CREPIN Olivia
LAFAURIE Aurore
IMAC 2

RAPPORT DU PROJET
PROGRAMMATION ET SYNTHÈSE D'IMAGES

4 Le rapport

Vous fournirez **en version électronique** un rapport de 10 pages maximum. **Consacrez suffisament de temps au rapport car il représente une partie de la note finale.** Essayez de respecter au mieux les directives suivantes :

- Le rapport vise à expliciter vos choix algorithmique et fonctionnel. Les éventuelles formules d'interaction nouvelles par exemple. Ou les détails sur la technique d'ombrage utilisée.
- Ne perdez pas de temps à réexpliquer le sujet du projet, l'enseignant le connaît déjà, faites seulement un bref résumé de l'idée générale de votre application. De manière plus générale, ne détaillez pas des méthodes déjà expliquées dans l'énoncé mais plutot celles que vous avez développées.
- Un rapport sert aussi à montrer comment vous avez fait face aux problèmes (d'ordre algorithmique). Certains problèmes sont connus (on en parle dans l'énoncé), d'autres sont imprévus. Montrez que vous les avez remarqués et compris. Donnez la liste des solutions à ce problème et indiquez votre choix. Justifiez votre choix (vous avez le droit de dire que c'est la méthode la plus facile à coder). A savoir : "on a eu un super problème par ce que ça compilait pas ..." ne nous intéresse pas.
- Il ne doit figurer aucune ligne de code dans votre rapport. Un rapport n'est pas un listing de votre programme où vous détaillez chaque fonction. Vous devez par contre détailler vos structures de données et mettre du pseudocode pour expliquer vos choix algorithmiques.
  Il est autorisé d'utiliser des "raccourcis" tels que "`initialiser le tableau tab a 0`" plutôt que de détailler la boucle faisant la même chose.
- n'hésitez pas à mettre des images dans votre rapport pour illustrer vos propos et vos résultats.
- Enfin, il est **très important** de faire la liste de ce que vous avez fait, de ce qui fonctionne correctement et la liste des dysfonctionnements. Précisez quand il s'agit d'options que vous avez rajoutées en plus de ce qui était demandé. N'hésitez pas à montrer des timings et des screenshots.

Image de la simulation (début du jeu)

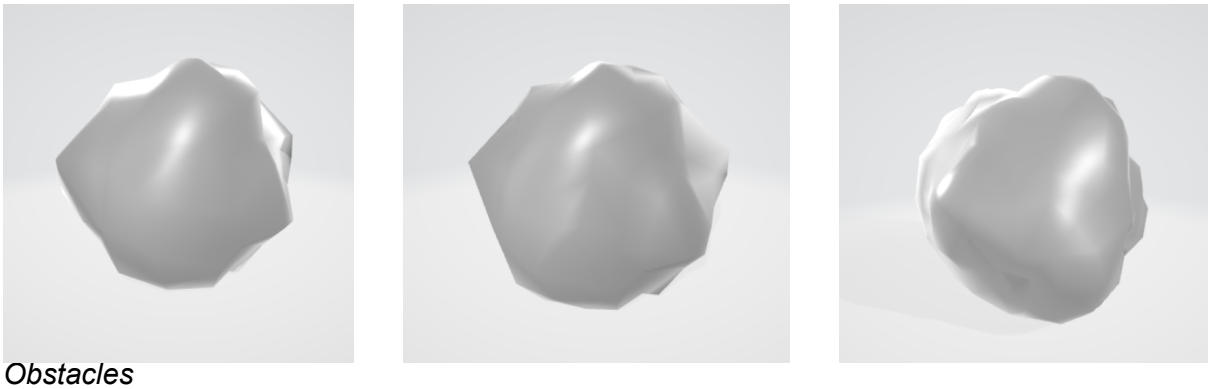
SOMMAIRE

Introduction

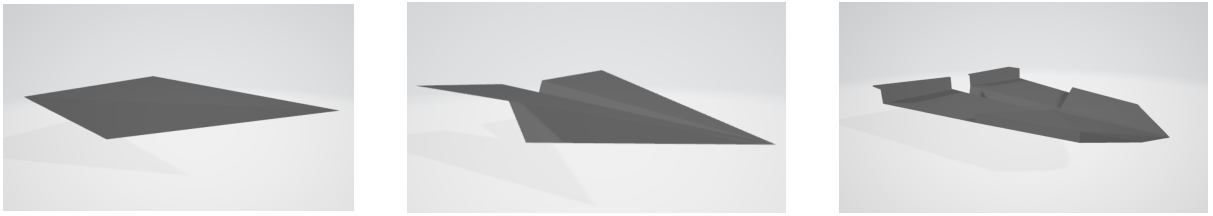
Notre programme autorise l'utilisateur à manipuler les paramètres des boids et les paramètres de l'environnement à sa convenance à partir de l'interface ImGUI. Il devient maître de l'application. Il est possible de se déplacer dans la cage grâce à son clavier et sa souris et de générer des obstacles que les boids pourront éviter.

I - Les Modèles 3D et LOD

Les modèles 3D ont été réalisés sur Blender, les boids et les obstacles possèdent 3 différents niveaux de résolutions que vous pouvez voir ci-dessous. Leurs textures ont été réalisées avec du VertexPaint.



Obstacles



Boids

Pour charger les modèles, nous avons décidé d'écrire notre propre parser, afin de découvrir le fonctionnement des fichiers .obj mais aussi de ne pas être dépendant d'une bibliothèque. Cela nous a donc permis de gérer au mieux les vertex color puisqu'il fallait lire 6 informations sur un vertex au lieu de 3 (x, y, z et r, g, b).

De plus, même si nous avons globalement utilisé les vertex colors, nous avons choisi d'avoir un cube avec une texture pour englober la simulation. Nous avons donc créé une classe *Texture* pour pouvoir la dessiner. Ce qui est intéressant avec la gestion de ses 2 techniques de couleurs est que l'application se fait via le même shader en précisant via un booléen l'utilisation ou non de textures.

Nous avons créé deux classes et une structure afin de gérer les modèles et leurs différentes résolutions. La classe *ModelsLOD* permet de définir plusieurs niveaux de détails pour un objet et ainsi de dessiner la bonne instance. La classe *Model* quant à elle génère et construit l'objet. Enfin, la structure nommée *ModelParams* permet d'initialiser les paramètres physiques du modèle. Cela permet d'uniformiser les dessins un peu partout dans le code.

II - La Cubemap

Pour l'environnement nous avons décidé de mettre une skybox, aussi appelée cubemap dans ce cas précis. Nous voulions un décor représentant un environnement global qui ainsi ne bougerait pas du point de vue de la caméra et donnerait un sentiment de profondeur.

En partant d'une HDRI, il a fallu découper cette image de manière à ce qu'elle puisse être dessinée comme un cube. Ensuite, en important toutes les faces correspondant au cube et en fixant ainsi toutes les textures correspondantes à une cubemap, il ne reste plus qu'à dessiner.

Pour ce qui est du dessin, il est important de dessiner la skybox en dernier, car il est nécessaire de changer la fonction de profondeur. Cette fonction de profondeur modifiée permet de donner l'impression que l'environnement ne bouge pas et donc qu'il est possible de se déplacer de manière infinie.

Afin d'ajouter une interactivité avec l'utilisateur, nous avons choisi de permettre, via un *slider,* de modifier l'exposition de l'environnement. Même si cela n'affecte pas les autres modèles par des ombres, cela permet de donner une ambiance plus lumineuse.

III - L'Arpenteur et les Obstacles

Pour l'arpenteur et son usage, nous n'avons pas trouvé utile de lui créer sa propre classe. En effet, en sachant qu'il allait suivre la caméra et donc dépendre de la position, de la rotation et de l'orientation de cette dernière, nous avons choisi de juste lui initialiser un modèle et bien sûr lui définir ses paramètres avec la structure *ModelParams*.
Après avoir réalisé cela, nous avons été confrontées à un problème, l'arpenteur ne restait pas à sa position, c'est-à-dire, au centre de l'écran. En se déplaçant dans l'environnement, l'arpenteur se décalait et lorsque l'on pivotait la caméra, il ne prenait pas en compte le changement et restait à sa dernière position. Une des solutions envisageables était qu'à la place de gérer des vecteurs individuels, nous passions les données de notre arpenteur en matrice et appliquions les transformations ensuite. Une autre solution possible était de décomposer la *ViewMatrix* et de récupérer les paramètres intéressants. Nous avons opté pour une autre solution qui nous semblait plus simple et qui évitait de devoir décomposer pour recomposer la *ViewMatrix* : un *getter.* Pour obtenir la rotation de la caméra, nous avons créé un *getter* qui récupère le vecteur du *forward axis* de la caméra. Ensuite, nous avons remarqué que la *ViewMatrix* subissait des transformations qui provoquent ce décalage de position pour l'arpenteur. En suivant la même méthode que pour le *forward axis,* nous avons créé un *getter* qui récupère la position de la caméra avant transformation de la matrice.

En ce qui concerne les obstacles nous avons décidé de laisser l'utilisateur les placer avec sa souris au clic droit. Grâce à l'interface ImGUI, il peut supprimer la liste de tous les obstacles pour les remplacer. En laissant le choix de position à l'utilisateur une problématique apparaît : si le clic est trop proche, les obstacles se superposent. Il faut donc éviter cela. Nous avons travaillé sur les obstacles d'abord en 2D avant de passer notre projet en 3D. La solution pour laquelle nous avons opté consiste à définir à rayon autour de l'obstacle déjà placé et de comparer leur longueur : `comparer la distance entre le`

`clic et 3 fois le rayon de l'obstacle.` Après le passage en 3D, une autre solution est possible, si le clic est proche d'un autre obstacle, on positionne le nouvel obstacle avec un z différent. Bien sûr, ce z doit toujours être en dehors du rayon de l'obstacle placé. Cette deuxième solution suit le même principe que la première, nous avons donc décidé de garder cette dernière qui était déjà implémentée.
Le passage à la 3D nous a posé des difficultés. En effet, après avoir cliqué à une certaine position, l'obstacle se positionnait avec un décalage conséquent. Cela était dû au fait qu'on avait la position du clic dans l'espace de l'écran alors qu'on souhaitait celle dans l'espace du monde. Pour obtenir la position du clic dans le monde 3D, le plus simple pour nous était de manipuler directement les matrices Projection et View :

```
inverser la View Projection Matrix.
récupérer la position du clic sur l'écran.
multiplier la VP Matrix avec la position du clic.
```

IV - Les Lights

Les lumières ponctuelles sont générées à partir du modèle de réflection Bling-Phong. Pour en afficher plusieurs nous avons procédé de telle manière :

```
Initialisation d'une constante NB_LIGHTS.
Initialisation d'un tableau uniforme de vecteurs de positions.
Pour toutes les NB_ LIGHTS, additionner et appliquer le
Bling-Phong en fonction de leur position.
```

Nous aurions pu créer une structure *Lights,* si l'on souhaitait modifier certains paramètres comme le shininess d'une light à l'autre. Cependant, dans notre cas d'utilisation, seules les différentes positions des lights nous intéressaient. Nous n'avons donc pas jugé nécessaire d'en créer une.

Une des lumières ponctuelles ne bouge pas afin de pouvoir dessiner des ombres visibles, tandis que la deuxième lumière ponctuelle est accrochée à l'arpenteur afin de lui donner de la visibilité.

De plus, nous avons mis une lumière ambiante afin de ne pas avoir un environnement complètement noir.

```
vec4 ambient = vec4(0.4, 0.5, 0.4, 1.);
```

La couleur ici définie a des nuances de vert afin de faire ressortir la couleur de l'herbe.

V - Le Shadow Mapping

Pour la gestion des ombres, nous avons décidé de nous y prendre en plusieurs fois. En effet, le *shadow mapping* peut être divisé en plusieurs étapes distinctes. La première est de pouvoir créer une texture, appelée *depth map*, et de sauvegarder dedans la profondeur du point de vue d'une lumière ponctuelle. Il y a donc un changement de projection car la projection d'une lumière ponctuelle n'est pas identique à celle du point de vue d'une caméra. Il s'agit donc de faire une première passe de dessin afin d'écrire dans la texture et donc de dessiner à minima les *boids*. Nous enregistrons donc la position des *casters* d'ombre. Pour

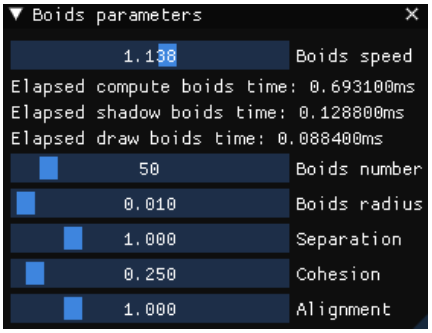
ce qui est des *receivers*, nous avons décidé que toute la simulation pourrait être impactée par les ombres.

Au moment du dessin, il suffit uniquement de penser à activer la *depth map* avant le dessin des *receivers*. Dans le shader, les profondeurs sont comparées entre celle du point de vue de la caméra et celle du point de vue de la lumière. Il est alors décidé si l'objet devrait avoir de l'ombre sur lui ou non. Le calcul se fait donc dans le shader, au moment où les lumières sont calculées et on retrouve la combinaison entre la couleur de l'objet, la lumière qui lui arrive dessus et l'ombre qui pourrait être affichée.

Un paramètre modifiable dans l'interface utilisateur permet aussi de choisir de dessiner ou non les ombres. Cela permet parfois d'obtenir une meilleure lisibilité de la simulation.

VI - L'interface ImGUI

Nous avons souhaité que l'utilisateur ait la main sur le plus de paramètres possibles. Nous avons donc ajouté le contrôle des boids, ainsi que l'affichage du temps de calcul, du temps de création des ombres et du dessin des boids. Cela peut être intéressant pour l'utilisateur s'il souhaite optimiser la simulation pour son ordinateur.



Il a aussi le contrôle de l'affichage des ombres, du passage à l'instancing, du niveau d'exposition de la skybox et il peut replacer la caméra à son origine. De plus, l'onglet "Help" donne des informations sur les contrôles de la simulation.

VII - L'Instancing

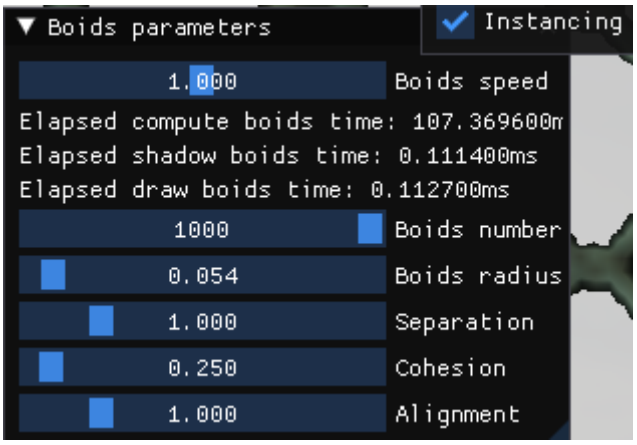
Pour pouvoir aller plus loin dans le projet, nous avons décidé de travailler sur les performances de la simulation. Pour cela, il a fallu regarder les temps de calcul et de dessins de nos *boids*. Il se trouve que, malgré le fait que nos boids soient de simples avions en papier avec assez peu de *vertex* notamment avec le LOD, la simulation est très vite ralentie par le nombre de modèles à afficher. Plus le nombre est important, plus le temps est long.

Après quelques recherches, nous avons décidé d'implémenter l'instancing à notre projet. Il s'agit d'une méthode de rendu permettant de rendre plusieurs instances d'un même modèle

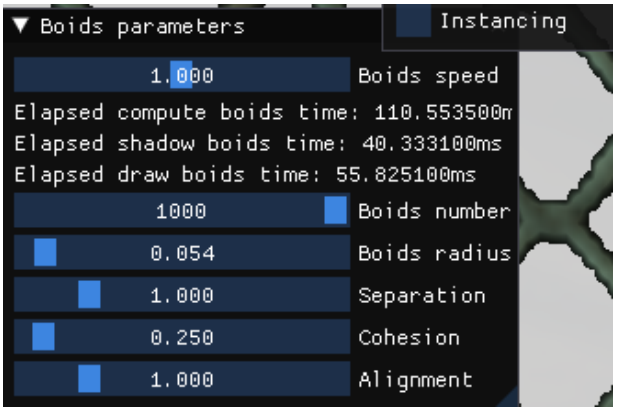
d'un seul appel OpenGL. Il suffit juste de passer toutes les matrices modèles à utiliser et la méthode se débrouille d'elle-même pour tracer toutes les instances.
Il a donc fallu créer une méthode de la classe *Model* afin de pouvoir différencier le dessin classique du dessin instancié.

De plus, afin de gagner plus de temps, nous avons vu que les calculs côté GPU étaient plus rapides qu'en CPU. Sachant qu'il est nécessaire de calculer toutes les matrices modèles pour ensuite dessiner, nous avons choisi de construire ces matrices directement dans le shader après avoir envoyé les informations nécessaires au GPU.

Au niveau des performances obtenues, quelques chiffres sont visibles pour la simulation comme le temps nécessaire pour dessiner tous les *boids* avec et sans *instancing*. Il est possible d'activer et de désactiver cette méthode de dessin pour faire des comparaisons.
En effet, avec 1000 *boids* dessinés on obtient les chiffres suivants :



Statistiques avec l'instancing activé



Statistiques sans instancing

Il est important de noter que l'instancing désactive la notion de dessin avec LOD dans notre projet. En effet, il aurait fallu trier et dessiner 3 modèles différents de manière instanciée, ce qui aurait pu être encore plus coûteux en temps. De plus, nos modèles de *boids* ne contenant que très peu de points, le coût de changement de modèle aurait été important.

Conclusion

En conclusion, nous avons essayé d'ajouter au projet quelques fonctionnalités afin de faire correspondre le sujet ainsi que nos souhaits sur le projet. En effet, nous voulions pouvoir obtenir une simulation ayant un temps de calcul et de dessin relativement optimisé. L'instancing a donc très vite été une option permettant de gagner en performances et ainsi de pouvoir dessiner en un temps record. Une des améliorations possibles de ce projet aurait d'ailleurs pu être de faire un algorithme de tri permettant de ne calculer les comportements des boids qu'avec ses voisins les plus proches (octree par exemple). Cette amélioration aurait permis d'obtenir des calculs de position réduits et ainsi d'obtenir une simulation optimisée.

État des lieux final

	Fonctionne	**Dysfonctionne**	**Non Fait**
Boids 3D	**x**		
Cohésion	**x**		
Séparation	**x**		
Alignement	**x**		
ImGui	**x**		
Cube d'environnement	**x**		
Restriction dans un cube	**x**		
Modèles 3D (parsing avec vertex colors et textures)	**x**		
LOD	**x**		
Caméra 3e personne	**x**		
Arpenteur fixe à la caméra	**x**		
2 lumières ponctuelles	**x**		
Ombres	**x**		
En plus			
Instancing	**x**		

Choix de l’apparition d’ombres	**x**		
Obstacles au clic	**x**		
HDRI et choix de son exposition	**x**		