

OpenGL - Boids

Antoine Leblond
Mathurin Rambaud

Table des matières

1	Introduction	2
1.1	Mise en contexte de la réalisation du projet	2
2	Programmation 2D avec P6	3
2.1	Création de la classe Boid	3
2.2	Gérer les paramètres de simulation	4
3	Programmation 3D en OpenGL	5
3.1	Créations des modèles 3D (LODs)	5
3.2	Lecture des modèles 3D en OpenGL	6
3.3	Optimisation du code et des fichiers	8
3.4	Textures et lumières	10

1 Introduction

1.1 Mise en contexte de la réalisation du projet

L'idée principale de ce projet est de simuler des comportements collectifs en utilisant l'interface de programmation OpenGL. Pour se faire nous allons utiliser le principe des Boids. Les Boids sont des objets qui se déplacent dans un espace selon des règles dans le but de reproduire des mouvements réalistes, on retrouve parmi eux 3 règles principales :

- La cohésion qui vise à maintenir la proximité entre les Boids d'un même groupe. Chaque Boid est attiré vers le centre de masse des Boids voisins. Cela crée un effet d'attraction qui permet aux Boids de rester ensemble, formant ainsi un groupe. La cohésion favorise la formation de groupes, où les Boids sont attirés les uns vers les autres et se déplacent globalement dans la même direction.
- L'alignement concerne l'orientation des Boids au sein du groupe. Chaque Boid a tendance à aligner sa direction de déplacement avec celle de ses voisins. Cela crée un effet de synchronisation où les Boids suivent la tendance du groupe dans son ensemble. L'alignement permet aux Boids de se déplacer dans une formation cohérente.
- La séparation sert à éviter les collisions entre les Boids. Chaque Boid cherche à maintenir une distance minimale avec ses voisins pour éviter les chevauchements ou les interférences. Cela crée un effet de répulsion entre les Boids, les poussant à s'éloigner les uns des autres. La séparation permet aux Boids d'éviter les regroupements excessifs.

Si l'on applique ces 3 règles en même temps sur nos Boids alors nous aurons un résultat satisfaisant quant aux comportements de ces derniers. L'idée est alors dans un premier temps de programmer ces comportements en 2D pour l'instant afin d'obtenir le meilleur résultat possible et lorsque l'on est satisfait il faudra adapter ce code pour l'implémenter dans l'interface d'OpenGL afin de cette fois-ci déplacer nos objets dans une scène 3D. De plus, OpenGL nous permet d'ajouter certaines fonctionnalités comme les textures ou encore les lumières qui permettront d'enjoliver notre scène finale et mettre en valeur nos Boids.

2 Programmation 2D avec P6

2.1 Création de la classe Boid

Nos Boids sont caractérisés par 4 paramètres différents :

- La taille (type : float)
- La vitesse (type : float)
- La position (type : vec2 en 2D | vec3 en 3D)
- La direction (type : vec2 en 2D | vec3 en 3D)

Ces paramètres sont initialisés dès la créations des Boids grâce au constructeur avec des valeurs cohérentes par rapport à notre scène. Les positions et directions des Boids sont aléatoires dans notre scène pour varier notre simulation.

Vient ensuite les méthodes de notre classe Boid. Celle-ci est composée des méthodes suivantes :

- move_boid : actualise la position de notre Boid en fonction de sa vitesse et de sa direction
- avoid_walls : permet aux Boids de ne pas sortir de la scène, lorsqu'ils sont trop proches des murs ils partent dans la direction opposée de manière fluide. Cette fonction prend en paramètre les dimensions de la fenêtre ainsi que la distance à laquelle les murs sont des bords de la fenêtre.
- display_boid : affiche le Boid à sa position. Cette fonction prend en paramètre le contexte p6 dans lequel se déroule la simulation.
- update_boid : met à jour la taille et la vitesse du Boid. Cette fonction prend en paramètre deux float qui représentent les nouvelles valeurs. Cette fonction sert notamment lorsque l'utilisateur décide de manipuler les paramètres grâce à l'interface, il faut alors mettre à jour les nouvelles valeurs.
- separate : cette fonction est responsable de la séparation entre nos Boids afin d'éviter les collisions. Elle prend en paramètre un vecteur de Boids où sont stockés tous les Boids de notre scène, une distance

pour laquelle la séparation s'applique et enfin une force de séparation. Globalement cette fonction vérifie la distance d'un Boid par rapport aux autre Boid de la scène et si celle-ci est inférieure à la distance passée en paramètre, alors on lui applique une force proportionnelle à la distance entre les deux Boids. On remarquera que si plusieurs Boids sont à distance alors les forces s'additionnent.

- align : De la même manière que separate, align prend en paramètre les mêmes variables sauf que cette fois-ci elle va faire tendre la direction d'un Boid vers la direction des autres Boid à proximité.
- cohesion : Pareil que pour separate et align, cohesion prend en paramètre les mêmes variables sauf qu'elle va rapprocher les Boids les uns des autres s'il sont suffisament proche par rapport à la distance minimale.

Une chose à noter sur ces 3 dernières méthodes, il aurait peut-être été intéressant lorsque le nombre de Boids devient très grand d'implémenter un system de Quadtree afin d'optimiser le programme pour éviter de vérifier les collisions avec tous les Boids à chaque fois.

2.2 Gérer les paramètres de simulation

P6 dispose d'une library GUI nommée Dear ImGui qui permet d'avoir une interface avec laquelle l'utilisateur peut interagir. Dans notre cas, on va pouvoir utiliser cette librairie afin de modifier les paramètres de nos Boids et pouvoir observer le résultat en temps réel, notamment les différentes règles qui s'appliquent à ces derniers.

```
ctx.imgui = & {
    ImGui::Begin("Parameters");
    ImGui::SliderFloat("Size", &boid_size, 0.005f, 0.05f);
    ImGui::SliderFloat("Speed", &boid_speed, 0.005f, 0.02f);
    ImGui::SliderFloat("Separation", &separation_strength, 0.0f, 0.1f);
    ImGui::SliderFloat("Alignment", &alignment_strength, 0.0f, 0.1f);
    ImGui::SliderFloat("Cohesion", &cohesion_strength, 0.0f, 0.1f);
    ImGui::SliderFloat("Wall distance", &wall_distance, 0.25f, 0.75f);
    ImGui::SliderFloat("Background alpha", &background_alpha, 0.f, 1.f);
    ImGui::End();
};
```

On peut restreindre la valeur de nos paramètres en ajoutant des bornes pour garder un minimum de sens à notre simulation, mais globalement l'utilisateur est libre de changer les valeurs de son choix. À noter que dans la simulation 3D la variable `wall_distance` n'existe plus puisque les murs de notre scène correspondent tout simplement aux murs de notre cube englobant notre simulation ainsi que `background_alpha` qui permettait en 2D d'afficher un tracé derrière les Boids, cependant il y a l'ajout d'un bouton "LD/HD" permettant de changer le niveau de détails de nos objets. La possibilité de modifier les paramètres en temps réel permet d'observer comment les Boids réagissent à des changements soudains de leur environnement.

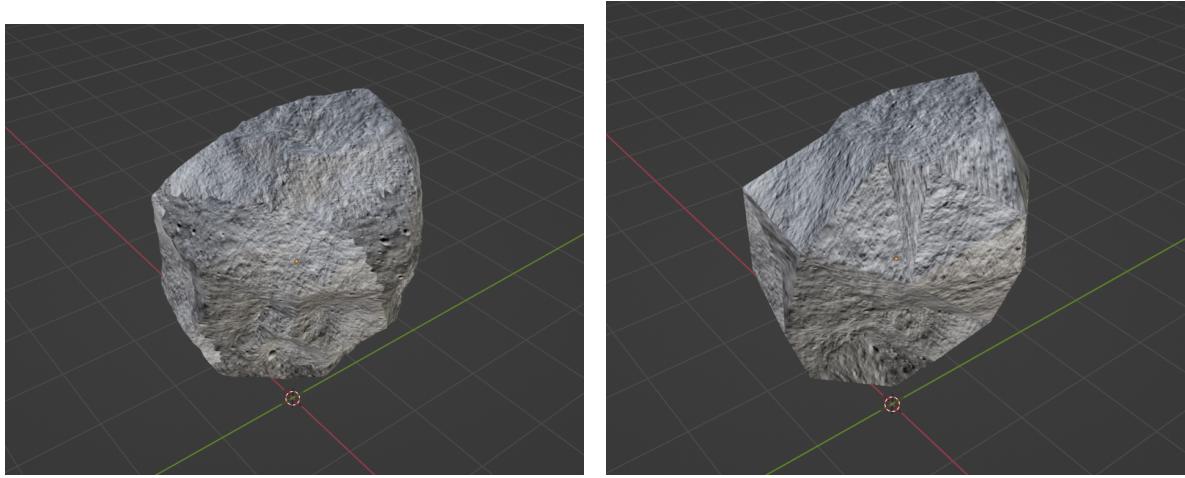
3 Programmation 3D en OpenGL

3.1 Créations des modèles 3D (LODs)

On attaque maintenant la grosse partie du projet, à savoir travailler sur OpenGL. Avant toute chose puisque notre but est de représenter des Boids simulant un comportement de groupe, il fallait savoir ce que nous voulions représenter afin de créer les modèles 3D correspondants. Lorsqu'on nous a introduit le phénomène de Boids on nous a montré une vidéo qui représentait des vaisseaux spatiaux dans l'espace, nous avons donc eu l'envie de reproduire la même idée. Il y a alors 5 modèles 3D différents : un cube qui représentera simplement le décors, un vaisseau et un astéroïde de haute qualité ainsi que la version basse qualité de ces deux derniers modèles pour respecter le principe de LOD. Sur blender il existe une fonction nommée "Decimate" qui permet tout simplement de réduire le nombre de sommets de notre objet tout en gardant sa forme globale, elle est utile lorsque l'on souhaite par exemple faire des modèles en low poly ou simplement dans notre cas pour avoir un modèle moins lourd. Enfin nous le verrons juste après dans ce rapport mais la fonction que nous avons créé pour lire les fichiers .obj en OpenGL fonctionne uniquement sur des modèles 3D avec des faces qui comportent 3 sommets, il est donc important avant d'exporter notre modèle de le triangulariser. Heureusement Blender possède une fonction qui permet d'effectuer cette tâche. Nous avons alors créé ces différents modèles puis nous les avons exporté en fichier .obj afin de pouvoir les lire en OpenGL, en voici un petit aperçu :



(a) Vaisseau spatial HD (gauche) et Vaisseau spatial LD (droite)



(b) Astéroïde HD (gauche) et Astéroïde LD (droite)

3.2 Lecture des modèles 3D en OpenGL

Puisque ça ne pouvait pas être aussi simple, maintenant que nous avons nos fichiers .obj il va falloir trouver un moyen de les lire en OpenGL grâce à une fonction communément appelée "Parser". Le template que l'on nous donne possède une struct nommée ShapeVertex contenue dans glimac, voici à quoi elle ressemble :

```
struct ShapeVertex{
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec2 texCoords;
};
```

ShapeVertex est composé d'un vecteur pour stocker les positions des vertex, d'un vecteur pour stocker les normales des vertex et enfin d'un vecteur pour stocker les coordonnées des textures. C'est tout ce qu'il nous faut

pour pouvoir afficher nos objets en 3D, car lorsque l'on regarde comment est fait un fichier .obj il ressemble à ceci :

```
v 100.000000 100.000000 -100.000000
/*...*/
vn -0.0000 1.0000 -0.0000
/*...*/
vt 0.625000 0.500000
/*...*/
f 5/5/1 3/3/1 1/1/1
/*...*/
```

Ce qui suit le préfix v (vertex) correspond à ce que l'on doit stocker dans le vecteur position, ce qui suit vn (vertex normal) dans le vecteur normal et enfin vt (vertex texture) dans le vecteur texCoords. Avec un peu d'acharnement nous avons réussi à créer notre propre Parser pour lire des fichiers .obj comportant des faces de maximum 3 sommets, voici un pseudocode très simplifié de la fonction :

```
fonction loadObjFile(nomFichier: chaîne de caractères) -> vecteur de ShapeVertex
    vertices <- nouveau vecteur de ShapeVertex
    fichier <- ouvrir le fichier nomFichier
    si le fichier n'a pas pu être ouvert, afficher msg erreur et retourner vecteur vide
    positions, normals, texCoords <- nouveaux vecteurs de vec3 et vec2
    ligne <- chaîne de caractères
    tant que l'on peut lire une ligne du fichier
        ligne <- lire une ligne du fichier
        ss <- nouveau stringstream avec la ligne lue
        type <- extraire le premier mot de ss
        si type est égal à "v", extraire x, y et z de ss et ajouter un nouveau vec3 avec
        ces coordonnées à positions
        sinon si type est égal à "vn", extraire x, y et z de ss et ajouter un nouveau vec3 avec
        ces coordonnées à normals
        sinon si type est égal à "vt", extraire x et y de ss et ajouter un nouveau vec2 avec
        ces coordonnées à texCoords
        sinon si type est égal à "f"
            pour i allant de 0 à 2, extraire la chaîne de caractères vertexStr de ss
            vss <- nouveau stringstream avec vertexStr
            extraire les indices posIndex, texIndex et normalIndex de vss en utilisant le caractère '/'
            ajouter un nouveau ShapeVertex avec les positions, normales et coordonnées de
            texture correspondantes à vertices
        fermer le fichier
        retourner vertices
    fin fonction
```

C'est assez imbuvable comme ça, mais ça marche (youpi) !

3.3 Optimisation du code et des fichiers

En OpenGL ou du moins dans notre code, de nombreuses lignes viennent à se répéter notamment lors de la création des textures, des vbo, des vaos, etc... et si notre code n'est pas assez organisé il est très facile de se perdre. C'est pourquoi nous avons créé des fichiers avec des classes nous permettant d'avoir une vision plus claire de ce que nous faisons. Nous allons prendre l'exemple du fichier Programs.hpp, celui-ci est composé de deux structures : OneTextureProgram ainsi que OneTextureLightProgram. À quoi servent ces structures ? Tout simplement à charger les shaders correspondants et récupérer la location des différentes variables uniformes. Puisque celles-ci ne sont pas les mêmes en fonction des shaders utilisés, il est important d'en avoir plusieurs. Récupérer les locations des variables uniformes peut prendre beaucoup de place dans le code, c'est pourquoi on décide de regrouper tout dans un seul fichier, voici un exemple :

```
struct OneTextureProgram {
    p6::Shader m_Program;

    GLuint uMVPMatrix;
    GLuint uMVMatrix;
    GLuint uNormalMatrix;
    GLuint uTexture;

    OneTextureProgram()
        : m_Program{p6::load_shader("shaders/3D.vs.glsl", "shaders/tex3D.fs.glsl")}
    {
        uMVPMatrix     = glGetUniformLocation(m_Program.id(), "uMVPMatrix");
        uMVMatrix      = glGetUniformLocation(m_Program.id(), "uMVMatrix");
        uNormalMatrix = glGetUniformLocation(m_Program.id(), "uNormalMatrix");
        uTexture       = glGetUniformLocation(m_Program.id(), "uTexture");
    }
};
```

Les fichiers Setup.hpp et Setup.cpp permettent de créer les textures, les vbo, les vao ainsi que de dessiner nos objets dans la scène. Dit comme ça ça a peut-être l'air de rien, mais c'est de nombreuses étapes et lignes de code qui se répètent c'est pourquoi faire des fonctions qui prennent en paramètre les bonnes variables permet d'alléger (énormément) notre main et ainsi avoir quelque chose de beaucoup plus lisible. Voilà un exemple :

```

void create_vao(GLuint& vao, GLuint& vbo)
{
    static constexpr GLuint VERTEX_ATTR_POSITION = 0;
    static constexpr GLuint VERTEX_ATTR_NORMAL = 1;
    static constexpr GLuint VERTEX_ATTR_TEXCOORDS = 2;

    glGenVertexArrays(1, &vao);

    glBindVertexArray(vao);

    glEnableVertexAttribArray(VERTEX_ATTR_POSITION);
    glEnableVertexAttribArray(VERTEX_ATTR_NORMAL);
    glEnableVertexAttribArray(VERTEX_ATTR_TEXCOORDS);

    glBindBuffer(GL_ARRAY_BUFFER, vbo);

    glVertexAttribPointer(VERTEX_ATTR_POSITION, 3, GL_FLOAT, GL_FALSE, sizeof(glimac::ShapeVertex),
                          (const GLvoid*)offsetof(glimac::ShapeVertex, position));
    glVertexAttribPointer(VERTEX_ATTR_NORMAL, 3, GL_FLOAT, GL_FALSE, sizeof(glimac::ShapeVertex),
                          (const GLvoid*)offsetof(glimac::ShapeVertex, normal));
    glVertexAttribPointer(VERTEX_ATTR_TEXCOORDS, 2, GL_FLOAT, GL_FALSE, sizeof(glimac::ShapeVertex),
                          (const GLvoid*)offsetof(glimac::ShapeVertex, texCoords));

    glBindBuffer(GL_ARRAY_BUFFER, 0);

    glBindVertexArray(0);
}

```

Ce qui nous permet ensuite dans le main d'avoir quelque chose de très lisible comme ceci :

```

////////// VBOS & VAOS //////////

size_t                                nb_objects = 5;
std::vector<GLuint>                  vbos(nb_objects);
std::vector<GLuint>                  vaos(nb_objects);
std::vector<std::vector<glimac::ShapeVertex>> shapes;

shapes.push_back(loadObjFile("./assets/models/spaceshipLD.obj"));
shapes.push_back(loadObjFile("./assets/models/spaceshipHD.obj"));
shapes.push_back(loadObjFile("./assets/models/background.obj"));
shapes.push_back(loadObjFile("./assets/models/asteroidLD.obj"));
shapes.push_back(loadObjFile("./assets/models/asteroidHD.obj"));

for (size_t i = 0; i < nb_objects; i++)
{
    create_vbo(vbos[i], shapes[i]);
    create_vao(vaos[i], vbos[i]);
}

```

Et pareil lors de la libération des ressources, il suffit d'appeler les fonctions pour libérer les buffers en bouclant sur les vecteurs de vbo et vao (de même pour les textures).

3.4 Textures et lumières

Une autre partie importante du projet sont les textures et les lumières. Lors des TP nous avons réussi à faire les deux mais jamais en même temps. L'idée dans ce projet est alors de pouvoir projeter de la lumière sur nos éléments 3D quand bien même ils sont texturés. Pour ce faire il a fallu modifier notre shader afin qu'il prenne en compte ces deux éléments. De plus, dans notre scène il y a deux types de lumières : les pointLight qui agissent comme un soleil et les directionalLight qui éclairent devant le joueur, il a donc fallu implémenter les fonctions correspondantes. Voilà à quoi ressemble notre shader en sortie :

```
#version 330 core

in vec3 vPosition_vs;
in vec3 vNormal_vs;
in vec2 vTexCoords;

uniform vec3 uKd;
uniform vec3 uKs;
uniform float uShininess;

uniform vec3 uDirLightDir_vs;
uniform vec3 uDirLightColor;

uniform vec3 uPointLightPos_vs;
uniform vec3 uPointLightColor;
uniform float uPointLightIntensity;

uniform sampler2D uTexture;

out vec4 fFragColor;

vec3 blinnPhongPoint(vec3 position, vec3 normal, vec3 kd, vec3 ks, float shininess, vec3 uLightIntensity,
                      vec3 uLightPos_vs)
{
    vec3 viewDir = normalize(- position);
    vec3 lightDir = normalize(uLightPos_vs - position);
    float distanceToLight = length(uLightPos_vs - position);
    vec3 attenuatedLightIntensity = uLightIntensity / (distanceToLight);
    vec3 halfVector = normalize(lightDir + viewDir);

    float diffuse = max(0.0, dot(normal, lightDir));
    float specular = pow(max(0.0, dot(normal, halfVector)), shininess);

    return kd * attenuatedLightIntensity * diffuse + ks * attenuatedLightIntensity * specular;
}

vec3 blinnPhongDir(vec3 position, vec3 normal, vec3 lightDir, vec3 lightIntensity, vec3 kd, vec3 ks,
                    float shininess)
{
    vec3 viewDir = normalize(-position);
    vec3 halfVector = normalize(lightDir + viewDir);

    float diffuse = max(0.0, dot(normal, lightDir));
    float specular = pow(max(0.0, dot(normal, halfVector)), shininess);
```

```

        return kd * lightIntensity * diffuse + ks * lightIntensity * specular;
    }

void main()
{
    vec4 texture = texture(uTexture, vTexCoords);
    vec3 diffuseColor = texture.xyz;

    vec3 normal = normalize(vNormal_vs);
    vec3 lightDir = normalize(uDirLightDir_vs);

    vec3 color = blinnPhongPoint(vPosition_vs, normal, uKd, uKs, uShininess, uPointLightIntensity *
        uPointLightColor, uPointLightPos_vs);
    color += blinnPhongDir(vPosition_vs, normal, lightDir, uDirLightColor, uKd, uKs, uShininess);

    fFragColor = vec4(color * diffuseColor, texture.w);
}

```

Encore une fois ce n'est pas très accueillant à l'oeil mais les lumières fonctionnent. Par ailleurs nous avons aussi créé des struct afin de pouvoir contenir les informations liées aux lumières (position, direction, couleur, intensité...). Il suffit ensuite de les créer dans le main avec les paramètres souhaités :

```

PointLight pointLight(glm::vec3(0.0f, 10.0f, 0.0f), glm::vec3(1.0f, 1.0f, 1.0f), 15.0f);

DirectionalLight directionalLight(glm::vec3(1.0f), glm::vec3(0.0f));

```

En conclusion nous sommes content du résultat que nous avons obtenu, mais nous nous sommes rendu compte de la difficulté à manier une telle technologie. C'est vrai que lorsque aujourd'hui on manipule des moteurs de jeu comme Unity ou Unreal on ne se doute pas forcément du nombre d'étapes colossal qui peut se passer en arrière plan. C'est un projet conséquent qui a demandé pas mal de recherches et de réflexions, mais nous avons beaucoup appris et c'est le plus important.



Nos Boids en 2D



Puis en 3D !