# Rapport de projet INF443

Tess BRETON

**Objectif** : Créer une île éclairée par la lumière d'un phare, puis ajouter un feu d'artifice en interaction avec le clavier.

## 1 Premiers éléments de la scène

### 1.1 Terrain

J'ai créé le terrain en utilisant le code du TP sur les textures pour pouvoir ajuster les paramètres d'un **bruit de Perlin**, afin d'obtenir le rendu souhaité.

Pour obtenir une **couleur différenciée selon l'altitude** (beige jusqu'à la surface, puis vert) avec une transition douce, j'ai créé une fonction `vec3 evaluate_terrain_color(float z)` qui interpole les couleurs entre deux paliers. J'utilise ensuite cette fonction dans la fonction `update_terrain`.

### 1.2 Mouvement de l'eau

Pour animer la surface de l'eau, j'ai utilisé un vertex shader `waterVert.glsl`. La modélisation de la surface de l'eau n'étant pas au coeur de mon projet, j'ai choisi de simplement déplacer verticalement les sommets en superposant trois sinusoïdes de directions différentes [1].

### 1.3 Végétation et objets

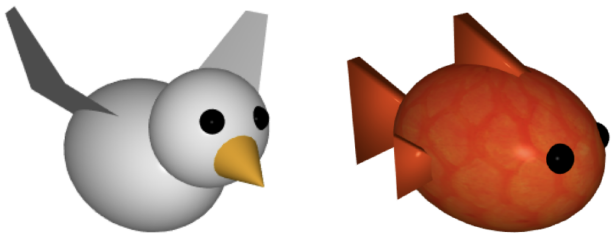
Pour ajouter une végétation assez réaliste, il fallait placer les éléments de manière aléatoire sur le terrain (et au-dessus de la surface de l'eau, en $z > 0$) tout en évitant qu'ils s'intersectent. J'ai donc créé une fonction `std::vector<cgp::vec3> generate_positions_on_terrain`, qui prend notamment en argument l'espacement `spacing` minimal accepté entre deux éléments. Pour une exécution rapide, j'ai utilisé une méthode vue en cours qui consiste à **diviser le terrain en cases** de largeur `spacing`.

Pour l'herbe et les fleurs, j'ai utilisé des ***billboards***. Le reste des objets est créé à partir de fichiers `.obj` ou de primitives.

1. NB : Comme dans le reste du projet, j'ai utilisé le GUI pour ajuster les paramètres (amplitudes, fréquences) afin d'obtenir un rendu qui me convenait ; pour plus de clarté, j'ai supprimé le code correspondant dans la version finale pour ne laisser dans le GUI que les paramètres pertinents.

### 1.4 Animaux animés

J'ai repris le travail effectué en TP d'animation hiérarchique sur l'oiseau, et j'ai appliqué la même méthode pour créer un poisson rouge. Pour les nageoires du poisson, j'ai créé un fichier `triangle.cpp` avec une fonction `mesh create_triangle3D_mesh(vec3 p0, vec3 p1, vec3 p2, vec3 height)` pour construire le *mesh* triangulaire voulu.



(a) Oiseau (b) Poisson

FIGURE 1 – Animation hiérarchique d'un oiseau et d'un poisson rouge

### 1.5 Illumination : Soleil, brouillard et lucioles

Pour l'illumination de base, je me suis inspirée du code du TD2 : j'ai placé une source de lumière principale (un **soleil**), à laquelle j'ai imposé un mouvement elliptique couplé à un changement de couleur pour simuler la tombée de la nuit. J'ai également implémenté un effet de **brouillard** avec une distance caractéristique assez grande pour ne pas gêner la visibilité de la scène.

Pour ajouter des **lucioles**, j'ai adapté le code du Soleil en diminuant la taille et la distance d'atténuation, et en changeant le mouvement procédural pour le rendre plus aléatoire. Pour l'atténuation, pour avoir une transition plus douce aux bords, j'ai choisi un facteur $\alpha$ de la forme `tanh(length(luciole_position - fragment.position) / dlmax_luciole)`. Les lucioles ne sont dessinées qu'une fois le soleil couché, et l'intensité de leur lumière est alignée sur le mouvement du soleil.

De nombreux paramètres d'illumination sont réglables via le **GUI** (couleurs des sources de lumière, couleur du brouillard, paramètres du modèle de Phong, gamma correction, fréquence de révolution du Soleil...).

## 2 Lumière du phare

Pour éclairer ma scène avec la lumière du phare, je me suis inspirée du schéma suivant présenté dans [1] :

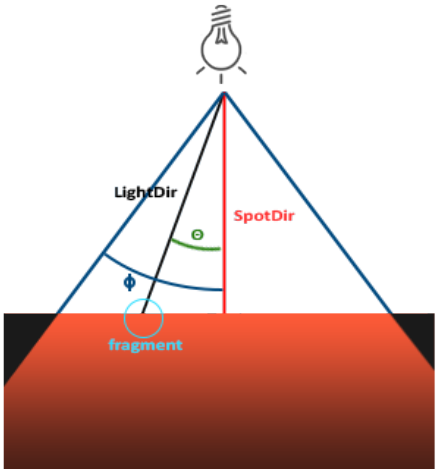



FIGURE 2 – Géométrie du spot

J'ai commencé par suivre la méthode de [1] pour obtenir un **spot simple** (sans atténuation aux bords), défini par une direction `SpotDir` et un angle de cutoff $\phi$ (on stocke en fait $\cos(\phi)$ dans la variable `Cutoff`). Il s'agit alors essentiellement de calculer le produit scalaire entre le vecteur `LightDir` (du fragment considéré vers la source de lumière) et `SpotDir`, que l'on compare à $\cos(\phi)$.

Pour obtenir un **effet d'atténuation aux bords**, toujours en suivant la méthodologie de [1] : on ajoute un `OuterCutoff`, puis on interpole l'illumination linéairement entre `Cutoff` et `OuterCutoff`.

(a) Sans atténuation

(b) Avec atténuation

FIGURE 3 – Atténuation des bords dans la zone éclairée par la lumière du phare

Le code correspondant à l'illumination par la lumière du phare se trouve à la fin du shader `/scene/frag.glsl`. À nouveau, beaucoup de paramètres sont réglables via le GUI.

# 3 Feu d'artifice

## 3.1 Modélisation

Pour modéliser un feu d'artifice, j'ai choisi d'utiliser un **système de particules** dont l'évolution est déterminée par **simulation physique**. Les particules sont lancées avec des vitesses aléatoires (dans des intervalles définis), puis soumises seulement à leur poids. Chaque particule a une durée de vie aléatoire, à partir de laquelle on définit l'évolution de sa taille par interpolation linéaire. Les particules sont représentées par des ***billboards*** portant une image d'étoile. Pour obtenir un joli rendu, j'ai ajouté une part d'aléatoire dans les couleurs des particules d'un même feu : on perturbe aléatoirement la couleur principale du feu (`Firework.color`).

## 3.2 Implémentation

Tout le code relatif au feu d'artifice se trouve dans le fichier `firework.hpp`. Pour créer les feux d'artifice, j'ai défini une classe `Particle` (pour chaque particule d'un feu), une classe `Firework` (qui gère l'ensemble des particules d'un feu), et une classe `FireworkManager` (qui gère l'ensemble des feux à afficher et mettre à jour).

Un appel à `fireworkManager.launchFirework(position, color)` initialise un nouveau feu d'artifice et l'ajoute au `FireworkManager`. Si toutes les particules d'un feu sont éteintes, alors on supprime le feu de la liste `fireworks`. Pour introduire une interaction avec le clavier dans mon projet, j'ai décidé de lancer un nouveau feu à chaque fois que l'utilisateur appuie sur une touche (voir `void scene_structure::keyboard_event()` dans `scene.cpp`). L'affichage peut être ralenti si l'on demande trop de feux au même moment.

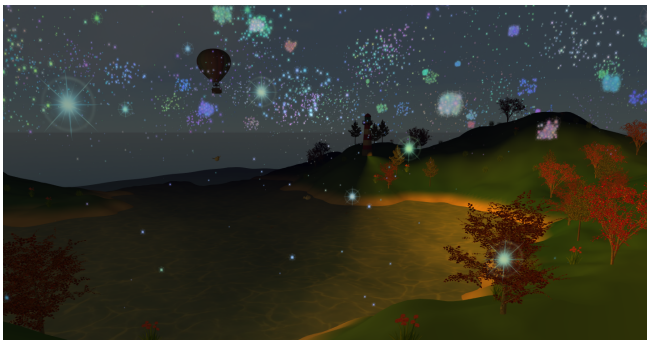


FIGURE 4 – Rendu final

# Références

[1] Joey De VRIES. URL : `https://learnopengl.com/Lighting/Light-casters`.