Rapport de Projet
Boids 3D

Elève ingénieur :
BEKKAR Fady Zakaria

Tuteurs académiques :
M. BIRI Venceslas
M. FOUCHY Jules

IMAC 2 2022-2023

2 juin 2023

Table des matières

1 Introduction

Projet 2D Projet 3D

1.1 Présentation du projet

Durant ce projet semestriel de synthèse d'image qui vise à modéliser des boids en 3D selon certaines conditions données. Afin de suivre, le plus rigoureusement possible les consignes, on parlera des choix algorithmiques et des fonctionnalités du projet. On explorera tout d'abord l'architecture du projet en particulier les implémentation des différentes classes. Par la suite on explicitera certaines implémentations en passant que ce soit l'éclairage ou le mouvement des objets dans la scène. Enfin on abordera les quelques fonctionnalités du projet.

1.2 Architecture du projet

1.2.1 Dossier d'entête

- **main.h**
- **camera.h**
- **ebo.h**
- **texture.h**
- **shaderclass.h**
- **stb.h**
- **model.h**
- **mesh.h**

1.2.2 Dossier source

- **main.cpp** : Ce fichier contient la fonction principale (main) du programme. Il est responsable du point d'entrée de l'application.
- **camera.cpp** : Ce fichier contient les définitions et les implémentations des fonctions liées à la caméra. Il gère les mouvements de la caméra et les transformations de la vue.
- **glad.c** : Ce fichier contient les définitions et les implémentations des fonctions nécessaires pour charger les pointeurs de fonction de l'API OpenGL.
- **ebo.cpp** : Ce fichier contient les définitions et les implémentations des fonctions liées aux Element Buffer Objects (EBOs). Les EBOs sont utilisés pour spécifier l'ordre des vertices lors du rendu.
- **texture.cpp** : Ce fichier contient les définitions et les implémentations des fonctions liées aux textures. Il gère le chargement et l'application des textures sur les objets 3D.
- **shaderclass.cpp** : Ce fichier contient les définitions et les implémentations des fonctions liées aux shaders. Il gère la compilation, le lien et l'utilisation des shaders OpenGL.
- **stb.cpp** : Ce fichier contient les définitions et les implémentations des fonctions de la bibliothèque STB utilisées pour le chargement d'images et de textures.

— **model.cpp** : Ce fichier contient les définitions et les implémentations des fonctions liées aux modèles 3D. Il gère le chargement et le rendu des modèles 3D à partir de fichiers.

— **mesh.cpp** : Ce fichier contient les définitions et les implémentations des fonctions liées aux maillages 3D. Il gère les données des vertices, des normales et des textures des maillages.

1.2.3 Dossier shaders

— **default.frag** : Ce fichier contient le code source du fragment shader par défaut.

— **default.vert** : Ce fichier contient le code source du vertex shader par défaut.

— **light.frag** : Ce fichier contient le code source du fragment shader pour l'éclairage.

— **light.vert** : Ce fichier contient le code source du vertex shader pour l'éclairage.

— **cube.frag** : Ce fichier contient le code source du fragment shader pour le cube qui englobe la simulation.

— **cube.vert** : Ce fichier contient le code source du vertex shader pour le cube qui englobe la simulation.

1.2.4 Dossier texture

À cet emplacement vous trouverez les différentes images qui servent à des textures pour l'application.

2 Modèles 3D

2.1 Les boids

Tout d'abord au début du projet avant de créer une Model et un loader j'avais pour idée des boids représentants un poisson et un requin. Ce sont deux modèles que vous pourrez retrouver sur mon git au format .blend. Les images ci-dessous représente le boid qui aurait été le boid prédateur tant dis que celle d'après, les poissons sont les boids les plus nombreux.

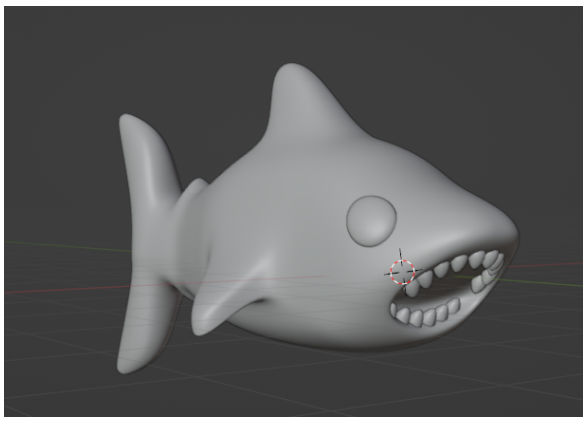

FIGURE 1 – Requin

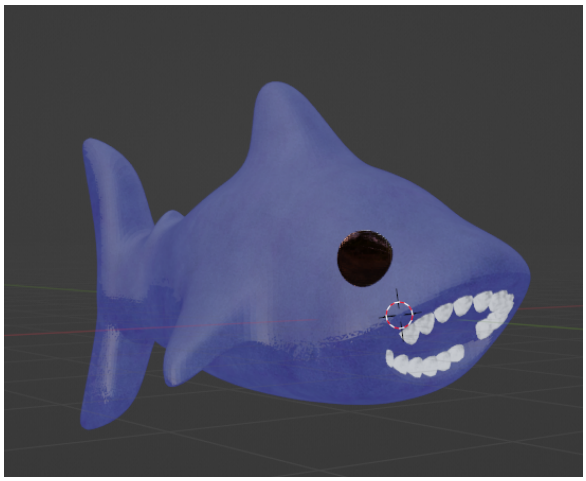

FIGURE 2 – Requin avec texture

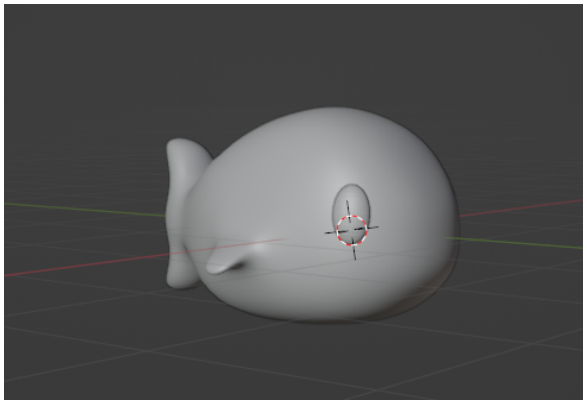

FIGURE 3 – Poisson

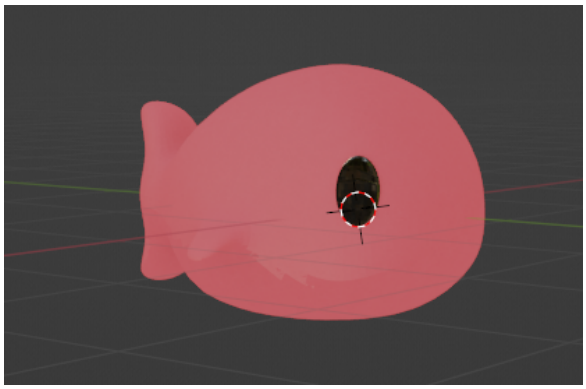

FIGURE 4 – Poisson avec texture

Ayant un loader me permettant de load seulement des gltf 1.0, on utilisera un modèle de lapin trouver sur internet.

2.1.1 Implémentation des boids

Les algorithmes relatifs aux boids ne sortent pas de l'ordinaire. Ils ont été codé par trois comportements importants qui sont la séparation, l'alignement et la cohésion. Un des obstacles à leur développement a été de set correctement leur position et notamment gérer les collisions avec les bords. Ce problème était plus abordable en 2D car dès lors où on a retranscrit les algorithmes en 3D on se rend compte des problèmes lier aux vélocités. En effet, on a remarqué à la fin que les vélocités ne s'incrémentent que positivement, ce qui fait qu'à chacune des rencontres avec les parois l'objet revient dans le sens dans lequel il est venu. Ainsi, l'hypothèse première pour régler ce problème serait d'incrémenter les vitesses positivement et négativement quand il le faut, dans les méthodes de comportement des boids. Ce probème constitue une des différences majeures avec le projet en 2D, malheuresement je n'ai pas su le régler à temps.

2.2 L'arpenteur

Même principe que les boids, j'avais modélisé un arpenteur original qui pourrait aller avec l'idée générale de la scène :

3 Fonctionnalités

3.1 Les lumières

3.1.1 Direct light

La fonction 'direcLight()' calcule l'éclairage d'une surface en prenant en compte les principes de l'éclairage ambiant, diffus et spéculaire.

- L'éclairage ambiant représente la lumière ambiante présente dans l'environnement. C'est une lumière uniforme qui éclaire la surface de manière égale de tous les côtés, indépendamment

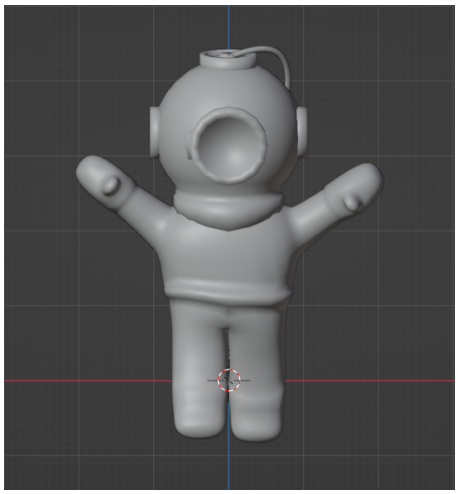

FIGURE 5 – Plongeur

FIGURE 6 – Plongeur avec texture

de la direction de la source de lumière. Dans la fonction, la valeur ‘ambient‘ représente l'intensité de cet éclairage ambiant.

- L'éclairage diffus est causé par la diffusion de la lumière sur une surface rugueuse. Lorsque la lumière frappe une surface, elle se réfléchit dans toutes les directions, créant ainsi une lumière diffuse. La quantité de lumière diffusée dépend de l'angle entre la normale de la surface et la direction de la source de lumière. Si l'angle est élevé, plus la surface est éclairée, et si l'angle est faible, la surface reçoit moins de lumière diffuse. La fonction calcule l'éclairage diffus en prenant le produit scalaire entre la normale de la surface et la direction de la lumière, et en s'assurant que la valeur est positive.

- L'éclairage spéculaire représente les reflets brillants qui se produisent sur une surface lisse lorsqu'une source de lumière est réfléchie de manière réfléchie. Cela crée des points de lumière brillants qui changent en fonction de l'angle de vue. La quantité de lumière spéculaire dépend de l'angle entre la direction de réflexion de la lumière et la direction de la vue de la caméra. Plus l'angle est petit, plus la lumière spéculaire est intense. La fonction calcule l'éclairage spéculaire en utilisant la réflexion de la direction de la lumière par rapport à la normale de la surface, puis en prenant le produit scalaire entre cette direction de réflexion et la direction de la vue.

En combinant ces trois composantes d'éclairage (ambiant, diffus et spéculaire) avec les textures de la surface et la couleur de la lumière, la fonction ‘direcLight()‘ permet de calculer la couleur finale de la surface éclairée. Cela donne un rendu réaliste en simulant les interactions de la lumière avec la surface, en prenant en compte les propriétés de diffusion et de réflexion.

3.1.2 Spot light

La fonction ‘spotLight()‘ simule l'éclairage d'une surface par une lampe projecteur (spotlight). Une lampe projecteur émet de la lumière dans une direction spécifique.

- L'éclairage ambiant représente la lumière ambiante présente dans l'environnement qui est réfléchie uniformément sur toutes les surfaces. Cela ajoute une certaine quantité de luminosité à la surface, indépendamment de la direction de la source de lumière. Dans la fonction ‘spotLight()‘, l'éclairage ambiant est défini par la variable ‘ambient‘.

- L'éclairage diffus simule la diffusion de la lumière par une surface rugueuse. Il dépend de l'angle entre la normale de la surface et la direction de la source de lumière. Plus l'angle est proche de 90 degrés, plus l'éclairage diffus est intense. Dans la fonction 'spotLight()', la composante diffus est calculée en prenant le produit scalaire entre la normale de la surface et la direction du projecteur lumineux ('lightDirection'). La valeur maximale de ce produit scalaire est utilisée pour déterminer l'intensité de l'éclairage diffus.

- L'éclairage spéculaire simule la réflexion de la lumière par une surface lisse et brillante. Il dépend de l'angle entre la direction de réflexion (calculée en utilisant la direction opposée de la lumière réfléchie) et la direction de la vue. Plus l'angle entre ces deux directions est petit, plus l'éclairage spéculaire est intense. Dans la fonction 'spotLight()', la composante spéculaire est calculée en utilisant le modèle de réflexion de Phong. La quantité de lumière spéculaire est déterminée en élevant à la puissance une valeur basée sur le produit scalaire entre la direction de la vue et la direction de réflexion ('specAmount').

On calcule aussi l'intensité en fonction de l'angle entre la surface éclairée et le centre du cône de lumière du projecteur. Cette mesure d'angle est réalisée en calculant le produit scalaire entre la direction opposée de la lumière ('-lightDirection') et une direction de référence (dans ce cas, une direction vers le bas). En comparant cet angle à des seuils prédéfinis ('outerCone' et 'innerCone'), une valeur d'intensité ('inten') est déterminée. Cette valeur d'intensité permet de contrôler l'atténuation de la lumière en fonction de sa position par rapport au cône de lumière. Les valeurs d'intensité sont clippées entre 0 et 1 pour assurer une plage valide.

En combinant les éclairages ambiant, diffus et spéculaire avec les textures de la surface et la couleur de la lumière, la fonction 'spotLight()' calcule la couleur finale de la surface éclairée par la lampe projecteur.

3.1.3 Profondeur

On utilise majoritairement deux fonctions, présentes dans le fragment shader pour donner des impressions de profondeur à la scène. Elles sont :

— **float linearizeDepth(float depth)** est utilisée pour convertir une valeur de profondeur normalisée en une valeur de profondeur linéaire. Elle utilise une formule basée sur la projection perspective pour effectuer cette conversion.

— **float logisticDepth(float depth, float steepness, float offset)** utilise la fonction linearizeDepth pour obtenir une valeur de profondeur linéaire, puis applique une transformation logistique pour ajuster cette valeur. La transformation logistique permet de contrôler la plage de valeurs de profondeur pour obtenir des effets spécifiques. Les paramètres steepness et offset ajustent la pente et le décalage de la courbe logistique respectivement, permettant de modifier l'intensité de l'effet.

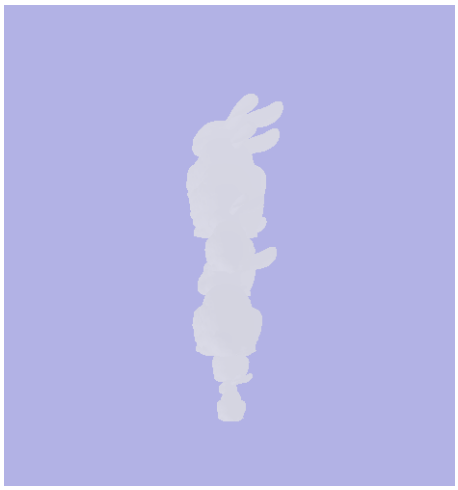

FIGURE 7 – depth buffer eloignée

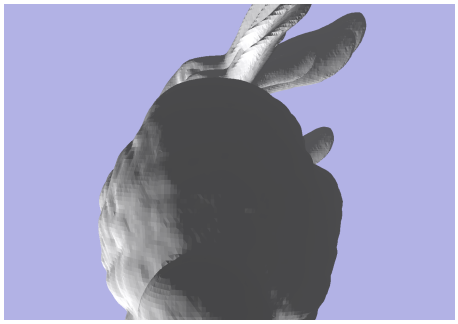

FIGURE 8 – depth buffer proche

3.1.4 Transparence

Vous remarquerez que le cube est transparent, étant donné qu'il n'a pas de texture il me semblait plus naturel de le rendre transparent afin de voir les boids dans un environnement restreint clairement. On le réalise simplement en jouant sur la variable uniform alpha du shader correspondant au cube.

4 Conclusion

En conclusion ce projet a aidé trop appréhender ce qui m'a empêché de clairement voir mes limites. Néanmoins, il m'a permis d'apprendre assez de choses en programmation comme en OpenGL en ce qui concerne le rendement de scène en proposant éclairages multiples et des algorithmes variés,