegan` et `getCapturer`.

Calcul de la fin du jeu

Tout d'abord, on vérifie les niveaux de vie des joueurs quand ils entrent en collision. Nous avons plusieurs cas de figure :

1. les joueurs ont le même nombre de points de vie : meurent tous les deux et l'ordinateur gagne ;
2. les joueurs n'ont pas le même nombre de points de vie : le joueur ayant le plus gagne.

Lorsque qu'un joueur entre en collision avec la planète cible, le compteur de capture démarre et le joueur capturant l'objectif gagne si et seulement si le joueur adverse n'a pas réussi à détruire la planète (pour signifier que le joueur capturant la planète est vulnérable à toutes les attaques le temps que les armes de la planète s'établissent). Il est utile de noter que le joueur adverse peut s'écraser sur la cible si le joueur la capturant est dessus. S'il a plus de vie que la planète cible, il gagne. Nous avons plusieurs cas concernant la capture :

1. les deux joueurs arrivent exactement au même moment :
   (a) ont le même nombre de points de vie : l'ordinateur gagne ;
   (b) ont un nombre différent : le joueur ayant le plus de points de vie gagne ;
2. le joueur arrivant en premier commence la capture.

Le système de capture n'est pas fonctionnel et donc le code associé n'a pu être testé.

3 Conclusion

Le projet ayant dû être réalisé à 2 personnes, dans une période de temps assez courte et sur un jeu choisi demandant plus de temps de développement que ce que nous pouvions accorder, il n'a malheureusement pu être terminé. De plus, de nombreux problèmes sont apparus durant le développement, notamment des problèmes de permissions nous empêchant d'avancer, ou encore sur la machine virtuelle sur le bureau distant. En effet, sur celle-ci, quelques jours avant le rendu du projet, il nous était impossible de l'utiliser (car le service était hors ligne ou que la limite des personnes pouvant l'utiliser simultanément était atteinte) et quand nous le pouvions, les objets générés par notre code ne s'affichaient pas ou de façon inattendue. Grâce à l'ordinateur personnel de Raphaëlle, nous avons pu réaliser les quelques tests afin de vérifier mais il est difficile de réaliser du code que l'on ne peut pas vérifier !

Malgré cela, nous avons réussi à afficher des objets texturés, gérer les différents écrans de l'application, déplacer les joueurs dans l'espace grâce à la caméra fixée sur eux.