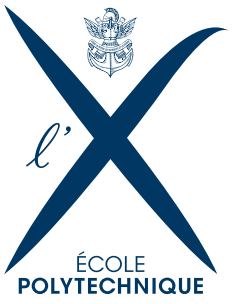

Marius Debussche,
Lilian Bichot

1er juin 2021

SYSTÈME STELLAIRE À GÉNÉRATION PROCÉDURALE

INF443: Rapport de projet

Table des matières

1 Introduction **2**

2 Planète **2**
2.1 Le corps . . . 2
2.2 Shaders . . . 3
2.2.1 L'eau . . . 3
2.2.2 L'atmosphère . . . 3
2.3 Versatilité . . . 3

3 Système physique **4**

4 Végétation **5**

5 Ciel **5**

6 Interaction **5**
6.1 Caméra . . . 6
6.2 Déplacement . . . 6

7 Optimisations et astuces **6**

8 Bibliographie **7**

1 Introduction

Ce projet a pour objectif de générer un système solaire (avec divers astres en rotation et révolution) différent du nôtre.

D'autre part, le joueur a la possibilité non seulement de tourner la caméra mais aussi de se déplacer totalement librement entre les astres, et même sur leur surface !

C'est un défi de pouvoir se déplacer dans une scène si grande sans problème de précision sur la représentation des nombres en mémoire, en particulier de la profondeur.

Enfin, si nous avons choisi de générer des planètes qui nous semblaient intéressantes et de composer un système solaire complet pour pouvoir ensuite l'explorer, la généricité de notre code et l'implémentation d'une interface utilisateur versatile permettent le fonctionnement d'un mode d'édition d'astre, dans lequel l'utilisateur a totale liberté sur les paramètres de notre code pour créer le corps céleste de son choix.

2 Planète

Les éléments de la scène les plus importants sont évidemment les planètes. Nous avons réalisé un système de génération versatile qui permet de générer des planètes très diversifiées à partir d'un nombre relativement restreint de paramètres.

2.1 Le corps

Le corps de la planète est initialement une sphère centrée sur l'origine. Nous avons choisi de réaliser notre propre *mesh* pour une sphère afin d'avoir des triangles d'une taille à peu près identique, contrairement à la *UV Sphere* dont la concentration de triangles aux pôles est plus élevée. Pour notre modèle de sphère, nous générons un octaèdre centré en zéro. Chaque vecteur est ensuite normalisé pour obtenir une sphère. Ainsi, augmenter la résolution de la sphère augmente uniformément (ou presque) le niveau de détail d'une planète [1].

Pour créer une planète, il suffit ensuite de créer une fonction qui donne la distance d'un point sur la sphère à son centre. Nous avons superposé différents bruits de Perlin pour créer des surfaces intéressantes comme des montagnes et des océans, et combiné tout cela en utilisant un masque également généré par bruit de Perlin.

Pour rajouter des détails à la surface de la planète, une *normal map* est utilisée.

Concernant la couleur, nous avons pris en compte à la fois la hauteur et la pente des sommets de la planète. Ces valeurs sont envoyées au *fragment shader*(en utilisant le *layout* de la couleur, le canal rouge pour la hauteur, le canal vert pour la pente) qui calcule la couleur adaptée au fragment. Si on passait la couleur des sommets et pas leur pente et leur hauteur, le *fragment shader* interpolerait la couleur des sommets et non pas leur pente et leur hauteur qui donne un résultat plus précis.

2.2 Shaders

Les *shaders* sont très efficaces pour augmenter la qualité du rendu graphique. Nous avons décidé d'utiliser les *shaders* pour afficher de l'eau (dans un premier temps) sur les planètes. Le problème est que le *shader* n'est appelé que pour les fragments de la planète, et l'eau peut très bien déborder de l'image de la planète et avoir besoin d'être affichée alors qu'il n'y a pas de fragment de planète derrière. C'est pourquoi il a fallu implémenter du *post processing*, à savoir faire le rendu des planètes (sans eau) dans une texture intermédiaire grâce à un *framebuffer* [2]. Il faut aussi conserver le tampon de profondeur, car nous en avons besoin pour calculer la couleur de l'eau. Après ce rendu intermédiaire effectué, il faut ensuite passer ces deux textures à un *shader* qui fera leur rendu à l'écran en calculant la couleur du fragment pour afficher l'eau.

2.2.1 L'eau

Pour l'eau, lors du rendu final, nous comparons la valeur de la profondeur au niveau du fragment à la distance caméra - sphère d'eau. C'est une étape particulièrement technique car la profondeur dans le *fragment shader* n'est pas la même que celle de notre scène : il faut d'abord récupérer la vraie valeur dans la texture passée au *shader*, puis calculer la bonne direction pour retrouver la position du fragment dans le repère de la caméra (d'autant plus que la profondeur est en réalité la profondeur depuis le plan de la caméra, et non pas depuis la caméra elle-même). Si la profondeur est plus grande que la distance à l'eau, cela signifie que ce fragment doit afficher de l'eau. On affiche ensuite une couleur différente en fonction de la profondeur d'eau (typiquement plus sombre). On a ainsi une impression de profondeur dans les océans.

Nous calculons aussi l'éclairage de Phong à la surface de l'eau qui est facile à déterminer, car la normale du fragment se calcule directement à partir de sa position sur la sphère. C'est évidemment possible de le désactiver pour avoir des océans qui semblent produire leur propre lumière, comme nous avons fait sur Vulkan avec les lacs de magma.

2.2.2 L'atmosphère

Pour l'atmosphère, nous nous sommes inspiré de la vidéo de Sebastian Lague [3] et de l'article [4]. Il suffit d'estimer la valeur de deux intégrales en chaque fragment, et ce pour trois valeurs de longueur d'onde différentes (ce qui est très gourmand en calculs pour obtenir un résultat satisfaisant). Physiquement on estime, le long du rayon par lequel passe le fragment considéré, la quantité de lumière reçu par le soleil. A chaque point, on tire un nouveau rayon dans la direction de la source de lumière et on regarde quelle est l'intensité lumineuse reçue, qui dépend donc de la quantité de gaz par lequel passe la lumière.

On peut donc observer une couleur dans le ciel lorsqu'on se pose sur une planète, et même un changement de cette couleur quand la source de lumière se couche.

2.3 Versatilité

Si les paramètres présentés plus hauts ne semblent pas adéquats pour implémenter un soleil, des astéroïdes, des nuages, ou encore de la glace, les plages de valeurs de certains paramètres permettent d'obtenir des effets intéressants.

Pour le soleil, il suffit de placer le niveau des océans légèrement au-dessus de celui du sol, et de lisser ce dernier. On donne alors au liquide une couleur jaune, et au sol une couleur ocre pour que la semi-transparence du liquide fasse de jolies nuances. Enfin, on donne à l'atmosphère un blanc éclatant pour simuler la luminosité de l'astre.

Pour obtenir un corps similaire à un astéroïde, comme Companion le satellite de Gwen, i.e. une forme peu sphérique et une surface abîmée, on augmente fortement la persistance et la frequency gain du bruit de Perlin. Pour avoir des plissements de la croûte linéaires et sinueux, comme sur la surface d'Ee l'étoile blanche du système binaire, on augmente la *sharpness* du *mountain noise* pour avoir un relief très localisé, tout en maintenant le *mountain blend* relativement bas pour éviter de prendre de l'altitude.

Pour obtenir de la brume ou des nuages, comme sur Gwen, on peut placer le niveau des océans légèrement au-dessus de celui du sol, tout en gardant des montagnes qui piquent au travers. On rend alors transparente la couleur de faible épaisseur des océans, et blanche la couleur d'épaisseur importante. On modifie les valeurs du *depth multiplier* et du *water blend multiplier* jusqu'à obtenir l'effet désiré.

Pour avoir des vapeurs émanant des mers, comme celles vertes de Scylla le satellite d'Oculus, on place une atmosphère à peine au-dessus de la surface des océans, et on augmente fortement son *density falloff* tout en maintenant un *scattering strength* conséquent.

Enfin, pour que la surface d'une planète soit similaire à de la glace, comme sur Aethedis Prime, il suffit de la rendre blanchâtre, de lui ajouter des reflets spéculaires dont les étendue et luminosité seront choisis à la main, et enfin de lisser le sol en diminuant la Persistance et le *frequency gain* du bruit de Perlin.

3 Système physique

Les planètes sont en mouvement dans le système solaire. Le mouvement est purement physique, chaque planète agit sur toutes les autres. Ce système est pris en compte dans une classe qui garde en mémoire tous les composants physiques qu'elle a attribué et qui met à jour leur position en intégrant un pas de temps fixe. Chaque composant physique peut retourner sa position, qui est une combinaison linéaire des deux positions intermédiaires (on n'a pas la position exacte en t car on intègre par pas de temps constants).

Une fois les planètes crées, nous avions remarqué des bugs graphiques lors de l'exploration de cette immense scène. Le problème vient justement de la taille de la scène et la représentation de la position des sommets en mémoire. Une planète a une taille de l'ordre de 50 unités, et se situe à une distance d'environ 50 000 unités du centre du repère, ce qui représente presque la moitié de la précision de la mantisse d'un flottant. En changeant complètement le système physique pour que chaque position soit relative à la caméra, ces problèmes graphiques furent résolus. Le problème étant qu'il fallait prendre en compte une nouvelle accélération pour les planètes : celle de l'accélération de la caméra du fait que son référentiel puisse être accéléré lors d'un déplacement.

La rotation est cinématique, à chaque boucle, il suffit de tourner les planètes sur elles-mêmes proportionnellement au temps qui s'est écoulé.

4 Végétation

Les plantes tentaculaires sont des *mesh* entièrement paramétrés à la main, sans utiliser aucune primitive, notamment aucun cylindre.

La colonne vertébrale de la plante, i.e. la courbe suivie par le tronc, est une spline cardinale, régie par les équations vues en cours. Nous avons donc placé les points d'interpolation de sorte à obtenir la forme de spirale désirée.

Une fois les points d'interpolation placés et l'équation de courbe obtenue, des points (que nous nommerons "vertèbres") sont placés régulièrement sur chaque tronçon de spline, i.e. entre chaque paire de points d'interpolation.

Ensuite, des cercles orthogonaux à la spline sont placés autour de chaque vertèbre. Leur rayon est issu d'une interpolation entre les rayons des cercles des points d'interpolation initiaux, qui sont choisis par l'utilisateur. Puis les points desdits cercles sont reliés pour former les triangles, tout comme dans la construction des *mesh* usuels. Le dégradé de couleur est implémenté à cette étape.

Enfin, les plantes sont animées avec le principe de hiérarchie vu en TP. Leur placement sur la planète Gwen est issu de paramètres aléatoires qui déterminent leurs taille, position et orientation.

Et pour leur permettre de garder leur position sur la planète malgré le mouvement de celle-ci, comme dans la structure *hierarchy*, on multiplie les coordonnées de chaque plante par le transform de la planète. parler de la réorientation des plantes pour qu'elles suivent la surface : multiplie plante par transform de planète

5 Ciel

La *skybox* est un cube de taille 1, texturé avec une image de bonne résolution trouvée sur internet. Le cube n'a pas de matrice de modèle, son référentiel est le référentiel global. Pour que l'observateur ait l'impression que l'apparence de la *skybox* est à l'infini, il suffit de ne pas prendre en compte les déplacements de la caméra, mais seulement ses rotations, lors du passage dans le référentiel de la caméra. Pour éviter que les planètes passent derrière la *skybox*, on met sa troisième coordonnée égale à sa quatrième coordonnée. Comme ça, une fois le changement de référentiel effectué, elle est à la distance maximale du *clipspace*, et apparaît donc derrière les planètes.

6 Interaction

Les interactions sont les commandes que peut effectuer le joueur pour influencer ce qu'il voit à l'écran.

6.1 Caméra

Nous voulions un moyen d'explorer cette scène. La caméra type *fps* nous semblait particulièrement adaptée pour apprécier la grandeur de la scène. Nous avons utilisé la classe *camera_head* de *vcl* liée au déplacement de la souris qui est capturée.

6.2 Déplacement

Les touches du clavier Z, S, Q et D permettent de déplacer la caméra. Le déplacement dans l'espace se fait par jetpack : l'action de ces touches crée une force qui s'applique sur le joueur (en réalité sur toutes les planètes car le joueur est le centre du référentiel).

Le déplacement sur une planète est plus compliqué. Il s'agit de vérifier à chaque instant si le joueur rentre en collision avec une planète, auquel cas on adapte sa position et son orientation afin qu'il se tienne bien debout. Sa vitesse doit ensuite être égale à celle de la planète, à laquelle il faut rajouter une vitesse due à la rotation de la planète (sinon le joueur immobile sur une planète glisserait à sa surface lors d'une rotation). Pour que le joueur puisse se déplacer sur une planète, il ne faut plus le soumettre à une force mais lui imposer une vitesse constante selon une direction définie par les actions des touches, ce qui ressemble plus à une marche.

7 Optimisations et astuces

— La génération des planètes à chaque lancement était relativement longue (parfois plusieurs minutes). En parallélisant cette génération, on peut largement abaisser ce temps de chargement en utilisant tous les *threads* du processeur.
— Les planètes possèdent beaucoup de sommets donc nous affichons un *mesh* simplifié à grande distance pour augmenter les performances et diminuer l'*aliasing*. De plus les planètes derrière la caméra ne sont pas rendues.
— La scène est très grande. Initialement, la matrice de projection allait de 0.1 à 10 000 ce qui créait des problèmes visuels pour le *post processing* (le tampon de profondeur n'avait pas assez de précision). Nous faisons un rendu en deux fois comme proposé dans l'article [5]. Il suffit donc d'adapter la méthode de rendu qui s'effectuait déjà en plusieurs fois pour le *post processing*, en utilisant deux matrices de projection différentes.
— Finalement, au lieu d'écrire les paramètres de chaque planète dans le code, nous pouvons exporter ces paramètres dans un fichier binaire, qui sont ensuite importés au lancement du programme. Cependant, il faut prendre en compte que le programme peut être compilé différemment et avec des *paddings* différents pour la même classe. Il faut donc construire une table à l'exécution du programme (chapitre 7 de [6]) qui associe chaque paramètre avec son offset dans la classe.

8 Bibliographie

Références

[1] Oscar Sebio Cajaraville. « Four Ways to Create a Mesh for a Sphere ». In : (). URL : https://medium.com/game-dev-daily/four-ways-to-create-a-mesh-for-a-sphere-d7956b825db4.

[2] Joey de Vries. « Framebuffers ». In : (). URL : https://learnopengl.com/Advanced-OpenGL/Framebuffers.

[3] Sebastian Lague. *Coding Adventure : Atmosphere*. URL : https://www.youtube.com/watch?v=DxfEbulyFcY.

[4] Sean O'Neil. « Accurate Atmospheric Scattering ». In : (). URL : https://developer.nvidia.com/gpugems/gpugems2/part-ii-shading-lighting-and-shadows/chapter-16-accurate-atmospheric-scattering.

[5] « Depth Buffer Precision ». In : URL : https://www.khronos.org/opengl/wiki/Depth_Buffer_Precision.

[6] Jason Gregory. *Game Engine Architecture*. Third edition. CRC Press, 2019.