
Rapport de projet

Simuler un océan

Master IMAGINE

**BÈS Jean-Baptiste, COMBOT Evan, JEAN Louis,
KERBAUL Loïc**

Encadrante : FARAJ Noura

20 mai 2024



**UNIVERSITÉ DE
MONTPELLIER**

Remerciements

Nous tenons à exprimer notre profonde gratitude à M^{me} Faraj pour son encadrement et ses conseils précieux tout au long de notre projet.

Nous souhaitons également remercier toutes les personnes qui nous ont apporté des retours constructifs et des conseils avisés tout au long du développement. Leur soutien et leurs contributions ont été d'une grande aide dans l'avancement de notre projet.

Table des matières

1	Introduction	4
1.1	Glossaire	5
1.2	Structure du rapport	7
2	Organisation, méthodes et outils	8
2.1	Organisation	8
2.1.1	Répartition	8
2.1.2	Objectifs du projet	8
2.1.3	GitHub	9
2.2	Méthodes et outils	10
2.2.1	C++ et OpenGL	10
3	Préambule	11
3.1	Définition d'une vague	11
3.2	Les différentes catégories pour modéliser les vagues océaniques	12
3.2.1	Les ondes de surface	12
3.2.2	Une somme d'ondes de surface	13
3.3	Etat de l'art des méthodes intégrables en informatique graphique	13
3.3.1	Les domaines d'applications	14
4	Réalisations	15
4.1	Modèles de simulation de vagues	15
4.1.1	Modèle sinusoïdal	15
4.1.2	Modèle de Gerstner	20
4.1.3	Quelques améliorations	25
4.1.4	Modèle basé sur un spectre océanographique	27
4.2	Système de flottabilité	32
4.2.1	Premier essai de simulation de flottabilité	32
4.2.2	Utilisation de la force de flottabilité	33
4.2.3	Application d'une force de restitution	34
4.3	Éclairage et rendu	35
4.3.1	Modèle d'éclairage de Phong	35
4.3.2	Skybox	37
4.3.3	Effet de Fresnel	38
4.3.4	Combinaison des différents effets d'éclairage	39
4.3.5	Physically Based Rendering (PBR)	41
4.4	Gestion du son	43
4.5	Interface utilisateur	44
4.5.1	DearImGui	44
4.5.2	Prise en main de l'application	44
5	Problèmes rencontrés	46
5.1	Calcul des normales	46

5.2	Le modèle spectral	46
6	Bilan du projet	47
6.1	Autocritique	47
6.2	Enseignements tirés	47
6.3	Perspectives	47
7	Conclusion	49
8	Annexes	50
8.1	Shaders du modèle sinusoïdal	50
8.1.1	Vertex shader du modèle sinusoïdal simple	50
8.1.2	Vertex shader de la somme des sinusoïdes	51
8.2	Shaders du modèle de Gerstner	54
8.2.1	Vertex shader du modèle de Gerstner simple	54
8.2.2	Vertex shader de la somme des vagues de Gerstner	54
8.3	Shaders d'éclairage / de rendu	59
8.3.1	Fragment shader de l'éclairage de Phong	59
8.3.2	Fragment shader de la skybox	60
8.3.3	Fragment shader du PBR	61
9	Bibliographie	64

Table des figures

2.1	Diagramme de Gantt	9
3.1	Illustration d'une vague	11
3.2	Exemple d'une onde de surface - Onde de Rayleigh	12
4.1	Premier rendu (filaire) d'une vague sinusoïdale	17
4.2	Premier rendu (filaire) d'une somme de vagues sinusoïdales, ici avec 8 vagues	19
4.3	Premier rendu d'une vague de Gerstner	22
4.4	Premier rendu (filaire) d'une somme de vagues de Gerstner, ici avec 5 vagues	24
4.5	Simulation avec modèle sinusoïdal composé de 30 vagues, avec et sans FBM	26
4.6	Opération <i>butterfly</i> à deux entrées	28
4.7	Opération <i>butterfly</i> à quatre entrées	28
4.8	Exemple d'un rendu basé sur un spectre de Phillips	32
4.9	Sphères qui flottent sur une vague sinusoïdale	35
4.10	Exemple de l'illumination de Phong sur une somme de sinusoïdes	36
4.11	Skybox utilisée dans notre simulation	37
4.12	Rendu final de la somme de sinusoïdes	40
4.13	Rendu final de la somme de vagues de Gerstner	40
4.14	Exemple de rendu utilisant le PBR	42
4.15	Exemple de l'interface ImGui	44

CHAPITRE 1. Introduction

Certains films emblématiques tels que *Titanic* et *Waterworld*, ainsi que des jeux vidéo populaires comme *Sea of Thieves* et *Assassin's Creed Black Flag*, ont captivé les spectateurs avec leurs représentations saisissantes d'océans. Derrière ces reproductions impressionnantes se cachent des modèles et des méthodes complexes de simulation des vagues et des océans. Ce projet de TER (Travail d'Étude et de Recherche) vise à explorer et à implémenter ces techniques de simulation.

L'objectif principal de notre projet est de mettre en œuvre plusieurs types de simulations de vagues basées sur des modèles mathématiques, afin de modéliser de manière réaliste le comportement des océans. Nous avons choisi d'utiliser OpenGL pour développer ces simulations en raison de sa flexibilité et de sa puissance dans le rendu graphique en temps réel.

Nous sommes tous les quatre étudiants en Master Informatique parcours IMAGINE et notre formation nous a permis d'acquérir une solide expérience en OpenGL. Ce choix technologique nous a donc paru évident pour la réalisation de ce projet.

1.1 Glossaire

Shader : Programme informatique utilisé dans le domaine de la programmation graphique pour contrôler l'apparence visuelle des objets rendus à l'écran.

Fragment shader : C'est un programme qui calcule la couleur finale de chaque pixel dans une scène 3D.

Vertex shader : C'est un programme qui transforme les attributs des sommets d'un objet 3D vers l'espace d'affichage.

Compute Shader : Il est utilisé dans le domaine du calcul parallèle sur GPU. Le compute shader est conçu pour effectuer des calculs généraux sur les données sans nécessairement générer de pixels ou de triangles pour le rendu à l'écran.

Maillage : C'est une structure de données composée de points (sommets), de lignes (arêtes) et de faces (polygones) qui définissent la géométrie d'un objet tridimensionnel.

Normale : Les normales sont essentielles dans le rendu 3D car elles influencent la manière dont la lumière interagit avec les surfaces, ce qui est crucial pour calculer l'éclairage et les ombres.

FFT (Fast Fourier Transform) : Une technique algorithmique utilisée pour calculer efficacement la transformation de Fourier d'une fonction discrète. Elle est largement utilisée dans le traitement du signal et l'analyse des données pour convertir des signaux temporels en leur représentation fréquentielle.

IFFT (Inverse Fast Fourier Transform) : C'est la transformée inverse de Fourier rapide. La fonction permet de passer du domaine fréquentiel au domaine temporel.

Spectre océanique : Un modèle mathématique pour décrire la distribution de l'énergie des vagues océaniques en fonction de leur fréquence et de leur direction.

Spectre de Jonswap : Un modèle spectral couramment utilisé pour représenter la distribution de l'énergie des vagues océaniques en fonction de leur fréquence et de leur direction. Il est souvent utilisé dans les simulations océanographiques pour générer des vagues réalistes en tenant compte des conditions météorologiques et océanographiques. Ce modèle est notamment adapté pour simuler l'échelle de Beaufort.

Échelle de Beaufort : Une échelle de classification empirique utilisée pour estimer la force du vent et ses effets sur la surface de la mer et sur la terre. Cette échelle, inventée par l'amiral britannique Sir Francis Beaufort au début du 19^{ème} siècle, va de 0 (calme plat) à 12 (ouragan). Chaque valeur de l'échelle est associée à une description de l'état de la mer et des effets observés sur terre, tel que la vitesse du vent, la hauteur des vagues, et les dommages aux structures et aux arbres. L'échelle de Beaufort est utilisée dans la navigation maritime, la météorologie, et d'autres domaines où la force du vent est importante.

Spectre de Phillips : Un modèle spectral qui décrit la distribution de l'énergie des vagues

océaniques en fonction de leur fréquence et de leur direction. Il est basé sur les travaux du météorologue britannique John Miles Read Phillips et est utilisé pour modéliser les vagues marines en tenant compte de leur origine sous l'influence du vent.

Bruit de Perlin : Le bruit de Perlin est une méthode de génération de bruit cohérent développée par Ken Perlin. Il est largement utilisé dans la synthèse d'images pour créer des textures naturelles, des terrains et d'autres phénomènes aléatoires.

Diamant carré : Le diamant carré est un algorithme utilisé pour la génération de terrains et de formes géométriques en utilisant une approche basée sur des grilles. Il est particulièrement efficace pour la génération de terrains en 2D.

Simplex : Le bruit simplex est une amélioration du bruit de Perlin, développée également par Ken Perlin. Il offre une meilleure qualité de bruit, une meilleure efficacité computationnelle et une meilleure adaptation à des dimensions supérieures, ce qui le rend populaire pour la génération de terrains en 2D et 3D, ainsi que pour d'autres applications nécessitant des bruits aléatoires cohérents.

1.2 Structure du rapport

Ce rapport est structuré comme suit :

- **Organisation, méthodes et outils** : Cette section détaille notre organisation de travail, ainsi que les méthodes et outils utilisés tout au long du projet, notamment OpenGL et GitHub.
- **Préambule** : Nous y définissons ce qu'est une vague, les différentes catégories pour modéliser les vagues océaniques, et les ondes de surface.
- **État de l'art** : Un aperçu des différentes approches existantes et des technologies utilisées pour la simulation des vagues.
- **Réalisations** : Une description détaillée des modèles de simulation que nous avons implémentés, ainsi que des fonctionnalités supplémentaires telles que l'interface utilisateur avec ImGui et la gestion du son avec miniaudio.
- **Problèmes rencontrés** : Une discussion sur les défis et les obstacles rencontrés au cours du projet.
- **Bilan du projet** : Une réflexion sur le projet, incluant une autocritique, les enseignements tirés, et les perspectives futures.
- **Conclusion** : Un résumé des travaux réalisés et des résultats obtenus.
- **Annexes** : Des informations supplémentaires, du code source, et de la documentation technique.

La simulation d'océans est un domaine fascinant qui combine les aspects théoriques de la physique des fluides avec des techniques avancées de rendu graphique. À travers ce projet, nous avons exploré plusieurs méthodes pour modéliser les vagues, tels que les modèles sinusoïdaux, de Gerstner et spectraux utilisant la transformée de Fourier rapide. Nos résultats montrent des visualisations réalistes et interactives d'océans, ouvrant la voie à de futures améliorations et applications dans divers domaines.

Ce rapport détaillera nos découvertes et réalisations, offrant un aperçu complet de notre démarche et des techniques utilisées pour atteindre nos objectifs.

CHAPITRE 2. Organisation, méthodes et outils

2.1 Organisation

2.1.1 Répartition

Notre équipe est composée de quatre membres, chacun ayant des rôles et des responsabilités spécifiques :

- **Jean-Baptiste Bes** : Responsable de la modélisation des vagues.
- **Evan Combot** : Développeur principal OpenGL.
- **Louis Jean** : Intégration de l'interface utilisateur avec ImGui et rendu.
- **Loïc Kerbaul** : Gestion du son et du système de flottabilité.

Nous avons organisé des réunions hebdomadaires pour suivre l'avancement du projet et résoudre les problèmes rencontrés.

2.1.2 Objectifs du projet

Les objectifs principaux du projet étaient de mettre en place trois modèles de mouvement des vagues. Tout d'abord, nous voulions implémenter un modèle de vagues sinusoïdales, ensuite notre second objectif était d'implémenter un modèles des vagues de Gerstner, autrement appelé houle trochoïdale. Enfin, le dernier modèle est un modèle spectral couplé à l'utilisation d'une transformée de Fourier rapide.

Dans le but d'être efficace dans la réalisation de ce projet nous avons mis en place un diagramme de Gantt.

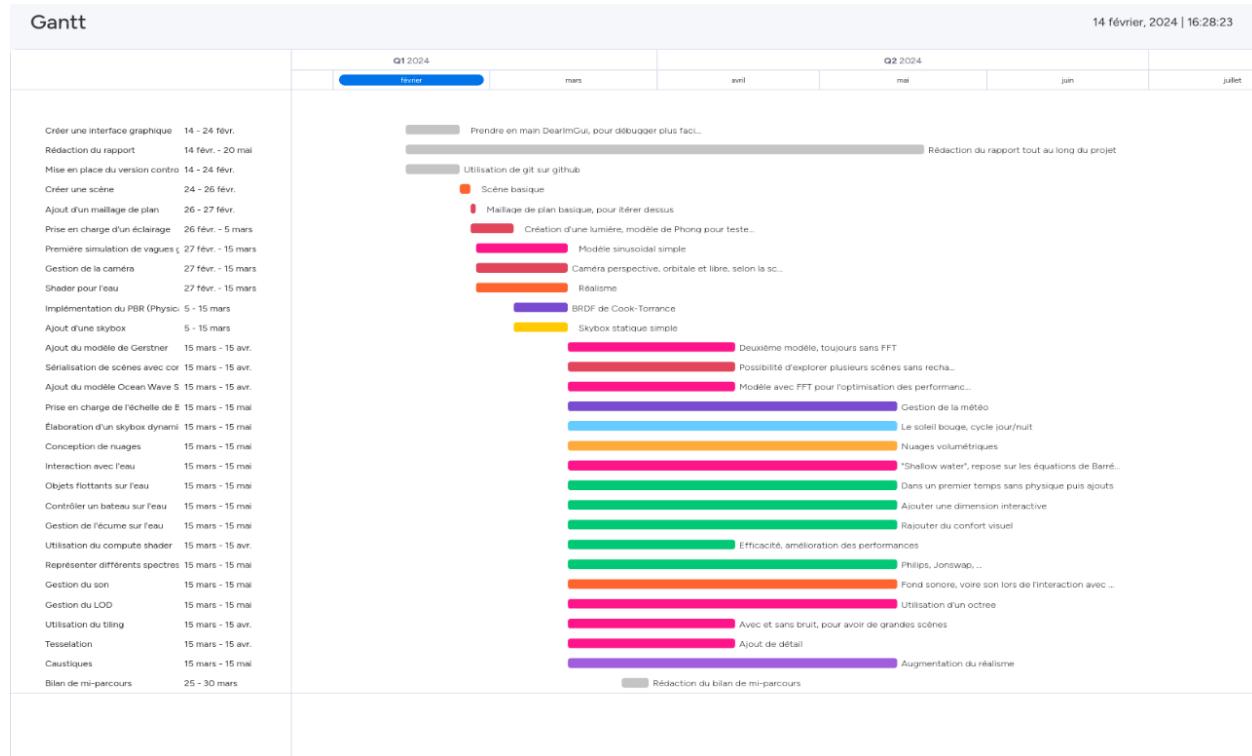


Figure 2.1: Diagramme de Gantt

2.1.3 GitHub

Pour gérer efficacement notre projet, nous avons choisi d'utiliser GitHub. Cet outil nous a permis de fusionner nos travaux, de suivre les changements et de collaborer facilement. Chaque membre de l'équipe pouvait ainsi apporter ses contributions aux différentes parties du projet, tout en gardant une version centralisée et bien organisée du code. GitHub nous a également aidés à gérer les problèmes et les tâches en créant des tickets pour les fonctionnalités à implémenter et les bugs à corriger. De plus, les outils de revue de code de la plateforme ont été essentiels pour assurer la qualité et la cohérence de notre code tout au long du projet.

Lien du dépôt GitHub de notre projet : <https://github.com/louis-jean0/TER-OceanGL.git>.

2.2 Méthodes et outils

2.2.1 C++ et OpenGL

Pour ce projet, nous avons choisi de travailler avec le langage de programmation C++, et nous avons choisi de ne pas utiliser un moteur existant tel que Unity ou Unreal Engine. À la place, nous avons décidé de programmer notre propre application en utilisant C++ avec OpenGL. OpenGL est une API (Interface de Programmation d'Applications) graphique multi-plateforme et open-source. Elle est largement utilisée pour le rendu 2D et 3D dans les applications informatiques. Cette API permet aux développeurs de créer des applications graphiques interactives, offrant une grande flexibilité et un contrôle détaillé sur le rendu graphique.

CHAPITRE 3. Préambule

3.1 Définition d'une vague

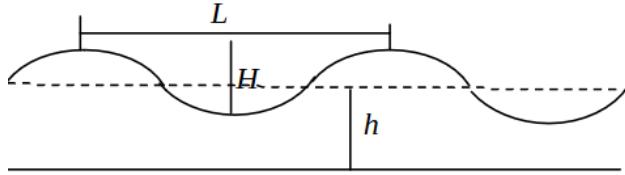


Figure 3.1: Illustration d'une vague

Une vague peut être définie comme une perturbation périodique de la surface de l'eau. Elle se produit généralement en réponse à des forces extérieures telles que le vent, les courants marins ou les mouvements sismiques. Les vagues peuvent varier en taille, en forme et en intensité, et sont souvent classées en différentes catégories en fonction de leur origine et de leur comportement.

Voici les caractéristiques d'une vague :

- T , la période qui représente le temps qui sépare le passage de deux crêtes successives.
- L , la longueur d'onde qui représente la distance qui sépare le passage de deux crêtes successives.
- f , la fréquence est le nombre de périodes par seconde, est égale à $\frac{1}{T}$ et est exprimée en Hertz (Hz).
- k , le nombre d'ondes de la vague égal à $\frac{2\pi}{L}$.
- g , la gravité terrestre, en principe égale à 9.81 m/s^2 .
- C , la célérité de la vague qui représente la vitesse moyenne de propagation des crêtes, qui est égale à $\frac{L}{T}$.
- H , la hauteur de la vague qui représente la différence de niveau entre un creux et une crête successive.
- A , l'amplitude qui est égale à $\frac{H}{2}$.
- h , la profondeur qui représente la distance entre la surface de l'eau et le sol.
- ω , la pulsation de la vague, soit le nombre de radians effectués par seconde, égale à $\frac{2\pi}{T}$.

3.2 Les différentes catégories pour modéliser les vagues océaniques

Les théories des vagues océaniques sont classées en trois catégories.

- **Équations détaillées de la mécanique des fluides :** Introduites par Navier-Stokes en 1820, la résolution classique de ces équations par la méthode des éléments finis présente des défis, notamment en termes de stabilité numérique et de temps de calcul. Cela a conduit au développement de théories alternatives plus simples à mettre en œuvre.
- **Ondes de surface :** Cette théorie décrit la surface des vagues comme un mouvement ondulatoire régulier, aussi appelé la houle. La représentation mathématique d'une onde de surface se fait par une équation paramétrique.

La superposition d'ondes de surface, fondée sur l'hypothèse de linéarisation des mouvements, permet de modéliser la génération de vagues océaniques sous l'effet du vent en combinant plusieurs ondes. Cette approche est largement utilisée dans l'étude des phénomènes océanographiques.”

Dans notre cas, les modèles basés sur les ondes de surface sont préférables car nous avons seulement besoin de déplacer les sommets d'un maillage pour obtenir la forme de l'onde et nous n'avons pas besoin de simuler les conditions réelles d'un fluide. L'utilisation des équations de Navier-Stokes dans un cadre discret nécessiterait d'utiliser des méthodes basées sur des particules.

3.2.1 Les ondes de surface

Les ondes de surface sont des perturbations qui se propagent le long de la surface d'un milieu, tel que l'interface entre l'air et l'eau ou entre deux liquides de densités différentes. Dans le contexte océanographique, les ondes de surface sont principalement associées aux mouvements de l'eau à la surface de l'océan, provoqués par des forces externes telles que le vent, les marées ou les séismes. Ces ondes peuvent avoir des caractéristiques variées, notamment leur hauteur, leur période et leur longueur d'onde, qui dépendent des conditions environnementales et des forces qui les ont générées. Les ondes de surface jouent un rôle crucial dans de nombreux phénomènes océaniques, tels que la formation des vagues, les courants marins côtiers et les interactions avec les structures côtières.

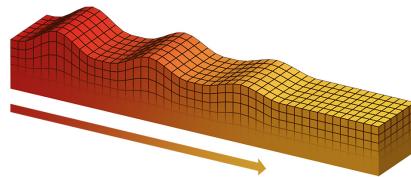


Figure 3.2: Exemple d'une onde de surface - Onde de Rayleigh

3.2.2 Une somme d'ondes de surface

Lorsque plusieurs ondes de surface se propagent simultanément à la surface de l'eau, elles peuvent interagir et se superposer pour former une somme d'ondes de surface. Cette somme d'ondes résulte de la combinaison des ondes individuelles, chacune ayant sa propre amplitude, période et direction de propagation. La formation de cette somme d'ondes peut donner lieu à des phénomènes intéressants tels que l'interférence constructive et destructive, où les crêtes et les creux des différentes ondes se renforcent ou s'annulent mutuellement. Ces interactions peuvent entraîner des variations complexes de la hauteur, de la période et de la direction des vagues, ce qui peut avoir des implications importantes pour la navigation maritime, la sécurité côtière et d'autres activités en mer. La compréhension de la formation et du comportement des sommes d'ondes de surface est donc essentielle pour modéliser avec précision les conditions océaniques et prévoir les phénomènes météorologiques marins.

3.3 Etat de l'art des méthodes intégrables en informatique graphique

Dans le domaine de la modélisation des vagues océaniques, de nombreuses approches ont été développées pour représenter différents types de vagues, chacune avec ses propres caractéristiques et applications spécifiques. Voici un aperçu des principales approches :

- **Vagues sinusoïdales** : Les vagues sinusoïdales sont un modèle simple et couramment utilisé pour représenter les vagues océaniques. Elles sont caractérisées par une forme régulière et périodique, avec une amplitude, une période et une longueur d'onde constantes. Bien que ce modèle soit limité dans sa capacité à capturer les caractéristiques réalistes des vagues, il reste utile pour des applications telles que l'enseignement et la visualisation.
- **Vagues de Gerstner** : Les vagues de Gerstner sont un modèle plus avancé qui prend en compte les effets de non-linéarité et de dispersion dans la propagation des vagues. Elles permettent de représenter des vagues plus réalistes avec des profils de crêtes et de creux asymétriques. Ce modèle a été créé par le baron von Gerstner en 1802 et est largement utilisé dans les jeux vidéo et les simulations maritimes pour sa capacité à produire des vagues plus réalistes.
- **Vagues aléatoires** : Les vagues aléatoires sont des modèles utilisés pour simuler des conditions de vagues réalistes, en tenant compte de la variabilité spatiale et temporelle des vagues océaniques. Ces modèles peuvent être basés sur des processus stochastiques tels que l'utilisation de bruit pour générer des profils de vagues avec une apparence aléatoire. Ils sont largement utilisés dans les applications de recherche scientifique et d'ingénierie océanique pour étudier les phénomènes de houle et de tsunami. Des exemples de modèles de vagues aléatoires incluent le bruit de Perlin, le diamant carré et le simplex. En outre, pour obtenir des détails plus fins et des variations de hauteur réalistes, on utilise souvent le "displacement mapping" pour déplacer les sommets de la surface en 3D, ce qui permet de représenter les caractéristiques détaillées des vagues.

- **Vagues basées sur un modèle spectral :** Une approche couramment utilisée pour générer des vagues réalistes est de les synthétiser à partir de modèles spectraux. Cette méthode permet de générer des profils de vagues en combinant différentes fréquences et amplitudes de manière contrôlée, souvent à l'aide de la transformée de Fourier rapide (FFT). Elle est utilisée dans les simulations numériques et les visualisations interactives pour produire des animations de vagues réalistes. Cette technique est souvent utilisée en combinaison avec des modèles de spectres de vagues, tels que le spectre de JONSWAP (Joint North Sea Wave Project) et le spectre de Phillips, qui fournissent des distributions d'énergie en fonction de la fréquence pour générer des vagues avec des caractéristiques réalistes en termes de hauteur, de période et de direction.
- **Modèles basés sur la théorie des ondes non linéaires :** Ces modèles prennent en compte les effets de non-linéarité dans la propagation des vagues, ce qui permet de représenter des phénomènes tels que la formation de vagues solitaires et les interactions entre vagues. Ils sont utilisés dans les applications de recherche avancée pour étudier les phénomènes complexes de la dynamique des vagues, comme les tsunamis et les vagues scélérites.

Ces différentes approches offrent des compromis entre réalisme et complexité, et sont utilisées dans divers contextes, de la recherche scientifique à l'industrie du divertissement. Le choix du modèle dépend souvent des besoins spécifiques de la simulation et des ressources disponibles pour sa mise en œuvre.

3.3.1 Les domaines d'applications

Le rendu physiquement réaliste d'un océan en 3D est crucial dans de nombreux domaines, notamment la simulation maritime, l'animation pour les films et les jeux vidéo, la réalité virtuelle, et la recherche scientifique. Dans le domaine de la simulation maritime, la capacité à modéliser avec précision le comportement de l'océan est essentielle pour l'entraînement des marins, la conception de navires et les opérations en mer. Pour l'industrie du divertissement, un océan réaliste est nécessaire pour créer des effets spéciaux convaincants dans les films et les jeux vidéo. De même, dans le domaine de la réalité virtuelle, un océan réaliste peut améliorer l'immersion des utilisateurs dans des environnements virtuels. Enfin, dans la recherche scientifique, la modélisation réaliste de l'océan est utilisée pour étudier divers phénomènes océanographiques, tels que les vagues, les courants marins et les interactions avec les structures côtières.

CHAPITRE 4. Réalisations

4.1 Modèles de simulation de vagues

Dans notre projet, nous utilisons un plan comme maillage de base pour représenter la surface de l'océan. Le plan agit comme une surface sur laquelle nous appliquons des déformations pour simuler les vagues et les mouvements de l'eau. Cette approche nous permet de modéliser l'océan de manière efficace en utilisant un maillage simple et en appliquant des techniques avancées pour rendre les effets de l'eau de manière réaliste et interactive.

4.1.1 Modèle sinusoïdal

Notre implémentation de ce modèle repose sur l'article [1]. C'est une excellente ressource pour commencer à étudier l'implémentation des vagues en informatique graphique.

4.1.1.1 Modèle sinusoïdal simple

Le modèle sinusoïdal est l'une des approches les plus simples pour simuler des vagues océaniques. Ce modèle repose sur l'utilisation de fonctions sinusoïdales pour représenter les oscillations régulières de la surface de l'eau. Cette méthode est particulièrement efficace pour des simulations où la précision et le réalisme ne sont pas des priorités absolues, mais où la simplicité et la rapidité de calcul sont essentielles.

Principe de base

Le modèle sinusoïdal repose sur l'équation de la vague sinusoïdale, qui peut être exprimée mathématiquement comme suit :

$$H(x, z, t) = A \sin(k(D \cdot P) + \phi_t)$$

où :

- $H(x, z, t)$ est la hauteur de la vague à la position (x, z) et au temps t ,
- A est l'amplitude de la vague, c'est-à-dire la hauteur maximale de la vague,
- k est le nombre d'onde, défini par $k = \frac{2\pi}{\lambda}$, où λ est la longueur d'onde,
- D est le vecteur directionnel de la vague,
- P est le vecteur position (x, z) sur la surface de l'eau,
- ϕ_t est la phase de la vague, dépendant du temps et de la vitesse de propagation de la vague, définie par $\phi_t = S \times k \times t$, avec S la vitesse de propagation de la vague.

Implémentation en OpenGL

Pour implémenter le modèle sinusoïdal en OpenGL, nous avons utilisé des shaders pour calculer la hauteur des vagues en temps réel. Le vertex shader est responsable de la déformation des sommets du maillage de plan en fonction de l'équation sinusoïdale. Les détails de ce shader sont présentés en Annexe 8.1.1.

Calcul des normales

Les normales de la surface sont essentielles pour un éclairage réaliste. Pour le modèle sinusoïdal, les normales peuvent être calculées en utilisant les dérivées partielles de la surface de la vague par rapport aux coordonnées x et z . Nous utilisons une méthode analytique pour trouver la binormale et la tangente, avant de calculer un produit vectoriel entre ces deux données pour trouver la normale.

La binormale s'écrit comme étant $\mathbf{B}(x, z, t) = (1, \frac{\partial}{\partial x} H(x, z, t), 0)$ avec

$$\frac{\partial}{\partial x} H(x, z, t) = AkD_x \cos(k(D \cdot P) + \phi_t)$$

La tangente est calculée par $\mathbf{T}(x, z, t) = (0, \frac{\partial}{\partial z} H(x, z, t), 1)$ avec

$$\frac{\partial}{\partial z} H(x, z, t) = AkD_z \cos(k(D \cdot P) + \phi_t)$$

Enfin, la normale est donnée par $\mathbf{N}(x, z, t) = \mathbf{B}(x, z, t) \times \mathbf{T}(x, z, t)$.

Les normales sont ensuite normalisées pour garantir des vecteurs unitaires. Cette méthode est intégrée directement dans le vertex shader.

Résultats et performances

Le modèle sinusoïdal singulier est une méthode simple et efficace pour générer des vagues. En utilisant une seule fonction sinusoïdale, il est possible de représenter les oscillations régulières de la surface de l'eau. Cette méthode est particulièrement adaptée pour des simulations où la rapidité de calcul et la simplicité sont des priorités. Le modèle sinusoïdal singulier permet de simuler des vagues avec une faible complexité computationnelle, ce qui le rend très performant même sur des systèmes avec des ressources limitées.

Cependant, le réalisme de ce modèle est limité. Les vagues générées par une seule fonction sinusoïdale manquent de diversité et de détails. Les crêtes et les creux des vagues sont uniformes et réguliers, ce qui ne reflète pas fidèlement la complexité des vagues océaniques réelles.

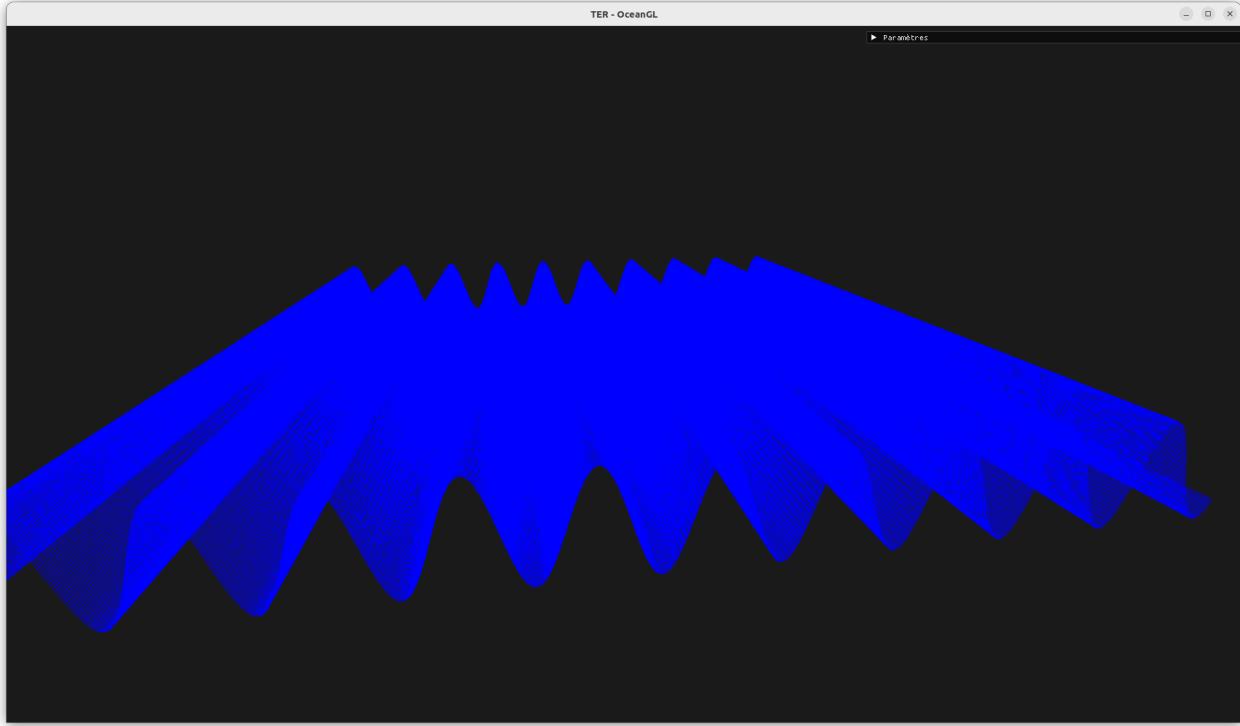


Figure 4.1: Premier rendu (filaire) d'une vague sinusoïdale

4.1.1.2 Somme de sinusoïdales

Pour obtenir des vagues plus réalistes, nous pouvons combiner plusieurs ondes sinusoïdales avec différentes amplitudes, longueurs d'onde, fréquences et directions. Cette approche permet de simuler des phénomènes plus complexes et de capturer une plus grande variété de comportements de vagues.

Principe de base

La somme de sinusoïdales consiste à additionner plusieurs ondes sinusoïdales individuelles. Mathématiquement, cela peut être exprimé par :

$$H(x, z, t) = \sum_{i=1}^N A_i \sin(k_i(D_i \cdot P) + \phi_{i,t})$$

où :

- $H(x, z, t)$ est la hauteur du plan déformé par les vagues à la position (x, z) et au temps t ,
- N est le nombre de vagues composantes,
- A_i est l'amplitude de la i -ème vague,

- k_i est le nombre d'onde de la i -ème vague, défini par $k_i = \frac{2\pi}{\lambda_i}$,
- D_i est le vecteur directionnel de la i -ème vague,
- P est le vecteur position (x, z) sur la surface de l'eau,
- $\phi_{i,t}$ est la phase de la i -ème vague, dépendant du temps et de la vitesse de propagation de la vague, définie par $\phi_t = S \times k \times t$, avec S la vitesse de propagation de la vague.

En combinant ces vagues, nous pouvons créer des motifs d'ondes plus complexes et réalistes.

Dans notre implémentation, chaque vague rajoutée voit ses attributs alloués aléatoirement.

Implémentation en OpenGL

Pour implémenter cette somme de sinusoïdales en OpenGL, nous adaptons le vertex shader pour calculer la hauteur des vagues en additionnant plusieurs termes sinusoïdaux. Les détails de ce nouveau shader sont visibles en Annexe 8.1.2.

Calcul des normales

Pour une seule vague sinusoïdale, les normales peuvent être calculées analytiquement grâce à la simplicité de l'équation de la vague. Cependant, pour la somme de sinusoïdales, le calcul analytique des normales devient plus complexe en raison de la superposition de multiples ondes avec différentes amplitudes, fréquences et directions. Nous n'avons pas réussi à implémenter ces normales de manière analytique.

Méthode des différences finies

Pour surmonter cette complexité, nous utilisons la méthode des différences finies pour calculer approximativement les normales de la surface. Cette méthode consiste à estimer les dérivées partielles de la surface en prenant de petites différences dans les directions x et z .

En pratique, les dérivées partielles de la hauteur de la vague $H(x, z, t)$ par rapport à x et z sont approximées par les différences finies, en prenant les différences de hauteur des points voisins. Soit ϵ un petit incrément, alors :

La dérivée partielle par rapport à x est approximée par $\frac{\partial H}{\partial x} \approx \frac{H(x+\epsilon, z, t) - H(x, z, t)}{\epsilon}$.

La dérivée partielle par rapport à z est approximée par $\frac{\partial H}{\partial z} \approx \frac{H(x, z+\epsilon, t) - H(x, z, t)}{\epsilon}$.

Avec ces approximations, on peut déduire les vecteurs binormale et tangente.

La binormale est alors $\mathbf{B}(x, z, t) = \left(1, \frac{H(x+\epsilon, z, t) - H(x, z, t)}{\epsilon}, 0\right)$.

La tangente est $\mathbf{T}(x, z, t) = \left(0, \frac{H(x, z+\epsilon, t) - H(x, z, t)}{\epsilon}, 1\right)$.

Enfin, la normale est calculée en prenant le produit vectoriel de ces vecteurs, comme pour la sinusoïdale simple.

Les normales sont ensuite normalisées pour garantir des vecteurs unitaires. Cette méthode est intégrée directement dans le vertex shader pour fournir des normales précises même pour des surfaces de vagues complexes générées par la somme de sinusoïdales.

Résultats et performances

La somme de sinusoïdales permet de générer des vagues plus complexes et réalistes par rapport à l'utilisation d'une seule sinusoïde. Cette approche est toujours relativement simple à implémenter et à calculer, mais elle offre une flexibilité accrue pour simuler divers types de vagues en fonction des paramètres choisis.

Cependant, l'ajout de multiples vagues sinusoïdales augmente légèrement la complexité des calculs, ce qui peut affecter les performances, notamment sur des systèmes moins puissants. Malgré cela, cette méthode reste efficace pour des simulations en temps réel avec un bon compromis entre réalisme et performance.

En conclusion, la somme de sinusoïdales constitue une amélioration notable par rapport à une seule vague sinusoïdale, permettant de capturer une plus grande variété de comportements et de fournir une base solide pour des modèles plus sophistiqués.

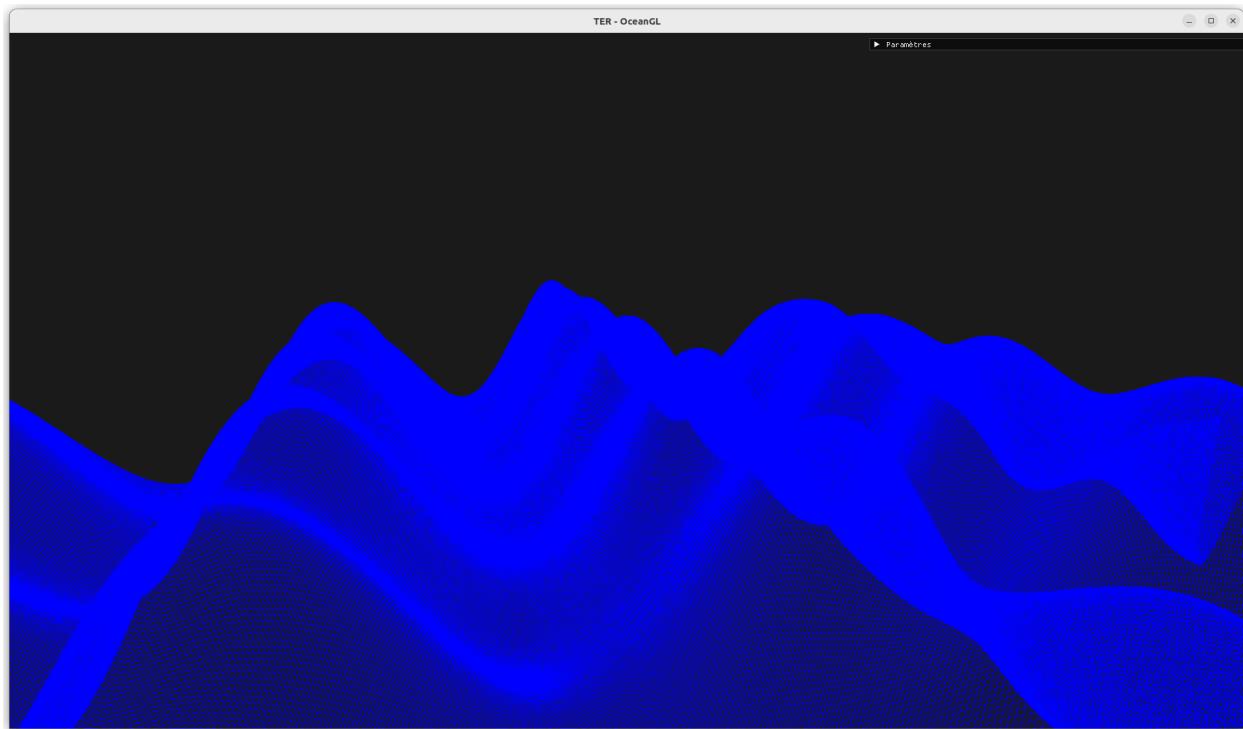


Figure 4.2: Premier rendu (filaire) d'une somme de vagues sinusoïdales, ici avec 8 vagues

4.1.2 Modèle de Gerstner

Là encore, nous nous sommes aidés de l'article [1] pour réaliser ce modèle.

4.1.2.1 Modèle de Gerstner simple

Le modèle de Gerstner est une méthode plus avancée pour simuler des vagues océaniques, offrant une représentation plus réaliste des vagues que le modèle sinusoïdal. Ce modèle utilise des ondes sinusoïdales, mais introduit également un déplacement horizontal des points de la surface de l'eau, ce qui permet de créer des vagues avec des crêtes plus nettes et des creux plus prononcés.

Principe de base

Le modèle de Gerstner repose sur une équation paramétrique qui décrit à la fois le déplacement vertical et horizontal des points de la surface de l'eau. L'équation paramétrique d'une vague de Gerstner peut être exprimée comme suit :

$$\begin{cases} x(t) = x_0 + Q \cdot A \cdot D_x \cos(\omega \cdot (D \cdot P) + \phi_t) \\ y(t) = A \sin(\omega \cdot (D \cdot P) + \phi_t) \\ z(t) = z_0 + Q \cdot A \cdot D_z \cos(\omega \cdot (D \cdot P) + \phi_t) \end{cases}$$

où :

- $x(t), y(t), z(t)$ sont les coordonnées des points sur la surface de l'eau au temps t ,
- x_0, z_0 sont les positions initiales des points sur la surface de l'eau,
- A est l'amplitude de la vague,
- $\omega = \sqrt{\frac{g \cdot 2\pi}{L}}$ est la fréquence angulaire de la vague, avec L la longueur d'onde,
- $\phi_t = S \cdot \frac{2\pi}{L} \cdot t$ est la phase temporelle de la vague, avec S la vitesse de propagation de la vague,
- $Q = \frac{\text{Steepness}}{\omega \cdot A}$ est le facteur de steepness (pente),
- D est la direction de propagation de la vague,
- P est la position initiale.

Le facteur de steepness Q contrôle la pente des vagues, avec des valeurs plus élevées produisant des crêtes plus nettes et des creux plus prononcés.

Dans notre implémentation, chaque vague rajoutée voit ses attributs alloués aléatoirement.

Implémentation en OpenGL

Pour implémenter le modèle de Gerstner en OpenGL, nous devons ajuster le vertex shader pour calculer à la fois les déplacements verticaux et horizontaux des points de la surface de l'eau en fonction de l'équation de Gerstner. Voici en Annexe 8.2.1 le vertex shader correspondant.

Calcul des normales

Dans notre implémentation du modèle de Gerstner, nous avons utilisé une méthode analytique pour calculer les normales de la surface de l'eau. Contrairement à la méthode des différences finies, cette approche utilise les dérivées analytiques des équations de vague de Gerstner pour obtenir des normales précises.

Pour calculer les normales analytiquement, nous devons d'abord calculer les vecteurs binormale et tangente. Les normales sont ensuite obtenues en prenant le produit vectoriel de ces deux vecteurs.

La binormale \mathbf{B} est calculée en prenant la dérivée partielle de la position par rapport à x :

$$\mathbf{B}(x, z, t) = \left(1, \frac{\partial}{\partial x} H(x, z, t), 0\right)$$

où $\frac{\partial H}{\partial x}$ est la dérivée partielle de la hauteur de la vague par rapport à x .

La tangente \mathbf{T} est calculée en prenant la dérivée partielle de la position par rapport à z :

$$\mathbf{T}(x, z, t) = \left(0, \frac{\partial}{\partial z} H(x, z, t), 1\right)$$

où $\frac{\partial H}{\partial z}$ est la dérivée partielle de la hauteur de la vague par rapport à z .

Les calculs spécifiques des dérivées partielles dans notre shader sont les suivants :

$$\begin{aligned}\frac{\partial H}{\partial x} &= Q \cdot A \cdot \omega \cdot D_x \cdot \sin(k \cdot (D \cdot P) + \phi_t) \\ \frac{\partial H}{\partial z} &= Q \cdot A \cdot \omega \cdot D_z \cdot \sin(k \cdot (D \cdot P) + \phi_t)\end{aligned}$$

Enfin, la normale \mathbf{N} est obtenue en prenant le produit vectoriel des vecteurs binormale et tangente :

$$\mathbf{N}(x, z, t) = \mathbf{B}(x, z, t) \times \mathbf{T}(x, z, t)$$

Cette méthode analytique garantit des normales précises et est plus efficace que la méthode des différences finies. Elle est intégrée directement dans le vertex shader pour fournir des normales précises même pour des surfaces de vagues complexes générées par le modèle de Gerstner.

Résultats et performances

Le modèle de Gerstner offre une simulation des vagues beaucoup plus réaliste que le modèle sinusoïdal en raison de son traitement des déplacements horizontaux et verticaux. Cela permet de créer des vagues avec des crêtes et des creux plus prononcés, qui se rapprochent davantage du comportement réel des vagues océaniques.

Cependant, ce modèle est également plus complexe à calculer, ce qui peut avoir un impact sur les performances, surtout lorsqu'un grand nombre de vagues est simulé en temps réel. Malgré cette complexité accrue, le modèle de Gerstner reste efficace pour des simulations en temps réel et constitue un bon compromis entre réalisme et performance.

En conclusion, le modèle de Gerstner est une avancée significative par rapport au modèle sinusoïdal, permettant de capturer des détails plus fins et des comportements de vagues plus réalistes. Cette méthode est particulièrement utile dans les applications où le réalisme visuel est crucial, comme dans les films, les jeux vidéo, et les simulations d'ingénierie.

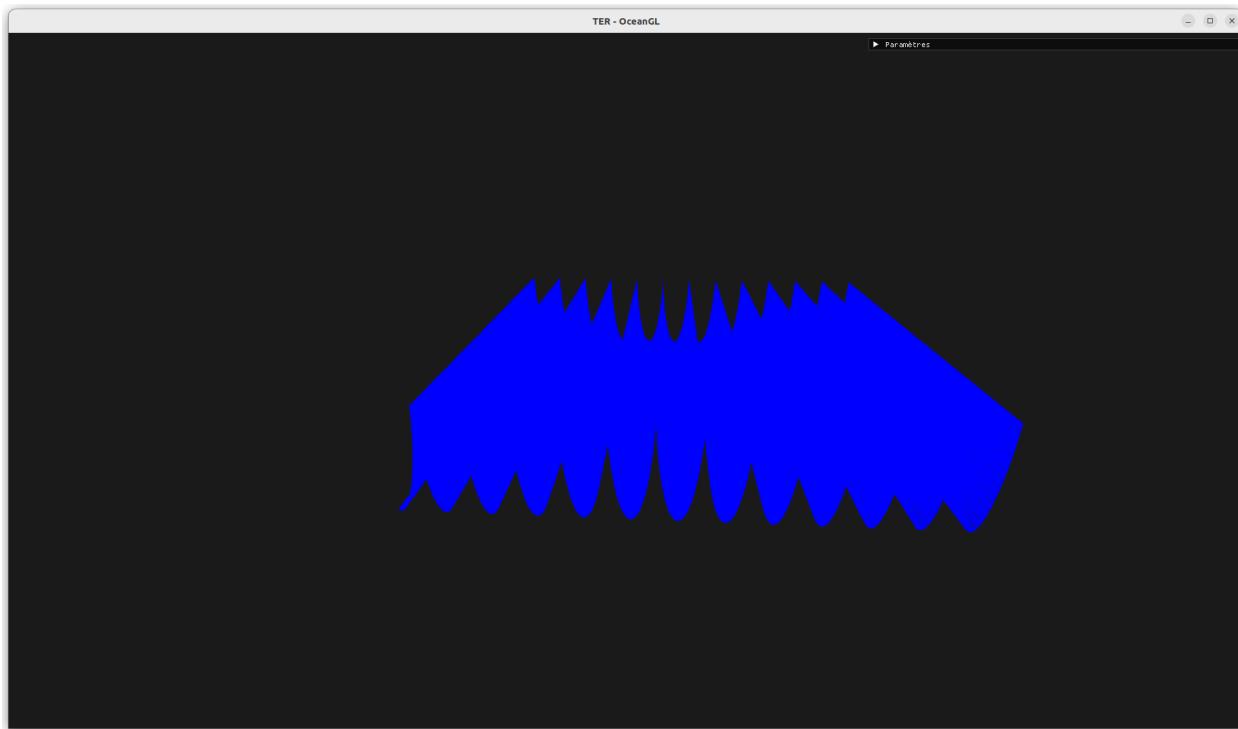


Figure 4.3: Premier rendu d'une vague de Gerstner

4.1.2.2 Somme de vagues de Gerstner

Pour obtenir une simulation de vagues encore plus réaliste, nous pouvons combiner plusieurs vagues de Gerstner avec différentes amplitudes, longueurs d'onde, fréquences et directions. Cette approche permet de simuler des phénomènes plus complexes et de capturer une plus grande variété de comportements de vagues.

Principe de base

La somme de vagues de Gerstner consiste à additionner plusieurs ondes de Gerstner individuelles. Mathématiquement, cela peut être exprimé par :

$$\begin{cases} x(t) &= x_0 + \sum_{i=1}^N Q_i \cdot A_i \cdot D_{x_i} \cos(\omega_i \cdot (D_i \cdot P) + \phi_{t_i}) \\ y(t) &= \sum_{i=1}^N A_i \sin(\omega_i \cdot (D_i \cdot P) + \phi_{t_i}) \\ z(t) &= z_0 + \sum_{i=1}^N Q_i \cdot A_i \cdot D_{z_i} \cos(\omega_i \cdot (D_i \cdot P) + \phi_{t_i}) \end{cases}$$

où :

- N est le nombre de vagues composantes,
- A_i est l'amplitude de la i -ème vague,
- $\omega_i = \sqrt{\frac{g \cdot 2\pi}{L_i}}$ est la fréquence angulaire de la i -ème vague, avec L la longueur d'onde,
- $\phi_{t_i} = S \cdot \frac{2\pi}{L_i} \cdot t$ est la phase temporelle de la i -ème vague, avec S la vitesse de propagation de la vague,
- $Q_i = \frac{\text{Steepness}}{\omega_i \cdot A_i}$ est le facteur de steepness (pente) de la i -ème vague,
- D_i est la direction de propagation de la i -ème vague,
- P est la position initiale.

En combinant ces vagues, nous pouvons créer des motifs d'ondes plus complexes et réalistes.

Implémentation en OpenGL

Pour implémenter cette somme de vagues de Gerstner en OpenGL, nous adaptons le vertex shader pour calculer la hauteur des vagues en additionnant plusieurs termes de Gerstner. Voir en Annexe 8.2.2 le vertex shader correspondant.

Calcul des normales

Tout comme pour une seule vague de Gerstner, les normales pour une somme de vagues de Gerstner sont calculées analytiquement en utilisant les dérivées des équations de vague de Gerstner. Cependant, dans ce cas, nous devons sommer les contributions de chaque vague pour obtenir les dérivées totales :

$$\begin{aligned} \frac{\partial H}{\partial x} &= \sum_{i=1}^N Q_i \cdot A_i \cdot \omega_i \cdot D_{x_i} \cdot \sin(\omega_i \cdot (D_i \cdot P) + \phi_{t_i}) \\ \frac{\partial H}{\partial z} &= \sum_{i=1}^N Q_i \cdot A_i \cdot \omega_i \cdot D_{z_i} \cdot \sin(k_i \cdot (D_i \cdot P) + \phi_{t_i}) \end{aligned}$$

Ensuite, nous calculons les vecteurs binormale et tangente pour chaque point de la surface en utilisant ces dérivées totales et prenons leur produit vectoriel pour obtenir les normales, comme déjà vu plus haut :

$$\begin{aligned}\mathbf{B}(x, z, t) &= \left(1, \frac{\partial}{\partial x} H(x, z, t), 0\right) \\ \mathbf{T}(x, z, t) &= \left(0, \frac{\partial}{\partial z} H(x, z, t), 1\right) \\ \mathbf{N}(x, z, t) &= \mathbf{B}(x, z, t) \times \mathbf{T}(x, z, t)\end{aligned}$$

Cette méthode analytique garantit des normales précises pour les surfaces de vagues complexes générées par la somme de vagues de Gerstner.

Résultats et performances

La somme de vagues de Gerstner permet de générer des vagues plus complexes et réalistes par rapport à l'utilisation d'une seule vague de Gerstner. Cette approche offre une flexibilité accrue pour simuler divers types de vagues en fonction des paramètres choisis.

Cette méthode reste efficace pour des simulations en temps réel avec un bon compromis entre réalisme et performance.

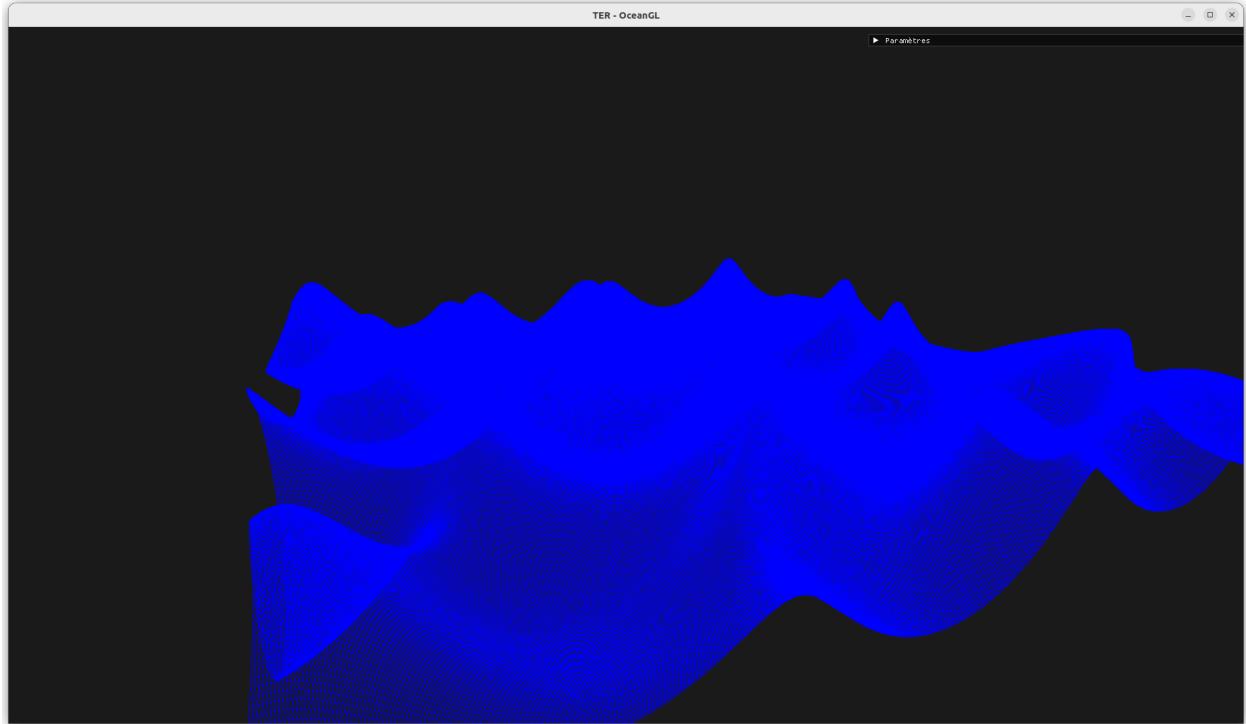


Figure 4.4: Premier rendu (filaire) d'une somme de vagues de Gerstner, ici avec 5 vagues

4.1.3 Quelques améliorations

Pour améliorer le réalisme de notre simulation de vagues océaniques, nous avons exploré et implémenté plusieurs techniques avancées. Deux de ces techniques sont le mouvement brownien fractionnaire (FBM) et le domain warping (DW).

4.1.3.1 Le mouvement brownien fractionnaire (FBM)

Le mouvement brownien fractionnaire (FBM) est une technique permettant de générer des vagues avec une certaine logique et complexité. Le FBM commence avec une vague initiale ayant une fréquence et une amplitude de base, ainsi qu'une direction aléatoire. Chaque vague est calculée et ajoutée à une somme de sinusoïdes, après quoi sa dérivée est également calculée.

Principe de base

Le FBM repose sur une série de vagues, où chaque nouvelle vague a une fréquence multipliée par un facteur strictement supérieur à 1, et une amplitude multipliée par un coefficient strictement inférieur à 1. Cela signifie que plus on ajoute de vagues, plus leur fréquence augmente, mais leur amplitude diminue, réduisant ainsi leur effet visuel global. Mathématiquement, cela peut être exprimé par :

$$f_{FBM}(x, z) = \sum_{i=0}^N G^i f(\lambda^i x, \lambda^i z)$$

où :

- N est le nombre de niveaux de récursivité,
- G est le facteur de gain qui contrôle l'amplitude des vagues à chaque niveau,
- λ est le facteur d'échelle qui contrôle la fréquence des vagues à chaque niveau,
- f est la fonction de base (sinusoïdale ou de Gerstner dans notre cas).

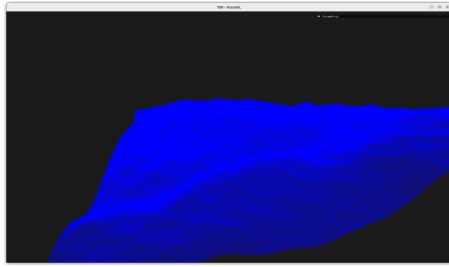
À chaque niveau de récursivité, la contribution des vagues diminue en fonction du facteur de gain G . Cela permet de limiter l'amplitude totale tout en ajoutant des détails plus fins à la surface de l'eau. De cette manière, on peut ajouter énormément de vagues sans que cela ne cause un désastre visuellement.

Implémentation

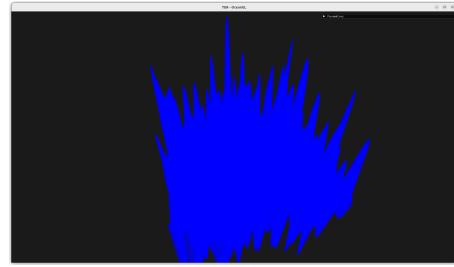
Pour intégrer le FBM dans nos modèles de vagues sinusoïdales et de Gerstner, nous avons modifié les shaders correspondants pour inclure la somme des fonctions de base à différents niveaux de récursivité. Voici en Annexe 8.1.2 ou Annexe 8.2.2 les shaders l'implémentant.

Résultats et performances

L'application du FBM aux modèles de vagues permet de générer des vagues avec des détails et une variabilité accrues, rendant la simulation plus réaliste. De plus, cette fonctionnalité n'augmente que très peu le coût computationnel.



(a) Avec FBM



(b) Sans FBM

Figure 4.5: Simulation avec modèle sinusoïdal composé de 30 vagues, avec et sans FBM

4.1.3.2 Le domain warping (DW)

Le domain warping (DW) est une technique utilisée pour créer l'effet de vagues qui se repoussent entre elles, ajoutant ainsi un niveau supplémentaire de réalisme à la simulation.

Principe de base

Le DW déforme les coordonnées d'entrée (x, z) en utilisant une fonction de perturbation avant d'appliquer la fonction de base f . Mathématiquement, cela peut être exprimé par :

$$f_{DW}(x, z) = f(p(x, z))$$

où :

- $p(x, z)$ est la fonction de perturbation qui déforme les coordonnées d'entrée,
- f est la fonction de base (sinusoïdale ou de Gerstner dans notre cas).

Implémentation

Dans notre cas, la dérivée partielle de la vague précédente est ajoutée à un accumulateur, et cette valeur est directement intégrée dans la formule de la vague, ce qui permet de créer un effet de repoussement entre les vagues. Voici en Annexe 8.1.2 ou Annexe 8.2.2 les shaders implémentant le DW.

Résultats et performances

L'application du DW aux modèles de vagues permet de créer des vagues avec des motifs plus complexes et variés, augmentant ainsi le réalisme de la simulation. Comme pour le FBM, cette technique n'augmente quasiment pas le coût computationnel, et les gains en termes de réalisme visuel sont convaincants.

4.1.3.3 Conclusion des améliorations

En conclusion, les techniques de FBM et de DW représentent des améliorations significatives pour la simulation des vagues océaniques, en introduisant des détails et des variations qui augmentent le réalisme visuel. Ces techniques, très légèrement plus coûteuses en termes de calcul, sont essentielles pour les applications où la qualité de la simulation est cruciale.

4.1.4 Modèle basé sur un spectre océanographique

Ce modèle, basé sur un spectre océanographique utilise la transformée de Fourier rapide (FFT). La transformée de Fourier rapide est un algorithme efficace pour calculer la transformée de Fourier Discrète (DFT) et son inverse. La DFT convertit une séquence de valeurs en une somme de fonctions sinusoïdales de différentes fréquences, facilitant ainsi l'analyse de la fréquence des signaux discrets. La FFT optimise ce processus, réduisant le temps de calcul de $O(n^2)$ de la DFT à $O(n \log n)$, où N est le nombre de points dans la séquence.

Avantages de la FFT dans un compute shader

En plus de réduire la complexité de calcul, l'implémentation de ce modèle dans un compute shader tirerait parti de la structure parallèle des GPU, soulageant ainsi la charge de calcul du CPU et améliorant ainsi les performances de rendu en temps réel. De plus, l'algorithme de la FFT est adapté à être utilisé dans des structures parallèles, grâce à son approche *diviser pour régner*.

Algorithme Cooley-Tukey

Le calcul de la FFT est basé sur l'algorithme Cooley-Tukey. Cet algorithme repose sur le principe de décomposition récursive d'une DFT de taille N en deux DFT de taille $N/2$. La DFT d'un signal discret f avec N valeurs est définie par l'équation suivante :

$$F[k] = \sum_{n=0}^{N-1} f[n] e^{-\frac{2\pi i k n}{N}}, n = 0, 1, \dots, N - 1$$

La DFT pour $N = 4$ peut être définie par :

$$F[k] = \sum_{n=0}^3 (-i)^{kn} f[n]$$

avec la simplification suivante :

$$e^{-i\pi/2} = -i$$

On peut maintenant écrire cette DFT sous la forme d'une matrice :

$$\begin{bmatrix} f[0] + f[1] + f[2] + f[3] \\ f[0] - if[1] - f[2] + if[3] \\ f[0] - f[1] + f[2] - f[3] \\ f[0] + if[1] - f[2] - if[3] \end{bmatrix}$$

L'algorithme Cooley-Tukey exploite la possibilité d'économiser des opérations en réarrangeant les termes dans la matrice :

$$\begin{bmatrix} (f[0] + f[2]) + (f[1] + f[3]) \\ (f[0] - f[2]) - i(f[1] - f[3]) \\ (f[0] + f[2]) - (f[1] + f[3]) \\ (f[0] - f[2]) + i(f[1] - f[3]) \end{bmatrix}$$

En précalculant la matrice de cette manière, nous réduisons de moitié le nombre de calculs. De plus, les termes pairs et impairs peuvent être calculés séparément.

Afin de calculer ces termes, l'algorithme Cooley-Tukey utilise ce qui s'appelle une *butterfly* opération. Cette opération prend deux nombres complexes en entrée ainsi qu'un nombre complexe fixe (dans cet exemple, ce nombre est égal à 1), et elle génère deux nombres complexes en sortie.

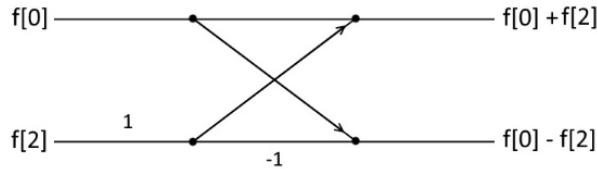


Figure 4.6: Opération *butterfly* à deux entrées

Les nombres complexes obtenus en sortie de cette opération peuvent ensuite être utilisés comme les entrées d'une autre opération *butterfly*.

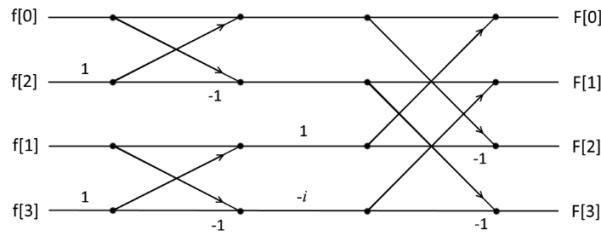


Figure 4.7: Opération *butterfly* à quatre entrées

On obtient alors la DFT de taille $N = 4$, il faut maintenant étendre l'algorithme à une taille N (avec N une puissance de 2). Pour cela, on va structurer la FFT en étapes. Chaque étape va prendre N nombres complexes en entrée, exécuter $N/2$ opérations *butterfly* et retourner N nombres complexes. Lors du calcul de la DFT de taille $N = 4$, nous avons pu faire la simplification suivante :

$$e^{-i\pi/2} = -i$$

Mais dans le contexte d'une DFT de taille N il est préférable d'utiliser le *twiddle* facteur, il représente les racines de l'unité et a pour valeur :

$$W_N = e^{-i\frac{2\pi}{N}}$$

Ce qui nous donne la matrice suivante dans le contexte d'une *butterfly* opération à quatre entrées :

$$\begin{bmatrix} W_4^0 & W_4^0 & W_4^0 & W_4^0 \\ W_4^0 & W_4^2 & W_4^1 & W_4^3 \\ W_4^0 & W_4^4 & W_4^2 & W_4^6 \\ W_4^0 & W_4^6 & W_4^3 & W_4^9 \end{bmatrix} \begin{bmatrix} f[0] \\ f[1] \\ f[2] \\ f[3] \end{bmatrix}$$

Grâce à la propriété des racines de l'unité, la matrice peut être simplifiée :

$$\begin{bmatrix} W_4^0 & W_4^0 & W_4^0 & W_4^0 \\ W_4^0 & W_4^2 & W_4^1 & W_4^3 \\ W_4^0 & W_4^0 & W_4^2 & W_4^2 \\ W_4^0 & W_4^2 & W_4^3 & W_4^1 \end{bmatrix} \begin{bmatrix} f[0] \\ f[1] \\ f[2] \\ f[3] \end{bmatrix}$$

Car :

$$W_N^k = W_N^{k \mod N}$$

Calcul de la carte des hauteurs

Le but est d'obtenir une texture de la carte des hauteurs qui représentera la forme de la vague. Cette carte des hauteurs est défini par une somme de sinusoïdes avec des amplitudes complexes :

$$h(x, z, t) = \sum_k \tilde{h}(k, t) \exp(ik \cdot (x, z))$$

où :

- k est le vecteur d'onde. Il détermine la direction de la propagation de la vague. Avec $k = (k_x, k_z)$, $k_x = 2\pi n/L$ et $k_z = 2\pi m/L$
- $L = V^2/g$
- V est la vitesse du vent.
- $g = 9.8m/sec^2$ est la constante gravitationnelle de la Terre.

Afin de calculer $\tilde{h}(k, t)$ on doit calculer le spectre de Phillips qui est donné par la formule suivante :

$$P_h(k) = A \frac{\exp(-1/(kL)^2)}{k^4} |\hat{k} \cdot \hat{w}|^2$$

où :

- w est la direction du vent.

- $|\hat{k} \cdot \hat{w}|^2$ est un facteur qui élimine les vagues perpendiculaires par rapport à la direction du vent grâce à la propriété du produit scalaire.

Grâce au spectre de Phillips on peut alors déterminer les amplitudes de Fourier (les amplitudes de chaque fréquence de notre fonction) :

$$\tilde{h}_0(k) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(k)}$$

où $(\xi_r + i\xi_i)$ sont des nombres indépendants d'une distribution normale gaussienne avec une moyenne de 0 et un écart-type de 1. Pour cela, on utilise de manière générale la méthode de Box-Muller.

Afin de pouvoir calculer les amplitudes de Fourier en fonction du temps, il nous faut déterminer la relation de dispersion :

$$w^2(k) = gk$$

La relation de dispersion exprime la corrélation entre les fréquences et l'amplitude des vecteurs d'onde.

On peut alors exprimer nos amplitudes de Fourier en fonction du temps et obtenir $\tilde{h}(k, t)$:

$$\tilde{h}(k, t) = \tilde{h}_0(k)\exp(iw(k)t) + \tilde{h}_0(-k)\exp(-iw(k)t)$$

Préparation au compute shading

Comme nous voulons obtenir une carte de hauteur à partir d'une composante spectrale. Nous devons appliquer une FFT inverse (IFFT). Nous pouvons alors définir une équation pour la IFFT à partir de la formule de calcul, obtenue plus haut, de $\tilde{h}(x, z, t)$.

On définit d'abord deux variables k et l avec les bornes :

$$0 < k, l < N - 1$$

On peut redéfinir le vecteur d'onde \mathbf{k} avec k et l :

$$\mathbf{k} = \left(\frac{2\pi k - \pi N}{L}, \frac{2\pi l - \pi N}{L} \right)$$

On obtient alors l'équation suivante :

$$h(n, m, t) = \frac{N \cdot N}{1} \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} \tilde{h}(k, l, t) \exp\left(\frac{i2\pi Nkn - \pi Nn}{N}\right) \exp\left(\frac{i2\pi Nlm - \pi Nm}{N}\right).$$

On peut simplifier et réarranger cette fonction pour obtenir l'équation IFFT 2D :

$$h(n, m, t) = \frac{1}{N^2} (-1)^n \sum_{k=0}^{N-1} \left[(-1)^m \sum_{l=0}^{N-1} \tilde{h}(k, l, t) \exp\left(i\frac{2\pi ml}{N}\right) \right] \exp\left(i\frac{2\pi nk}{N}\right)$$

On parle ici de IFFT 2D car cette IFFT est appliquée sur une texture (une donnée bidimensionnelle).

Algorithme IFFT

L'implémentation de la IFFT en compute shader est réalisée en cinq étapes.

- **Première étape :** Cette phase initiale de l'algorithme consiste à générer deux textures spectrales contenant les valeurs $\tilde{h}_0(k)$ et $\tilde{h}_0(-k)$, calculées à partir du spectre de Phillips.
- **Deuxième étape :** Durant cette étape, la texture contenant les composantes d'amplitude de Fourier $\tilde{h}(k, t)$ pour chaque instant t est générée.
- **Troisième étape :** Dans cette phase, on effectue N FFT 1D horizontales, chacune avec $\log_2 N$ étapes, en utilisant la texture des composantes d'amplitude de Fourier $\tilde{h}(k, t)$ comme entrée. Chaque ligne de la texture d'entrée est traitée comme un vecteur d'entrée pour les FFT 1D.
- **Quatrième étape :** À cette étape, N FFT 1D verticales sont exécutées, chacune avec $\log_2 N$ étapes, en utilisant le résultat de la troisième étape comme entrée. Comme précédemment, chaque colonne de la texture d'entrée est traitée comme un vecteur d'entrée pour les FFT 1D.
- **Cinquième étape :** Cette étape consiste à obtenir la texture finale qui représentera notre vague. Pour cela on va appliquer le calcul de la IFFT 2D vu plus haut. Au cours de cette opération, les amplitudes des hauteurs calculées sont ajustées en multipliant par $(-1)^m$ et $(-1)^n$, où m et n représentent respectivement les indices de colonne et de ligne. Enfin, le résultat est inversé en appliquant une multiplication par $1/N^2$, où N représente la dimension de la texture en entrée.

Résultats

Malgré nos travaux de recherche sur ce sujet, nous n'avons pas eu le temps nécessaire d'implémenter ce modèle en raison de la structure de base de notre code qui n'était pas adaptée à l'utilisation de multiples compute shaders. Toutefois, nous avons réussi à mettre en place un shader de calcul sur un modèle de vague sinusoïdale simple.



Figure 4.8: Exemple d'un rendu basé sur un spectre de Phillips

4.2 Système de flottabilité

En parallèle du travail effectué sur la simulation réaliste de vagues, nous avons aussi mis en place un système de flottabilité pour des objets simples, dans notre cas des sphères.

4.2.1 Premier essai de simulation de flottabilité

Comme première tentative pour mettre en place une simulation de flottabilité dans notre projet, nous avons commencé par utiliser un principe à 3 états pour les sphères flottantes. On considérera qu'à leurs apparitions, les sphères seront dans l'état 1.

- **Etat 1 :** La sphère se situe à un niveau supérieur de la surface du maillage de plan, et en dessous de leur point d'apparition initial `heightSpawn`. Dans ce cas, la sphère tombe sous l'effet de la gravité jusqu'à arriver dans l'état 2, au moment où sa hauteur sera inférieur à l'amplitude utilisée pour les vagues. Lors de sa chute, la vitesse de la sphère est conservée dans un accumulateur, celui-ci sauvegardant la vitesse à laquelle une sphère arrive à l'état suivant.
- **Etat 2 :** A cette étape, on utilise l'accumulateur précédent pour mettre à jour la position d'une sphère, en décrémentant peu à peu celui-ci. Ainsi, la sphère continuera sa chute dans l'eau, avant d'inverser son mouvement jusqu'à revenir vers la surface et passer dans son dernier état.
- **Etat 3 :** La sphère a atteint la bonne hauteur pour commencer son mouvement cyclique. La sphère va alors avoir un mouvement qui va osciller de manière régulière, de la même manière que le modèle sinusoïdale utilisé pour la simulation des vagues (voir 4.1.1), puisque c'est une formule identique qui est utilisée.

Cette manière de procéder s'est révélée peu convaincante dû au manque de réalisme dans les mouvements des sphères. En effet, il s'est révélé plus complexe que prévu d'avoir des transitions correctes pour les mouvements de ces dernières lorsque qu'elles passaient d'un

état à un autre. Nous ne sommes finalement pas parvenu à obtenir des résultats satisfaisants, et avons donc utilisé une seconde méthode, plus précise et réaliste que la première.

4.2.2 Utilisation de la force de flottabilité

Avec cette seconde méthode, nous avons fait en sorte de pouvoir paramétriser la flottabilité selon la constante de gravitation, ainsi que selon la masse volumique du milieu dans lequel les objets flotteront. Dans notre cas, nous avons utilisé $9.81m/s^2$ pour la gravité, et $1000kg/m^3$ pour masse volumique de l'eau.

De plus, il est possible de faire varier la masse et le volume des sphères, ce qui aura une influence sur leurs flottabilités, étant donné que le volume d'eau déplacé par chacunes d'entre elles variera en conséquence.

Ainsi, ce seront 2 forces opposées qui s'appliqueront sur chacune des sphères :

- La force de gravité :

$$F_{gravity} = m \times g$$

(où m est la masse de l'objet flottant, et g la constante de gravitation mentionnée précédemment)

- La force de flottaison :

$$F_{buoyancy} = \rho_{water} \times V_{displaced} \times g$$

(où ρ_{water} est la masse volumique du milieu, $V_{displaced}$ est le volume d'eau déplacé par l'objet flottant, et g est la constante de gravitation)

Cette force est celle évoquée dans le principe d'Archimède : *"Tout corps plongé dans un liquide subit une poussée verticale vers le haut égale au poids du volume de liquide déplacé"*.

Dans l'équation de la force de flottabilité, il est donc nécessaire d'estimer le volume d'eau déplacé par nos objets, dans notre cas des sphères, ce qui facilite grandement la recherche de cette valeur. En effet, celle-ci est donnée par :

- Dans le cas où la sphère est complètement immergée :

$$V_{displaced} = \frac{4}{3} \times \pi \times r^3$$

- Dans le cas où la sphère n'est que partiellement immergée :

$$V_{displaced} = \pi \times depthSphere^2 * (3 \times r - depthSphere) / 3$$

(où $radius$ est le rayon de la sphère, et $depthSphere$ est la profondeur d'immersion de

la sphère dans l'eau)

Une fois les 2 forces calculées, on peut alors déterminer la force nette qui s'appliquera à la sphère :

$$F_{net} = F_{gravity} + F_{buoyancy}$$

Puis, grâce à cette force nette, on peut déterminer l'accélération qui viendra mettre à jour la vitesse de déplacement de la sphère (sachant qu'elle est initialement nulle), elle même utilisée pour mettre à jour la position de la sphère :

$$a = \frac{F_{net}}{m}$$

$$v_{current} = v_{previous} + a \cdot \Delta t$$

$$pos_{current} = pos_{previous} + v_{current} \cdot \Delta t$$

4.2.3 Application d'une force de restitution

L'un des avantages de la seconde technique, par rapport à la première, est notamment le fait de pouvoir calculer une restitution de la force de flottabilité au moment où la sphère émerge de l'eau (dans le cas où cette force est suffisamment importante pour que ce soit le cas), et ainsi avoir un comportement plus réaliste pour nos objets flottants. Ainsi, si le point d'apparition initial d'une sphère est suffisamment haut, la sphère descendra à un tel point qu'au moment où elle atteindra à nouveau la surface de l'eau, la force de flottabilité sera assez importante pour que la sphère émerge de l'eau et soit projetée en l'air (avec une force amoindrie).

Initialement, nous avions fait en sorte que les sphères ne puissent pas émerger complètement de l'eau après y être tombé (de manière à ne pas avoir une animation qui paraisse surréaliste). Lors de l'implémentation de cette nouvelle fonctionnalité, nous avons donc fait face au fait qu'il était impossible de calculer la force de flottabilité au-delà de la surface. Les formules données précédemment ne furent donc plus suffisante. Pour contrer ce problème, il a fallu faire en sorte que lorsqu'une sphère atteignait la surface de l'eau, une force lui soit donnée afin qu'elle soit projetée dans les airs. Et c'est justement cette force qui doit dépendre de la force de flottabilité de la sphère au moment où elle atteint la surface.
Évidemment, pour que la sphère ne soit pas éjecté avec autant de force que celle de la flottabilité, il a été nécessaire d'utiliser un coefficient de restitution (modifiable via la fenêtre ImGui) avec lequel on divise la force de flottabilité à la surface. Par défaut, cette valeur a été mise à 1.7.

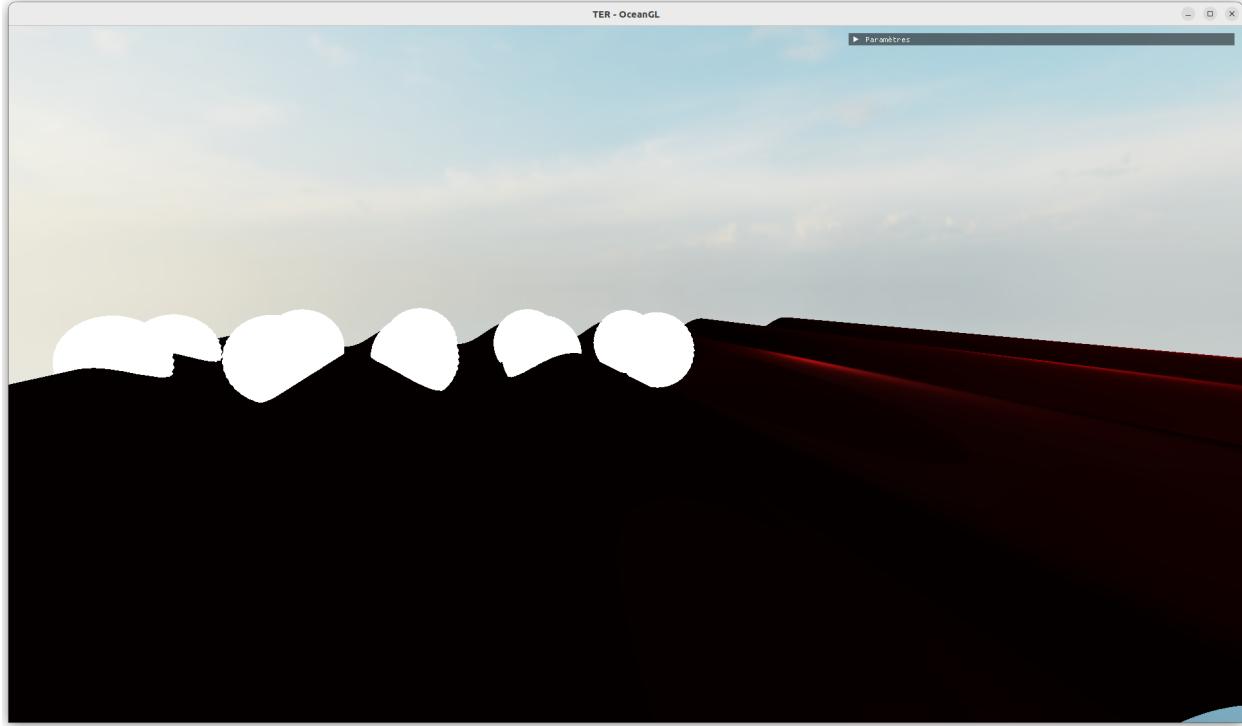


Figure 4.9: Sphères qui flottent sur une vague sinusoïdale

4.3 Éclairage et rendu

L'éclairage est un aspect crucial dans la simulation réaliste des vagues océaniques. Dans notre projet, nous avons principalement utilisé le modèle d'éclairage de Phong pour éclairer la scène, et nous avons également exploré l'utilisation du Physically Based Rendering (PBR) pour le modèle sinusoïdal simple. Aussi, nous avons mis en place une skybox, ainsi que des effets différents selon les hauteurs des vagues et le point de vue de l'utilisateur.

4.3.1 Modèle d'éclairage de Phong

Le modèle d'éclairage de Phong est une technique couramment utilisée pour simuler la manière dont la lumière interagit avec les surfaces. Il se compose de trois composants principaux : l'éclairage ambiant, diffus et spéculaire.

Éclairage ambiant

L'éclairage ambiant représente la lumière indirecte présente dans la scène, qui illumine uniformément tous les objets. Cette composante est indépendante de la position et de l'orientation de la surface.

Éclairage diffus

L'éclairage diffus simule la lumière directe provenant d'une source lumineuse et frappant une surface. L'intensité de l'éclairage diffus dépend de l'angle entre la direction de la lumière et la normale de la surface. Il est calculé en utilisant le produit scalaire entre le vecteur normal de la surface et le vecteur directionnel de la lumière :

$$I_{\text{diffus}} = I_{\text{lumière}} \cdot \max(\mathbf{L} \cdot \mathbf{N}, 0)$$

où $I_{\text{lumière}}$ est l'intensité de la lumière, \mathbf{L} est le vecteur directionnel de la lumière, et \mathbf{N} est le vecteur normal de la surface.

Éclairage spéculaire

L'éclairage spéculaire simule les reflets brillants sur les surfaces, créant des points lumineux là où la lumière est réfléchie directement vers l'observateur. Il est calculé en utilisant le produit scalaire entre le vecteur de vue et le vecteur de réflexion de la lumière :

$$I_{\text{spéculaire}} = I_{\text{lumière}} \cdot \max(\mathbf{R} \cdot \mathbf{V}, 0)^\alpha$$

où \mathbf{R} est le vecteur de réflexion, \mathbf{V} est le vecteur de vue, et α est le shininess, qui contrôle la brillance du reflet.

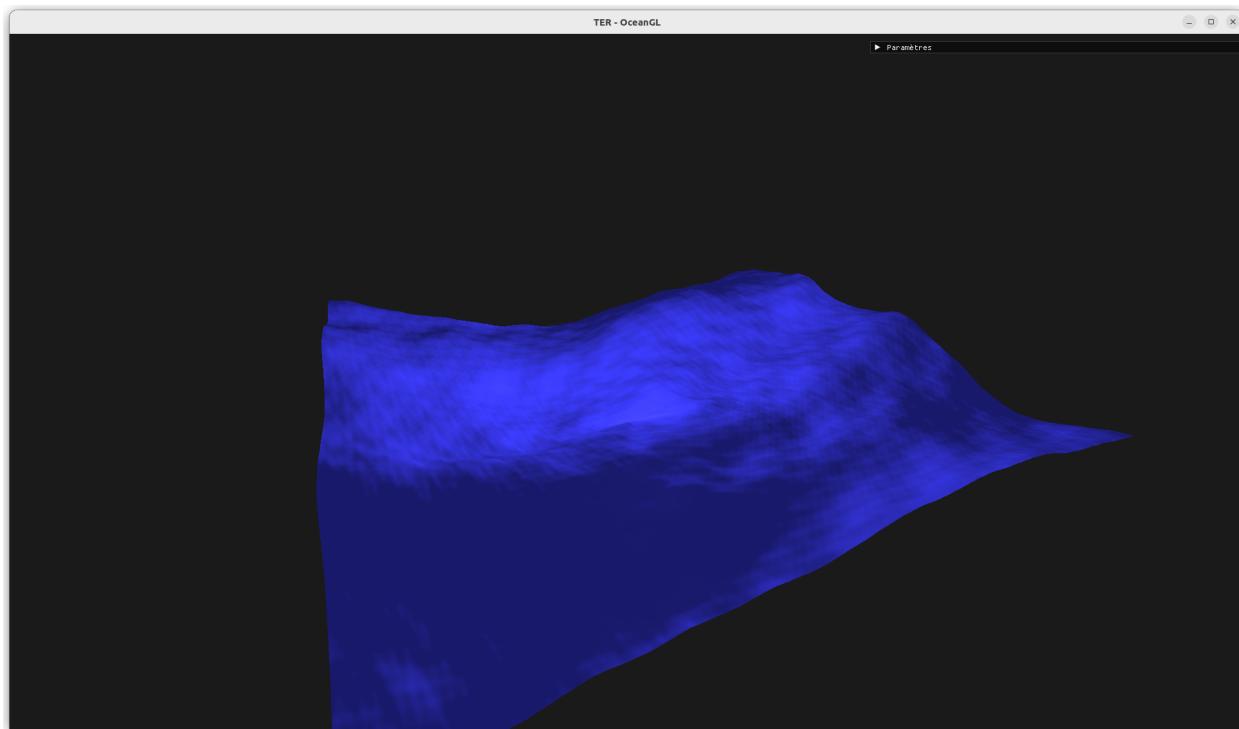


Figure 4.10: Exemple de l'illumination de Phong sur une somme de sinusoïdes

4.3.2 Skybox

Une skybox est une technique utilisée en graphisme 3D pour créer l'illusion d'un environnement lointain et infini autour de la scène. Elle consiste en une boîte qui entoure la scène et sur laquelle sont appliquées des textures représentant le ciel, l'horizon et éventuellement des éléments de décor lointains. La skybox permet de simuler un environnement réaliste sans avoir à modéliser des détails complexes à grande distance.

Implémentation de la skybox

Pour implémenter une skybox, nous utilisons un cube autour de la scène sur lequel sont mappées des textures représentant le ciel. Les coordonnées de texture du cube sont calculées de manière à correspondre aux directions de vue du spectateur.

Éclairage et réflexions

L'une des principales utilisations de la skybox dans notre projet est de simuler les réflexions sur la surface de l'eau. En utilisant un shader, nous pouvons récupérer la couleur de la skybox en fonction de la direction de réflexion calculée sur la surface de l'eau. Cela permet de créer des réflexions réalistes des nuages, du ciel et d'autres éléments présents dans la skybox.

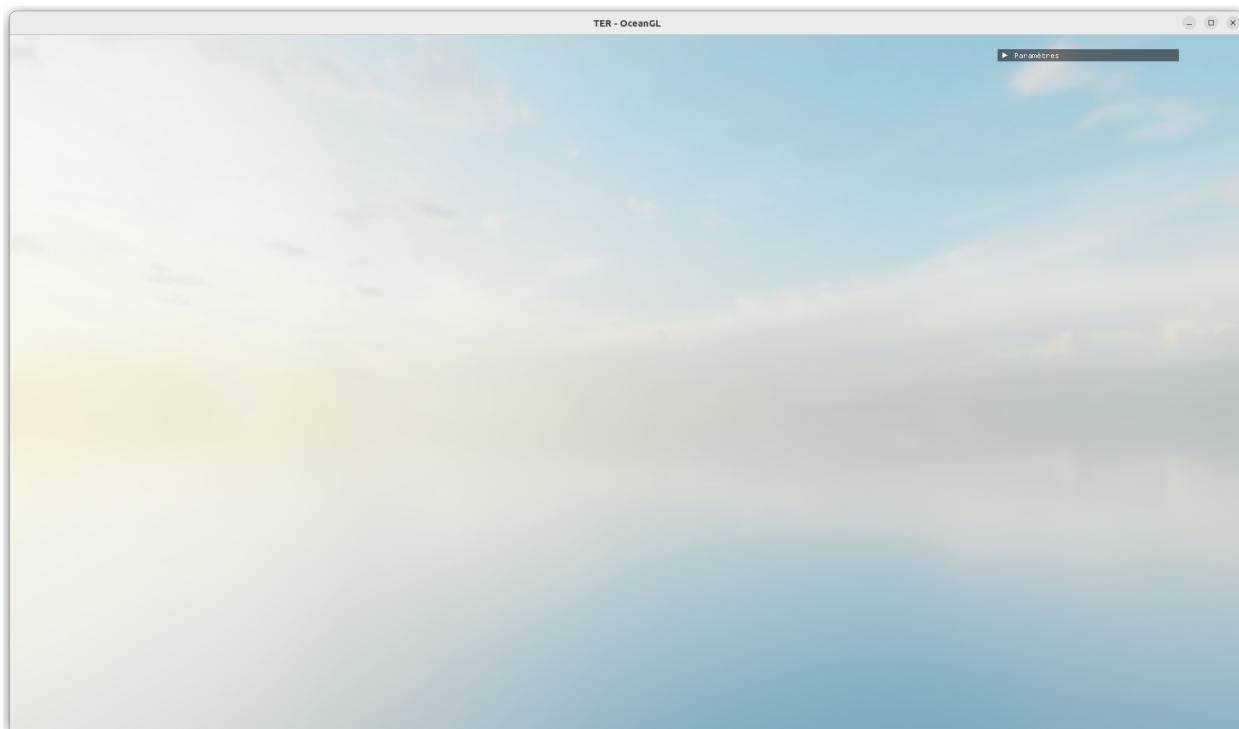


Figure 4.11: Skybox utilisée dans notre simulation

Le shader utilisé pour créer la skybox est détaillé en Annexe 8.3.2.

Résultats et performances

L'ajout d'une skybox améliore considérablement le réalisme visuel de la simulation en fournissant un contexte environnemental riche et en permettant des réflexions dynamiques sur l'eau. L'impact sur les performances est minimal, car la skybox est rendue en tant que cube map et ne nécessite pas de calculs complexes pour chaque fragment.

4.3.3 Effet de Fresnel

L'effet de Fresnel est un phénomène optique qui décrit la variation de la quantité de lumière réfléchie en fonction de l'angle d'incidence de la lumière par rapport à la surface. Plus l'angle d'incidence est grand, plus la réflexion est forte. Cet effet est crucial pour simuler des matériaux réalistes, notamment pour les surfaces d'eau.

Principe de base

L'effet de Fresnel peut être décrit mathématiquement à l'aide de la formule suivante pour le facteur de Fresnel F :

$$F(\theta) = F_0 + (1 - F_0)(1 - \cos \theta)^5$$

où :

- F_0 est le coefficient de réflexion à angle normal, souvent pris comme une constante,
- θ est l'angle entre le vecteur de vue et la normale de la surface,
- $\cos \theta$ est le cosinus de cet angle, calculé par le produit scalaire entre le vecteur de vue et la normale.

Cette formule est une approximation simplifiée de l'effet de Fresnel, connue sous le nom d'approximation de Schlick.

Application de l'effet de Fresnel

Dans notre projet, nous utilisons l'effet de Fresnel pour moduler la réflexion de la lumière sur la surface de l'eau. La réflexion est plus forte à des angles obliques et plus faible à des angles proches de la normale.

Le facteur de Fresnel est intégré dans le calcul de l'éclairage de la surface de l'eau, en influençant la contribution de la couleur réfléchie par la skybox.

Calcul du facteur de Fresnel

Pour calculer le facteur de Fresnel dans notre implémentation, nous utilisons la formule suivante :

$$F(\theta) = (1 - \cos \theta)^p + b$$

où :

- p est une puissance ajustable qui contrôle l'intensité de l'effet de Fresnel,
- b est un biais ajouté pour ajuster la réflexion de base.

Cette formule est similaire à l'approximation de Schlick, mais simplifiée pour une implémentation plus directe dans notre shader. Le produit scalaire entre le vecteur de vue et la normale est utilisé pour calculer $\cos \theta$, et le facteur de Fresnel en résultant est utilisé pour mélanger la couleur de la skybox avec la couleur de la surface de l'eau.

Résultats et performances

L'ajout de l'effet de Fresnel améliore considérablement le réalisme visuel de la surface de l'eau en fournissant des réflexions dynamiques qui varient en fonction de l'angle de vue. Cela donne à l'eau un aspect plus naturel et réaliste, en particulier lorsque la caméra se déplace autour de la scène. L'impact sur les performances est minimal, car le calcul du facteur de Fresnel est simple et efficace.

4.3.4 Combinaison des différents effets d'éclairage

Tous les effets d'éclairage sont combinés pour produire le rendu final. Le modèle de Phong fournit la base de l'éclairage avec les composants ambiant, diffus et spéculaire. L'effet de Fresnel ajoute une couche supplémentaire de réalisme en simulant les réflexions en fonction de l'angle de vue. Enfin, la skybox est utilisée pour fournir des réflexions dynamiques et un contexte environnemental réaliste.

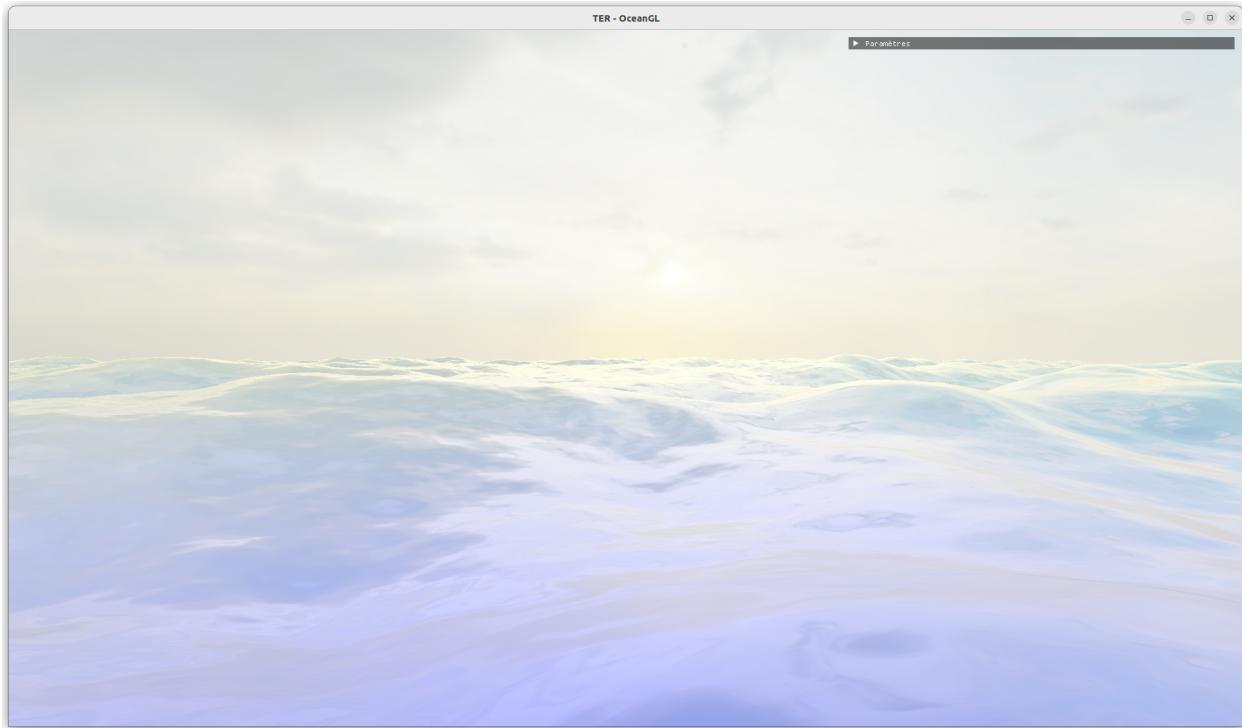


Figure 4.12: Rendu final de la somme de sinusoïdes

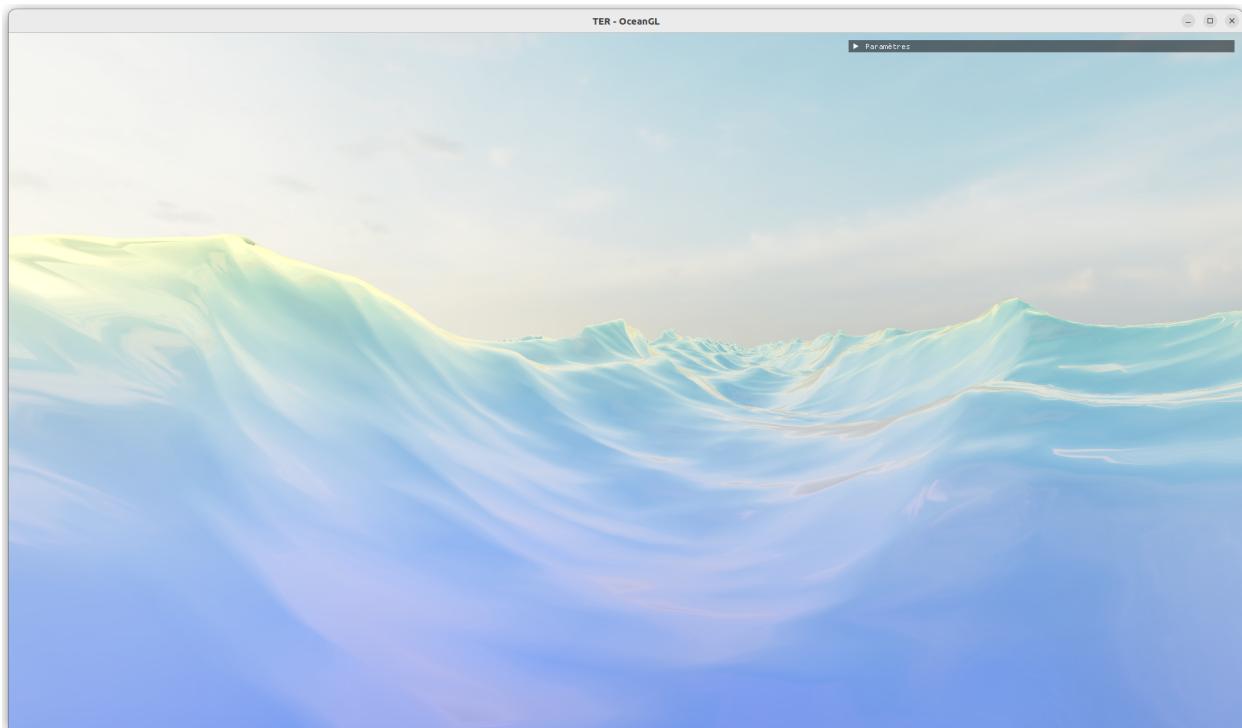


Figure 4.13: Rendu final de la somme de vagues de Gerstner

Le résultat final est une simulation d'océan qui bénéficie de l'illumination complexe et

réaliste, combinant les contributions de chaque composant pour offrir une expérience visuelle riche et immersive.

4.3.5 Physically Based Rendering (PBR)

Le Physically Based Rendering (PBR) est une méthode d'éclairage avancée qui simule l'interaction de la lumière avec les surfaces de manière plus réaliste en s'appuyant sur des principes physiques. Cette approche permet de créer des matériaux plus authentiques en tenant compte de la conservation de l'énergie et des propriétés microscopiques des surfaces, contrairement au modèle de Phong.

Principes fondamentaux du PBR

Le PBR repose sur deux propriétés matérielles essentielles : la rugosité et la réflectivité. La rugosité détermine la diffusion de la lumière sur une surface, tandis que la réflectivité, ou albédo, contrôle la quantité de lumière réfléchie. Ces propriétés sont utilisées pour calculer de manière réaliste l'éclairage direct et indirect sur une surface.

- **Rugosité** : La rugosité affecte la manière dont la lumière est diffusée sur la surface. Une surface rugueuse diffuse la lumière de manière plus dispersée, créant des reflets plus doux. En revanche, une surface lisse produit des reflets nets.
- **Réflectivité** : La réflectivité, ou albédo, est la proportion de lumière réfléchie par la surface. Elle est utilisée pour modéliser la couleur et la brillance des matériaux.

Modèle d'éclairage basé sur la physique

Le PBR utilise un modèle d'éclairage basé sur la physique, qui comprend plusieurs composants :

- **Éclairage diffus** : Simule la lumière diffusée de manière aléatoire lorsqu'elle frappe une surface. Calculé en utilisant la loi de Lambert.
- **Éclairage spéculaire** : Simule les reflets brillants sur les surfaces. Utilise le modèle de Fresnel-Schlick pour calculer la réflectance spéculaire en fonction de l'angle d'incidence.
- **Occlusion ambiante** : Ajoute des ombres douces dans les coins et les crevasses pour simuler la lumière indirecte.

Implémentation du PBR

Nous avons implémenté le PBR pour le modèle sinusoïdal simple afin de tester ses effets et d'améliorer le réalisme visuel de notre simulation. Bien que nous n'ayons pas étendu l'application du PBR au-delà de ce modèle, les résultats obtenus sont convaincants.

Le fragment shader utilise les équations de base du PBR pour calculer la couleur finale des pixels en tenant compte de la lumière ambiante, diffuse et spéculaire. L'implémentation détaillée du PBR dans le fragment shader est présentée en Annexe 8.3.3.

Résultats et perspectives

Cette méthode offre un potentiel important pour améliorer encore le réalisme de la simulation, notamment en simulant des matériaux variés et en capturant les interactions complexes de la lumière.

Malheureusement, nous n'avons pas eu le temps d'étendre l'utilisation du PBR à d'autres modèles de vagues et à différents matériaux pour explorer davantage ses avantages et ses applications dans la simulation océanique.

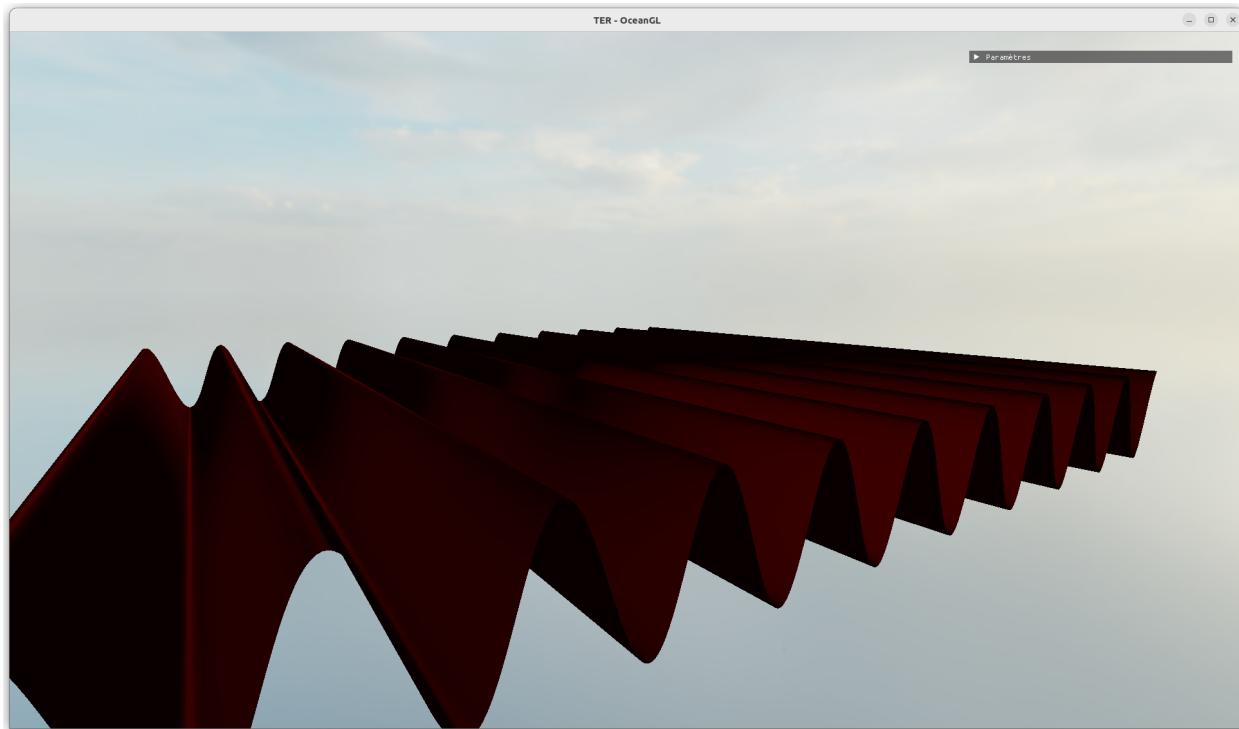


Figure 4.14: Exemple de rendu utilisant le PBR

L'implémentation du PBR dans le fragment shader est détaillée en Annexe 8.3.3.

4.4 Gestion du son

Dans le but de renforcer le réalisme de la simulation d'océan que nous avons réalisé, nous souhaitions avoir la possibilité de jouer des sons à l'intérieur de notre programme. Pour cela, nous avons décidé d'utiliser la librairie miniaudio (dont la documentation est disponible ici).

L'utilisation de cette librairie présente de nombreux avantages :

- Afin d'inclure la librairie dans notre projet, il nous suffit d'ajouter un unique fichier source (ici).
- miniaudio fonctionne sur les principaux systèmes d'exploitation (Windows, macOS, Linux, iOS, Android, etc...).
- Une API haut niveau est fournie pour utiliser la librairie, celle-ci étant très simple d'utilisation, en plus d'être très bien documentée. Sur le dépôt GitHub de miniaudio, un grand nombre d'exemples est fourni, ces derniers exposant les bonnes pratiques à adopter pour une bonne utilisation de la librairie.
- Il est possible d'utiliser différents types de fichiers audio, notamment `wav` et `mp3`. Dans notre cas, nous avons fait le choix d'utiliser des fichiers `mp3`, qui fournissent une qualité audio suffisante, et avec un bon taux de compression pour des fichiers légers.
- La librairie présente aussi l'avantage de pouvoir initialiser et nettoyer correctement la mémoire, afin de jouer plusieurs sons.

Après avoir donc ajouté cette librairie à notre projet, nous avons récupéré plusieurs sons libres de droit sur internet, et avons fait en sorte qu'ils puissent être joués dans notre simulation :

- Plusieurs bruitages de mouette, qui se lancent de manière aléatoire (en prenant la précaution que ces bruitages ne puissent pas se lancer en simultané).
- Un son de vague, durant environ une minute. Celui-ci est joué en boucle jusqu'à la fermeture de l'application. Pour cela, nous utilisons un callback (une fonction de rappel), qui est appelée automatiquement au moment où le son se termine. Une telle fonction est justement possible grâce à la librairie miniaudio, en respectant une signature particulière.

4.5 Interface utilisateur

4.5.1 DearImGui

Dans le but d'avoir des paramètres contrôlables de manière interactive, nous avons intégré la bibliothèque Dear ImGui dans notre projet. Cette bibliothèque est dédiée à la création d'interfaces utilisateur immédiates (Immediate Mode GUI), ce qui est particulièrement adapté pour des applications nécessitant des ajustements rapides et en temps réel des paramètres.

Nous avons choisi cette bibliothèque pour sa simplicité d'utilisation, sa flexibilité et sa compatibilité ainsi que ses performances.

4.5.2 Prise en main de l'application

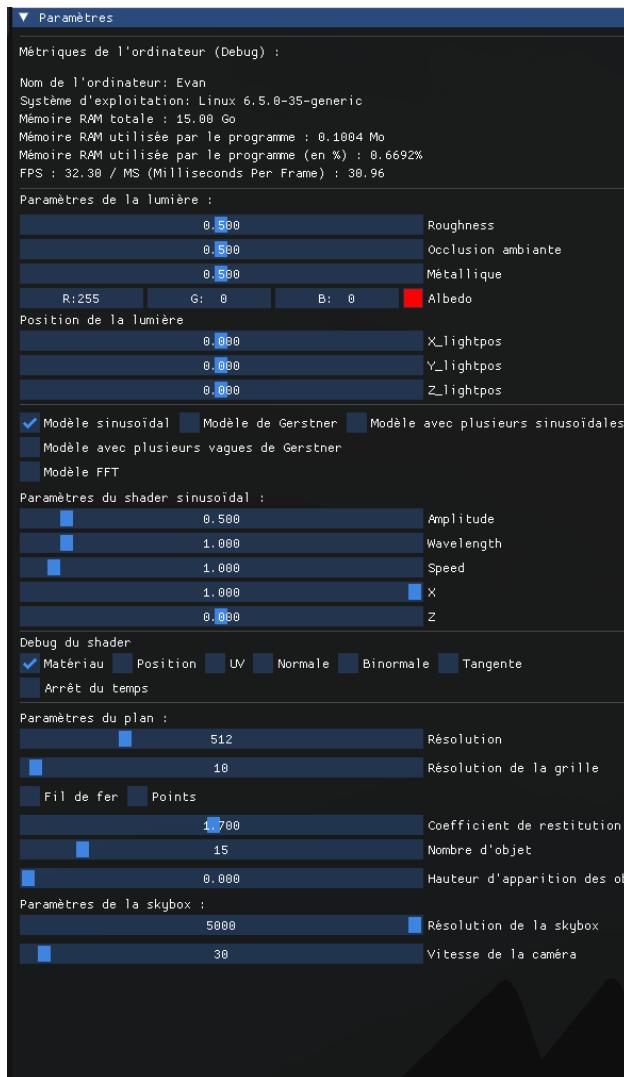


Figure 4.15: Exemple de l'interface ImGui

Cette interface utilisateur permet d'afficher les informations du système de l'utilisateur. En dessous, on peut modifier les paramètres de la lumière et choisir le modèle souhaité (simple vague sinusoïdale, plusieurs vagues sinusoïdales, vague de Gerstner, plusieurs vagues de Gerstner et le modèle expérimental basé sur la transformée de Fourier rapide). Suivant le modèle choisi, différentes options sont paramétrables. Ensuite, il y a les différentes options du shader qui permettent de le déboguer. On peut ensuite modifier les paramètres du plan et de la skybox. Pour la flottabilité, on peut modifier la hauteur d'apparition des objets flottants, modifier leur nombre (entre 1 et 100), et définir le coefficient de restitution de la force d'éjection (voir 4.2.3).

Liste des contrôles

- **Déplacement de la caméra :** les touches **ZQSD** (respectivement mouvement avant, gauche, arrière, droit).
- **Monter / descendre la caméra :** respectivement **Espace / Ctrl.**
- **Rotation de la caméra :** la rotation de la caméra se fait avec les mouvements de la souris.
- **Apparition du curseur :** la touche **T** permet de faire appaitre le curseur afin de pouvoir modifier les paramètres depuis l'interface ImGui.
- **Disparition du curseur :** la touche **E** permet de cacher le curseur afin de revenir au contrôle de la rotation de la caméra.
- **Apparition de sphères :** la touche **L** permet de faire appaitre des objets flottants (dans notre cas des sphères) dans la scène.

CHAPITRE 5. Problèmes rencontrés

5.1 Calcul des normales

Nous avons eu beaucoup de mal à mettre en place les normales pour la somme de vagues sinusoïdales. Ce problème nous a coûté beaucoup en terme de temps et nous a empêché d'avoir un éclairage correct pendant une bonne partie du développement. Nous nous sommes résolus à utiliser la méthode des différences finies pour contourner cette difficulté, et cela s'est avéré assez concluant.

5.2 Le modèle spectral

Nous avons rencontré plusieurs problèmes durant l'implémentation du modèle spectral qui ont mené à notre incomplétion du modèle. L'algorithme de la transformée de Fourier rapide est basé sur des concepts avancés, ce qui nous a demandé beaucoup de travail de recherche pour comprendre comment l'implémenter dans un compute shader. Mais également, la structure de notre code. Malgré la réussite de notre implémentation d'un compute shader, la structure de notre programme n'était pas adaptée pour l'utilisation de plusieurs compute shaders en parallèle ce qui a été problématique car l'implémentation du modèle spectral en compute shader requiert une utilisation de plusieurs compute shader en parallèle.

CHAPITRE 6. Bilan du projet

6.1 Autocritique

Bien que notre moteur soit fonctionnel et que son architecture soit correcte, nous estimons que nous aurions pu optimiser davantage le code et améliorer l'architecture du moteur. Par exemple, nous n'avons pas réussi à implémenter le modèle spectral, principalement en raison de difficultés rencontrées lors de l'association de plusieurs textures aux compute shaders, ce qui constitue un problème d'architecture. Concernant les problèmes rencontrés avec les calculs des normales, nous pensons que les résultats peu satisfaisants sont dûs à des erreurs dans le calcul des dérivées partielles.

6.2 Enseignements tirés

Au cours de ce projet, plusieurs enseignements ont été tirés, notamment en termes de gestion du temps, de planification et de développement de compétences techniques. Voici quelques-uns des principaux enseignements :

- **Gestion du temps** : Il est essentiel d'établir un planning réaliste et de respecter les échéances pour mener à bien un projet dans les délais impartis.
- **Communication** : Une communication efficace au sein de l'équipe est cruciale pour assurer une collaboration harmonieuse et éviter les malentendus.
- **Compétences techniques** : Ce projet a permis de consolider nos compétences en programmation, en conception logicielle et en résolution de problèmes techniques.
- **Adaptabilité** : Face aux imprévus et aux difficultés rencontrées, il est important de rester flexible et de trouver des solutions alternatives.
- **Collaboration** : Travailler en équipe nous a permis d'apprendre à partager des responsabilités, à coopérer et à tirer parti des forces de chacun.

6.3 Perspectives

Nous aurions souhaité intégrer la simulation d'ondes spectrales basée sur la transformée de Fourier rapide (FFT) dans notre projet, pour capturer plus finement les caractéristiques complexes des vagues océaniques. Cette approche aurait ajouté un niveau supplémentaire de réalisme à notre simulation.

De plus, l'inclusion du modèle "shallow water", basé sur les équations de Barré de Saint-Venant, aurait renforcé le réalisme en prenant en compte les interactions entre les différentes couches d'eau et les effets de la topographie du fond marin sur la propagation des vagues.

Nous aurions également aimé intégrer des particules utilisant des techniques de la mécanique des fluides numérique (CFD) pour modéliser plus précisément les interactions entre les vagues et les objets flottants, ainsi que les phénomènes de turbulence et de dispersion dans l'eau. Cette addition aurait non seulement amélioré le réalisme visuel de notre simulation, mais aussi permis une exploration approfondie des processus physiques sous-jacents à l'écoulement de l'eau. Cependant, en raison de contraintes de temps et de ressources, nous n'avons pas pu concrétiser cette ambition dans ce projet.

L'ajout d'une skybox dynamique avec des nuages volumétriques aurait également été une amélioration significative, créant un environnement atmosphérique plus immersif et des jeux de lumière dynamiques en fonction des conditions météorologiques. Malheureusement, cette fonctionnalité n'a pas pu être incluse.

Utiliser d'autres spectres océanographiques comme Jonswap ou Double Jonswap aurait enrichi la variété des vagues simulées. Ces spectres, couramment utilisés pour modéliser les vagues en fonction de conditions météorologiques spécifiques, auraient permis de capturer une plus grande diversité de comportements des vagues, contribuant à un réalisme accru de la simulation.

Globalement, nous sommes satisfaits des résultats obtenus. Les objectifs minimums que nous avions établis ont été atteints avec succès. Cependant, intégrer un modèle plus réaliste basé sur la transformée de Fourier rapide (FFT) aurait optimisé la fidélité aux phénomènes océaniques réels. Nous espérons explorer ces pistes dans de futurs projets pour améliorer encore nos simulations océaniques si l'occasion venait à se re-présenter.

CHAPITRE 7. Conclusion

Le projet que nous avons mené nous a offert une expérience enrichissante et passionnante dans le domaine de la simulation et de la modélisation océanique. Tout au long de ce travail, nous avons exploré et expérimenté diverses techniques pour reproduire de manière réaliste le mouvement des vagues et la dynamique des océans.

Cette expérience nous a permis de développer nos compétences en programmation, en analyse numérique et en compréhension des phénomènes physiques sous-jacents. Nous avons également pris conscience des limites et des possibilités des différentes approches de modélisation, ce qui nous a amenés à réfléchir sur les axes d'amélioration et les pistes de recherche futures dans ce domaine.

En conclusion, ce projet a été une véritable aventure intellectuelle et technique pour nous tous. Nous sommes fiers du travail accompli et des compétences acquises tout au long de ce parcours, et nous sommes impatients d'appliquer ces connaissances dans nos projets et métiers futurs.

CHAPITRE 8. Annexes

8.1 Shaders du modèle sinusoïdal

8.1.1 Vertex shader du modèle sinusoïdal simple

```

1 #version 460 core
2
3 layout (location = 0) in vec3 aPos;
4 layout (location = 1) in vec2 aTex;
5
6 uniform float time;
7 uniform float amplitude;
8 uniform float wavelength;
9 uniform float speed;
10
11 out vec3 fragPos;
12 out vec2 texCoord;
13 out vec3 normal;
14
15 vec3 CalculateNormal(vec3 Pos, float amplitude, float k, float omega
16   , float time) {
16     float delta = 0.01;
17     vec3 PosX = Pos + vec3(delta, 0.0, 0.0);
18     vec3 PosZ = Pos + vec3(0.0, 0.0, delta);
19
20     Pos.y += amplitude * sin(k * Pos.x - omega * time);
21     PosX.y += amplitude * sin(k * PosX.x - omega * time);
22     PosZ.y += amplitude * sin(k * PosZ.z - omega * time);
23
24     vec3 tangentX = PosX - Pos;
25     vec3 tangentZ = PosZ - Pos;
26
27     return normalize(cross(tangentZ, tangentX));
28 }
29
30 void main() {
31   float k = 2.0 * 3.14159 / wavelength;
32   float omega = speed * k;
33
34   vec3 pos = aPos;
35   pos.y += amplitude * sin(k * pos.x - omega * time);
36
37   normal = CalculateNormal(aPos, amplitude, k, omega, time);
38
39   fragPos = pos;
40   texCoord = aTex;

```

```

41     gl_Position = projection * view * model * vec4(pos, 1.0);
42 }
43

```

8.1.2 Vertex shader de la somme des sinusoïdes

```

1 #version 460 core
2
3 layout (location = 0) in vec3 aPos;
4 layout (location = 1) in vec2 aTex;
5 layout (location = 2) in vec3 aNormal;
6
7 // Variables de sorties
8 out float height;
9 out vec3 pos;
10 out vec3 normal;
11 out vec2 tex;
12 out vec3 binormale;
13 out vec3 tangente;
14 out vec3 positionWorld;
15 out vec3 normalWorld;
16
17 uniform mat4 model;
18 uniform mat4 view;
19 uniform mat4 projection;
20 uniform float time;
21
22 uniform float Amplitude_min;
23 uniform float Amplitude_max;
24 uniform float Amplitude_FBM;
25 uniform float L_min;
26 uniform float L_max;
27 uniform float S;
28 uniform vec3 Direction;
29 uniform float PI;
30 uniform float Gain_A;
31 uniform float Gain_W;
32 uniform float L_FBM;
33
34 uniform int nbVagues;
35 uniform int seed;
36 uniform int FBM;
37 uniform int DW;
38
39 float rand(vec2 co) {
40     return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) *
41                 43758.5453);

```

```

41 }
42
43 vec3 randomDir(float seed) {
44     float angle = rand(vec2(seed, 0.0)) * 2.0 * PI;
45     float x = cos(angle);
46     float y = sin(angle);
47     return vec3(x, 0.0, y);
48 }
49
50 vec3 Add_Wave(vec3 Pos, float S, vec3 Direction, float time) {
51     vec3 newPos = Pos;
52
53     float wave = 0.0f;
54     float L;
55     float w;
56     float A;
57
58     if (FBM == 0) {
59         for (int i = 0; i < nbVagues; i++) {
60             L = mix(L_min, L_max, rand(vec2(float(i), seed)));
61             w = (2.0 * PI) / L;
62             A = mix(Amplitude_min, Amplitude_max, rand(vec2(float(i),
63                 seed)));
64             float randomPhase = rand(vec2(float(i) + 0.1, seed)) *
65                 2.0 * PI;
66             vec3 randomDirection = normalize(reflect(Direction,
67                 normalize(vec3(
68                     rand(vec2(float(i) + 0.2, seed)),
69                     rand(vec2(float(i) + 0.3, seed)),
70                     rand(vec2(float(i) + 0.4, seed))))));
71
72             float phi_t = (S * w) * time + randomPhase;
73             float sinTerm = sin(dot(randomDirection, Pos) * w +
74                 phi_t);
75
76             wave += A * sinTerm;
77         }
78     } else if (FBM == 1) {
79         float wFBM = (2.0 * PI) / L_FBM;
80         float AFBM = Amplitude_FBM;
81
82         for (int i = 0; i < nbVagues; i++) {
83             vec3 randomDirection = randomDir(float(i) + seed);
84             float phi_t = (S * wFBM) * time;
85             float sinTerm = sin(dot(randomDirection, Pos) * wFBM +
86                 phi_t);
87
88             wave += AFBM * sinTerm;
89
90     }
91 }
```

```

85         wFBM *= Gain_W;
86         AFBM *= Gain_A;
87     }
88 }
89
90 newPos.y += wave;
91 return newPos;
92 }
93
94 vec3 CalculateNormal(vec3 Pos, float S, vec3 Direction, float time,
95   out vec3 dPosdx, out vec3 dPosdz) {
96     float eps = 0.01; // Small offset for finite difference
97     vec3 dX = vec3(eps, 0.0, 0.0);
98     vec3 dZ = vec3(0.0, 0.0, eps);
99
100    vec3 posX = Add_Wave(Pos + dX, S, Direction, time);
101    vec3 posZ = Add_Wave(Pos + dZ, S, Direction, time);
102    vec3 posY = Add_Wave(Pos, S, Direction, time);
103
104    dPosdx = normalize(posX - posY);
105    dPosdz = normalize(posZ - posY);
106
107    return normalize(cross(dPosdz, dPosdx));
108 }
109
110 void main() {
111   vec3 dPosdx, dPosdz;
112   vec3 accPos = Add_Wave(aPos, S, Direction, time);
113   vec3 accNor = CalculateNormal(aPos, S, Direction, time, dPosdx,
114                                 dPosdz);
115
116   vec3 newWave = accPos;
117   gl_Position = projection * view * model * vec4(newWave, 1.0f);
118
119   pos = newWave;
120   normal = accNor;
121   tangente = dPosdx;
122   binormale = dPosdz;
123   height = newWave.y;
124   tex = aTex;
125   normalWorld = normalize(mat3(transpose(inverse(model))) * normal
126 );
127   positionWorld = (model * vec4(newWave, 1.0f)).xyz;
128 }
```

8.2 Shaders du modèle de Gerstner

8.2.1 Vertex shader du modèle de Gerstner simple

```

1 #version 460 core
2
3 layout (location = 0) in vec3 aPos;
4 layout (location = 1) in vec2 aTex;
5
6 uniform float time;
7 uniform float amplitude;
8 uniform float wavelength;
9 uniform float speed;
10 uniform float steepness;
11 uniform vec2 direction;
12
13 out vec3 fragPos;
14 out vec2 texCoord;
15
16 void main() {
17     float k = 2.0 * 3.14159 / wavelength;
18     float omega = speed * k;
19     float Q = steepness / (k * amplitude);
20     float phi = 0.0;
21
22     vec3 pos = aPos;
23     float theta = k * dot(direction, pos.xz) - omega * time + phi;
24
25     pos.x += Q * amplitude * direction.x * cos(theta);
26     pos.y += amplitude * sin(theta);
27     pos.z += Q * amplitude * direction.y * cos(theta);
28
29     fragPos = pos;
30     texCoord = aTex;
31
32     gl_Position = projection * view * model * vec4(pos, 1.0);
33 }
```

8.2.2 Vertex shader de la somme des vagues de Gerstner

```

1 #version 460 core
2
3 layout (location = 0) in vec3 aPos;
4 layout (location = 1) in vec2 aTex;
```

```

5 layout (location = 2) in vec3 aNormal;
6
7 // Colors vertices
8 out float height;
9 out vec3 pos;
10 out vec3 normal;
11 out vec2 tex;
12 out vec3 binormale;
13 out vec3 tangente;
14
15 // General variables
16 uniform mat4 model;
17 uniform mat4 view;
18 uniform mat4 projection;
19 uniform float time;
20
21 uniform float PI;
22 uniform float Amplitude_min;
23 uniform float Amplitude_max;
24 uniform float Amplitude_FBM;
25
26 uniform float g;
27 uniform float L_min;
28 uniform float L_max;
29 uniform float Steepness;
30 uniform float S;
31 uniform vec3 Direction;
32
33 uniform int nbVagues;
34 uniform int seed;
35 uniform int FBM;
36 uniform float Gain_A;
37 uniform float Gain_W;
38 uniform float L_FBM;
39
40 float rand(vec2 co) {
41     return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) *
42                 43758.5453);
43 }
44
45 vec3 randomDir(float seed) {
46     float angle = rand(vec2(seed, 0.0)) * 2.0 * PI;
47     float x = cos(angle);
48     float y = sin(angle);
49     return vec3(x, 0.0, y);
50 }
51 vec3 Add_Wave(vec3 Pos, float S, vec3 Direction, float time) {
52     vec3 newPos = aPos;

```

```

53    vec3 newWave = vec3(0.0f);
54    vec3 finalWave = vec3(0.0f);
55
56    float w;
57    float Q;
58    float L;
59
60    float wFBM = sqrt(g * ((2.0 * PI) / L_FBM));
61    float AFBM = Amplitude_FBM;
62
63    if(FBM == 0) {
64        for(int i = 0; i < nbVagues; i++) {
65            L = mix(L_min, L_max, rand(vec2(float(i), seed)));
66            float randomAmplitude = mix(Amplitude_min, Amplitude_max,
67                rand(vec2(float(i), seed)));
68            float randomPhase = rand(vec2(float(i) + 0.1, seed)) *
69                2.0 * PI;
70            vec3 randomDirection = normalize(reflect(Direction,
71                normalize(vec3(rand(vec2(float(i) + 0.2, seed)), rand(
72                    vec2(float(i) + 0.3, seed)), rand(vec2(float(i) +
73                    0.4, seed))))));
74            float randompeak = mix(Steepness, 0.5f, rand(vec2(float(
75                i), seed)));
76            w = sqrt(g * ((2.0 * PI) / L));
77            Q = randompeak / (w * randomAmplitude * nbVagues);
78            float phi_t = (S * (2.0 / L)) * time + randomPhase;
79            newWave.x += ((Q * randomAmplitude) * randomDirection.x *
80                cos((w * dot(randomDirection.xyz, aPos.xyz)) +
81                    phi_t));
82            newWave.z += ((Q * randomAmplitude) * randomDirection.z *
83                cos((w * dot(randomDirection.xyz, aPos.xyz)) +
84                    phi_t));
85            newWave.y = randomAmplitude * sin(w * dot(
86                randomDirection.xyz, aPos.xyz) + phi_t);
87            finalWave += newWave;
88        }
89    } else if(FBM == 1) {
90        for(int i = 0; i < nbVagues; i++) {
91            vec3 randomDirection = randomDir(float(i) + seed);
92            Q = Steepness / (wFBM * AFBM * nbVagues);
93            float phi_t = (S * (2.0 / L_FBM)) * time;
94            newWave.x += ((Q * AFBM) * randomDirection.x * cos((wFBM *
95                dot(randomDirection.xyz, aPos.xyz)) + phi_t));
96            newWave.z += ((Q * AFBM) * randomDirection.z * cos((wFBM *
97                dot(randomDirection.xyz, aPos.xyz)) + phi_t));
98            newWave.y = AFBM * sin(wFBM * dot(randomDirection.xyz,
99                aPos.xyz) + phi_t);
100           finalWave += newWave;
101           wFBM *= Gain_W;
102       }
103   }

```

```
88         AFBM *= Gain_A;
89     }
90 }
91
92 newPos += finalWave;
93 return newPos;
94 }
95
96 vec3 ComputeBinormale(vec3 Pos, float S, vec3 Direction, float time)
97 {
98     vec3 binormal = vec3(1.0, 0.0, 0.0);
99     vec3 finalWave = vec3(0.0f);
100    float wFBM = sqrt(g * ((2.0 * PI) / L_FBM));
101    float AFBM = Amplitude_FBM;
102
103    for(int i = 0; i < nbVagues; i++) {
104        vec3 randomDirection = randomDir(float(i) + seed);
105        float phi_t = (S * (2.0 / L_FBM)) * time;
106        float WA = wFBM * AFBM;
107        float SA = sin(wFBM * dot(randomDirection.xyz, Pos.xyz) +
108                      phi_t);
109        binormal.x -= randomDirection.x * WA * SA;
110        binormal.z -= randomDirection.z * WA * SA;
111        binormal.y = 1.0 - Steepness * WA * SA;
112        wFBM *= Gain_W;
113        AFBM *= Gain_A;
114    }
115
116
117    return normalize(binormal);
118 }
119
120 vec3 ComputeTangente(vec3 Pos, float S, vec3 Direction, float time)
121 {
122     vec3 tangent = vec3(0.0, 0.0, 1.0);
123     vec3 finalWave = vec3(0.0f);
124     float wFBM = sqrt(g * ((2.0 * PI) / L_FBM));
125     float AFBM = Amplitude_FBM;
126
127     for(int i = 0; i < nbVagues; i++) {
128         vec3 randomDirection = randomDir(float(i) + seed);
129         float phi_t = (S * (2.0 / L_FBM)) * time;
130         float WA = wFBM * AFBM;
131         float SA = sin(wFBM * dot(randomDirection.xyz, Pos.xyz) +
132                         phi_t);
133         tangent.x -= randomDirection.x * WA * SA;
134         tangent.z -= randomDirection.z * WA * SA;
135         tangent.y = 1.0 - Steepness * WA * SA;
136         wFBM *= Gain_W;
137         AFBM *= Gain_A;
```

```

133 }
134
135     return normalize(tangent);
136 }
137
138 vec3 ComputeNormal(vec3 Pos, float S, vec3 Direction, float time) {
139     vec3 normal = vec3(0.0, 1.0, 0.0);
140     vec3 finalWave = vec3(0.0f);
141     float wFBM = sqrt(g * ((2.0 * PI) / L_FBM));
142     float AFBM = Amplitude_FBM;
143
144     for(int i = 0; i < nbVagues; i++) {
145         vec3 randomDirection = randomDir(float(i) + seed);
146         float phi_t = (S * (2.0 / L_FBM)) * time;
147         float WA = wFBM * AFBM;
148         float CA = cos(wFBM * dot(randomDirection.xyz, Pos.xyz) +
149                         phi_t);
150         normal.x -= randomDirection.x * WA * CA;
151         normal.z -= randomDirection.z * WA * CA;
152         normal.y = 1.0 - Steepness * WA * CA;
153         wFBM *= Gain_W;
154         AFBM *= Gain_A;
155     }
156
157     return normalize(normal);
158 }
159
160 void main() {
161     vec3 accPos = Add_Wave(aPos, S, Direction, time);
162     vec3 accBin = ComputeBinormale(aPos, S, Direction, time);
163     vec3 accTan = ComputeTangente(aPos, S, Direction, time);
164     vec3 accNor = ComputeNormal(aPos, S, Direction, time);
165
166     vec3 newWave = accPos;
167     gl_Position = projection * view * model * vec4(newWave, 1.0f);
168
169     binormale = accBin;
170     tangente = accTan;
171     normal = accNor;
172     height = newWave.y;
173     pos = newWave;
174     tex = aTex;
175 }
```

8.3 Shaders d'éclairage / de rendu

8.3.1 Fragment shader de l'éclairage de Phong

```

1 #version 460 core
2
3 out vec4 FragColor;
4
5 in float height;
6 in vec3 pos;
7 in vec2 tex;
8 in vec3 normal;
9 in vec3 binormale;
10 in vec3 tangente;
11
12 uniform int Debug;
13 uniform vec3 lightPosition;
14 uniform vec3 lightColor;
15 uniform vec3 viewPosition;
16 uniform float ambientStrength;
17 uniform float diffuseStrength;
18 uniform float specularStrength;
19 uniform samplerCube skyboxTexture;
20
21 void main() {
22     vec3 colBase = vec3(0.6, 0.8, 0.98); // Cr te
23     vec3 colCreux = vec3(0.01, 0.36, 0.7); // Creux
24
25     float gradient = 0.8;
26     float mixF = smoothstep(gradient, gradient + 0.1, height);
27     vec3 baseColor = mix(colBase, colCreux, mixF);
28
29     // Phong lighting model
30
31     // Ambient
32     vec3 ambient = ambientStrength * lightColor;
33
34     // Diffuse
35     vec3 lightDir = normalize(lightPosition - pos);
36     float diff = max(dot(normal, lightDir), 0.0);
37     vec3 diffuse = diffuseStrength * diff * lightColor;
38
39     // Specular
40     vec3 viewDir = normalize(viewPosition - pos);
41     vec3 reflectDir = reflect(-viewDir, normal);
42     vec3 halfwayDir = normalize(lightDir + viewDir);
43     float spec = pow(max(dot(normal, halfwayDir), 0.0), 32);
44     vec3 specular = specularStrength * spec * lightColor;

```

```

45 // Fresnel effect
46 const float FresnelPower = 1.0;
47 const float FresnelBias = 0.1;
48 const float FresnelScale = 0.1;
49 vec3 fresnelNormal = normalize(vec3(normal.x * FresnelScale,
50     normal.y, normal.z * FresnelScale));
51 float fresnelFactor = pow(1.0 - dot(viewDir, fresnelNormal),
52     FresnelPower) + FresnelBias;
53 vec3 fresnelColor = fresnelFactor * lightColor;

54 // Final color computation
55 vec3 skyboxColor = texture(skyboxTexture, reflectDir).rgb;
56 vec3 finalColor = ambient + diffuse + specular + fresnelColor;
57 finalColor = mix(baseColor, finalColor, 0.7);
58 finalColor = mix(finalColor, skyboxColor, fresnelFactor);
59 finalColor = clamp(finalColor, 0.0, 1.0);

60 if(Debug == 0) {
61     FragColor = vec4(finalColor, 1.0);
62 } else if(Debug == 1) {
63     FragColor = vec4(pos * 0.5 + 0.5, 1.0);
64 } else if(Debug == 2) {
65     FragColor = vec4(tex, 0.0, 1.0);
66 } else if(Debug == 3) {
67     FragColor = vec4(normal * 0.5 + 0.5, 1.0);
68 } else if(Debug == 4) {
69     FragColor = vec4(binormale * 0.5 + 0.5, 1.0);
70 } else if(Debug == 5) {
71     FragColor = vec4(tangente * 0.5 + 0.5, 1.0);
72 }
73 }
74 }
```

8.3.2 Fragment shader de la skybox

```

1 #version 330 core
2
3 out vec4 FragColor;
4 in vec3 TexCoords;
5
6 uniform samplerCube skyboxTexture;
7
8 void main()
9 {
10     FragColor = texture(skyboxTexture, TexCoords);
11 }
```

8.3.3 Fragment shader du PBR

```

1 #version 460 core
2
3 out vec4 FragColor;
4
5 in float height;
6 in vec3 pos;
7 in vec2 tex;
8 in vec3 normal;
9 in vec3 binormale;
10 in vec3 tangente;
11
12 // PBR In
13 in vec3 TangentLightPos;
14 in vec3 TangentViewPos;
15 in vec3 TangentFragPos;
16 in vec4 positionWorld;
17 in vec3 o_positionWorld;
18 in vec3 o_normalWorld;
19
20 const float PI = 3.14159265359;
21
22 uniform samplerCube skyboxTexture;
23
24 // Functions
25
26 float DistributionGGX(vec3 N, vec3 H, float roughness)
27 {
28     float a = roughness*roughness;
29     float a2 = a*a;
30     float NdotH = max(dot(N, H), 0.0);
31     float NdotH2 = NdotH*NdotH;
32
33     float nom    = a2;
34     float denom = (NdotH2 * (a2 - 1.0) + 1.0);
35     denom = PI * denom * denom;
36
37     return nom / denom;
38 }
39
40 float GeometrySchlickGGX(float NdotV, float roughness)
41 {
42     float r = (roughness + 1.0);
43     float k = (r*r) / 8.0;

```

```

44
45     float nom    = NdotV;
46     float denom = NdotV * (1.0 - k) + k;
47
48     return nom / denom;
49 }
50
51 float GeometrySmith(vec3 N, vec3 V, vec3 L, float roughness)
52 {
53     float NdotV = max(dot(N, V), 0.0);
54     float NdotL = max(dot(N, L), 0.0);
55     float ggx2 = GeometrySchlickGGX(NdotV, roughness);
56     float ggx1 = GeometrySchlickGGX(NdotL, roughness);
57
58     return ggx1 * ggx2;
59 }
60
61 vec3 fresnelSchlick(float cosTheta, vec3 F0)
62 {
63     return F0 + (1.0 - F0) * pow(clamp(1.0 - cosTheta, 0.0, 1.0),
64                                   5.0);
65 }
66
67 uniform int Debug;
68
69 // Light
70 uniform vec3 ViewPos;
71 uniform float roughness;
72 uniform float ao;
73 uniform float metallic;
74 uniform vec3 albedo;
75
76 void main() {
77     vec3 N = normalize(o_normalWorld);
78     vec3 V = normalize(TangentViewPos - TangentFragPos);
79     vec3 L = normalize(TangentLightPos - TangentFragPos);
80     vec3 H = normalize(V + L);
81
82     vec3 F0 = vec3(0.04);
83     F0 = mix(F0, albedo, metallic);
84
85     vec3 envColor = texture(skyboxTexture, reflect(-V, N)).rgb;
86
87     float NDF = DistributionGGX(N, H, roughness);
88     float G = GeometrySmith(N, V, L, roughness);
89     vec3 F = fresnelSchlick(max(dot(H, V), 0.0), F0);
90
91     vec3 numerator = NDF * G * F;

```

```
91 float denominator = 4.0 * max(dot(N, V), 0.0) * max(dot(N, L),
92     0.0) + 0.0001;
93 vec3 specular = numerator / denominator;
94
95 vec3 kS = F;
96 vec3 kD = (1.0 - kS) * (1.0 - metallic);
97
98 float NdotL = max(dot(N, L), 0.0);
99
100 vec3 Lo = (kD * albedo / PI + specular) * envColor * NdotL;
101
102 vec3 ambient = vec3(0.03) * albedo * ao;
103 vec3 color = ambient + Lo;
104
105 if(Debug == 0) {
106     FragColor = vec4(color, 1.0);
107 } else if(Debug == 1) {
108     FragColor = vec4(pos, 1.0);
109 } else if(Debug == 2) {
110     FragColor = vec4(tex, 0.0, 1.0);
111 } else if(Debug == 3) {
112     FragColor = vec4(normal, 1.0);
113 } else if(Debug == 4) {
114     FragColor = vec4(binormale, 1.0);
115 } else if(Debug == 5) {
116     FragColor = vec4(tangente, 1.0);
117 }
118 }
```

CHAPITRE 9. Bibliographie

- [1] NVIDIA. *GPU Gems Chapter 1: Effective Water Simulation from Physical Models*. URL: <https://developer.nvidia.com/gpugems/part-i-natural-effects/chapter-1-effective-water-simulation-physical-models>.
- [2] Jerry Tessendorf. “Simulating ocean water”. In: *Simulating Nature: Realistic and Interactive Techniques* (2004). URL: https://peoplecomputing.clemson.edu/~jtessen/reports/papers_files/coursenotes2004.pdf.
- [3] R. Olajos. *Water Shader*. URL: https://fileadmin.cs.lth.se/cs/Education/EDAF80/seminars/2022/sem_4.pdf.
- [4] K. Lantz. *Ocean simulation part one: using the discrete Fourier transform*. URL: <https://www.keithlantz.net/2011/10/ocean-simulation-part-one-using-the-discrete-fourier-transform/>.
- [5] Oliver Staadt. “Real-Time Open Water Environments with Interacting Objects”. In: *ResearchGate* (). URL: https://www.researchgate.net/publication/221314832_Real-Time_Open_Water_Environments_with_Interacting_Objects.
- [6] P. García-Navarro et al. “The shallow water equations and their application to realistic cases”. In: *SpringerLink* (2018). URL: <https://link.springer.com/article/10.1007/s10652-018-09657-7>.
- [7] wHiteRabbiT i wHiteRabbiT. *Gerstner Waves*. URL: <http://whiterabbit-studio.com/article?art=1LvAi6AsIe-m0aASaPnHbyoge9tjntZcxvYhbll8JRnU>.
- [8] Jean-Marc Cieutat. “Modélisation physiquement réaliste de simulation d’entraînement maritime”. PhD thesis. Université de Bordeaux 1, 2003.
- [9] Jocelyn Fréchot. “Realistic simulation of ocean surface using wave spectra”. PhD thesis. LaBRI - Laboratoire bordelais de recherche en informatique, 2008.
- [10] Florian-Johannes Flügge. “Realtime GPGPU FFT Ocean Water Simulation”. PhD thesis. Technische Universität Hamburg-Harburg, 2008. URL: https://tore.tuhh.de/bitstream/11420/1439/1/GPGPU_FFT_Ocean_Simulation.pdf.
- [11] Khronos Group. *OpenGL*. URL: <https://www.khronos.org/opengl/>.
- [12] David Reid. *miniaudio*. URL: <https://miniaud.io/>.
- [13] Figure 1. <https://developer.nvidia.com/waveworks>.
- [14] Figure 4. https://fr.wikipedia.org/wiki/Houle_trochoïdale.