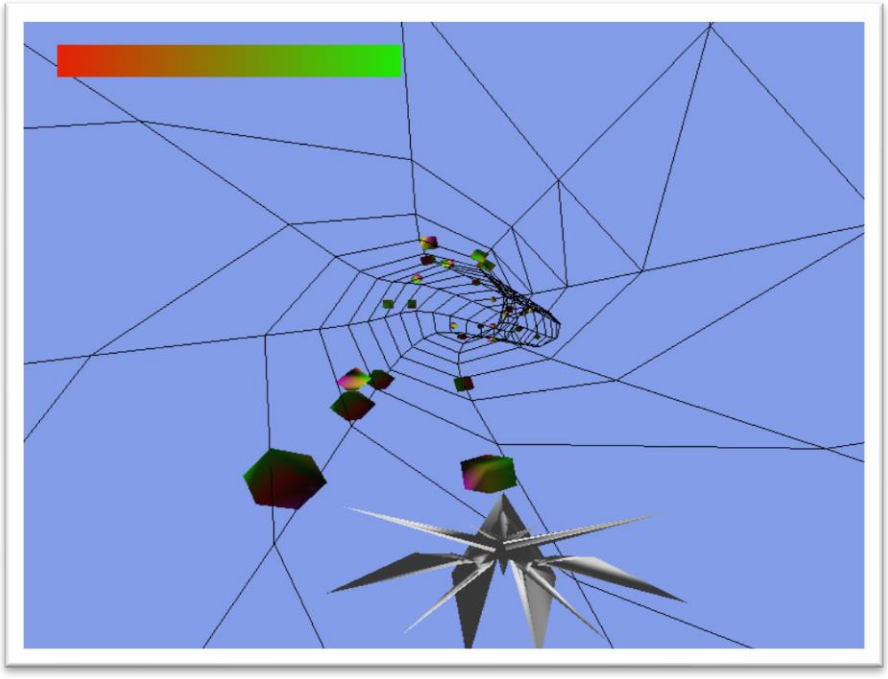
Guillaume Gisbert
Groupe B
Xavier Jannin

TSI : JEU

# Synthèse d'image

## Compte Rendu



# Jeu OpenGL : Perlin Vertex

Dans le cadre du module de Traitement et Synthèse d'Image, nous avons réalisé un petit jeu 3D fonctionnant à l'aide d'OpenGL, en C++.

Notre jeu consiste à contrôler un vaisseau se déplaçant dans un tube non linéaire, et qui doit éviter les obstacles qui se dirige vers lui. L'objectif est ainsi de survivre le plus longtemps. Petit à petit, le jeu s'accélère, ce qui augmente progressivement la difficulté.

La problématique principale durant la réalisation de ce jeu a été de permettre la création d'un tube qui n'est pas linéaire, c'est-à-dire qui suit un chemin généré aléatoirement.

Ce compte rendu est ainsi décomposé en 3 parties : la première partie explique comment générer un tube non linéaire ; la deuxième présente les éléments du jeu (le joueur et les obstacles) mais surtout comment un élément peut suivre et se déplacer à travers le tube, et enfin la troisième partie détaille quelques aspects de l'affichage, c'est-à-dire la caméra et l'interface (HUD).

# 1 – Génération d'un tube non linéaire (class Path)

Pour afficher un tube non linéaire, il est d'abord nécessaire de générer un chemin constitué de points dont les valeurs sont générées aléatoirement de manière continue, pour que l'ensemble apparaisse cohérent. Puis, il faut placer des points autour du chemin pour créer le contour du tube, et enfin il faut produire les indices des triangles qui permettront de visualiser le tube.

## 1 - Génération de valeurs aléatoires continues

Pour permettre la création d'un chemin non linéaire, il devient nécessaire d'utiliser des valeurs aléatoires. Cependant, les valeurs données par la méthode `rand()`, de la librairie standard `stdlib`, n'évolue pas graduellement.

Ainsi, pour permettre une génération progressive, nous avons simplement copié et adapté l'algorithme trouvé sur une page internet, qui s'inspire du bruit de Perlin : "Procedural Generation Part 1 - 1D Perlin Noise". Cette algorithme s'occupe de générer des valeurs intermédiaires (par une interpolation cosinus) entre des valeurs générées aléatoirement.

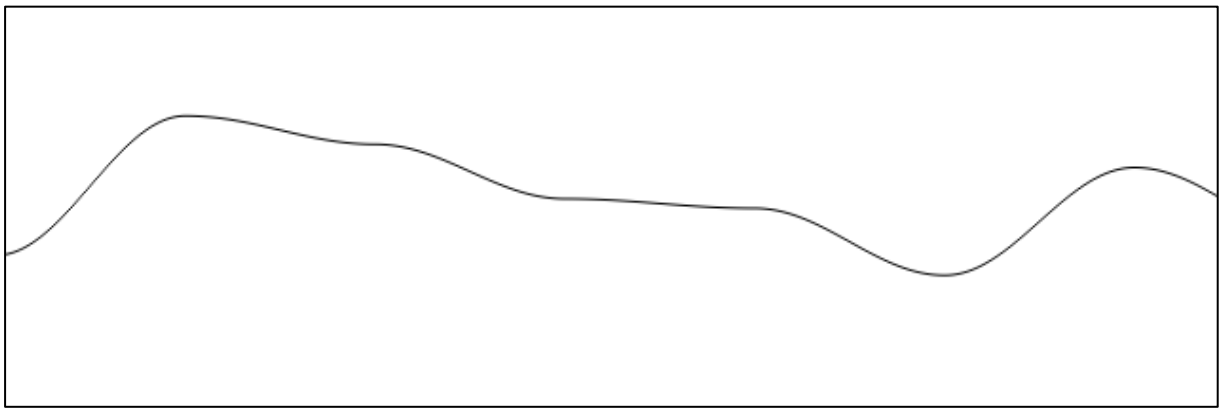


*figure 1 - Courbe générée à partir de l'algorithme de bruit de Perlin*

Nous avons alors développé une classe `Perlin` qui utilise cet algorithme pour ensuite avoir plusieurs instances de "générateurs aléatoires continus".

## 2 - Génération du chemin

Avant de générer un tube, il est d'abord nécessaire de déterminer le chemin de celui. Ainsi, ce chemin est constitué d'un ensemble de points stockés dans une liste, dont les valeurs sont obtenues à partir de "générateurs aléatoires continus".

Les coordonnées X et Y de chaque point sont créées à partir de générateurs `Perlin`, et les coordonnées Z sont produites à partir d'un compteur, qui s'incrémente à chaque nouveau point. En réalité, si toutes les coordonnées étaient produites par des "générateurs aléatoires continus", le chemin créé ressemblerait à un ensemble de morceaux de droite reliés entre eux, dont leurs extrémités correspondraient aux valeurs aléatoires, que les générateurs interpolent. En effet, générer une courbe cosinus en fonction d'un cosinus donne une droite ! (cela permet aussi d'éviter un risque de croisement du chemin). Et aussi, de cette manière, le chemin avance toujours dans la direction Z.

Ainsi, chaque point se trouve proche du précédent mais l'ensemble de ces points ressemble à de l'aléatoire.

Pour simplifier l'utilisation de cette liste de points, nous avons utilisé un conteneur `vector` qui permet de plus facilement ajouter et supprimer des éléments d'une liste, ce qui sera utile pour générer le chemin (et donc le tube) progressivement.

## 3 - Génération du tube

Le tube est constitué d'un ensemble de cercle placé en chaque point du chemin et orienté pour être "perpendiculaire" à la droite tracée par le point où il se situe et le point précédent (autrement dit, la droite est normale au cercle). Pour afficher le tube à partir des cercles, il suffit de correctement générer les triangles entre le cercle pour créer des cylindres entre chaque point du chemin.

Pour réaliser toutes ces étapes, il est donc nécessaire de d'abord générer un cercle de référence, puis d'en déduire les points de tous les cercles placés en chaque point du chemin, et enfin de déterminer les indices des triangles qui permettront de former le tube.

### Génération du cercle de référence :

Le cercle de référence est modélisé à partir de deux paramètres : son rayon $R$ et le nombre de point $N$ composant le tour.

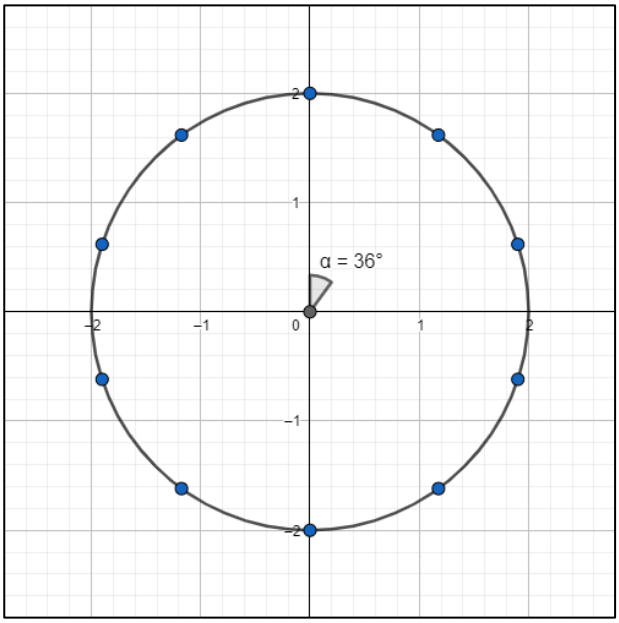



*figure 2 - Cercle de référence généré avec 10 points et de rayon $R = 2$*

Le cercle est alors placé sur le plan $(x, y)$ centré à l'origine du repère, et tous les points sont équitablement répartis autour du cercle à une distance $R$, grâce aux fameuses fonctions trigonométriques $cos$ et $sin$. De plus, nous pouvons observer que les points sont espacés d'un angle $\frac{2\pi}{N}$. Dans notre utilisation, il n'est pas nécessaire d'ajouter les coordonnées du centre dans la liste des points du cercle de référence.

**Placement des points du cercle :**

A partir du cercle de référence, il devient possible de placer des points autour du chemin pour pouvoir ensuite générer le tube.

Ainsi, en chaque point du chemin, les points du cercle doivent être correctement positionnés mais aussi orientés. Pour l'orientation, nous avons décidé que la normale de chaque cercle devait être orientée en direction du point précédent du chemin (sauf pour le premier point qui est dirigé sur le suivant). Pour réaliser l'orientation, il est alors possible de générer une matrice de rotation en fonction d'un vecteur, et ce vecteur est dirigé entre les points en cours et précédent du chemin. A partir de la matrice de rotation, il devient possible de placer tous les points à la bonne position pour chaque point du chemin, par le calcul suivant :

$$points = rotation * cercle + position$$

L'ensemble des points est ainsi sauvegardé dans une unique liste, qui servira de buffer pour envoyer toutes les coordonnées de chaque point sur la carte graphique.

**Indices des triangles pour composer le tube :**

Enfin, lorsque tous les points du tube sont correctement placés, il ne reste plus qu'à générer les indices des points pour former les triangles qui représenteront le tube.

Ainsi, comme les points des cercles ont été ajoutés progressivement dans la liste de points, il devient facile, à partir du nombre de points $N$ qui composent le cercle, de localiser l'indice du premier point de chaque cercle, et par conséquent les indices des autres points. De plus, comme les points des cercles ont été ajoutés dans le même ordre, cela signifie que chaque indice d'un cercle se trouve "en face" du même indice du cercle précédant et suivant.

Par conséquent, la génération des indices pour former les cercles se réalise de manière indépendante de la liste des points formant le tube. Les indices des triangles sont alors générés pour suivre la forme suivante :

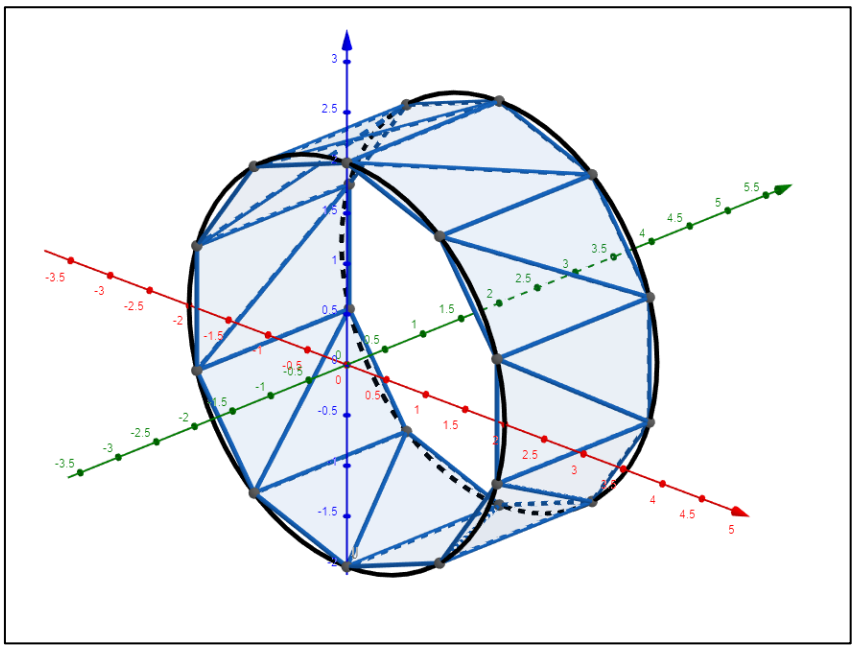


*figure 3 - Tube généré entre deux cercles de points*

C’est-à-dire prendre 2 points de chaque cercle, et tracer le quadrilatère formé par ces points par deux triangles, puis passer aux points suivants ; et cela pour tous les cercles.

A partir du buffer des points et aussi le buffer contenant les indices des triangles, il est possible de générer un tube non linéaire avec OpenGL :

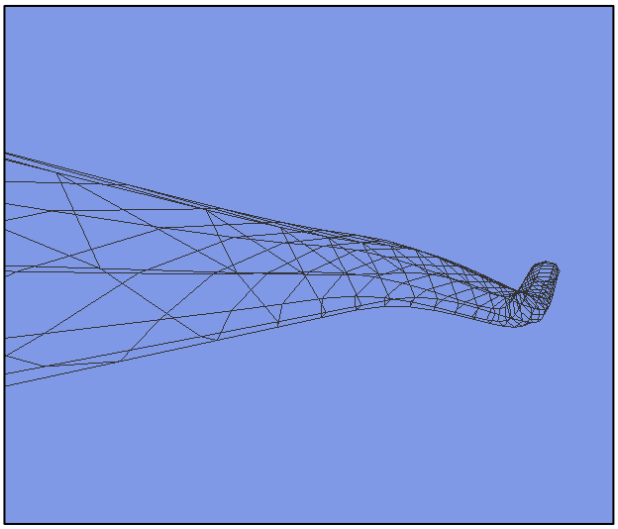


*figure 4 - Tube visualisé grâce à OpenGL*

**Affichage progressif du tube :**

Pour limiter le nombre de points nécessaire utilisés pour générer le tube, celui-ci est construit progressivement en fonction de la position du joueur.

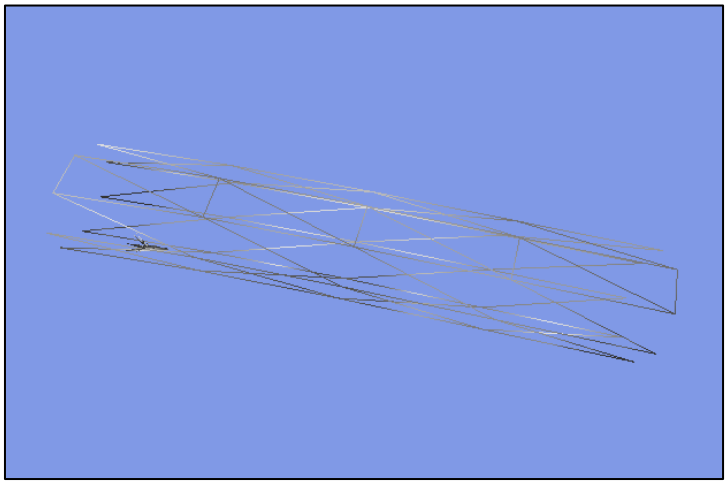


*figure 5 - Portion du tube qui suit le joueur*

Ainsi, lorsque le joueur avance, le programme vérifie que le dernier point du chemin (c’est-à-dire devant le vaisseau) se trouve à une distance du joueur supérieur à une valeur ***M*** définie. Si ce n’est pas le cas, des points sont ajoutés pour respecter cette condition.

Inversement, si le premier point du chemin (c’est-à-dire derrière le vaisseau) se trouve à une distance supérieure à une valeur ***m***, celui-ci est supprimé. Et ainsi de suite, si le nouveau premier point ne respecte pas la condition.

Ainsi, l'utilisation du containeur `vector` est très utile ici, pour facilement manipuler la liste de points du chemin pour seulement afficher une portion du tube.

L'ensemble de ces étapes permet donc de produire un tube dont le chemin est produit aléatoirement et progressivement. Il est à présent nécessaire de créer un composant capable de se déplacer sur le chemin et sur le tube pour ainsi placer des éléments de jeu comme le joueur et les obstacles.

## 2 – Eléments de jeu : Joueur et Obstacles (class PathAgent)

Le tube n'étant pas linéaire, le déplacement d'éléments n'est pas aussi simple que d'aller tout droit. Nous avons ainsi développé un composant capable de suivre le chemin du tube et de correctement positionner l'élément sur le tube. Le joueur et les obstacles seront alors facilement intégrable au jeu. De plus, cette intégration permettra une détection des collisions simple à partir des données de position des éléments.

### 1 - Elément suivant le chemin

A partir des points du chemin, il est possible de créer un élément qui se déplace en suivant ce chemin.

Pour cela, l'élément se place en un point du chemin et se déplace progressivement en direction du point suivant ou précédent, en fonction de la vitesse positive ou négative de celui-ci. Ainsi, lorsque l'élément dépasse le point auquel il se dirigeait, la distance dépassée est modifiée pour être dirigée en direction du point suivant. Par conséquent, l'élément suit petit à petit tous les points du chemin jusqu'à ce qu'il n'y en ait plus.

Lorsque l'élément a atteint une extrémité du chemin, celui-ci peut être détruit.

### 2 - Elément sur le tube

En connaissant la position d'un élément sur le chemin, il devient possible de déterminer sa position sur le tube.

Pour cela, il est nécessaire de connaître le vecteur dirigé perpendiculairement à la direction de l'élément, c'est-à-dire la direction entre les deux points dont se déplace l'élément, pour ainsi se fixer sur le tube. Ce vecteur appartient alors au plan dont la direction entre deux points du chemin correspond à la normale de celui-ci ; par conséquent, c'est aussi le vecteur le plus court pour se diriger vers le tube.

Ainsi, un simple produit vectoriel avec un vecteur dirigé selon l'axe X permet à la fois de calculer ce vecteur perpendiculaire, mais aussi de l'orienter pour être toujours dirigé dans la même direction, c'est-à-dire dans une "orientation de référence".

Finalement, pour obtenir la position sur le tube, il ne reste plus qu'à tourner le vecteur autour de la direction des points du chemin, selon un angle définit dans les paramètres de l'élément, puis de le normaliser, afin de l'ajouter à la position sur le chemin en prenant en compte le rayon du tube.

Grâce à l'ensemble de ces étapes, il devient donc possible de déplacer des éléments sur le tube. Ces éléments correspondront, dans le jeu, au joueur mais aussi à tous les obstacles.

## 3 - Joueur et obstacles

A partir de la classe `PathAgent`, qui s'occupe de déplacer un élément sur le chemin du tube et permet de le placer sur le tube, le joueur et les obstacles sont facilement ajoutables au jeu.

Ainsi, tout le déplacement est directement géré par cette classe. Par conséquent, les classes du joueur (`Player`) et des obstacles (`Cube`) hérite de la classe `PathAgent`. Cette classe possède alors des paramètres pour définir la position, l'orientation, la direction, la vitesse du déplacement de l'élément sur le chemin et le tube. Les classes `Player` et `Cube` s'occupent en plus de l'affichage de leur élément respectif, en envoyant les bonnes valeurs sur la carte graphique.

Ainsi lors de la génération d'une image, les positions du joueur et de tous les obstacles sont mises à jour, puis tous ces éléments sont affichés.

Voici quelques aspects constituants le Joueur et les Obstacles :

**Joueur :**

- Le joueur est un modèle 3D créé à partir du logiciel Blender.
- Les flèches gauche et droite du clavier permettent de modifier son orientation dans le tube. Il est aussi possible d'accélérer la vitesse du joueur avec la flèche du haut (ou de ralentir avec la flèche du bas).
- Le joueur se dirige vers la fin du chemin composant le tube, ce qui est l'inverse des obstacles qui se dirigent vers le début du chemin (et donc en direction du joueur).

**Obstacles :**

- Les obstacles sont des cubes représentés par 12 triangles (= 6 faces * 2 triangles).
- La classe `Cube` représente un unique obstacle se déplaçant sur le chemin.
- L'ensemble des obstacles sont enregistrés dans une liste (plus précisément dans un `vector`) remplie d'éléments de la classe `Cube`.
- A un rythme régulier : un obstacle est ajouté à la fin du chemin en direction du joueur, et est positionné à une orientation aléatoire sur le tube.
- Les obstacles possèdent aussi une vitesse de rotation (qui est tout le temps nulle dans le jeu)
- Lorsqu'un obstacle atteint un bord du chemin, il est supprimé.

## 4 - HitBox : détection des collisions

Pour gérer les collisions entre le joueur et les obstacles, une fonction est déclenchée lors de la génération de chaque image.

Cette fonction teste pour chaque obstacle si la position de celui-ci sur le chemin est suffisamment proche de la position du joueur sur le chemin. Puis, si c'est le cas, la fonction détermine la différence d'angle entre l'obstacle et le joueur. Enfin, si les deux éléments sont proches, et la différence d'angle est suffisamment faible pour signifier qu'il y a un obstacle qui touche le joueur, le joueur perd 1 point de vie et l'obstacle disparait.

Ainsi, cette méthode de détection d'obstacles n'est pas une vraie HitBox qui utiliserait des points des modèles du joueur et de l'obstacle pour déterminer s'il y a contact, mais cette une bonne solution intermédiaire adaptée à la construction propre du jeu.

## 3 – Caméra et HUD

Bien que la génération du tube et la gestion du joueur et des obstacles sont des éléments importants du jeu, il ne faut pas oublier de gérer aussi tous les aspects visuels, tel que la caméra mais aussi l'interface.

### 1 - Caméra

Pour une caméra générée avec un ordinateur, il est toujours plus intéressant de modifier la position de chaque élément dans le monde au lieu de renvoyer toutes les coordonnées des points de chaque élément à afficher.

Ainsi, le shader affichant les éléments du jeu utilise la position de la caméra, ainsi que son orientation, pour positionner tous les points des éléments.

Dans notre jeu, la caméra se positionne et s'oriente de la même manière que le joueur, en étant légèrement décalé à l'arrière pour pouvoir voir le vaisseau.

### 2 - HUD : Affichage de la vie

Pour afficher les points de du joueur, il est nécessaire d'ajouter une interface (HUD en anglais pour *Head Up Display*) qui permet de préciser à l'utilisateur la valeur de ce paramètre.

**Barre de vie :**

Au début de la partie, le joueur possède 8 vies, qui sont représentées visuellement par une barre en haut à gauche de l'écran. Cet élément est ainsi indépendant de la caméra et des autres éléments de jeu.

Pour simplifier son implémentation, il est ainsi possible d'utiliser un shader spécifique à la barre de vie, ce qui permet de l'isoler de tous les autres composants du jeu à afficher. La barre est alors générée par un unique quadrilatère, et un fragment shader spécifique a été développé pour l'afficher par un dégradé de couleur du rouge vers le vert.

Pour conclure, ce projet nous a permis de réaliser un jeu en découvrant le fonctionnement d'OpenGL, ainsi que le langage C++. Conceptualiser et réaliser correctement tous les éléments composant ce jeu a pris un certain temps, mais le résultat correspond à ce que nous imaginions.

Cependant, nous nous sommes aussi rendu compte de la difficulté pour correctement paramétrer le jeu pour que celui-ci soit agréable à utiliser. En effet, la caméra n'est pas exactement fixée sur le joueur, et il arrive parfois que le vaisseau soit difficilement visible.

Aussi, nous n'avons pas pris beaucoup de temps pour réfléchir sur la manière dont la difficulté pouvait augmenter : pour l'instant, la vitesse augmente progressivement. Il aurait été intéressant de générer plusieurs vagues d'obstacles avec des spécificités : par exemple, leur vitesse de déplacement est différente, leur orientation change au cours du temps, la taille du tube diminue ou augmente, la taille des ennemies augmente ou diminue, les obstacles sont différents, … . Tout cela correspond à des pistes que nous aurions pu explorer, mais que nous n'avons pas eu le temps de les appliquer.

Ce projet était donc très intéressant, et donne envie de découvrir toutes les fonctionnalités que permet OpenGL, afin de mieux comprendre la synthèse d'image dans des applications très variés.

## Organisation du temps (environ) :

- Affichage d'un cube : 3h
- Affichage d'un modèle 3D : 2h
- Classe *Path*, sur le rendu : 1h + 4h + 1h + 2h + 1h30
    - Création de la classe mat3 (pour produit matriciel : mat3 * vec3)
    - Correction de l'orientation des cercles (en direction du précédent)
- Perlin Noise : 6h
    - Modification de la méthode de création du bruit
    - Difficulté à adapter un programme Javascript en C++ (gestion des types)
    - Résultat apparaissant faux au départ est en réalité correct
- Perte de temps :
    - Mesure de temps entre deux instants : 1h
- *Path Agent* : 4h + 2h
- *Joueur* : 3h
- Génération des obstacles : 1h
- Hitbox : 2h
- Caméra : 2h
- Affichage de la barre de vie : 2h