




The Colorless Journey

A Firefly Story

Synthèse d'images | IMAC2 | Juin 2023

Lauriane Gélébart



Partie 1 - Présentation de l'environnement

Pour ce projet de synthèse d'image, j'ai réalisé un environnement en 3D avec OpenGL. Le protagoniste (une luciole) se promène dans un monde qu'on pourrait voir comme un monde de papier. Dans cet espace où tout est blanc, il émet de la lumière jaune qui éclaire et permet de découvrir l'environnement qui l'entour. L'autre source de couleur est les bords, qu'on peut voir au choix comme d'autres petites lucioles ou comme des orbes colorées, et qui tourne autour d'une stèle magique. L'idée est d'attirer l'attention de l'utilisateur sur cette zone, car cette pierre magique permet de déclencher l'action du jeu : le retour de la couleur (et de la luminosité) dans l'environnement.

L'entièreté du décor a été créée au préalable sur blender par mes soins. Aucun objet et aucune texture n'ont été importés. Les objets n'ont pas beaucoup de détails, car les textures le leur donnent. En plus de donner un style "dessin d'enfant", cette idée permet de gagner en performance. En effet, les modèles sont simples et font rarement plus de 3 000 lignes.

Les éléments de décors : sapins, champignon sont placés et distribués de manière aléatoire dans l'espace. Les seuls décors placés manuellement sont ceux qui sont ponctuels comme la maison, les rails ou la pierre magique.

Partie 2 - Comment afficher des objets

J'ai modélisé mon environnement sur Blender. Pour l'importer, j'ai codé 2 fonctions loader qui lisent respectivement le fichier .obj et son fichier .mtl associé. Les fichiers sont lus lignes par lignes. En fonction du début de la ligne, les coordonnées sont stockées dans le tableau vertexData.

Chaque fichier obj est un objet qui, dans la majorité des cas, est constitué de deux ou trois matériaux différents. Aussi, j'ai pris la décision de séparer et donc d'afficher les objets en parties, une par matériau. Cette classe nommée ModelPart possède 6 attributs :

- le nom du matériau associé (string)
- un vecteur de vertex de données (float)
- un vecteur d'indice (float)

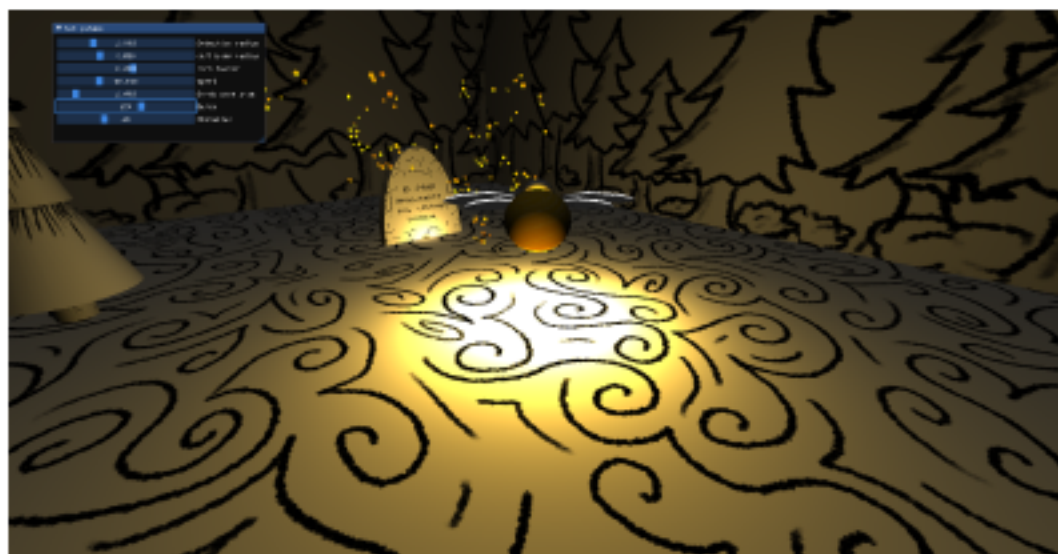
Ainsi qu'un VAO, un VBO et un IBO.

La classe Objet stocke ainsi un vecteur de ModelPart et via une boucle for affiche l'objet au complet.

Pour ce qui est des matériaux, plusieurs objets peuvent avoir le même, par exemple le puits et la maison utilisent le matériau brick. Il n'était donc pas cohérent de stocker directement un matériau dans ModelPart, et d'en créer un à chaque ligne "usemtl" du fichier obj. Aussi l'environnement de jeu stocke indépendamment des objets, une map associant un nom à un matériau. De cette façon, avec l'attribut nom de ModelPart on peut retrouver son matériau associé.

Partie 3 - Les boids

Les boids de l'environnement sont de petites boules colorées qui se déplacent autour de la stèle magique. Leur comportement est modifiable grâce à l'IHM.



Ils ont leur propre programme et leurs propres shaders. En effet, à la différence des objets de la scène, les boids n'ont pas de textures. Lors de leur initialisation, le constructeur leur donne une couleur aléatoire entre le jaune et l'orange en utilisant un random sur la valeur de vert. Les boids ont plusieurs fonctions de collisions qui se font de deux manières différentes dans le programme. La première utilisée pour les collisions avec les objets utilise un calcul de normal de l'objet pour changer instantanément la valeur du vecteur directeur du boid et le faire repartir dans l'autre sens.

```
vec3 normal <- boidPosition - obstaclePosition
normaliser(normal)
boidVelocity <- normal * 0.01
```

La deuxième, utilisée avec le bord de la zone dans laquelle les boids évoluent, fait des vérification en x, en y, et en z et utilise un facteur de virage :

```
Si positionBoid.x > positionSteleMagique.x + tailleZoneBoid
  boidVelocity.x <- boidVelocity.x - ihmTurnFactor
Fin si
Si positionBoid.x < positionSteleMagique.x - tailleZoneBoid
  boidVelocity.x <- boidVelocity.x + ihmTurnFactor
Fin si
[...]
```

C'est une méthode plus coûteuse en calcul, car pour chaque boid on effectue 6 vérifications. Pour autant, j'ai choisi de conserver cette solution, car elle permet d'avoir de très beaux virages des boids. Leur comportement est beaucoup plus intéressant, car ils ne donnent pas du tout l'impression de rebondir contre une paroi. À savoir que le facteur de virage est un des éléments modifiables dans l'IHM.

Partie 4 - Mise en place de LOD

Les boids sont de petites sphères colorées. Aussi, de loin, il est judicieux de réduire le nombre de triangles qui composent cette sphère. Pour faire fonctionner ce principe, à la différence des autres objets de la scène, les boids n'ont pas un, mais deux modèles. Il suffit donc ensuite de passer de l'un à l'autre en fonction de la distance entre un boid et la caméra. On peut tout de même remarquer que même si les lods sont moins lourds à charger, à chaque frame il faut faire le calcul de la distance avec la caméra pour chaque boid. J'avais à un moment pensé à la place, à calculer la distance entre la caméra et la pierre magique autour de laquelle gravitent les boids. L'idée était qu'approximativement les boids ont une position proche de celle de la pierre. Mais je n'ai pas retenu cette idée, car avec la mise en place de l'IHM, je laisse la possibilité à l'utilisateur d'agrandir la zone de déplacement des boids. Ainsi, comme cette zone peut faire la taille de l'environnement, se baser sur la position de la pierre n'a plus de sens. Pour régler ce problème de calcul récurrent de la distance, la mise en place d'un quadtree fragmentant en zones les positions des boids, pourrait être une amélioration intéressante pour le programme.

Par la suite, on pourrait également imaginer poursuivre dans cette voie avec tous les objets de la scène. Par exemple, sur certains modèles comme les champignons qui de loin ne se voient pas trop, les LOD pourraient permettre un gain de performance. Si l'on garde cet exemple du champignon, à l'heure actuelle, l'objet utilisé fait toujours plus de 2000 lignes. Dans l'image du dessous le champignon sur la droite a un fichier obj de 650 lignes.



Une des améliorations possibles du projet pourrait ainsi être la mise en place de l'LOD sur tous les objets. Encore une fois un quadtree serait pertinent pour ne pas avoir à calculer des distances entre tous les objets et la position de la caméra.

Partie 5 - La caméra

La caméra était un sujet intéressant. En cours nous en avons vu deux types. La trackball caméra et la freefly caméra. Pour résumer, la trackball caméra permet de contrôler la vue de la caméra en fonction des mouvements de la souris autour d'un point central, tandis que la freefly caméra permet une navigation libre dans l'environnement virtuel en utilisant les mouvements de la souris et des touches pour déplacer et orienter la caméra. À première vue une freefly caméra semblait adéquate. Pourtant la fonctionnalité de la trackball de tourner autour d'un objet central peut également être intéressante. En effet, comme l'on souhaite suivre à la troisième personne l'arpenteur dans la scène, on veut (comme dans n'importe quel jeu vidéo à la troisième personne) pouvoir tourner par rapport à la position de son axe Y et pas par rapport à celui de la caméra. Ainsi, faire une trackball qui se déplace est plus pertinent. J'ai donc repris une partie du code des tps et l'ai adapté en ajoutant des fonctions `moveFront` et `moveLeft`. Une fois cette partie réalisée, j'ai décidé que finalement ma luciole se dirigerait en fonction de l'angle Y de ma caméra, aussi la caméra tourne autour de mon modèle, mais lui tourne également. Cette rotation de la caméra se fait via la souris. Une fois cette caméra mise en place, une nouvelle problématique à laquelle je n'avais pas pensé, s'est posée lorsque j'ai voulu interagir avec l'IHM. Comme chaque mouvement de la souris fait tourner la caméra, on ne pouvait pas changer les paramètres de l'IHM sans que ce ne soit désagréable car la caméra tourne. Aussi, j'ai ajouté la possibilité de désactiver cette rotation de la caméra en fonction de la souris en appuyant sur la touche "Alt".

Partie 6 - Jeu de textures

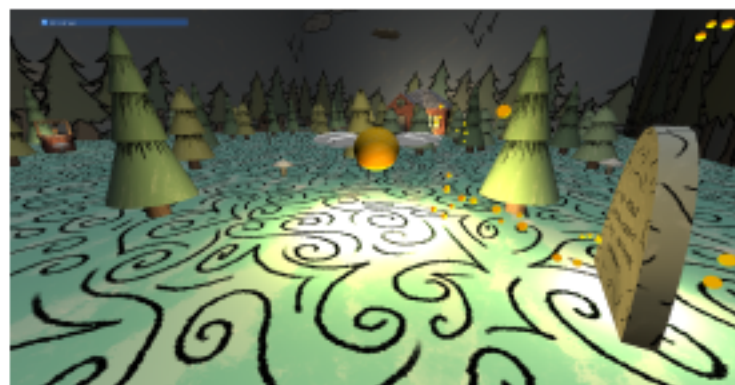
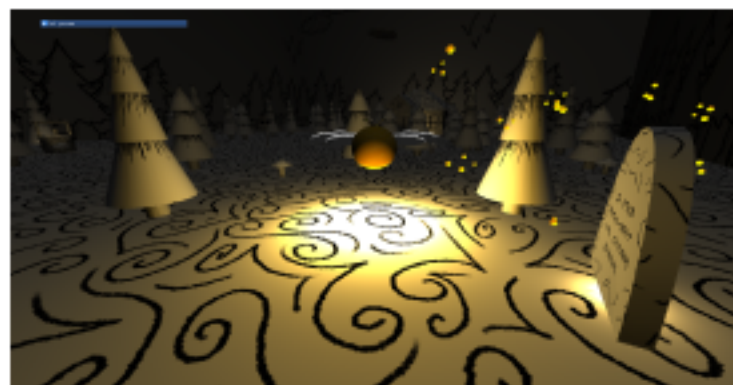
Pour aller plus loin que les consignes, j'ai décidé de créer une interaction changeant les textures et les lumières. En effet, en appuyant sur "Espace" près de la pierre magique autour de laquelle gravitent les boids, l'environnement change l'entièreté des textures du jeu. Toutes les textures qui étaient blanches retrouvent leurs couleurs.

Cette interaction, qui donne son nom au projet, est possible grâce à plusieurs points. Premièrement, comme dit plus haut, j'ai modélisé l'entièreté de mes objets, mais ce que je n'ai pas précisé c'est que j'ai également dessiné moi-même toutes les textures. Aussi, pour chaque texture, j'ai créé la version blanche et la version en couleurs.



Par la suite, dans mon programme, j'ai créé un vecteur de textures stocké dans ma structure Material. Un material a donc plusieurs textures. Comme j'ai codé le loader, lorsque je lis le chemin de la texture dans le fichier .mlt, j'en lis en réalité 2 séparés d'un espace. Cette subtilité nécessite de modifier le fichier au préalable pour ajouter le deuxième chemin de la texture. C'est un choix qui peut-être discutable, car il me contraint à modifier systématiquement tous les .mlt que j'utilise dans le programme. Pour autant, je pense que c'est la solution la plus simple pour récupérer 2 chemins de texture pour un même material. Une fois stockées dans un vecteur de textures, je n'ai qu'à choisir l'indice que je souhaite. J'ai défini 0 comme les textures sans couleurs et 1 pour les colorées. Cette manière de faire pourrait permettre d'ajouter un bon nombre de textures pour tous les objets et de continuer le jeu avec de nouvelles interactions.

Pour lancer ce changement, je détecte l'appui de la touche espace et vérifie que l'on est bien proche de la pierre magique. Si c'est le cas, je passe l'attribut `_color` du `GameEnvironment` de `false` à `true`. Vous pourrez remarquer que cela a également pour effet de modifier l'éclairage de la scène. En effet, j'augmente également l'intensité de la lumière que j'envoie à mon shader.



Partie 7 - Affichage des informations

Comme j'ai mis en place certains contrôles particuliers (Alt pour la caméra et Entrée pour l'interaction) je trouvais très important de prévenir les utilisateurs. C'est pourquoi j'ai cherché un moyen d'afficher des informations. La technique retenue est d'ajouter des parallélépipèdes rectangles face à la caméra, donc simplement vus comme des rectangles et d'y ajouter une texture explicative des touches.

Ce rectangle ne devait pas prendre toute la taille de l'écran, mais être placé au centre de l'image, laissant ainsi voir l'environnement derrière. L'idée était également de bloquer le déplacement lorsqu'un menu s'affiche. Pour en sortir, il suffit de faire "Entrée" ou "Échappe". Cette technique permet de rendre plus user friendly la navigation dans l'environnement.



J'ai créé un programme spécifique pour cette classe (Panel) d'objets un peu particulier. Comme un menu ne fait pas réellement partie de l'environnement 3 d, il ne fallait pas que les lumières soient prises en compte. Les objets de ce type ont de cette manière, un fragment shader différent dans lequel seule la texture est affichée.



Aussi, une fois le panneau de début du jeu mis en place, je me suis dit, pourquoi ne pas en ajouter dans le jeu. L'idée était d'indiquer à l'utilisateur qu'il y a sûrement une interaction autour de la pierre en affichant une phrase mystérieuse (traduction du texte écrit sur la pierre). Mais les choses se sont compliquées à ce moment-là. En effet, si le premier menu est simple à placer puisqu'on connaît la position de la caméra et son orientation au début du jeu, il est impossible de prévoir à quel endroit le joueur entrera dans un périmètre suffisamment proche de la pierre pour afficher le panneau. Pour le placer visuellement au même endroit que le premier, j'ai calculé la position sphérique et l'inclinaison du panneau avec l'angle en X et Y de la caméra :

```
Soit vec3 viewMatrixPosition, _position
float angleX, angleY, x, y, z
Fonction appears(viewMatrix)
    viewMatrixPosition <- position viewMatrix
    angleX <- - angle autour de l'axe X viewMatrix
    angleY <- - angle autour de l'axe Y viewMatrix
    x <- sin(90 + angleX) * cos(angleY) * 0.35
    y <- cos(90 + angleX) * 0.35
    z <- sin(angleY) * sin(90 - angleX) * 0.35
    _position <- (viewMatrixPosition.x + x, viewMatrixPosition.y + y,
viewMatrixPosition.z + z)
    _angleX <- angleX
    _angleY <- angleY
    [...]
fin fonction
```

Ainsi, désormais, je peux ajouter à n'importe quel endroit du jeu un panneau d'information. C'est une fonctionnalité dont je suis assez contente, car elle rend l'environnement utilisable par n'importe qui sans devoir mettre une notice à lire au préalable.



Partie 8 - Les ombres

Il n'y a pas d'ombres sur le projet. Pourtant, j'avais commencé à en mettre en place, mais par manque de temps, je n'ai jamais terminé. Aussi, on peut trouver dans le projet une classe Shadow non utilisée, un programme, un fragment shaders et un vertex shader spécifiques. Le chantier des ombres prévoyait d'utiliser la technique de shadow mapping. Le principe : rendre une première fois la scène du point de vue de la lumière, avant de la rendre du point de vue de la caméra. De cette manière, on crée des ombres en testant si un pixel est visible ou non depuis la source lumineuse. Puis on stocke la profondeur de chaque surface que voit la lumière. On crée ainsi une carte d'ombres qu'on utilise par la suite comme une texture dans le deuxième rendu (vision caméra).

Là où j'en suis, j'ai créé une fonction qui initialise un FBO. Puis je peux rendre la scène du point de vue de la caméra. La suite du travail aurait notamment consisté à utiliser ma shadow map comme texture. J'avais commencé à coder une fonction de calcul des ombres dans le fragment shader, mais faute de temps, je n'ai jamais pu terminer sa mise en place.

Voici la fonction ShadowCalculation que j'aurai aimé faire fonctionner dans le fragment shader de mes objets.



```

Soit vec4 fragPosLightSpace, vec3 projCoords
float closestDepth,currentDepth,shadow
Fonction ShadowCalculation(fragPosLightSpace)
    // Effectuer la division en perspective
    projCoords <- fragPosLightSpace.xyz / fragPosLightSpace.w
    // Transformer en plage [0,1]
    projCoords <- projCoords * 0.5 + 0.5
    // Obtenir la valeur de profondeur la plus proche de la perspective de la
    lumière (en utilisant les coordonnées projCoords.xy dans la plage [0,1])
    closestDepth <- texture(shadowMap, projCoords.xy).r
    // Obtenir la profondeur du fragment actuel depuis la perspective de la
    lumière
    currentDepth <- projCoords.z
    // Vérifier si la position actuelle du fragment se trouve dans l'ombre
    Si currentDepth > closestDepth alors
        shadow <- 1.0
    Sinon
        shadow <- 0.0
    Fin si
    Retourner shadow
Fin fonction

```

Partie 9 - Conclusion

Dans les différentes parties ci-dessus, vous avez pu constater qu'il reste des axes d'amélioration au projet, notamment la mise en place des ombres. En effet, l'une des choses que m'a apprises ce projet (en plus de progresser en OpenGL bien sûr...), c'est faire choix. Le temps était limité, au vu de la charge de travail demandée ce semestre, j'ai dû faire une croix sur certaines fonctionnalités, ou certaines optimisations pour en privilégier d'autres. Par exemple, pour moi, la mise en place des panneaux d'information est une fonctionnalité utile, je pense que privilégier l'UX rend l'environnement plus simple et agréable à utiliser. J'ai également choisi de mettre en place une fonctionnalité originale (changement de couleur des textures) au détriment d'autres éléments de la consigne. Je trouve que cette interaction apporte vraiment quelque chose à ce projet. C'est une fonctionnalité que j'aime beaucoup et que je suis fière de montrer. Elle n'est pas la plus complexe, mais elle donne une âme, un sens et une originalité à ce projet. C'est sur ça que je veux conclure. C'était un projet ambitieux, d'autant plus que je l'ai réalisé seule, mais avant tout, c'était un projet très intéressant et créatif. L'environnement 3d que je rends ne respecte pas tous les critères, mais visuellement (même sans ombres) je le trouve abouti, et l'aime beaucoup.

The page features decorative swirls in the corners, rendered in a simple line-art style. These swirls are located in the top-left, top-right, bottom-left, and bottom-right corners, framing the central text area.

Lauriane Gélébart
Juin 2023