RAPPORT DU PROJET FILE 2023

Black Seas

Hélène HASSAN
Virgile HENRY
Brice PERES
Paul COLINOT

Table des matières

1 Résumé

BlackSeas est un jeu multijoueur de navigation en 3D réalisé entièrement à partir de rien (*from scratch*) à l'aide de notre propre moteur de jeu : Gear. Le développement s'est réalisé entièrement en Rust accompagné de la spécification OpenGL pour la partie graphique.
L'objectif de ce projet de grande ampleur était d'apprendre et prendre conscience de la difficulté du développement d'un jeu en son entièreté, le tout en manipulant un langage qui nécessite une certaine rigueur : Rust.
Au moment du rendu, nous avons réalisé la grande majorité des fonctionnalités que nous voulions implémenter dans notre jeu. L'objectif de ce rapport est de retracer les grands axes de développement qui ont permis la naissance de notre jeu.

2 Introduction

BlackSeas est un projet né d'une passion pour l'informatique et les maths entre quatre amis. À l'heure où la majorité des jeux sont développés à l'aide de moteurs de jeux puissants et qui facilitent beaucoup la vie des développeurs, nous avons décidé d'emprunter une autre voie pour développer notre jeu : *from scratch* qui signifie que nous avons tout réalisé depuis la case départ : sans moteur de jeu, sans rien. La seule chose que nous nous sommes autorisés reste l'utilisation de la bibliothèque glfw pour pouvoir afficher une fenêtre, ainsi que d'autres bibliothèques mathématiques en Rust. Pourquoi from scratch ? Parce que nous envisageons ce projet comme l'opportunité d'apprendre à faire les choses nous-mêmes et de découvrir au maximum le monde du développement de jeux vidéos, c'est-à-dire pas seulement savoir développer un jeu, mais savoir développer un moteur de jeu. L'objectif de ce projet était également de nous amuser tout au long du semestre.

Dans les prochaines parties, nous tâcherons de présenter les grands axes de développement de BlackSeas et de Gear, puis nous montrerons l'état du jeu au moment du rendu dans la section **Résultats** afin d'apporter une conclusion générale dans **Discussion**. Une partie **Références** est également présente pour les papiers de recherche ou articles dont nous nous sommes inspirés pour réaliser certains aspects du projet.

3 Développement réalisé

3.1 Structure du code

Pour essayer de garder un code clair tout au long du projet, nous avons planifié une certaine architecture sur l'ensemble du code, et nous avons nommé les bibliothèques centrales afin de faciliter la communication sur le développement des ces différents modules.
On peut donc séparer le projet en trois couches :

- La "Foundry" :
La Foundry est le nom donné à la bibliothèque représentant l'ECS du projet. Un ECS, "Entity Component System", est une architecture permettant de simuler un ensemble d'entités ayant des comportements arbitrairement similaires ou différents, sans passer par une conception objet qui peut être contraignante. Elle sera expliquée plus en détails dans sa propre section.

- Gear :
Gear est le moteur de jeu qui fait tourner Black Seas. Il a été conçu par dessus la Foundry, et pensé pour pouvoir développer n'importe quel jeu, pas forcément le notre. Ces objectifs, certes contraignants au début, ont permis de nous forcer à écrire un code extensible, ce qui nous a permis de pouvoir ajouter des fonctionnalités sans avoir à revenir sur ce qui était déjà implémenté.

- Black Seas :
Black Seas est le produit final. Il a été fait par dessus Gear, et est séparé en deux projet : Black Seas et Black Seas Server, permettant un jeu en multijoueur.

3.2 Foundry

La Foundry est une bibliothèque permettant de construire un Entity Component System (ECS).
Le besoin d'un ECS vient de la nécessité à simuler un monde, avec plusieurs types d'entités différentes, mais qui peuvent avoir des comportements similaires. La première approche à un tel système serait une architecture objet avec de l'hérédité. On pourrait par exemple créer une classe pour les objets physiques, qui subissent la gravité et ont des collisions. Chaque entité qui souhaite subir la physique peut alors hériter de cette classe. Certaines classes posent problème car elles veulent peut-être subir uniquement les collisions, mais pas la gravité, comme le terrain par exemple. Il faudrait alors découpler ces deux comportements. On pourrait s'en sortir avec de l'héritage multiple, mais étendre ce type de système peut vite devenir long et fastidieux et les nombreux cas particuliers vont nous forcer à reprendre du code existant.

Une bonne solution est donc la création d'un Entity Component System (ECS).

Le principe d'un système composants - entités - systèmes, est de faire abstraction de la notion d'entité, qui n'est qu'une collection de composants. Les entités ne sont plus qu'un identifiant, qui regroupe cet ensemble de composants. Les composants sont des blocs (*containers*) pour des données, attachés aux entités. Enfin, les systèmes permettent de modifier les données des composants selon des comportements définis.

Par exemple, on peut créer un composant de type "RigidBody", qui représente les objets qui subissent la physique, et un composant de type "Collider", qui représente un objet de collision. On peut écrire un système physique qui va itérer sur tous les Collider, résoudre les collisions, puis sur les Rigidboy, et appliquer différentes forces (gravité, vent...). Ensuite, il suffit que les objets qui soient affectés par les forces et les collisions possèdent les deux composants et que ceux comme le terrain qui ne veulent pas subir de forces n'aient que le Collider. Sans même ajouter de code pour les cas particuliers, le système physique ne va simplement pas itérer sur cette entité.

En pratique, créer un tel système est plus complexe qu'un modèle objet. Il faut trouver un moyen de stocker les composants, et surtout une méthode pour itérer sur un archétype. Un archétype est une collection de types de composants. Itérer sur un archétype précis est la clé de tout système, car chaque système va avoir besoin de plusieurs composants d'une même entité pour fonctionner. Par exemple, le système des collisions va avoir besoin des composants de collisions, mais aussi des positions et vitesse de chaque objet.

Table des composants :

id entité/composant	entité 1	entité 2	entité 3	entité 4	entité 5
Position	⟨ Vecteur3 ⟩	⟨ Vecteur3 ⟩	⟨ Vecteur3 ⟩		⟨ Vecteur3 ⟩
Vitesse	⟨ Vecteur3 ⟩		⟨ Vecteur3 ⟩		

On peut visualiser l'ensemble des composants comme un tableau à deux dimensions, une pour les identifiants des entités, et l'autre pour le type des composants. Une case de tableau peut être vide, car chaque composant n'existe pas forcément pour chaque entité (dans l'implémentation réelle, cette table est optimisée pour ne pas avoir ces trous).

Dans l'exemple ci dessus, les entités 1 et 3 ont des positions et des vitesses, on peut les interpréter comme des entités qui se déplacent dans notre monde. Les entités 2 et 5 sont statiques, car elle n'ont que des positions. Mais si un système a besoin d'itérer sur les positions des entités pour, par exemple, renverser le monde sur une symétrie verticale, il peut facilement le faire avec un itérateur et va s'appliquer aux entités 1, 2 3 et 5. Il n'agira pas sur l'entité 4, mais comme elle n'a pas de

position, cela n'a pas de sens d'essayer de la déplacer.

Itérateur sur les archétypes :

L'opérateur clé de la Foundry est l'itération sur les archétypes. Chaque système va avoir besoin d'itérer sur des archétypes différents. Il faut donc une méthode pour créer des itérateurs, et cela a été possible avec les macros. En Rust, les macros sont des règles d'écriture qui vont modifier et écrire du code à la compilation. Dans notre cas, elles ont été utilisés pour générer un itérateur sur un archétype quelconque, ce qui est intéressant dans notre cas car Rust est un langage fortement typé. Nous avons ainsi créé la macro :

```
iterate_over_components!(components; Component1, Component2, ...)
```

En passant à cette macro la table des composants de la foundry, et les types des composants sur lequels on va itérer, cette macro va générer du code de plusieurs centaines de lignes qui va créer un itérateur sur les types demandés. Cela fonctionne pour n'importe quel type, et pour n'importe quel nombre de composants.

3.3 Gear

Gear est la couche entre la Foundry et notre jeu, Black Seas. Ce moteur de jeu implémente énormément de fonctionnalités de base sur l'ECS de la Foundry, comme : la création de fenêtre, l'affichage dans cette fenêtre, la lecture des entrées utilisateurs, il fournit des primitives pour les modèles 3D, les positions des objets, etc. En ce sens, on considère Gear comme un moteur de jeu, sur lequel on a développé Black Seas.

3.3.1 Transform

On appelle un 'Transform' un Composant qui donne aux objet une position, rotation et taille dans l'espace. Ce composant est essentiel et est utilisé par la quasi-totalité des systèmes, car il donne la notion d'espace aux objets. Il permet aussi de hiérarchiser les objets dans le monde, ce qui est très utile pour créer des entités complexes comme les bateaux de Black Seas.

3.3.2 Recompilation de shader

Lors du developpement d'un jeu, il peut être fastidieux lorsque l'on travaille sur une fonctionnalité d'avoir à recompiler l'entièreté d'un projet et de se remettre en situation de test dans le jeu. C'est pourquoi Gear est capable de recompiler les shaders pendant l'éxécution. Ainsi, il est possible de changer le code et de recompiler les shaders du projet directement depuis le jeu sans avoir à passer par ces étapes.

3.3.3 Graphe de shaders

Afin de simplifier la gestion des Texture d'OpenGL, un système de graphes de shader a été implementé dans Gear. Chaque noeud d'un graphe est associé a un shader (compute ou fragment) qui va générer une ou plusieurs textures. Chaque noeud passe ensuite ses textures aux noeuds suivants qui peuvent les utiliser. Il est aussi possible d'invalider un noeud, pour signaler qu'il doit recalculer ses textures. Ainsi, lorsque l'on récupère la texture d'un noeud qui nous intéresse, le graphe se charge de recalculer les noeuds invalidés ainsi que leurs enfants avant de retouner la texture.

3.3.4 Effets en post processing

Chaque entité de Gear possédant un composant de type Caméra a un buffer associé. Cela permet de faire le rendu d'une même scène depuis plusieurs caméras et de rajouter certains effets supplémentaires par dessus les textures contenant

le rendu de la scène : les effets de post-processing. Ce graphe de shaders a été utilisé pour implémenter les différents effets en post-processing de Gear tels que le *tone mapping*, la *gamma correction*, le brouillard, le bloom, la pluie et les nuages.

3.3.5 Propriétés des matériaux

Un materiau est un wrapper sur les shader OpenGl permettant de transmettre les informations relatives à ce materiau au shader associé à un objet. Ces matériaux peuvent avoir plusieurs propriétés. Par exemple, un matériau ayant besoin de texture aura une propriété lui permettant de lui attacher des textures. Si de plus il peut être éclairé par des lumières ponctuelles, on lui rajoute cette propriété.

3.3.6 Format G-Mesh

Pour afficher des modèles 3D dans Gear, il a fallu trouver un moyen de charger ces données. Pour ce faire, nous avons créé notre propre format de fichiers pour les modèles 3D, "gmesh". Ce format a l'avantage de ne contenir que ce qui nous intéresse (sommets et triangles du mesh), d'être simple et rapide à lire pour Gear. Mais il vient avec un coût, il a fallu écrire un script python pour blender qui puisse exporter un modèle 3d depuis le logiciel Blender dans le format G-Mesh.

3.4 BlackSeas

3.4.1 Physique

En plus des fonctionnalités graphiques, nous avions besoin d'un moteur physique. Nous l'avons construit à l'aide de Gear en utilisant le même système d'ECS.

Nous voulons plusieurs fonctionnalités : premièrement, nous voulons avoir plusieurs personnages, sur un terrain, ou sur un bateau, sans que ceux-ci ne passent à travers les uns des autres.
Ensuite, les objets doivent être affectés par les forces physiques naturelles, comme la gravité, mais aussi la flottaison des bateaux, le vent contre les voiles, et les forces de frottement. Les réponses à ces forces doivent être réalistes, par exemple, elles pourront mettre en rotation des objets.
Nous voulons pouvoir déplacer ces personnages, c'est-à-dire prendre en compte des inputs et appliquer des forces correspondantes.

a. Représentation des éléments physiques

Pour représenter les objets, nous attribuons plusieurs composants à une entité : un Transform, pour représenter la position et la rotation d'un objet dans l'espace, un RigidBody, pour représenter la vitesse d'un objet, sa masse, sa vitesse de rotation et son inertie, et un Collider, c'est à dire la forme de la zone de collision de l'objet.
Si l'objet ne doit pas se déplacer, comme le terrain, on le rend statique.
Ainsi, les forces ajoutées vont modifier la vitesse du RigidBody, puis la vitesse va affecter la position.

Nous avons choisi d'implémenter les forces de manière à pouvoir prendre en compte l'inertie d'un objet ou non. Par exemple, la force de gravité est la même pour tous les objets, mais la force du vent dépend de la masse du bateau.

La force de gravité est une force constante vers le bas que subissent tous les corps rigides.
Plongés dans l'eau, ces corps rigides subissent la poussée d'Archimède.

Une autre force à considérer est celle du déplacement des joueurs. Pour cela, si le joueur est sur un objet, on calcule sa vitesse relative à cet objet, et on la compare à la vitesse à laquelle le joueur voudrait aller, afin de déterminer la force de déplacement. Le joueur peut aussi sauter, ce qui ajoute une force verticale de déplacement.

Ces forces seront appliquées au joueur, et une force opposée sera appliquée à l'objet sur lequel il se tient, pour modéliser l'action/réaction.

b. Traitement d'une frame

Si dans notre simulation, il n'y a qu'un seul objet se déplaçant en ligne droite, alors à la fin d'une frame, on veut que cet objet se soit déplacé de vitesse x temps. Mais, comment faire lorsqu'il y a un autre objet sur le chemin ?
Il existe des algorithmes permettant de détecter les collisions, mais nous avons eu l'envie de se servir de ce projet pour essayer un algorithme de collision continue de notre conception.

Dans le cas de deux objets, l'idée de l'algorithme est de trouver le temps d'intersection entre ces deux objets, puis d'avancer ces objets à l'instant de la collision, de résoudre cette collision, c'est à dire de modifier leurs vitesses pour qu'ils se repoussent, puis d'avancer les objets à la fin de la frame.

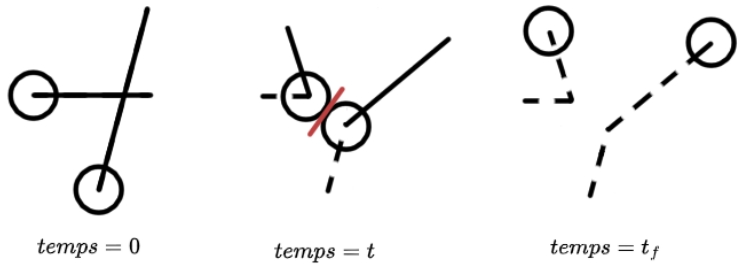


Figure 1 – Fonctionnement de l'algorithme avec deux objets

Si il y a plus de deux objets, alors on trouve la collision dont le moment d'intersection est le plus tôt, on déplace tous les objets à ce temps d'intersection, on résout, puis on recommence jusqu'à ce qu'il n'y ait plus de collisions avant la fin de la frame.

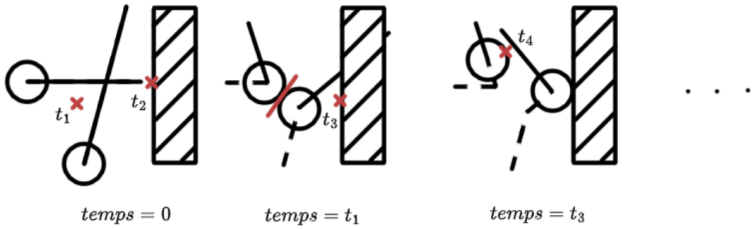


Figure 2 – Fonctionnement de l'algorithme avec plusieurs objets

```
Pour toutes les entités qui possèdent un Transform, RigidBody, et un Collider:
    Appliquer la gravité
    Appliquer la flottaison
    Pour tous les Joueurs, Appliquer les inputs

    temps = 0
    Tant que temps < temps_final:
        Trouver la première collision si elle existe
        Si il y a une collision, disons à t:
            Déplacer tous les objets à t
            Résoudre la collision
            temps = t
        Sinon:
            Déplacer tous les objets à temps_final
            temps = temps_final
```

Figure 3 – Pseudo-code de l'algorithme

c. Trouver le temps des différentes collisions

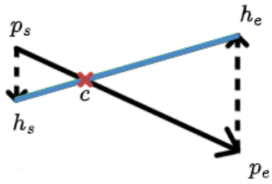
Dans notre simulation, nous avons plusieurs types d'objets qui peuvent entrer un collision les uns dans les autres. Par exemple, les joueurs sont des objets qui peuvent se déplacer, entrer en collision avec d'autre objets, comme les bateaux qui sont formés de triangles, ou avec le terrain qui est statique et généré procéduralement.

Pour ces différents besoins, nous avons séparé les détections de collisions en différents cas, dans lesquels nous modifions la représentation des objets pour faciliter les calculs.

i. Joueur / Terrain

Le terrain est généré procéduralement, donc pour représenter la forme de sa collision, nous utilisons une fonction qui permet d'obtenir la hauteur du terrain en un certain point, que nous appelons *height map*, et une autre fonction qui permet d'obtenir la normale au terrain en un certain point, appelée *normal map*.
Le joueur est représenté par un point situé au niveau de ses pieds, et c'est avec ce point que l'on teste la collision.

Avec p_s, p_e la position du joueur au début et à la fin de la frame t_f, h_s, h_e la hauteur du terrain à ces positions, alors comme la position et temps sont proportionnels à vitesse constante, on trouve le temps d'intersection t :



Avec le théorème de Thalès :

$$\frac{t_f}{t} = \frac{p_e - p_s}{c - p_s} = 1 - \frac{c - p_e}{c - p_s} = 1 - \frac{h_e - p_e}{h_s - p_s}$$

$$\text{D'où : } t = \frac{h_s - p_s}{h_s - p_s - h_e + p_e} t_f$$

Figure 4 – Calcul du temps d'intersection t entre un joueur et le terrain

ii. Joueur / Joueur

Pour les collisions joueurs/joueurs, on ne peut pas les représenter par des points pour calculer le temps d'intersection, sinon, ceux-ci peuvent être aussi proches que l'on veut. C'est pourquoi les joueurs sont considérés comme des cylindres aux yeux des autre joueurs. Nous voulons aussi que les joueurs restent droit, donc les cylindres sont toujours orienté selon l'axe vertical.

Pour tester la collision entre deux cylindres en mouvement, on se ramène d'abord au cas où un des cylindres est fixe et à l'origine, en soustrayant la vitesse et la position de l'un, a celles de l'autre.
Ensuite, comme les cylindres sont orientés selon le même axe, on peut séparer la collision en deux : une collision projetée selon l'axe vertical, une collision projetée selon le plan horizontal.

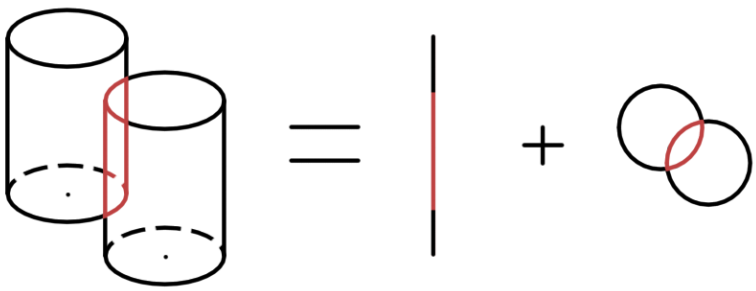


Figure 5 – Illustration du découpage de la collision en deux

Ainsi, nous devons trouver les intervalles d'intersections sur ces deux projections, et voir si ces intervalles se chevauchent. Selon si la première collision est verticale ou horizontale, on détermine si la collision sur le cylindre est sur le côté, au dessus ou en dessous.

Si le cylindre fixe est de taille h_1 et celui en mouvement est de taille h_2, alors trouver l'intervalle d'intersection entre les deux cylindres sur l'axe vertical revient à trouver l'intervalle d'intersection entre un point et l'intervalle $[-h_2, h_1]$. On peut trouver les bornes de cet intervalle temporel en divisant les bornes de l'intervalle spatial par la vitesse du point.

Si le cylindre fixe est de rayon r_1 et celui en mouvement est de rayon r_2, alors trouver l'intervalle d'intersection entre les deux cylindres sur le plan horizontal revient à trouver l'intervalle d'intersection entre un point et le disque centré à l'origine de rayon $r_1 + r_2$. C'est-à-dire, trouver t_1 et t_2 tels que $\forall t \in [t_1, t_2], ||p + tv|| \leq r_1 + r_2$, avec p et v la position et la vitesse du point. On peut mettre cette équation au carré et développer la norme pour obtenir un polynôme en t de degré 2, que l'on résout pour trouver les deux solutions, qui donnent les bornes de l'intervalle temporel.

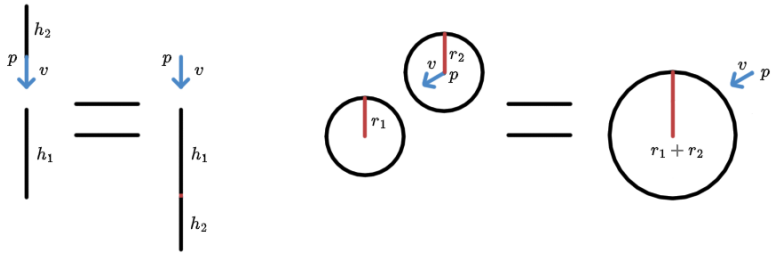


Figure 6 – Illustration de l'équivalence pour se ramener à un point contre un ensemble

iii. Joueur / Bateau

Un bateau, avec sa forme complexe, ne peut pas être représenté par une forme simple comme un cylindre ou une sphère. C'est pourquoi le bateau est représenté par un ensemble de triangles, qui forment le *mesh* de collision du bateau. Ainsi, pour trouver le temps de collision entre un joueur, que l'on représente cette fois-ci par une sphère, et un bateau, il faut tester la sphère contre chaque triangle et prendre la collision dont le temps d'intersection est le plus proche.

Pour cela, on veut ramener le test de sphère contre triangle à un test de point contre un ensemble de l'espace, comme dans le cas du cylindre. On peut alors découper cet ensemble en 7 : 3 sphères pour chaque sommet du triangle, 3 cylindres pour chaque côté du triangle, et 1 prisme pour la face du triangle. Le découpage est montré en figure 7.

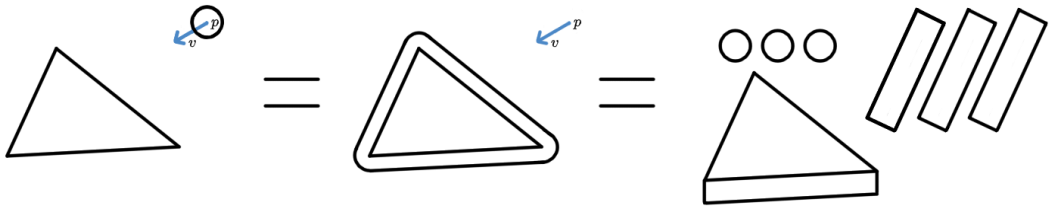


Figure 7 – Illustration de l'équivalence pour se ramener à un point contre un ensemble, puis de la découpe de celui-ci

Les tests de point contre sphère, cylindre ou prisme est semblable à ceux expliqués dans les collisions Joueur/Joueur, mais on peut les optimiser. Par exemple, on n'a pas besoin de détecter une collision sur les bords du prisme, car si il y en a une, alors il y a une collision avec un cylindre des côtés du triangle.

d. Résolution des collisions

Maintenant que l'on a trouvé le moment de la première collision et que l'on a déplacé tous les objets à cet instant, il faut modifier la vitesse des objets en collision pour simuler le fait qu'ils doivent se repousser.

Pour résoudre ces collisions, il faut appliquer une force égale et opposée aux deux objets en question. Pour trouver la formule de cette force, nous nous sommes référés aux articles de Chris Hecker[(3)]. Cette force va modifier la vitesse de l'objet si il peut se déplacer, et sa vitesse de rotation si il peut tourner.

3.4.2 Réseau

Black Seas est un jeu multijoueur en ligne. Il a donc fallu implémenter la communication réseau. Nous sommes parti sur une architecture avec une application dédié pour le serveur, Black Seas Server. Nous avons fait le choix de faire le serveur sur une application dédié, car cela permettait de bien séparer le code du serveur et du client, alors que si un des clients lançait le jeu en tant qu'hôte, et d'autres rejoignaient cette simulation en tant que client, il aurait fallu mettre tout le code du serveur et du client au même endroit.

Dans Gear, nous avons implémenté un template de système réseau. Ce template se construit à partir d'un ensemble de messages que peuvent s'envoyer le client et le serveur, et fournit les fonctionnalités de base pour envoyer ces messages via les protocoles TCP et/ou UDP. Le client ne peut envoyer des messages qu'au serveur, et le serveur a des fonctions pour envoyer des messages à un seul client, à tout les clients, ou à tout les clients sauf un.
Ensuite, dans Black Seas et Black Seas Serveur, nous avons utilisé les primitives réseau de Gear et construit les comportements souhaités pour le jeu.
Dans notre implémentation, le serveur est entièrement autoritaire. Les clients envoient leur commandes au serveur : se déplacer, activer les commandes d'un bateau, faire apparaitre un bateau, etc. Le serveur traite ces demandes, enlève celles qui ne sont pas autorisées (si les clients trichent), et renvoie aux clients l'état du monde (position des joueurs, des bateaux, état des bateaux). C'est donc le serveur qui s'occupe entièrement de simuler la physique.

Cette implémentation pose un soucis : il ya un délai entre le moment ou un joueur appuie sur une touche, et le moment ou ce joueur se voit bouger, équivalent au délai d'aller-retour des paquets sur le réseau (aussi appelé le 'ping'). Il existe des solutions à ces soucis, comme la prédiction client, l'interpolation / extrapolation, mais cela vient avec une implémentation complexe qui dépasse le temps et la portée du projet. De plus, comme nous jouons tout le temps sur un réseau local, le ping est suffisament faible pour ne pas être ressenti par les joueurs.

3.4.3 Océan

La simulation de l'eau (procédurale) est un sujet difficile dans le monde du jeu vidéo selon le degré de réalisme/d'effets que l'on veut ajouter à celle-ci. La majorité des jeux vidéos récents génèrent leur océan sur GPU (carte graphique) car le parallélisme permis par la carte graphique offre des performances indéniables concernant le temps de calcul. C'est le cas de Black Seas (la partie "client" de notre jeu) qui génère procéduralement l'entièreté de l'océan sur GPU. Cependant, si on ajoute un mode multijoueur avec un serveur autoritaire comme c'est le cas de Black Seas Server il est alors nécessaire de traduire la fonction de génération de l'océan de telle sorte à ce que l'on puisse l'exécuter sur processeur tout en garantissant le maintien des performances. En effet, notre manière d'envisager le mode multijoueur impose que ce soit le serveur qui calcule la flottaison des objets dans le monde, ceci via une fonction qui donnera la hauteur de l'océan en tout point (x, z) à chaque instant t :

$$\begin{aligned} h &: \mathbb{R}^3 \to \mathbb{R} \\ (x, z, t) &\mapsto h(x, z, t) \qquad (*) \end{aligned}$$

Le calcul de la hauteur de l'océan sur le serveur ne pose aucun problème dans le cas de modèles d'océan simples dont

la fonction d'océan est déjà sous la forme $(*)$ sur GPU. Cependant, pour des modèles plus complexes qui nécessitent une parallélisation de la fonction sur GPU, ce calcul de hauteur devient un vrai casse-tête.

a. Le modèle de Gerstner : la houle trochoïdale

Le modèle de Gerstner (appelé également *houle trochoïdale*) a été introduit par **Franz-Joseph von Gerstner** il y a plus de deux siècles et est depuis très longtemps utilisé dans le monde du jeu vidéo pour la simulation de l'océan car il s'agit d'un modèle simple, peu coûteux autant au niveau mémoire qu'en temps de calcul et ce dernier est facilement modulable pour donner à l'océan la forme que l'on souhaite. Sa simplicité en fait également un parfait candidat pour le mode multijoueur de Black Seas.

Le principe de ce modèle est simple à concevoir : chaque vague est modélisée par des fonctions sinusoïdales qui dépendent de plusieurs paramètres : la longueur d'onde λ de la vague ou plus précisément le nombre d'onde $k = \frac{2\pi}{\lambda}$, l'amplitude a, une direction du vent normalisée dans le plan (x, z) que l'on note d ainsi qu'un coefficient de raideur de la vague (plus ce coefficient est élevé, plus la vague est raide et "pointue") que l'on note s. Le déplacement d'un point de l'espace en un temps donné pour un océan composé d'une seule vague est donc la suivante ($c = \sqrt{\frac{9.81}{k}}$ correspond à la vitesse de la vague) :

$$f : \mathbb{R}^3 \to \mathbb{R}^3$$
$$(p, t) \mapsto \begin{cases} d_x * a * \frac{s}{k} * \cos(k * (< p, d > + c * t)) \\ \sin(k * (< p, d > + c * t) \\ d_z * a * \frac{s}{k} * \cos(k * (< p, d > + c * t)) \end{cases}$$

On peut noter que contrairement à une fonction sinusoïdale simple, ce modèle engendre un déplacement aussi bien *horizontal* que *vertical* des points. En effet, chaque point de la surface de la vague se déplace autour d'un cercle et non juste verticalement :

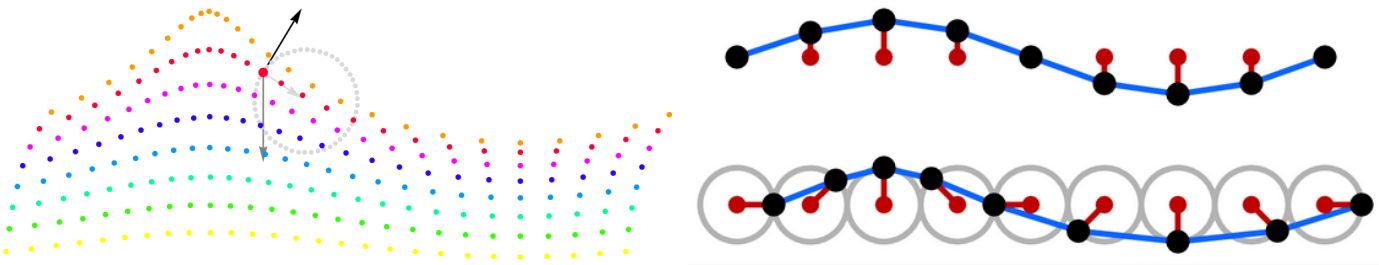
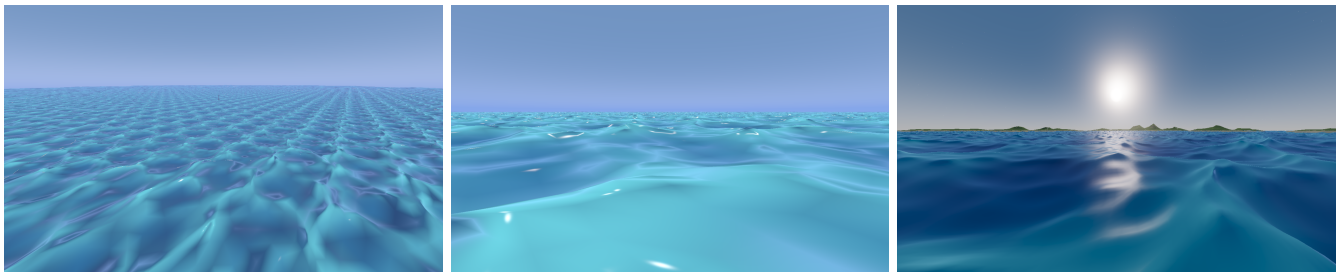


Figure 8 – Fonctionnement de la houle trochoïdale. Source : *wikipedia.com*

En théorie, plus on ajoute de vagues, plus on obtient un océan réaliste mais cela coûtera rapidement cher en terme de performances car on doit pour chaque point de l'océan calculer l'addition de beaucoup de vagues : c'est pourquoi la plupart des jeux ne vont jamais au-delà d'une vingtaine de vagues avec cette méthode.
Ci-dessous quelques illustration de l'océan avec ce modèle :



Toutefois, il est possible de modéliser l'océan par un très grand nombre de vagues en calculant la hauteur de l'océan non pas en un point donné mais pour un ensemble de points (une grille de points, pour être plus exact). L'addition de ce très grand nombre de fonctions sinusoïdales permet d'être effectué à l'aide de l'algorithme de Fast Fourier Transform. C'est ce que nous allons voir prochainement.

b. Une version avancée de Gerstner via l'utilisation de l'algorithme FFT (Fast Fourier Transform)

i) Explications

Jerry Tessendorf, un professeur à l'universié de Clemston aux États-Unis a rédigé un papier très célèbre et cité en 2001 concernant la simulation d'océan. Ce dernier décrit l'approche de la FFT pour simuler un océan réaliste. Le fonctionnement de l'algorithme n'est pas détaillé ici. L'idée consiste à modéliser la hauteur de l'océan comme une somme de sinusoïdes complexes avec des amplitudes qui dépendent du temps :

$$h \colon \mathbb{R}^3 \to \mathbb{R}$$
$$(x, z, t) \mapsto \sum_k \tilde{h}(k, t) exp(ik \cdot x)$$

Où $k = (\frac{2\pi n}{L_x}, \frac{2\pi m}{L_z})$ avec L_x, L_z qui correspondent aux dimensions de la grille de l'océan et n, m qui correspond aux indices de la vague courante : $-\frac{N}{2} \leq n, m \leq \frac{N}{2}$ donc le nombre de vagues total est de N^2. Les coefficients de Fourier contiennent une partie aléatoire. Les détails de cette formule sont très bien expliqués dans le papier de *Tessendorf*[4], référencé en fin de rapport si vous souhaitez avoir plus de détails.

ii) Implémentation

La FFT est certes un algorithme rapide lorsqu'on prend un nombre de vague élevé (sa complexité étant en $O(n^2 \log^2(n))$ où n est le nombre de vagues. Cependant, son implémentation sur CPU sous forme itérative est assez peu efficace et réduit énormément les performances lorsqu'on souhaite avoir un nombre de vagues suffisamment élevé pour atteindre un certain réalisme. Une solution serait donc de paralléliser l'algorithme pour pouvoir l'exécuter sur GPU. C'est une approche décrite dans un des papiers référencés en fin de chapitre[5][6].

L'idée de l'implémentation de la FFT sur GPU est d'utiliser des *compute shaders* (programmes permettant de faire du calcul sur GPU) dont le fonctionnement n'est pas détaillé ici. Chaque shader prend en entrée une texture et en sortie une autre texture, ce qui en fait l'outil idéal pour faire tourner l'algorithme. Dans notre version finale, l'algorithme tourne à l'aide d'une pipeline constituée de 4 compute shaders illustrée ci-dessous :

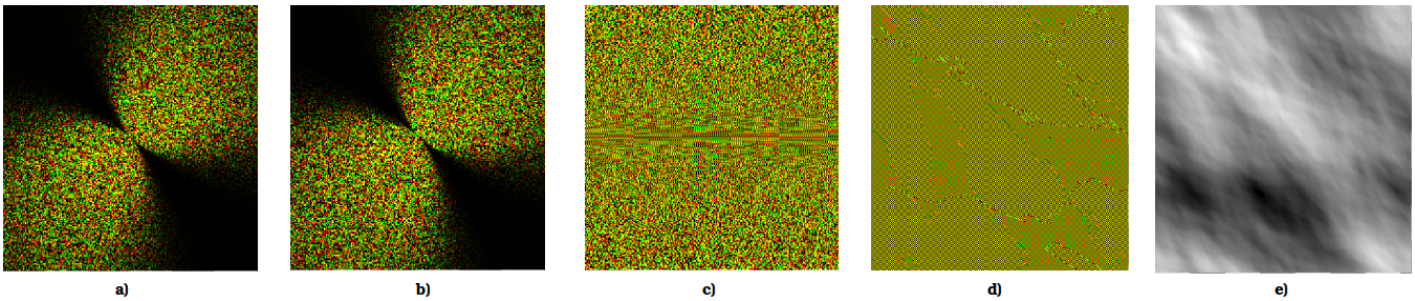
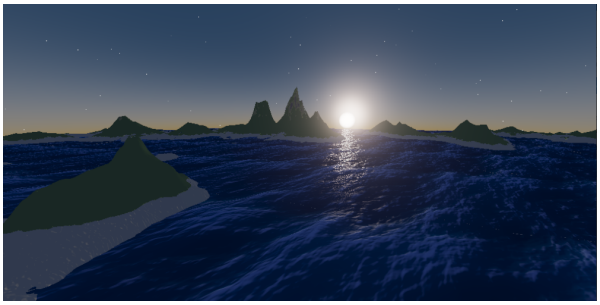
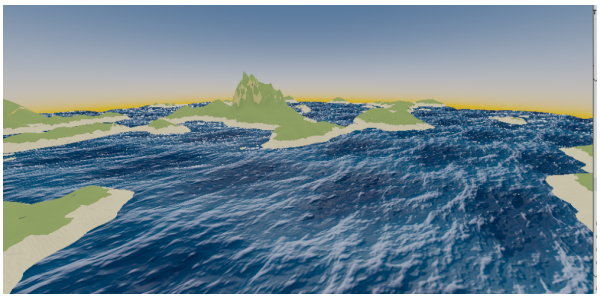


Figure 9 – Illustration de la pipeline de compute shaders pour le calcul de FFT. **a)** Texture des coefficients de Fourier en t=0 ($h_k(0)$) **b)** Texture des coefficients de Fourier à l'instant t ($h(k, t)$) **c)** FFT 1D sur les colonnes de la texture b) **d)** FFT 1D sur les lignes de la texture c) **e)** Inversion de la FFT (texture de la hauteur de l'océan)

Quelques illustrations de l'océan généré par FFT sont présentes ci-dessous. On peut voir que l'océan apparaît beaucoup plus "texturé" en raison du grand nombre de vagues. Le modèle d'illumination utilisé est également différent pour cet océan (il s'agit du modèle de Ward, tandis que l'océan précédent était illuminé avec le modèle de Blinn-Phong).

3.4.4 Terrain

La génération du terrain se fait à l'aide du graphe de shaders de Gear. Un premier noeud calcule la carte des hauteurs. Cette texture est ensuite récupérée dans un second noeud afin de calculer les normales du terrain. La méthode utilisée pour la génération de cette carte est inspirée d'une vidéo de Inigo Quilez[(9)]. On obtient alors une somme d'octaves de bruit que l'on passe à une fonction pour obtenir une répartition des hauteurs plus adaptée à notre terrain.

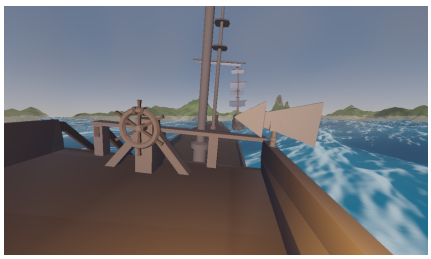
Le terrain est généré procéduralement. Cependant, cette génération dépend d'une graine, qui est commune aux joueurs et au serveur. De cette façon, chaque joueur va générer le même terrain, bien qu'il soit généré aléatoirement.
Enfin, nous avons créé un système qui va créer des portions de terrains, appelées "chunks", et les positionner autour du joueur afin d'afficher ce dernier. Cette méthode permet de varier le niveau de détails selon la distance du joueur au chunk, et ansi optimiser l'affichage du terrain en concentrant les détails à proximité du joueur.

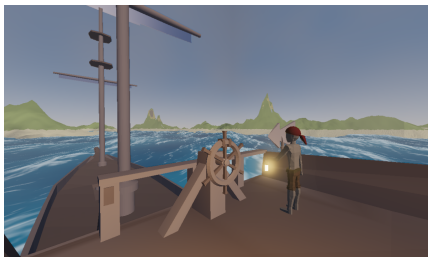
3.4.5 Skybox

La skybox est un plan auquel nous n'appliquons pas la matrice modèle. Le plan est situé de telle sorte à avoir la profondeur maximale pour que tous les autres objets soient dessinés devant. La direction dans laquelle le joueur regarde est alors récupérée avec les inverses des matrices de vue et de projection.
Le soleil ainsi que les étoiles possèdent une couleur avec des composantes supérieures à 1 pour leur appliquer du bloom.

4 Résultats

Dans l'ensemble, nous sommes extrêmement satisfaits du résultat. Même si certains détails sont manquants, avec du recul nous avons réussi à faire un jeu vidéo from scratch, sans moteur de jeu, répondant à des gros défis de programmation tels que la physique, le multijoueur ou la génération procédurale. Lorsque le jeu était jouable et que l'on s'approchait de la fin, il nous arrivait de nous perdre dans l'immersion du jeu lors des tests et de s'amuser ensemble. Voici quelques clichés qui montrent le résultat final :





5 Discussion

Finalement, que retenons-nous de ce projet ? Evidemment, comme tout projet aussi ambitieux avec des deadlines aussi courtes, il y a énormément de détails que l'on aurait aimé améliorer. Que ce soit dans les collisions, ou la synchronisation

de l'océan entre client et serveur, il y a encore quelques soucis. Cependant, il ne faut pas laisser ces quelques problèmes masquer tout ce que nous avons réussi à faire. Nous avons énormément appris dans ce projet, dans des domaines qui nous intéressaient tous et nous sommes très fiers de ce que nous avons rendu.

6 Références

6.1 Physique et Collisions

Collisions :

(2) Real Time Collision Detection, by Christer Ericson. (2004)

Rigid Body Dynamics :

(3) https ://chrishecker.com/Rigid_Body_Dynamics, by Chris Hecker (1996/1997)

6.2 Ocean

Articles de recherche :

(4) "Simulating Ocean Water", by Tessendorf, Jerry (2004).

(5) "Realtime GPGPU FFT Ocean Water Simulation", by Flynn-Jorin Flügge(2017).

(6) "Simulation of Ocean Surface", by Andrey Ufimtsev, Bachelor's Project (2015).

Pages web et vidéos :

(7) Wikipedia (pages : Fast Fourier Transform, Cooley-Tukey FFT algorithm)

(8) Nvidia, GPU Gems 1 et 2

(9) `https://youtu.be/BFld4EBO2RE?t=180` de Inigo Quilez