RAPPORT DU PROJET
D'ALGORITHMIQUE ET SYNTHÈSE D'IMAGE

I-ORGANISATION DU GROUPE

Au tout début du projet, avant de nous lancer dans le code, nous avons passé les quatres premières heures de cours dédiées à la réalisation du projet à réfléchir sur papier. En nous appuyant sur le sujet, nous avons établi ensemble les différentes étapes à suivre pour passer d'une height map, à une application permettant de nous déplacer au sein d'un environnement 3D, en détaillant à chaque fois les structures et les dépendances dont nous aurions besoin. En parallèle, nous avons aussi établi la liste des choses que nous avions déjà réalisées ou abordées au cours des différents TPs, afin de nous créer une première banque d'information.

Au cours de cette phase d'analyse, nous avons aussi distingué deux parties bien distinctes. Tout d'abord, la première, consistant en la lecture de fichiers externes et la création du quadtree, et une seconde correspondant à la réalisation d'une application réagissant en temps réel aux comportements de l'utilisateur.

Dans un premier temps, Flavien s'est donc occupé du code gérant la lecture du fichier .timac, et de l'instanciation d'un tableau de pixels à partir de la height map ainsi que du makefile. Tanguy s'est lui lancé dans la création du quadtree et dans l'élaboration de la fonction createQuadtree() permettant de l'instancier à partir du tableau de pixel de Flavien. Théo quant à lui, s'est attelé à la mise en place de l'environnement GLUT et SDL pour la gestion de l'application.

Par la suite, Flavien s'est occupé de l'intégration des décors au sein de la map, de la lumière, ainsi que des textures en binôme avec Théo qui s'est occupé de la caméra, des déplacements, ainsi que du mode de visualisation du quadtree. Tanguy quant à lui s'est occupé de l'intégration du quadtree à la map, du frustum culling, ainsi que du LOD et de la gestion des cracks engendrés par ce dernier.

Pour ce qui est de la gestion du code à plusieurs, nous avons utilisé un git via le logiciel GitKraken et de github. Nous avons tous travaillé sur la même branche de manière à limiter le nombre de conflits par commit.

II-ORGANISATION DU CODE

Pour ce qui est de l'organisation du code, nous avons décidé de répartir notre code selon le modèle utilisé au cours des TPs de synthèse d'image :

NOM DU DOSSIER	FONCTION
bin	Contient le fichier exécutable du projet
doc	Contient tous les documents relatifs au projet, notamment le fichiers .timac, la height map.
doc/img	Contient toutes les images de textures du projet
include	Contient tous les headers du projet
obj	Contient tous les fichiers .o du projet
src	Contient tous les fichiers .cpp du projet

NOM DU FICHIER	CONTENU
color.cpp	Contient toutes les méthodes liées à la structure Color3f.
geometry.cpp	Contient toutes les fonctions relatives aux calculs de vecteurs, de distances et de positions au sein de la map. Contient aussi les constructeurs et les méthodes liées aux structures Point3D, Vecteur3D, et Triangle.
preload.cpp	Contient toutes les fonctions relatives à la lecture de fichiers externes.
QuadTree.cpp	Contient le constructeur, les accesseurs, ainsi que toutes les méthodes liées au Quadtree, au frustum culling, au LOD, et à la gestion des cracks.
objects.cpp	Contient les constructeurs, les accesseurs, et toutes les méthodes relatives aux objets de la scène 3D, c'est-à-dire la caméra, le soleil, et les arbres.
draw.cpp	Contient toutes les fonctions permettant de dessiner les objets de la scène.
main.cpp	Contient la fonction main, ainsi que toutes les fonctions de gestion d'application GLUT.

III-FONCTIONNALITÉS DU PROJET

A) SYNTHÈSE D'IMAGE

1) Chargement et affichage du terrain

Pour la création de texture, nous avons tout d'abord créé un tableau de chaînes de caractères afin de rentrer les différents chemins des textures. Nous avons ensuite appelé notre fonction de création de texture qui prend en paramètre les chemins des textures et qui retourne les textures dans des tableaux. Nous avons créé 3 tableaux de textures pour plus de lisibilité : un pour la skybox, un pour le terrain et un pour les arbres.

Pour l'affichage des textures, chacune des fonctions qui les utilise prend en paramètre le tableau correspondant. La Skybox dispose de 12 textures (6 mode normal + 6 vue filaire). Pour le terrain, nous avons fait la moyenne de la hauteur des trois points constituant le triangle. Cette hauteur est ensuite comparée à nos trois différents niveaux de texture dans la fonction "bindTexture()" afin de savoir quelle texture est à appliquer parmi les quatre disponibles. Les textures sont cohérentes entre elles : plus l'on monte en altitude, plus elles sont enneigées. De la même manière, les textures appliquées aux arbres dépendent de la hauteur du point de la map sur lequel ils se trouvent.

2) Affichage de la végétation et de l'environnement

La végétation du terrain (hors textures du sol) correspond uniquement à l'affichage aléatoire des arbres en tant que billboards. Après avoir instancié le tableau de points contenu dans la structure PointChart (cf. partie B1 de ce rapport), on réalise un tirage aléatoire parmi ces points pour savoir où se trouveront nos arbres. Pour retenir leur position, nous avons ajouté un attribut booléen "tree" au sein de la structure Point3D, que l'on passe à true si le point est tiré au hasard. Lors de l'instanciation du quadtree, on enregistre aussi le paramètre "tree" de chaque point du PointChart pour y avoir directement accès dans la fonction drawHeightMap(). Enfin, lorsque l'on parcourt le quadtree, si un quad que l'on dessine comporte un arbre sur l'un de ses points, alors on stocke les coordonnées de ce point dans une structure TreeChart, que l'on parcourt avec la fonction drawTrees() une fois la map affichée entièrement, afin d'éviter tout problème de transparence.
Pour faire en sorte qu'ils soient tout le temps orientés vers la caméra, nous effectuons à chaque arbre une rotation correspondant à la latitude du regard de la camera sur sa position fixée à l'initialisation par rapport à l'axe z. Comme pour les textures de la map, nous avons décidé de faire varier les illustrations en fonction de l'altitude sur la map.

Une fois l'environnement réalisé, nous avons intégré la skybox. Sa taille s'ajuste automatiquement par rapport au zFar de la caméra, et lorsque l'on active la vue filaire, les textures de la skybox changent pour une version rétro wave.

3) Déplacement de la caméra

Pour plus de lisibilité, nous avoir décidé de créer une structure "Camera" regroupant l'ensemble des paramètres de cette dernière. La structure propose notamment une fonction sees() qui prend un quad en paramètre et qui renvoie "true" si le quad est dans le champ de

vision de la caméra, et “false” sinon. En mode fixe, la camera se déplace à une vitesse de marche, stockée dans la variable WALKING_SPEED dans le main.cpp. Lorsque l’on passe en mode libre en appuyant sur la touche ‘C’, on passe dans un mode de déplacement plus rapide, défini par la variable FLYING_SPEED.

4) Modèle d’illumination

Pour l’éclairage de notre map, nous avons utilisé le modèle d’illumination de Lambert : l'intensité de la lumière réfléchie par la texture est proportionnelle au cosinus de l'angle que fait la normale de chaque triangle avec la direction de la source de lumière incidente (soleil). Le soleil est une structure à part entière, dont la position peut varier. Une méthode de cette structure appelée getLight() retourne un coefficient de “réflexion” utilisée lors de l’affichage de chaque triangle de la heightmap, pour colorier chacun d’eux en fonction de ce même coefficient. Notre soleil tourne donc autour d’un point “origin”, stocké dans la structure Sun et initialisé au centre de la map dans la fonction initSun(). Ainsi, si l’on clique sur ‘L’, on passe en mode “loop”, et le soleil tourne autour de notre map. Nous avons aussi fait en sorte que lorsque le soleil n’est pas en mode “loop”, nous puissions changer sa position via les touches ‘K’ et ‘M’.

5) Visualisation du quadtree

Pour visualiser le quadtree, nous avons créé un mode filaire. On y accède via la touche ‘F’ du clavier. Les textures du terrain sont alors enlevées pour révéler les triangles composant la map. Nous avons fait le choix, comme dit précédemment, de laisser une skybox mais de changer le “thème”. Afin de voir le quadtree et le LOD, les triangles sont dessinés avec un code couleur (thème retro wave). Pour déterminer la couleur à appliquer aux différents triangles, on accède à l’attribut “height” du quad que l’on dessine afin de connaître sa hauteur dans le quadtree, et donc le niveau de détail auquel l’on se trouve. De cette manière, en se déplaçant sur la map, on peut voir le quadtree et le LOD en temps réel.

6) Réparation des cracks

Pour détecter les cracks, on vérifie dans la fonction LevelOfDetailReached() si le quad courant est un cran au-dessus du niveau de détail souhaité par rapport à sa distance de la caméra et des paliers du LOD. Si c’est le cas, dans la fonction dealWithCracks(), on part alors du coin du quad le plus proche de la caméra, et l’on vérifie si ses voisins directs font partie ou pas du même niveau de LOD. A chaque fois qu’un des voisins directs est en dehors de ce niveau de LOD, c’est qu’il y a un crack à réparer, et on appelle alors la méthode du quad permettant de corriger le crack sur le côté correspondant.

B) ALGORITHMIQUE

1) Chargement, création du quadtree

Struct Params :

Pour éviter d’avoir trop de variables à passer en paramètres pour récupérer les informations du fichier .timac, nous avons décidé de créer une structure Params regroupant tous ces

paramètres. Une seule variable de ce type est créée une seule fois dans la fonction init() du fichier main.cpp afin d'initialiser le quadtree et la caméra.

Struct PointChart :
La structure PointChart contient un tableau à deux dimensions de Point3D dont les coordonnées correspondent encore à la position de chaque pixels dans l'image, et à leur code rvb. Elle contient aussi une variable "width" et "height" indiquant la largeur et la hauteur du tableau courant. Une seule variable de ce type est créée une seule fois dans la fonction init() du fichier main afin d'instancier le QuadTree, puis elle est détruite.

Struct Node:
Notre structure Node, ou QuadTree, contient quelques paramètres en plus de ceux de base. Le paramètre "height" permet de stocker la hauteur d'un node au sein du quadtree afin d'éviter de devoir reparcourir l'arbre à chaque fois que l'on souhaite obtenir la hauteur d'un Node pour le LOD.
Un Node contient quatre Point3D dit "de références", contenant les coordonnées des quatres coins du quad qu'il représente, mais aussi quatre autres Point3D servant à stocker les coordonnées temporaires de ces quatres même coins dont le paramètre z peut varier afin de réparer les cracks provoqués par le LOD.
Enfin, un Node contient aussi un paramètre booléen indiquant si l'un des quatre points qui le compose comporte un arbre à générer, ce qui permet d'éviter de vérifier inutilement les quatres points lors du dessin de la map dans la fonction drawHeightMap.

Fonction createQuadtree() :
La fonction createQuadTree est une fonction récursive permettant de créer et d'instancier un quadtree en parcourant le PointChart passé en paramètre. Cette fonction est capable d'instancier un quadtree à partir d'une heightmap rectangulaire.

2) Utilisation du quadtree pour le dessin

Pour dessiner la map, on parcourt directement le quadtree et non pas un tableau instancier en parcourant le quadtree afin d'économiser de la mémoire. En dessinant la map, on instancie un tableau répertoriant tous les arbres présents sur la map, qui seront dessinés seulement une fois la map finie.

3) Utilisation du quadtree pour le frustum culling

A chaque quad du quadtree parcouru dans la fonction drawHeightMap(), on vérifie si le quad est dans le champ de vision de la caméra via la méthode sees() de la caméra qui renvoie true si c'est le cas, et false sinon. Pour qu'un quad soit considéré comme en dehors du champs de vision de la caméra, il faut que ses quatre points se trouvent tous du même côté de l'un des trois vecteurs constituant le champ de vision de la caméra. En l'occurrence, ici, il s'agit du côté gauche étant donné que l'on parcourt les trois vecteurs du champs de vision dans le sens des aiguilles d'une montre.

4) Utilisation du quadtree pour le LOD

Dans la fonction init() du fichier main.cpp, on fait appelle à la fonction initLODLevels(), qui initialise trois paliers de détails en fonction du zFar de la caméra. Par la suite, lorsque l'on parcours le quadtree dans la fonction drawHeightMap(), on fait appelle à la fonction LevelOfDetailReached(), qui détermine si le quad courant est à la bonne hauteur dans le quadTree, c'est à dire s'il a le bon niveau de détail, en comparant sa distance à la caméra aux différents paliers du LOD.

IV-DIFFICULTÉS RENCONTRÉES

A) Gestion des cracks

Actuellement, les conditions d'appel de la fonction dealWithCracks() dans LevelOfDetailReached() ne permettent pas de détecter les cracks sur des maps rectangulaires. En effet, une map rectangulaire implique un déséquilibre entre les enfants d'un quad à partir d'un certain niveau, ce qui entraîne l'apparition de "feuilles prématurées" se trouvant au niveau deux du quadtree alors que toutes les autres feuilles se trouvent au niveau un. Ainsi, ces feuilles sont rattachées à un parents ayant une hauteur enregistrée à trois. Or, pour le moment, la fonction de détection des cracks entre le niveau un et le niveau deux du quadtree n'est appelée que lorsque la hauteur du quad courant est égale à deux, ce qui ne prend pas en compte le cas évoqué plus haut. Malheureusement, je n'ai pas encore eu le temps de remédier à ce problème.

B) Ordre d'affichage des arbres

Les arbres sont censés être affichés du plus loin au plus proche, de manière à ce que la gestion des transparences permet de voir le décor se situant derrière les premiers éléments de la map transparents. Or, dans le cas de nos arbres, l'affichage ne se fait dans le bon ordre que lorsque l'on se déplace du haut de la map vers le bas. Encore une fois, nous avons manqué de temps pour trouver la solution à ce problème.

C) Calcul de la distance entre les quads et la caméra

Actuellement nous prenons en compte la coordonnée z dans le calcul des distances pour notre LOD. Cela ne pose pas de soucis pour l'instant car nous n'avons pas de très hautes élévation, mais une amélioration à prévoir serait de ne calculer la distance qu'en fonction des coordonnées du plan du sol x et y.

V-AMÉLIORATIONS POSSIBLES

Notre programme est stable et ne rencontre pas de problèmes. Cependant, on peut déjà imaginer des améliorations possibles.

La caméra actuelle ne propose pas des mouvements très fluide, on peut imaginer compléter son fonctionnement afin qu'elle soit une accélération et décélération pour la rendre plus agréable. Aussi, en mode POV, on pourrait la brider à la map afin qu'on ne puisse pas sortir au-delà.

On peut aussi penser à créer un “bac” pour notre height map. Cela permettrait, quand on est en mode caméra freefly, de voir la map en volume et pas seulement comme un plan. Pour cela il faudrait récupérer les bordures de la map, donner une hauteur et créer une forme complexe pour créer le bac.

LIEN VERS LE REPO GITHUB:
https://github.com/MrLorent/DIMENSION_PROJECT

Théo HOLAVILLE
Flavien LINEUC
Tanguy LORENT