

Projet de Synthèse d'Images et de Reconnaissance des Formes - Rapport

Denis Casteran - Cetre Cyril
SCIA 2018

rendu le 18 Janvier 2018

Introduction

Voici le rapport faisant état de notre travail sur le projet de synthèse d'image de 5ème année à EPITA. Le sujet était au choix parmi plusieurs projets de rendu. Nous avons choisi de partir sur un rendu d'eau avec son environnement, avec l'aide d'OpenGL.

1 Le projet WEWOR (Wildlife and Enhanced Water OpenGL Renderer)

1.1 Le sujet

Nous sommes partis du projet initial d'un rendu d'eau avec l'aide d'Opengl. Nous nous sommes rapidement rendu compte qu'afin de pouvoir tester un rendu d'eau propre, nous nous devions d'avoir un environnement minimal afin de pouvoir tester la réflexion et la réfraction. Nous avons donc fixé pour objectif d'avoir les éléments suivants :

- Un terrain généré à partir d'une heightmap.
- Un plan d'eau qui permettra d'effectuer le Water Renderering. La partie principale du travail de rendu graphique s'effectue dans sur ce plan.
- Une skybox qui nous permettra d'avoir un rendu esthétique et de travailler un peu plus la réflexion.

Nous sommes dans l'ensemble parvenus à obtenir tous ces éléments. Le rendu d'eau contient un certain nombre de techniques d'eau utilisé pour simuler une surface liquide que nous présenterons étape par étape dans le rapport ci-dessous.



FIGURE 1: le water rendering est très utilisé dans les jeux vidéos

1.2 Les difficultés du projet

Les difficultés de ce projet étaient à la fois techniques et mathématiques. Une difficulté technique majeure fut la prise en main d'OpenGL. Bien qu'étant un outil très puissant permettant d'effectuer des rendus graphique 2D et 3D, sa prise en main nécessite un temps d'adaptation et d'apprentissage très important. De plus, OpenGL n'a eu de cesse d'évoluer si bien que tous les tutoriels utilisant une version d'OpenGL inférieure à OpenGL3 sont complètement obsolètes.

Cette obsolescence a rendu difficile la recherche de sujets récents traitant de notre problématique. Enfin, nous avons dû travailler également avec des contraintes de performances, puisque le matériel dont nous disposions n'était pas équipé de cartes graphiques dédiée.

1.3 Les technologies utilisées

Nous avons choisi d'utiliser le langage C++, pour ses performances, sa compatibilité avec OpenGL ainsi que notre familiarité avec le langage. Nous avons utilisé également FreeImage pour lire nos images ainsi que GLAD comme extension d'OpenGL. Le développement a été effectué sous Linux et est compatible avec Ubuntu et ArchLinux.

2 OpenGL

OpenGL (Open Graphics Library) est une bibliothèque graphique très complète qui permet aux programmeurs de développer des applications 2D, 3D assez facilement. De nombreux jeux, comme QuakeIII utilisent OpenGL pour leur rendu graphique.

Nous avons utilisé de nombreuses extensions à Opengl, que nous allons détailler ci-dessous. Elles sont nécessaire à la compilation du projet.

2.1 Les Extensions

GLFW

GLFW est une bibliothèque Open Source multi-plateforme pour le développement OpenGL. Il fournit une API simple pour créer des fenêtres, des contextes et des surfaces, en recevant des entrées et des événements.

GLFW est écrit en C et supporte nativement Windows, macOS et de nombreux systèmes de type Unix utilisant le système X Window, tels que Linux et FreeBSD.

C'est grâce à cette bibliothèque que nous faisons l'interface entre l'utilisateur et la machine ainsi que l'affichage final.

GLAD

Glad produit des loader pour OpenGl adaptés à nos besoins basés sur les spécifications officielles du Khronos SVN. Il génère les fonctions openGL dont nous auront besoin au cours de ce projet.

FreeImage

Contrairement aux autres modules, FreeImage n'est pas directement lié à OpenGL. FreeImage est un projet de bibliothèque Open Source s'adressant aux développeurs qui souhaitent travailler sur les formats d'images populaires tels que PNG, BMP, JPEG, TIFF et autres, selon les besoins des applications graphiques d'aujourd'hui. Nous l'utilisons pour loader notre heightmap et lire son contenu.

2.2 Les concepts principaux d'OpenGL

Ce rapport n'a pas vocation à être un tutoriel complet à OpenGL, et ne serait certainement pas aussi bon que ceux déjà présent sur internet. Nous ne saurions que conseiller <https://learnopengl.com/> pour ceux souhaitant s'initier à l'OpenGL moderne. C'est une série de tutoriels bien conçus permettant d'appréhender les concept d'OpenGL en douceur.

Nous allons donc citer ici les concepts les plus importants indispensable à la compréhension de notre code OpenGL.

Les shaders

Un shader est simplement un programme exécuté non pas par le processeur mais par la carte graphique. Nous en utilisons deux types :

Vertex Shader : C'est l'étape qui va nous permettre soit de valider les coordonnées de nos sommets, soit de les modifier. Cette étape prend un vertex à part pour travailler dessus. S'il y a 3 vertices (pour un triangle) alors le vertex shader sera exécuté 3 fois.

Fragment Shader : C'est l'étape qui va définir la couleur de chaque pixel de la forme délimitée par les vertices. Par exemple, si vous avez défini un rectangle de 100 pixels par 50 pixels, alors le fragment shader se chargera de définir la couleur des 5000 pixels composant le rectangle.

Les shaders sont développés dans un langage réservé aux shaders : le GLSL.

Les vertex buffer objects (vbo)

Un vbo est un objet OpenGL qui contient des données relatives à un modèle 3D comme les vertices, les coordonnées de texture, les normales (pour les lumières), ... Sa particularité vient du fait que les données qu'il contient se trouvent non pas dans la RAM mais directement dans la carte graphique.

Un Vertex Buffer Object est donc une zone mémoire (buffer) appartenant à la carte graphique dans laquelle on peut stocker des données.

Les vertex array objects (vao)

Les Vertex Array Object (ou VAO) sont des objets OpenGL relativement proches des Vertex Buffer Object (VBO) dans le sens où il permettent de stocker des informations sur la carte graphique. Les vao, liés à une vbo, appor-

teront les informations nécessaire pour gérer la donnée dans ce même vbo et permettrons de gérer les données dans les shaders.

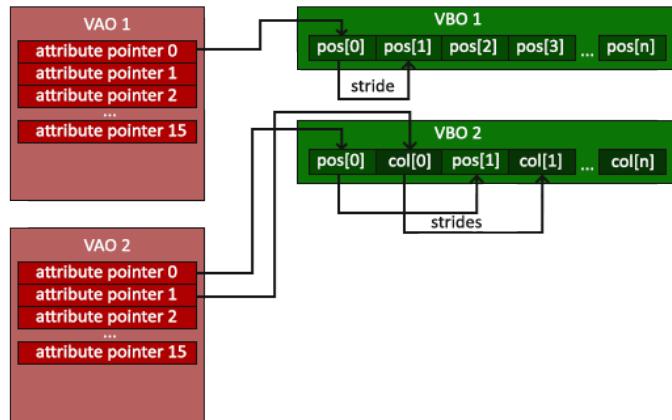


FIGURE 2: Schématisation du fonctionnement vao/vbo

2.2.1 Les textures

Une texture est une image 2D (bien qu'il existe des textures 1D et 3D) utilisée pour ajouter des détails à un objet. Bien que l'on puisse imaginer qu'une texture ne serve que pour afficher "l'habillage" du polygone, en réalité de très nombreuses opérations peuvent être travaillées sur les textures. Par exemple, une majeure partie des calculs du rendu de l'eau de ce projet sont effectués dans le fragment shader, qui travail directement sur les textures.

2.2.2 Les clip planes

Le clip plane est un mécanisme fourni par OpenGL qui permet de fournir des plans supplémentaires, par rapport auxquels la géométrie rendue sera découpée.

Ceci nous est utile pour éviter des réflexions inutiles ainsi qu'optimiser les performances pour ne pas avoir à rendre trois fois l'intégralité de la carte.

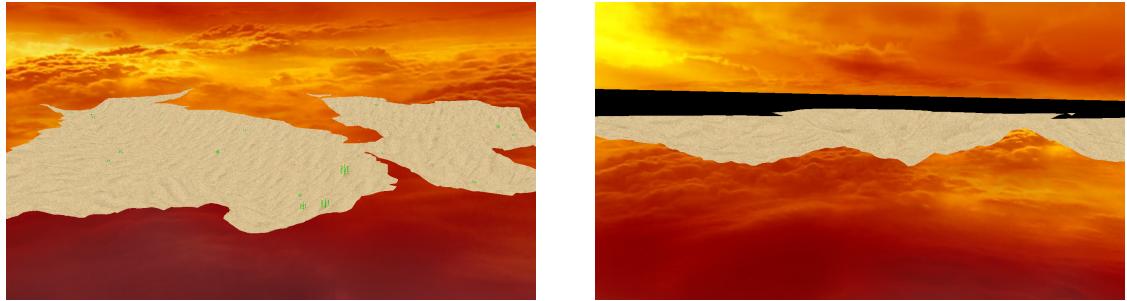


FIGURE 3: example of clip plane rendering under the water surface

Les Frame Buffer Objects (fbo)

Les FBO permettent de faire un rendu intermédiaire offscreen sans avoir à l'afficher à aucun moment. Ceci nous est utile pour rendre par exemple la réfraction ou la réflexion qui utilisent tous deux un fbo séparé.

3 Le rendu du terrain

Afin de parvenir à un rendu d'eau qui puisse être testé, il nous a déjà fallu créer une carte composée d'un terrain et de hauteurs afin de pouvoir s'assurer du bon fonctionnement de notre eau. Ce fut donc la première étape de notre travail.

3.1 La heightmap

Une heightmap est un moyen simple d'obtenir une information sur une hauteur. Chaque pixel de l'image correspond à un point de l'espace, la couleur du pixel code pour l'altitude du point. Par convention, un point d'altitude maximale sera blanc, un point d'altitude minimale noire.

Une fois lue, nous pouvons stocker dans un vbo l'ensemble des coordonnées de la map dans des points en 3 dimension, avec la coordonnée y la hauteur récupérée comme information dans la heightmap. Nous pouvons ensuite lier plusieurs textures à ce shader afin d'être paré à rendre plusieurs couleurs dans le fragment shader.



FIGURE 4: La heightmap avec une couleur unie.

Pour le rendu, le vertex shader est très simple, et place simplement le point là où il est sensé se trouver en fonction de la position de la caméra. Nous avons

simplement un clipboard qui nous permet de ne pas rendre certaines parties du terrain dans les frame buffer objects.

Pour le fragment shader, nous avons simplement plusieurs textures et nous affichons une texture différente en fonction de la hauteur des coordonnées du point. Afin d'effectuer une transition qui ne soit pas trop abrupte. Nous avons un intermédiaire entre nos deux textures qui nous permet de mixer les deux textures sur une portion se trouvant entre les deux.

Le résultat ainsi que le rendu de la hauteur peut-être modifié à l'aide de coefficients.



FIGURE 5: La heightmap après application des textures en fonction des hauteurs

3.2 La skybox

Afin d'embellir le rendu et de nous permettre de représenter l'horizon, le ciel et tout ce qui se trouve au final autour de notre rendu, nous avons besoin de la skybox.

Afin d'obtenir un rendu de ciel et de décor réaliste, nous n'avons au final d'uniquement besoin que d'une chose : que notre rendu soit englobé dans une boîte de six faces. Toutes les faces du cube seront dirigées vers l'intérieur car toute la scène sera contenue dans celui-ci. Chacune des faces du cube correspondra aux vues du Nord, Sud, Est, Ouest de la caméra ainsi que le Haut et le Bas. La texture qui sera appliquée sur chacune d'elle correspondra à ce que

l'on pourrait voir à une distance infinie, l'horizon.

Le principe est donc d'avoir un fond d'écran qu'il est impossible d'atteindre qui permet d'ajouter une vrai touche artistique au rendu. Comme nous le verrons par la suite, une skybox associée au rendu de l'eau permet de très beaux effets.



FIGURE 6: Un exemple de nos skybox

3.3 Les modèles

3.3.1 Construction des modèles

Dans un premier temps, les fichiers modèles sont chargés grâce à une bibliothèque externe. Nous avons fait le choix d'utiliser Assimp qui est une bibliothèque classique permettant d'importer des modèles. Le chargement de modèles avec leurs textures est donc fait avec Assimp.

Une fois le modèle chargé, il faut le charger et le traduire en une classe qui pourra être affiché avec OpenGL. La classe modèle correspond à cette classe, elle contient des mesh qui elles-mêmes contiennent les vertices qui seront affichés avec OpenGL.

Les différents objets que nous afficheront ne sont chargés qu'une fois à linitialisation de lapplication, les instances de cette classe correspondent donc à un template. Les différentes instances de chacun de ces templates est fait dans des parties différentes expliqués dans les parties suivantes.

3.3.2 Arbres et algues

Deux types de modèles statiques sont présent :

Sur la terre, il y a des arbres. Ces arbres de taille aléatoire se répartissent de manière aléatoire sur le terrain à la surface. La quantité d'arbre peut être modifiée en changeant la valeur de la macro TREE_PROBA .

Sous l'eau, il y a des algues. Ces algues de taille aléatoire se répartissent également de manière aléatoire sur le terrain sous la surface de l'eau. La quantité d'algues peut aussi être modifiée en changeant la valeur de la macro SEA-WEED_PROBA.

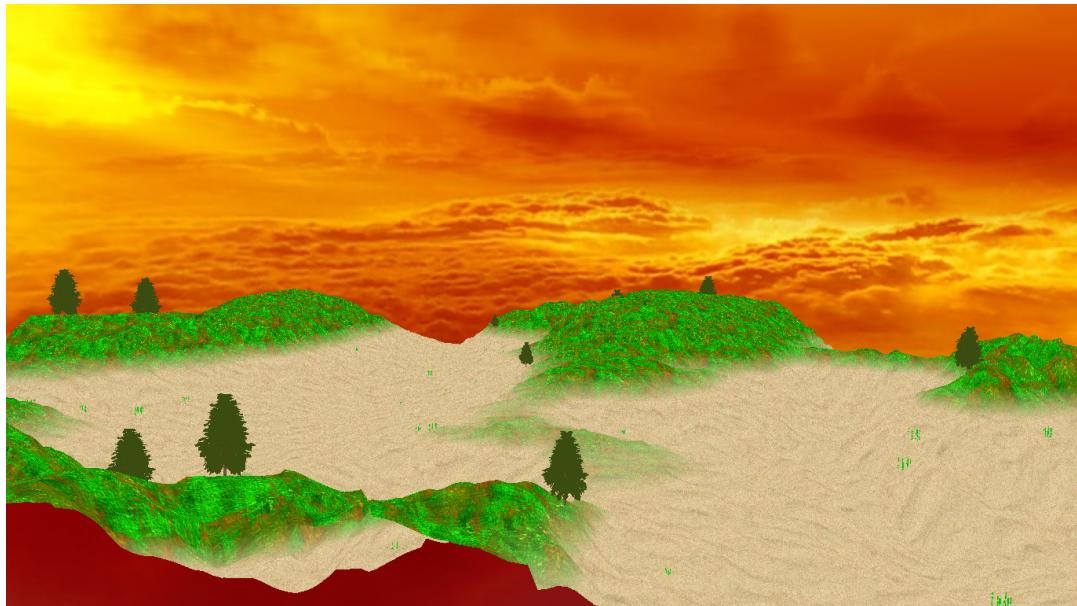


FIGURE 7: Le rendu des arbres et des algues

3.3.3 Poissons

Le projet contient également un modèle dynamique. Ce modèle est représenté par un poisson. A l'initialisation, des poissons de taille aléatoire sont positionnés aléatoirement sous la surface de l'eau. Ils prennent tous une direction aléatoire puis font des aller retours simples. La quantité de poisson masque le comportement naïf des poissons. La quantité de poissons peut évidemment être modifiée en changeant la valeur de la macro FISH_PROBA.



FIGURE 8: Les poissons

4 Le rendu de l'eau

Contrairement à ce que l'on pourrait croire, nul besoin de déplacer les vertices afin d'avoir un rendu d'eau réaliste. En effet, la majorité des effets de rendu d'eau se font sur la texture, ce qui signifie que notre plan d'eau n'est au final qu'un rectangle parfaitement plat !

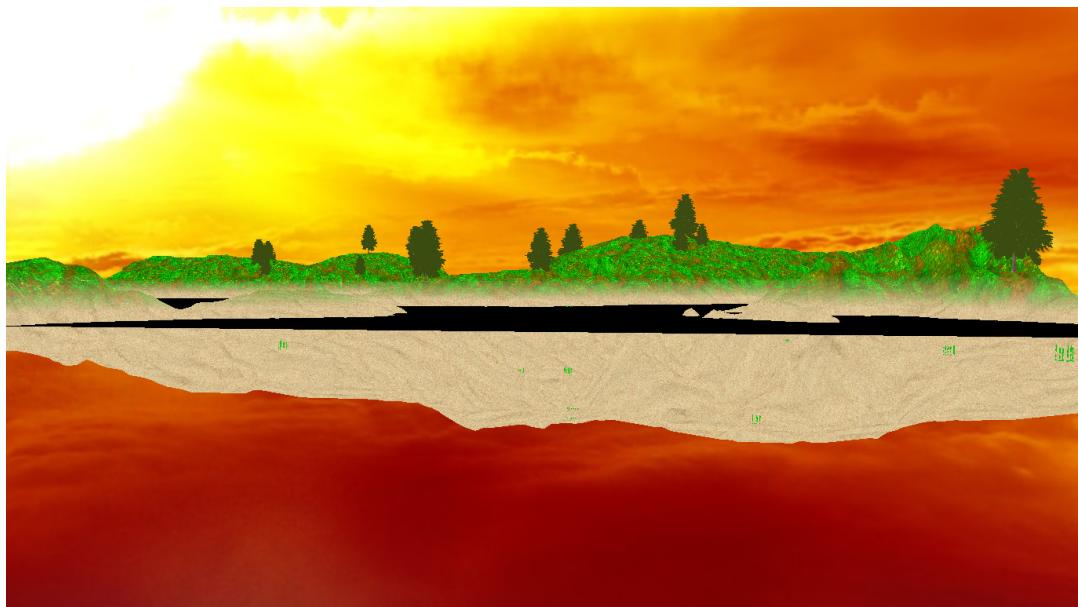


FIGURE 9: Le rendu d'eau sans aucune texture

4.1 La réfraction/réflexion

La première étape dans le rendu de l'eau (et non la moindre) est de pouvoir réfléchir le fond de l'eau et réfléchir le décor en face. Le principe est de rendre sur le plan d'eau deux choses : le fond de l'eau qui est réfléchie et le décor en face qui est réfléchie à la surface de l'eau.

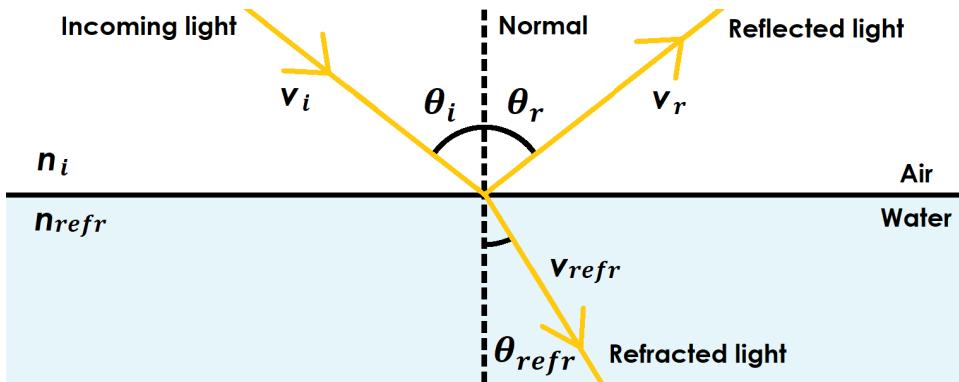


FIGURE 10: Schématisation du fonctionnement vao/vbo

Afin de se faire, c'est à cet endroit que nous allons utiliser les frame buffer objects. Dans un premier fbo, nous allons rendre la réfraction qui est simplement ce que nous voyons sous la surface de l'eau.

Pour la réflexion, nous devons avoir un nouveau fbo, mais cette fois-ci nous devons effectuer des opérations de caméra avant de rendre la scène. En effet, nous devons à la fois inverser le pitch de la caméra, qui est sa rotation selon l'axe y et translater la caméra d'une valeur fixe pour la placer sous le plan d'eau. Ceci va nous permettre de récupérer l'image inversée de notre plan et ainsi l'image réfléchie.

Dans chacun des plans, nous avons ajouté un clip plane, qui nous permet de ne pas rendre la totalité de la carte dans les fbo. Ceci nous permet également d'éviter des bugs avec des éléments qui ne devraient pas être réfléchis qui le sont quand même.

Enfin, Une fois en possession des deux textures, nous pouvons dans le buffer principal rendre l'eau. Il suffit de mixer la texture de réfraction et la texture de réflexion obtenue précédemment.

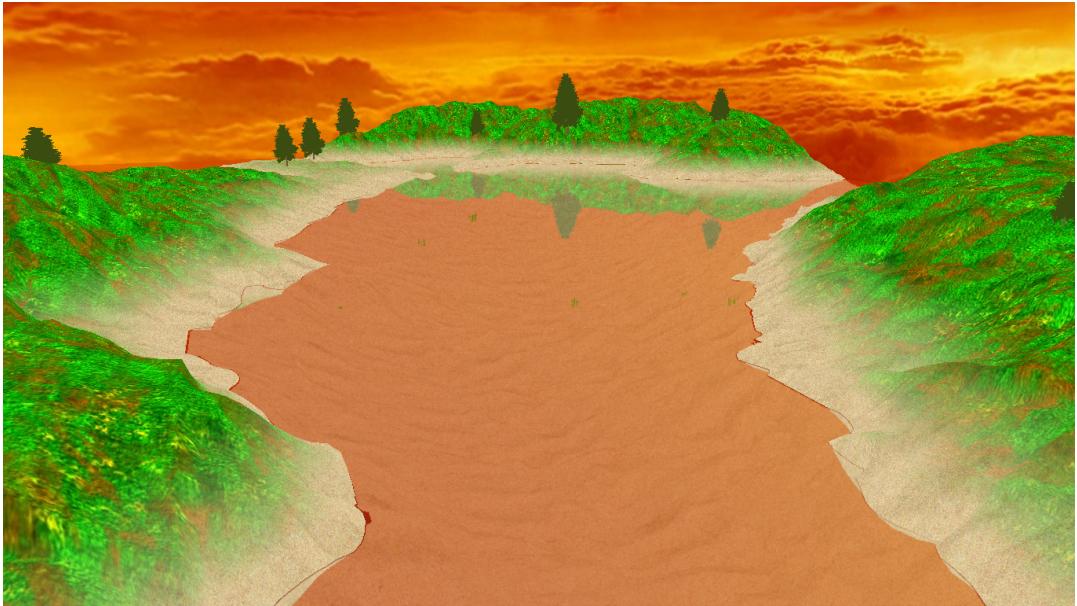


FIGURE 11: Le plan d'eau avec une simple réflexion/réfraction

4.2 La DuDv map

Une dudv est une texture composée de pixels rouge et vert d'intensité variables. Cette texture, trouvable très facilement sur internet, va nous permettre de représenter la distorsion de notre eau. Nous allons mapper la dudv map exactement comme pour une texture normale. La seul différence est qu'au lieu d'afficher cette dudv map, nous allons simplement utiliser les valeurs rouges et vertes de chaque partie de la texture.

Afin d'obtenir une distorsion réaliste et pas comprise seulement entre 0 et 1 (car chaque couleur de la dudv map est exprimée par un float), nous voudrions un chiffre compris entre -1 et 1. Pour se faire une astuce assez simple consiste à effectuer le calcul suivant :

$$val = val * 2.0 - 1.0;$$

Nous avons de nombreux facteurs qui nous permettent de diminuer ou d'augmenter le degré de distorsion, ou encore la vitesse à laquelle elle est réfléchie.

Un dernier élément à prendre en compte est de contraindre les valeurs des points à toujours se trouver strictement supérieur à 0 et strictement inférieur à 1. Ne pas faire cette opération cause des bugs d'animation sur les bords de l'écran.

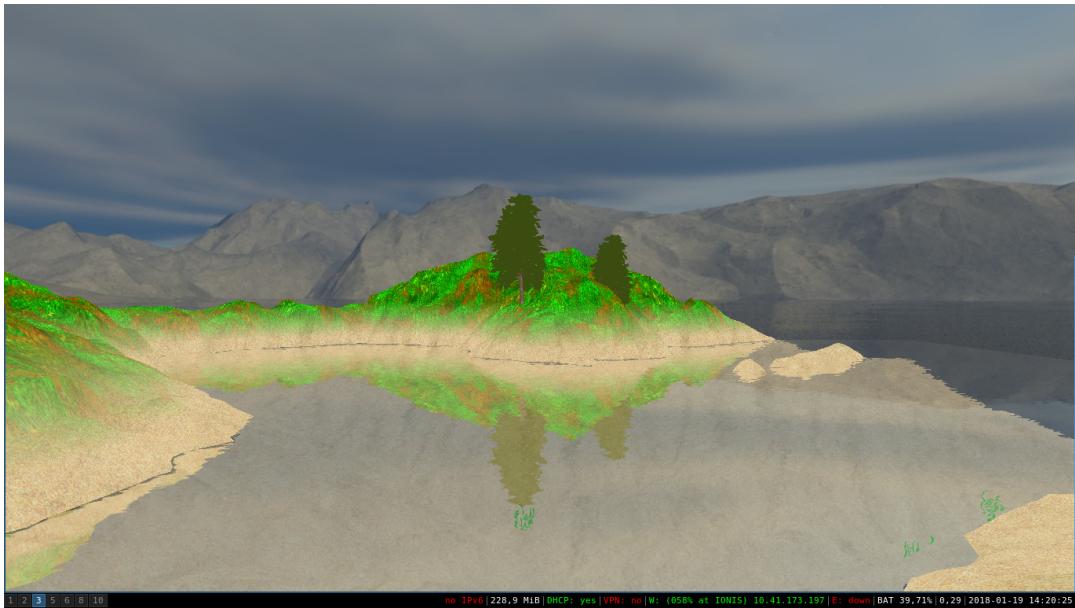


FIGURE 12: L'ajout de distorsion de la surface grâce à une dudv map

4.3 Le coefficient de Fresnel

Dans cette partie, il nous a fallu implémenter un phénomène physique bien connu. Jusqu'à maintenant, nous mélangeions réflexion et réfraction avec un coefficient égal pour rendre notre texture. Or il s'avère qu'en réalité la quantité de lumière réfléchie dépend de l'angle avec le rayon incident.

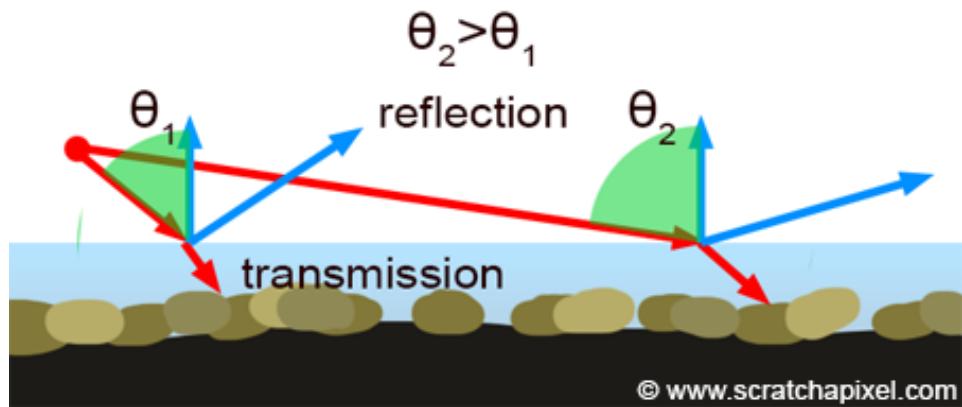


FIGURE 13: Lorsque l'angle entre la caméra et la normale à la surface est grand la quantité de lumière réfléchie est plus importante

Ce phénomène, représenté en physique par les équations de Fresnel, est ce que nous avons implémenté afin d'essayer de nous approcher d'un rendu

réaliste. Dans ce projet, nous avons donc calculé le coefficient par un produit scalaire de la normale au plan d'eau et du vecteur représentant la caméra. Nous avons ensuite remplacé le coefficient de mixage entre réflexion et réfraction arbitrairement à 0.5 par le résultat du produit scalaire.

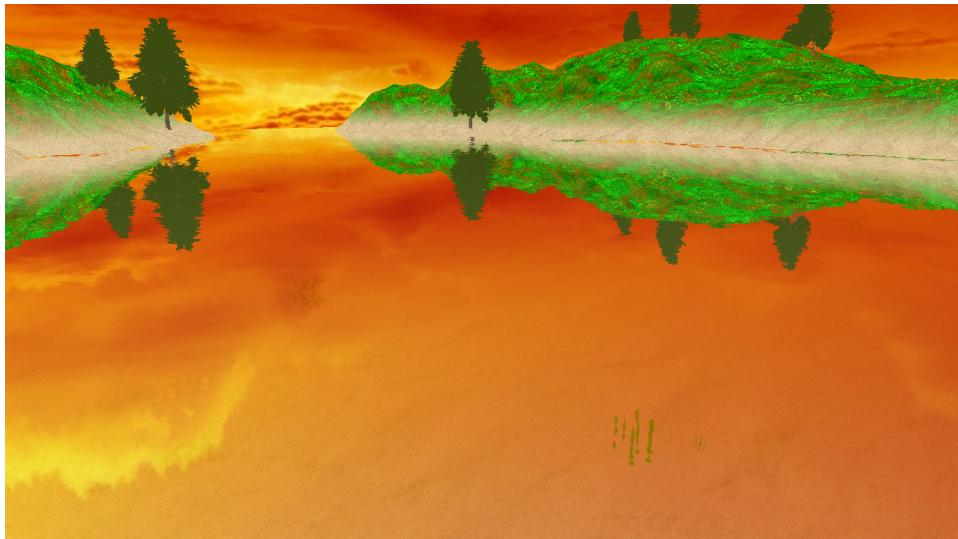


FIGURE 14: Lorsque l'angle entre la caméra et la normale à la surface est grand la quantité de lumière réfléchie est plus importante

4.4 L'amélioration du rendu de profondeur

Le principe ici est de prendre en compte la quantité d'eau se trouvant entre le fond de l'eau et la position de la où se trouve la caméra. Pour cela nous allons utiliser un depth buffer permettant d'apprécier la profondeur du bassin. Ce depth buffer se trouve dans le refraction fbo, car c'est à ce moment là que nous pouvons récupérer l'information de profondeur. Une fois cette distance calculée, Nous pouvons à l'aide de ceci calculer la profondeur pour chaque point dans le shader.

Une fois cette étape passée, il nous faut activer la fonctionnalité d'alpha blending d'OpenGL qui nous permet de travailler sur la transparence de la texture. Il nous suffit ensuite de modifier la transparence de la texture finale en fonction de la profondeur calculée.

4.5 L'ajout de bruit à la surface

Lors du calcul de la transformée de Fresnel, nous assumons pour le moment que le plan est complètement plat afin de calculer le coefficient réflexion/réfraction. Cependant, en réalité, une surface d'eau n'est jamais complètement plate.

Afin d'éviter cela, nous allons simplement utiliser le vecteur associé à une normal map afin de donner une impression de bruit supplémentaire qui donne un air naturel supplémentaire. Une normal map est une texture qui permet de simuler du relief avec des nuances de couleur. Le fonctionnement est un peu similaire à la dudv map quant à la manière de stocker les informations, seule leur utilisation diffère.

Nous avons choisi de ne pas mettre cette feature dans le rendu final car celle-ci rendais des vaguelettes dont l'effet n'était pas naturel. Nous n'avons pas été en mesure d'établir d'où venait le problème à cet effet même si l'effet semble pouvoir donner un effet de mouvement très appréciable.

5 Implementation

Nous allons détailler ici plus avant les détails de l'implémentation que nous avons choisi, afin de décrire plus précisément comment nous avons implémenté WEWOR.

5.1 L'initialisation du programme

L'initialisation du programme se fait principalement dans la classe Window et la classe DrawHandler, avec respectivement les fichiers du même nom. La classe Window, comme son nom l'indique se charge d'initialiser la fenêtre ainsi que tous les paramètres nécessaires pour cette initialisation. Une fois ceci terminé, on initialise la caméra ainsi que la classe input. On passe enfin au cœur de d'initialisation dans le drawHandler.

La classe Window est pour ainsi dire la classe principale de notre programme, c'est là que se trouve notre boucle d'exécution et c'est elle qui contient tous les éléments clefs au programme.

L'autre classe principale de ce programme est le DrawHandler. Le drawHandler est une classe responsable de l'initialisation, mais également responsable de dessiner tout ce que l'on va pouvoir voir à l'écran. Cela com-

prend l'heightmap, la skybox, le plan d'eau et les modèles qui ont tous une classe respectives et une ou plusieurs instances contenues dans le drawHandler. Le drawHandler à sa construction va se charger justement d'initialiser tous ces éléments. La deuxième fonction appelée systématiquement est la fonction draw qui sera appelée dans la boucle principale afin de rendre le décor.

Pourquoi deux classes séparées ?

L'intérêt du drawHandler vient essentiellement du fait que l'on ne va pas forcément appeler le draw du drawHandler une seule fois par boucle. En effet, on va appeler le draw une fois dans chaque fbo pour la réflexion et la réfraction puis une dernière fois pour rendre le décor. En plus d'être une séparation logique du code, ces deux classes permettent d'avoir une division logique bien définie.

5.2 Les input utilisateurs

Nous avons une classe *input* qui est initialisé une première fois dans la boucle principale. A chaque tour de boucle, les entrées de l'utilisateur sont vérifiés.

Dans un premier temps les touches du clavier sont vérifiés avec les déplacements de la caméra, le changement de skybox avec Entrée ainsi que la possibilité de quitter l'application avec Echap.

Les déplacements de la souris sont en suite vérifiés afin de permettre à l'utilisateur de modifier l'angle de vue de la caméra.

La classe *Input* possède la caméra et se charge de mettre elle même à jour la caméra directement.

5.3 La caméra

Pour les déplacements de la caméra au clavier, l'utilisateur a la capacité de se déplacer dans les 6 directions autour de lui dans l'espace avec les touches z, q, s, d, c et espace. Il peut également accélérer avec la touche shift.

Comme dit dans la partie précédente, l'utilisateur peut également utiliser la souris pour modifier l'angle de vue de la caméra.

5.4 La boucle principale

L'appel à la boucle principale se trouve dans la fonction *render_loop* de la classe *Window*. Cette boucle se comporte exactement de la même manière que l'on pourrait faire dans un jeu vidéo. Voici la liste dans l'ordre de ce que nous faisons dans la boucle.

- Nous récupérons les Input utilisateurs depuis le dernier appel
- Nous mettons à jour la position de la caméra (l'utilisateur) en fonctions des nouveaux inputs.
- Nous dessinons ensuite dans le premier fbo qui va nous permettre d'obtenir la réflexion
- Nous dessinons dans le second fbo pour pouvoir représenter la réfraction
- Nous pouvons enfin rendre dans le frame buffer principale afin de mettre à jour notre image.
- Tout en fin de boucle nous récupérons les input pour l'itération suivante de la boucle.

5.5 Les «rendering classes»

Nous allons ici parler de toutes les classes qui contiennent un objet qui sera rendu dans la carte. Nous parlons ici des classes *HeightmapHandler*, *Skybox*, *WaterRenderer*, *Tree* et *Fish*. Toutes ces classes ont une construction similaire. Le constructeur initialise tous les éléments nécessaires pour dessiner l'entité ensuite (vao, vbo, textures, ...). Enfin, chacune de ces classes implémente une fonction *draw* qui sera ensuite appelé par le *drawHandler* et qui permettra de dessiner l'ensemble de l'environnement.

6 Conclusion

Nous avons finalement des résultats assez satisfaisants, bien que beaucoup d'améliorations puissent être apportés. Ce projet nous a permis de nous familiariser avec OpenGL, qui est un excellent outil pour travailler sur du rendu 3D.

Cependant, pour arriver à un rendu réaliste, de nombreuses choses restent à faire, particulièrement sur le travail de la lumière. Nous aurions pu également creuser plus loin du côté de rendu de vagues car de nombreux travaux existent dans cette optique.

Nous avons réussi à mettre en place une bonne partie des techniques qui permettent de mettre en place un rendu d'eau assez réaliste sans être trop gourmand sur les ressources.

