

M2 SIS - Synthèse d'images avancée

BRIAND Maxime, PELLETIER Maxime et TOFFANIN Marc

28 mars 2016

1 OMBRAGES

1.1 SPOTLIGHT

Fichier relatifs au sujet : *spotlights.frag*, *shadow.frag*

Pour les lumières de types spotlight, la solution du cours a été utilisée. Elle consiste à faire une shadow map par lumière pour créer les ombres [1].

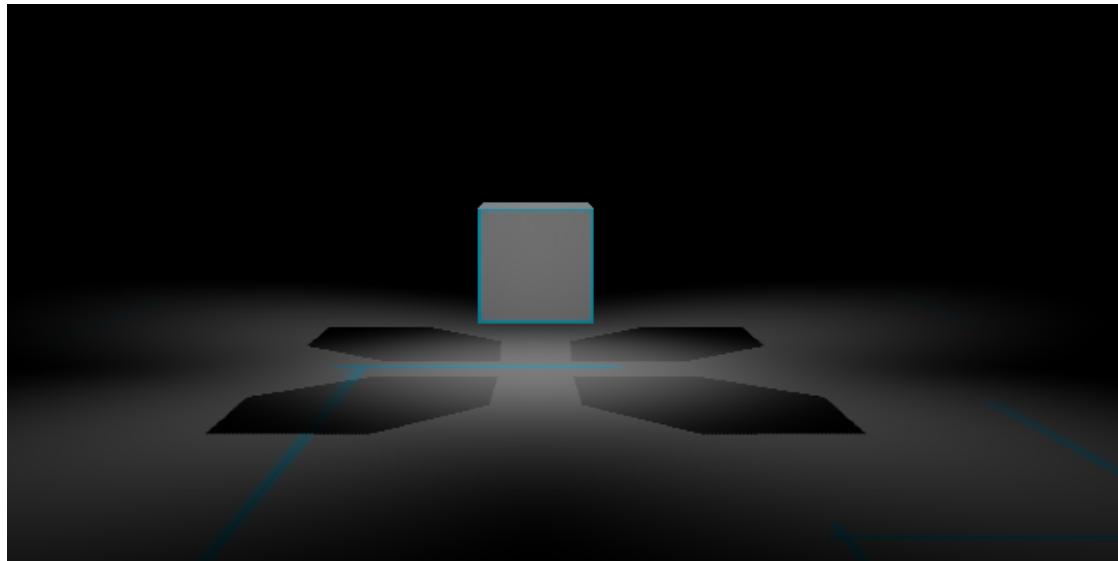


FIGURE 1.1 – Quatre spotlights éclairants le cube dans la première scène.

1.2 DIRECTIONAL LIGHT

Fichier relatifs au sujet : *directionallight.frag*, *shadow.frag*

Pour les lumières de types directional light, de la même manière que pour les spotlights, nous utilisons le shadow mapping pour créer les ombres. Au lieu d'utiliser une projection perspective, nous utilisons une projection orthographique pour simuler le fait que cette lumière vient de l'infini et que les rayons qui proviennent de cette source de lumière sont donc parallèles.

Cependant, cette lumière pose pas mal de problème au niveau de la qualité des ombres. En effet, cette lumière s'appliquant sur toute la scène extérieur, il est difficile de maintenir une qualité d'ombrages suffisante sur toute la scène. Pour corriger celà, la solution la plus utilisée est le Cascaded Shadow Mapping [2]. Au vu de la taille de la scène utilisée ici et du nombre d'objets qui font apparaître des ombres, la décision a été prise de simplement augmenter la résolution de la shadow map, passant de 1024×1024 à 4096×4096 .



FIGURE 1.2 – On peut voir l’ombrage créé par la lumière directionnelle sur la deuxième scène.
Une ombre apparaît sous le cube et dans le prolongement de l’arbre, ainsi que plus loin sur le plan, signifiant la limite de la shadow map pour cet exemple.

1.3 POINT LIGHT

Fichiers relatifs au sujet : *cubemap.vert*, *cubemap.geom*, *cubemap.frag*, *pointlight.frag*

Les lumières de type point light sont particulières à gérer. En effet il n'est pas possible de créer une seule shadow map dans une texture 2D, car la lumière peut générer des ombres dans toutes les directions. Pour gérer cette particularité, un choix commun en synthèse d'image est d'utiliser une projection par cube map ou sphere map, la première étant utilisée dans le projet.

Pour générer cette cube map, une solution naïve consiste à rendre 6 fois la scène en changeant la matrice monde vers vue lumière. Toutefois, une technique pour rendre cette cube map en une fois existe en utilisant le Geometry Shader. Dans OpenGL 4, il est possible de choisir sur quelle face du cube on rends au sein du Geometry Shader (grâce à la variable *glLayer*). Ceci nous permet donc de donner 6 matrices en uniform et pour chaque face rendre le point à travers ces 6 matrices sur les 6 faces correspondantes.

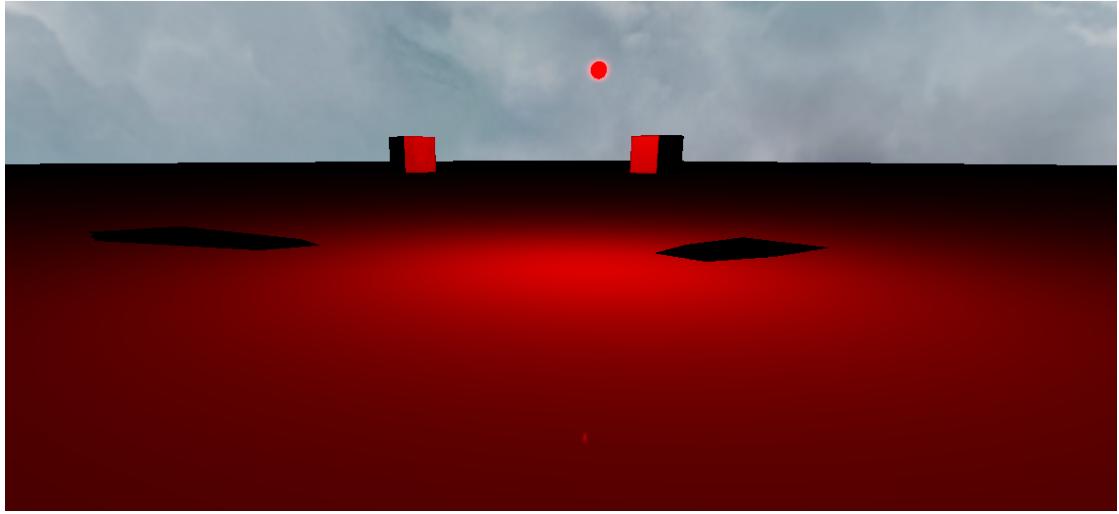


FIGURE 1.3 – On peut observer les deux ombres projetées par les deux cubes provenant d'une seule point light.

2 HDR ET GAMMA CORRECTION

Fichiers relatifs au sujet : *blit.frag*, *blitHDR.frag*

Avant d'aller plus loin, il est important d'aborder la notion de HDR (High Dynamic Range) et LDR (Low Dynamic Range). De base dans OpenGL quand un channel couleur obtient une valeur supérieure à 1.0, elle est clampée. C'est ce que l'on appelle le LDR (aucune valeur > 1.0). Cependant, si l'on met le framebuffer en float, alors les valeurs ne sont plus clampées. Ceci permet par exemple dans le cas de l'accumulation de lumière pendant la light pass du framebuffer de pouvoir avoir des endroits réellement lumineux par rapport aux autres, sinon toute les zones qui accumulent des couleurs et dépassent 1.0 seront en fait similaires.

Le problème vient donc après pour passer de HDR à LDR pour l'afficher sur un écran. C'est là qu'entre en scène le tone mapping. C'est une fonction qui va convertir l'image HDR en valeur LDR tout en appliquant un certain nombre d'effet gérés par l'utilisateur (exposition, saturation, contraste, colorimétrie ...). Dans notre cas nous avons utilisé la fonction qui a été appliquée au jeu Uncharted 2 [4]. A noter que le HDR n'est pas forcément possible sur toute les plateforme.

Ensuite, une deuxième problématique vient avec l'espace image utilisé [5][6]. De base il paraît normal que multiplier par deux une couleur la rende deux fois plus vive. Or les moniteurs n'étant pas parfaits, ils n'appliquent pas totalement cette règle et il faut donc convertir l'image dans le gamma du moniteur. Cependant nous voulons appliquer nos effets, lumières, ombrages dans un espace linéaire. Pour celà, on charge donc les images en sRGB et OpenGL se charge de faire la conversion gamma moniteur vers espace linéaire dans nos shaders pour

qu'avant d'afficher l'image finale nous n'ayons plus qu'à corriger le gamma.



FIGURE 2.1 – A gauche aucune correction, à droite HDR et gamma correction.

3 BLOOM ET BLUR

Fichiers relatifs au sujet : *bloom.frag*, *blur.vert*, *blur.frag*, *extractBright.frag*

Pour la scène Tron, les premières images qui viennent en tête sont composées de lasers, de lumières vives, de néons, etc. Si l'on rend ce genre de chose directement, on obtient une image fade qui ne véhicule pas bien l'intensité des lumières. Quand on regarde ce type de lumière, on doit avoir l'impression qu'elles sont intenses et donc ont tendances à se propager sur les médiums les entourants. Cet effet est appelé Bloom ou Glow en synthèse d'image [7].

Le principe est simple et se fait en trois étapes :

- Extraire dans une texture les parties qui ont les critères pour que l'effet soit appliqué
- Flouter ces parties
- Mixer l'image originale et cette nouvelle image

Les critères d'extraction sont donnés par l'utilisateur, dans notre cas c'est toute partie avec un channel couleur supérieur à 1.0 (c.f HDR et Tone mapping). Une fois ces parties extraites on les floute grâce à un flou Gaussien [7][8].

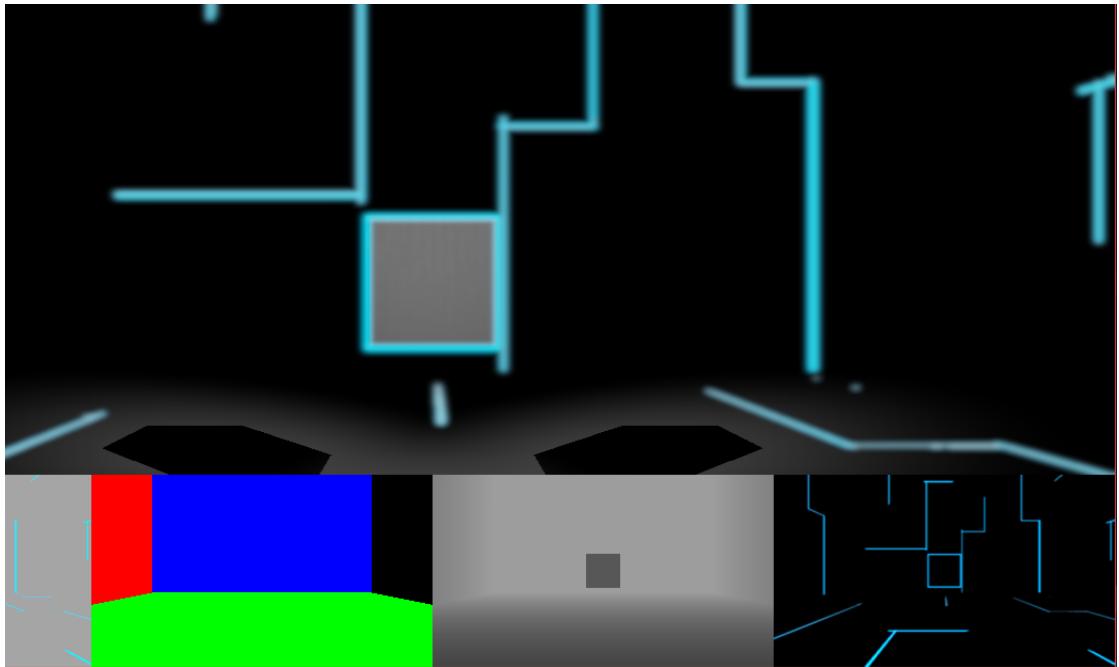


FIGURE 3.1 – On peut voir la différence entre cette figure où le bloom est activé et la figure 1-1. La vignette en bas à droite représente ce que l'extraction de partie brillante a extrait.

4 SCREEN SPACE LIGHT SHAFTS

Fichiers relatifs au sujet : *lightShaft.frag*

Dans le monde réel, quand la lumière traverse un média poussiéreux par exemple, on obtient ce que l'on appelle de la lumière volumétrique : le volume rempli de particules de poussières réagit à la lumière qui le traverse, contrairement à jusqu'à maintenant ou seulement le média frappé par la lumière était pris en compte. Si en plus la lumière est occultée, alors on obtient ce que l'on appelle des god rays, des light shafts ou encore daylight scattering of volumetric occlusion.

Pour achever cet effet, plusieurs solutions s'offrent à nous, plus ou moins complexes. Certaines sont globales et s'effectuent pendant la passe de rendu, certaines comme celle que nous avons choisie se passent en post process et uniquement en espace écran [9].

Cette solution est une solution proposée par Nvidia dans le GPU Gems 3 et consiste à lancer des rayons en espace écran depuis un point de la scène jusqu'à la lumière. Sachant que la géométrie de la scène est rendue en noir, tout au long de ce rayon des échantillons sont pris et leurs couleurs ajoutées. Si aucun obstacle n'entrave la lumière alors beaucoup plus d'échantillons ajouteront la couleur de la lumière à la somme totale, comparé à si un objet se

trouve devant la lumière sur la trajectoire du rayon.

On obtient donc cette notion de light shaft assez simplement en espace écran et si la géométrie n'est pas trop complexe topologiquement (ne pas rendre en wireframe par exemple ...) alors elle n'est pas très couteuse. Cependant, cet effet ne s'active que si la lumière (en l'occurrence le soleil) est visible en espace écran.

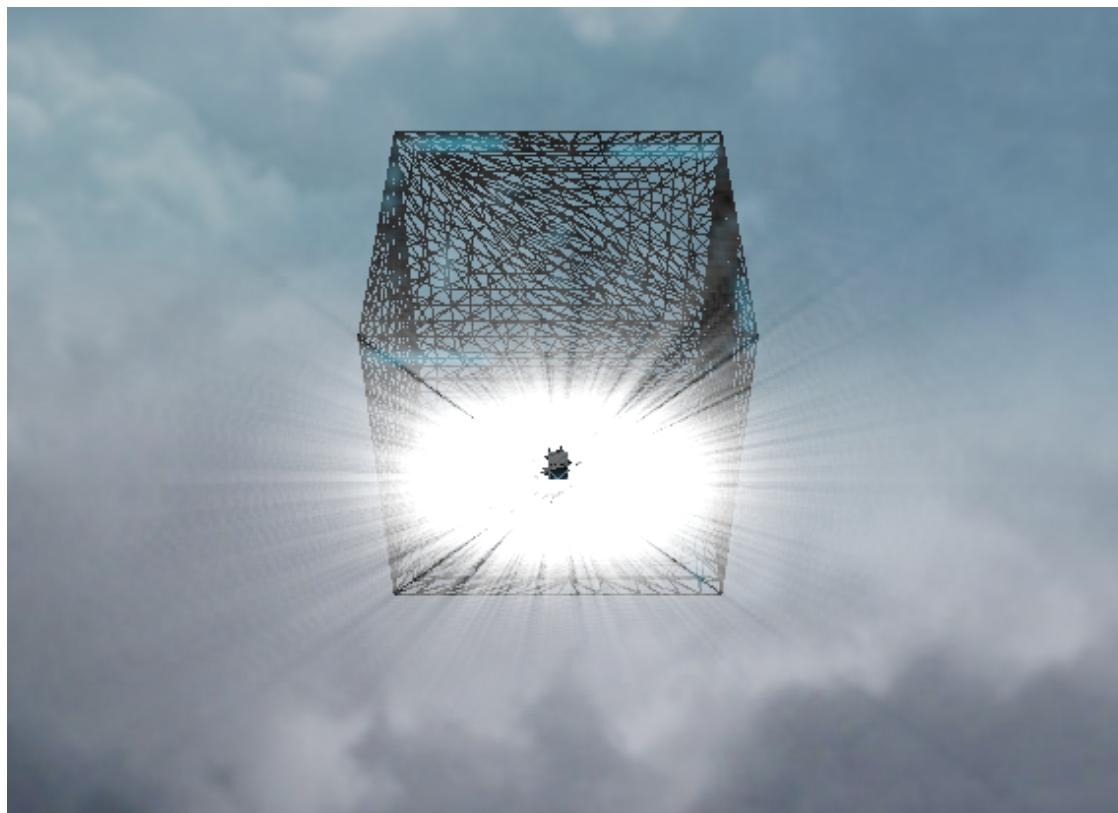


FIGURE 4.1 – On peut observer le light scattering engendré par l'affichage en wireframe des cubes de la première scène.

5 EXPLOSION ET TESSELATION

5.1 EXPLOSION

Fichiers relatifs au sujet : *explode.vert*, *explode.tcs*, *explode.tes*, *explode.geom*, *explode.frag*

A la fin de la première scène, une explosion des cubes survient. Cette explosion utilise deux shaders de OpenGL 4 : le Geometry Shader [10] ainsi que le Tesselation Shader [11].

Dans un premier temps, nous utilisons la tessellation pour passer d'un cube composé de 12 triangles à un cube composé de bien plus de triangles. Pendant cette tessellation, on calcule une nouvelle normale en faisant la différence entre l'emplacement du vertex et le centre du cube (pour créer une sphère). Cette normale est ensuite utilisée dans le Geometry Shader pour décaler chaque vertex émis par la Tessellation le long de la normale qui lui a été donnée, donnant l'impression que le cube explose.

Pour la reconstruction, il suffit de donner une magnitude de départ et de revenir à une magnitude de 0 le long de la normale.

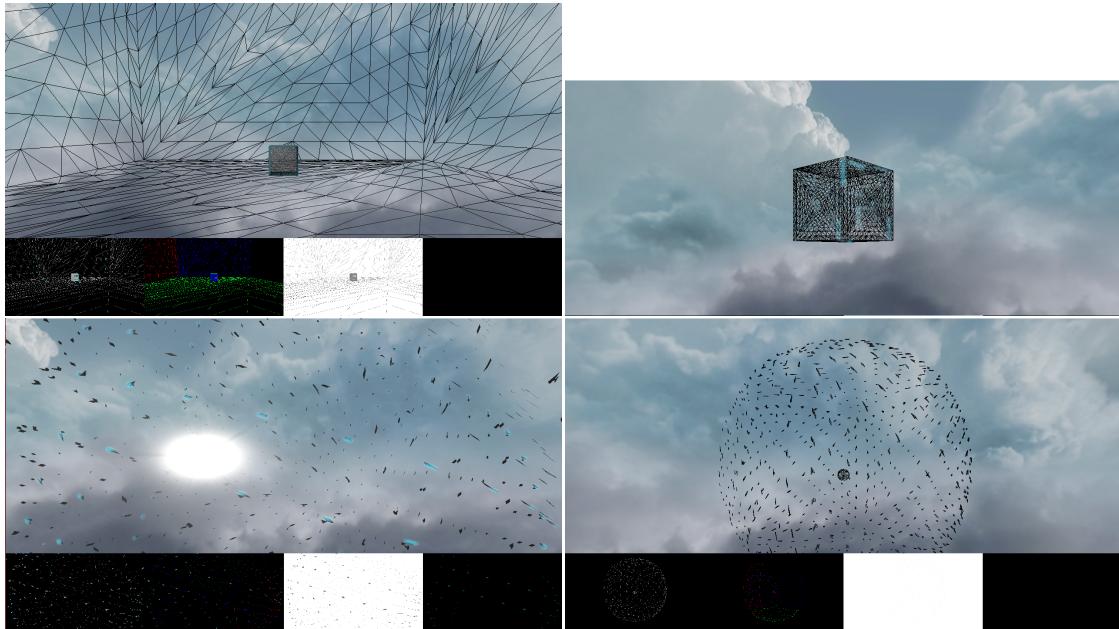


FIGURE 5.1 – En haut la démonstration de la tessellation, en bas l'explosion le long de la normale d'une sphère.

5.2 TESSELATION DE TERRAIN

Fichiers relatifs au sujet : *terrain.vert*, *terrain.tcs*, *terrain.tes*, *terrain.frag*, *terrainNormal.frag*

La tessellation est utilisée une autre fois dans le projet, cette fois ci pour créer le terrain extérieur de la scène 2 et l'effet de construction de ce terrain. De manière assez simple, on définit un nombre de tuiles constituant le terrain, chaque tuile étant définie par un quad [12]. Ce quad est shifté suivant le nombre de tuile pour correspondre à une partie du terrain à rendre, partie correspondant en fait à une height map. Une fois les coordonnées de texture et la position du quad définie, on effectue la tessellation pour augmenter la résolution du terrain et on applique la height map sur l'axe Y (la hauteur en OpenGL).

Au moment de charger la height map, on calcule aussi les normales en utilisant un filtre de Sobel pour faire la moyenne des slopes autour du point. Une fois cette normal calculée elle est appliquée pendant la tessellation pour que le terrain réagisse à la lumière directionnelle.

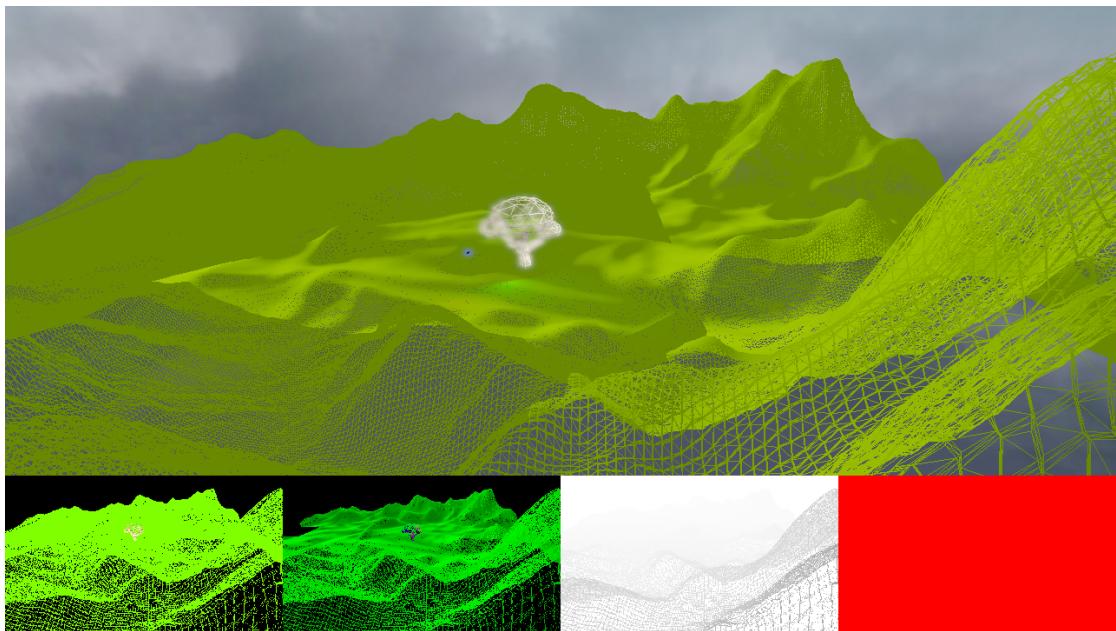


FIGURE 5.2 – Le terrain une fois tesselé et affiché en wireframe. On peut observer les différents patchs le constituant.

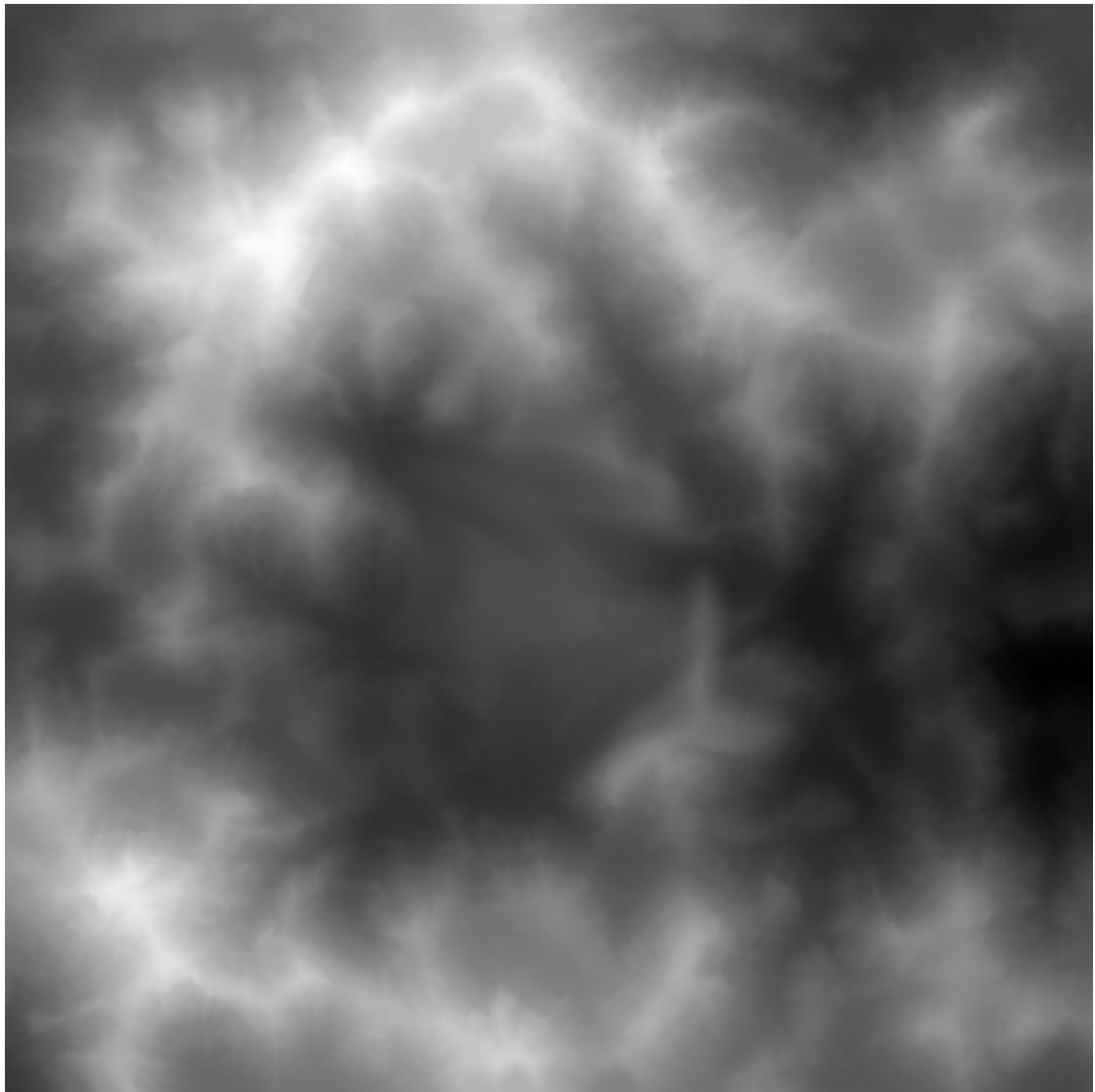


FIGURE 5.3 – La height map utilisée pour construire le terrain, noir représentant le level 0 et blanc un sommet.

6 SKYBOX ET SOLEIL

Fichiers relatifs au sujet : *skybox.vert*, *skybox.frag*, *sun.frag*

Enfin, parlons un peu de la skybox et du soleil. Pour la skybox on charge une cubemap et on retire la partie translation de la matrice de Vue pour ne pas les prendre en compte. Ensuite, dans le shader, si la profondeur dans le depth buffer est très proche de 1.0, alors on sait qu'il n'y a rien qui occlude la skybox et que l'on peut l'afficher.

De la même manière on y ajoute un soleil en donnant une position ainsi qu'un rayon et en affichant un cercle blanc (qui est ensuite modifié par le shader de light shaft).

7 CONCLUSION

En conclusion un certains nombres de solutions ont été implémentées dans ce projet, d'autres n'ont pas abouties (Screen Space Reflection par exemple). Le parti pris de n'utiliser quasiment aucun modèle (uniquement l'arbre de la scène deux) pour se focaliser sur le rendu était une contrainte intéressante. En effet, cela nous a obligé à chercher comment rendre dynamique un nombre sommaire de triangles, notamment en augmentant ce nombre avec la tessellation. Par exemple, il est très simple de rajouter du Level Of Detail (LOD) en GPU dans notre programme. On a donc réussi à aller un peu plus loin que ce qui était proposé dans le cours en utilisant des techniques qui font appelle à des étapes de la pipeline OpenGL qui ont peu été vues.

8 BIBLIOGRAPHIE / SITOGRAPHIE

- [1] <http://adrien.io/opengl-course-french/lesson-3-shadow-mapping/>
- [2] http://developer.download.nvidia.com/SDK/105/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.html
- [3] <http://learnopengl.com/#!Advanced-Lighting/Shadows/Point-Shadows>
- [4] <http://filmicgames.com/archives/75>
- [5] <http://frictionalgames.blogspot.fr/2013/11/tech-feature-linear-space-lighting.html>
- [6] http://http.developer.nvidia.com/GPUGems3/gpugems3_ch24.html
- [7] <http://learnopengl.com/#!Advanced-Lighting/Bloom>
- [8] <http://adrien.io/opengl-course-french/lesson-4-post-processing/>
- [9] http://http.developer.nvidia.com/GPUGems3/gpugems3_ch13.html
- [10] https://www.opengl.org/wiki/Geometry_Shader
- [11] <https://www.opengl.org/wiki/Tessellation>
- [12] <http://www.informit.com/articles/article.aspx?p=2120983>