Compte Rendu des TP 1 & 2 de Moteurs de jeux

M2 IMAGINA

Jérémy DAFFIX

https://github.com/jeremydaffix/hmin317-tp1

TP1

Question 1

La classe MainWidget représente un widget QT destiné à être inséré dans la fenêtre. Celle-ci hérite de la classe QOpenGLWidget, représentant un widget permettant d'afficher le résultat de commandes OpenGL.
L'objectif de MainWidget sera donc d'adapter un QOpenGLWidget aux besoins spécifiques de notre projet, en redéfinissant les méthodes adéquates : événements d'entrée utilisateur, d'affichage, etc.

La classe GeometryEngine contient les méthodes pour définir la géométrie OpenGL de nos objets 3D (pour le moment, un cube), et pour les afficher.

Fshader et Vshader sont des scripts de shader, qui permettent de paramétrer et modifier le processus de rendu, afin, par exemple, d'ajouter des « effets ».
Vshader est un shader pour les Vertex (sommets). Il convertit la position *a_position* d'un sommet 3D (repère monde) vers le repère écran (en effectuant un produit matriciel avec la matrice « Model View Projection »). Puis, il enregistre la coordonnée du point correspondant au sommet dans la texture, dans la variable *v_texcoord*.
Fshader est un shader pour les Fragments. Il associe le triangle courant à une couleur qu'il récupère dans la texture, à la coordonnée *v_texcoord*. Cette coordonnée est calculée à partir de la coordonnée donnée par le Fshader, en effectuant une interpolation.

Question 2

GeometryEngine::initCubeGeometry()

On crée un tableau des sommets du cube, *vertices*. Chaque sommet est associé à une position 3D, mais aussi à une coordonnée 2D dans la texture. Comme chaque face utilise une zone de texture différente, on est obligés de dupliquer les sommets pour chaque face, soit 6 * 4 = 24 sommets.

On crée ensuite un tableau des index, *indices*. Celui-ci définit le cube sous forme d'un « triangle strip » : pour chaque sommet d'indice *n* dans ce tableau, on créera un triangle avec les sommets *n – 1* et *n – 2*.

Enfin, on active et alloue les VBOs (Vertex Buffer Objects) : VBO0 pour accueillir la liste des sommets, et VB1 pour accueillir la liste des index du cube.

GeometryEngine::drawCubeGeometry(QOpenGLShaderProgram *program)

On active les 2 VBOs, celui des sommets et celui des index.

On récupère l'indice de la variable *a_location* (représentant la coordonnée dans le repère monde) utilisée dans les shaders, avec *attributeLocation()*. Cet indice permet ensuite d'activer la variable pour qu'elle puisse être utilisée dans les shaders, avec *enableAttributeArray()*. Puis, on associe le buffer dans lequel va piocher cette variable, avec *setAttributeBuffer()*.

Même procédé pour la variable *a_texcoord*, représentant des coordonnées dans la texture.

Enfin, avec *glDrawElements()*, on affiche les sommets en utilisant la convention de « triangle strip ».

Question 3

On crée 2 nouvelles méthodes pour la génération de la géométrie du plan, et pour son affichage : *initPlaneGeometry()* et *drawPlaneGeometry()*.

Pour générer les sommets et la topologie du plan, j'ai d'abord essayé de faire fonctionner l'algorithme pour un plan plus simple de 4x4 sommets.

La génération des sommets en eux-mêmes est simple. Il suffit de faire 2 compteurs (lignes et colonnes), et de calculer les positions x et y en conséquence (ainsi que les positions correspondantes sur la texture).

La liste des sommets pour dessiner le triangle strip se fait par groupes de 2 triangles formant un rectangle :
Sommet en haut à gauche, puis sommet en haut à droite, puis sommet en bas à gauche, puis de nouveau le sommet en bas à gauche (« déplacement »), puis sommet bas droit, et enfin sommet haut droit. Cela nous fait arriver au sommet haut gauche du prochain rectangle à dessiner.
A chaque fin de ligne, il faut également ajouter des sommets, pour se déplacer à la ligne suivante, puis au début de la 1ère colonne. Ce qui nous fait arriver en haut à gauche du 1er rectangle de la ligne suivante.

Une fois la logique pensée et implémentée pour un plan simple de 4x4 sommets, on peut ajouter un paramètre *nbrSommets* à la méthode pour généraliser à la création de plans de nxn sommets (dans la limite de la taille maximum des buffers OpenGL). On utilise aussi une variable *nbrFaces = nbrSommets – 1*. Enfin, un paramètre *size* permet de mettre à l'échelle le plan (= les sommets sont plus ou moins espacés les uns des autres).

L'affichage du plan reprend peu ou prou le code de celui du cube. Il faut surtout penser à passer le bon nombre d'indices à la fonction *glDrawElements()* :

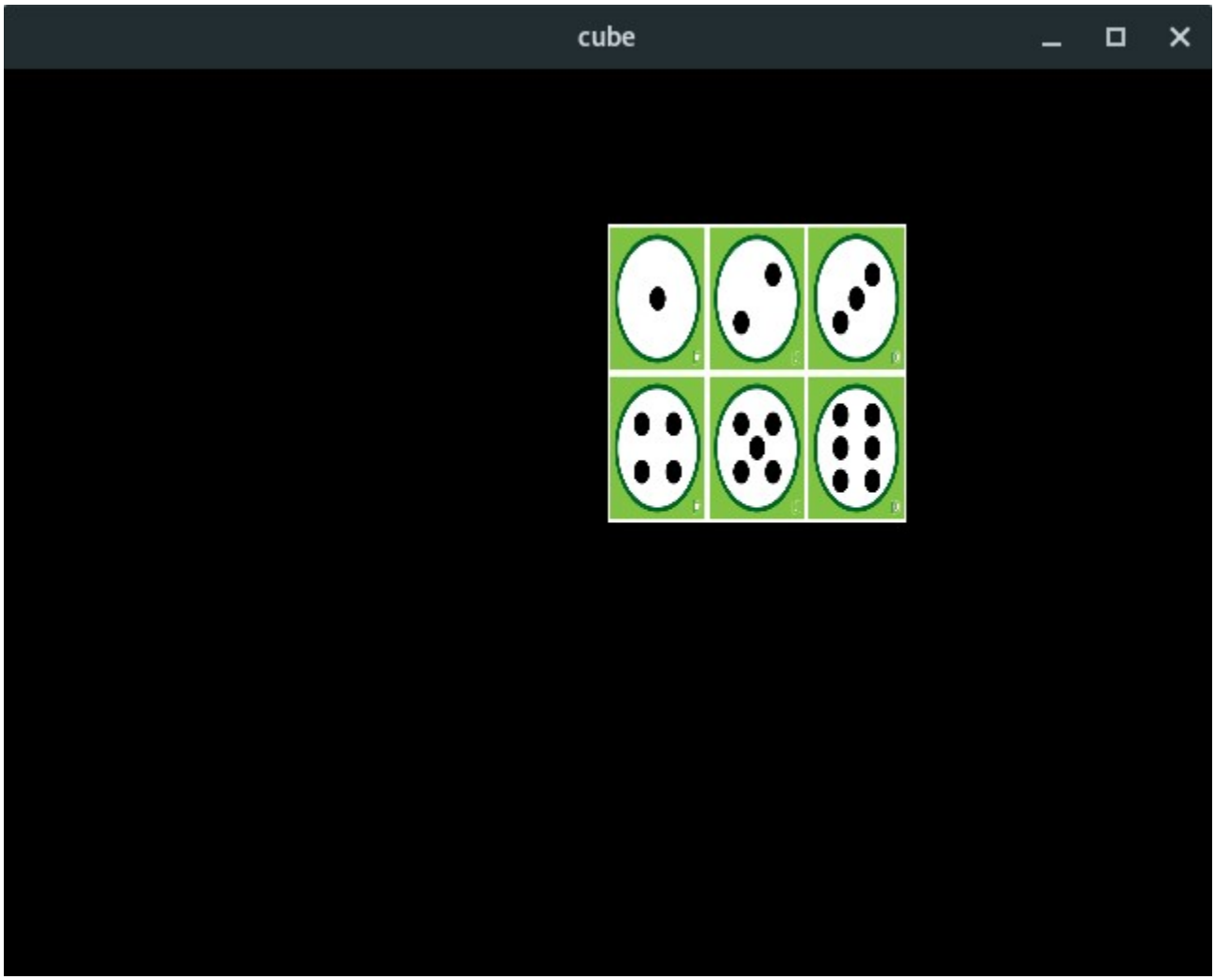
```
int nbrIndices = (nbrFaces * nbrFaces * 6) + (nbrFaces * 4);
[...]
glDrawElements(GL_TRIANGLE_STRIP, nbrIndices, GL_UNSIGNED_SHORT, 0);
```

Pour que cet affichage se passe bien, il faut également redéfinir les near et far planes pour s'adapter à la taille du plan !

```
const qreal zNear = 1.0, zFar = 100.0, fov = 45.0;
```

Enfin, on translate la matrice dans *paintGL()* pour définir la position fixe de la caméra :

```
matrix.translate(0.0, 0.0, -55.0);
```



Un plan 16x16 texturé

Question 4

On crée une méthode *initTerrainGeometry()* pour générer les sommets et la topologie d'un terrain. Il s'agit au final d'un plan, dont on aura modifié la hauteur (sur l'axe Z dans notre cas) de quelques sommets afin de créer un relief :

```
if(i > 3 * scaleRelief && i < 12 * scaleRelief && j > 3 * scaleRelief && j <
12 * scaleRelief) // partie relief
{
      if((i < 7 * scaleRelief || i > 9 * scaleRelief) && (j < 7 * scaleRelief
      || j > 9 * scaleRelief))
            vertices[pos].position = QVector3D((float)j * size, (float)i *
            size, - 2.0);

      else
            vertices[pos].position = QVector3D((float)j * size, (float)i *
            size, 2.0);
}
else // partie plate
{
      vertices[pos].position = QVector3D((float)j * size, (float)i * size,
      0.0);
}
```

On enlève la modification de la rotation lors du clic souris, pour que notre terrain soit stable.
On met la rotation de la caméra à une valeur fixe de manière à ce qu'elle soit légèrement inclinée sur l'axe X :

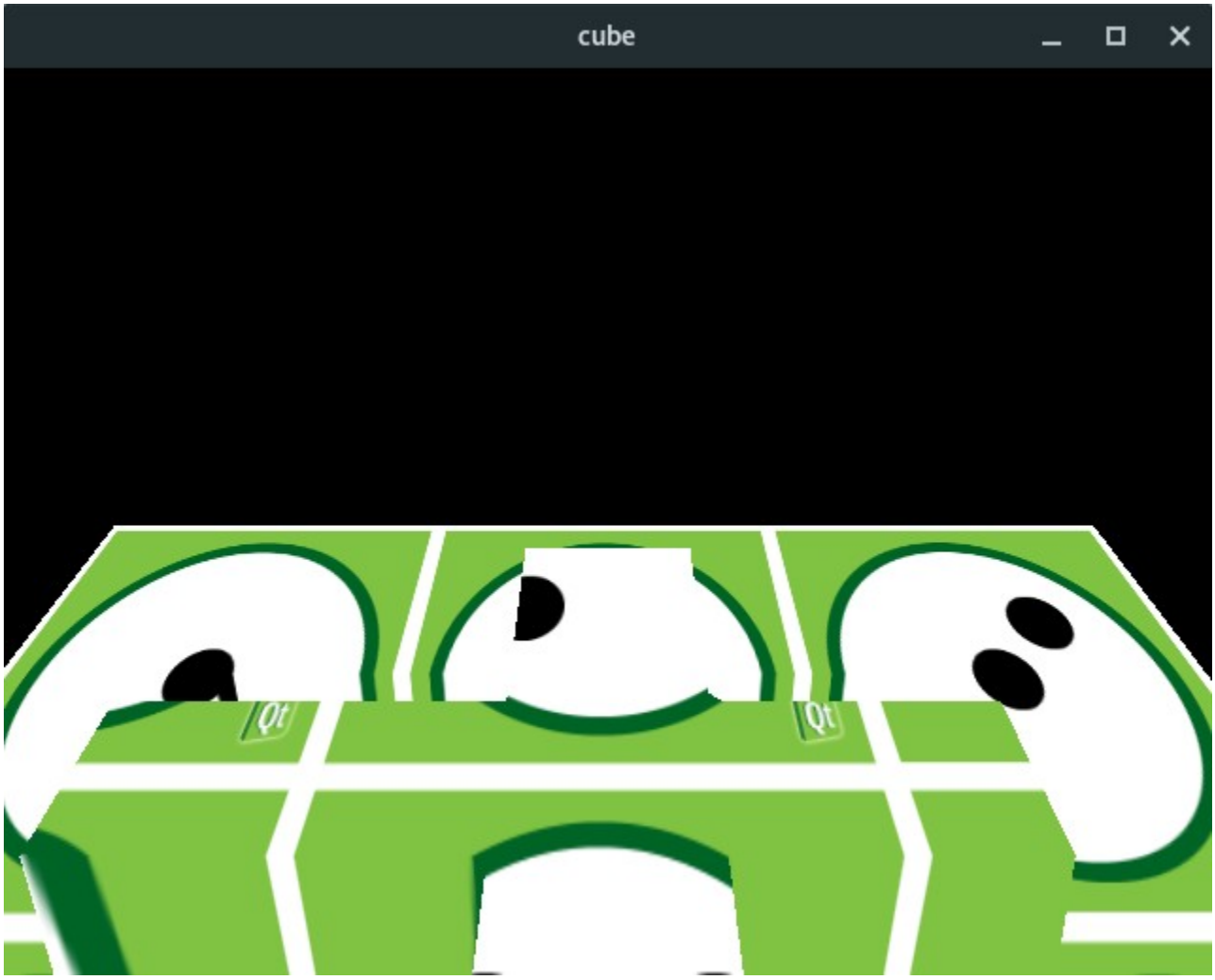
```
QQuaternion cameraRotation = QQuaternion::fromEulerAngles(-60, 0, 0);
```

On crée également un attribut de classe pour stocker la position de la caméra, que l'on modifiera dynamiquement :

```
QVector3D cameraPosition = QVector3D(-8.0, -8.0, -5.0);
```

Les événéments claviers sont gérés dans une méthode *keyPressEvent(e)*, qui est automatiquement appelée par QT. Son paramètre permet de récupérer le code de la touche qui a été pressée. Même chose pour *keyReleaseEvent(e)*, qui est appelée lorsqu'une touche est relâchée.

Le déplacement de la caméra n'est pas effectué dans les événements claviers en eux-mêmes, ceci afin de rester indépendant du framerate réel.
Dans KeyPressEvent et KeyReleaseEvent, on va simplement assigner une variable *movementDirection* représentant la direction courante (enum *DIRECTION* créée pour l'occasion).
La mise à jour du vecteur de position de la caméra en fonction de la direction est effectuée à intervalle fixe (= x fois par seconde) dans la méthode *timerEvent()*. Ainsi, même si l'affichage « lag », la caméra se déplacera à la même vitesse.



Un terrain avec relief (plan 16x16 avec Z modifié pour certains sommets)

TP2

Question 1

Une carte d'altitude est une image dont la couleur de chaque pixel représente une profondeur. Pour ce TP, j'ai récupéré sur Internet la heightmap d'une île au format PNG, et qui se charge comme n'importe quelle texture :

```
heightmap = new QImage(QImage(":/island_heightmap.png"));
```

On modifie ensuite le prototype de la méthode *initTerrainGeometry()* pour ajouter un paramètre *heightmap* de type *QImage**.

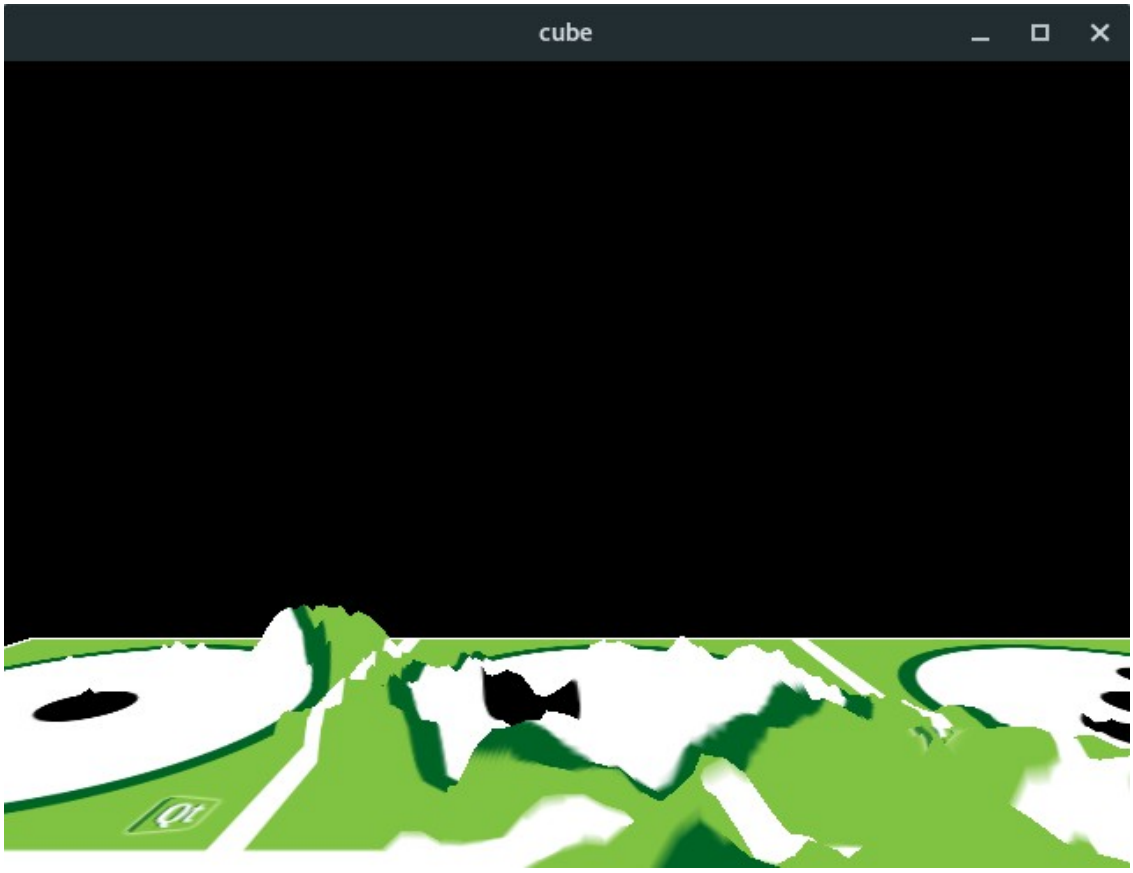
Si ce paramètre est *NULL*, on génère par défaut le relief tel que vu dans le TP1. Sinon, on va utiliser les pixels de l'image pour changer la hauteur Z des sommets du terrain.

A noter que la taille de la heightmap (5442x5442 px dans ce cas précis) n'est pas forcément égale au nombre de sommets du terrain (ici, 128x128 sommets, définis par un attribut *sizeTerrain* de la classe *GeometryEngine*). On va donc utiliser seulement un échantillon de cette image, grâce à un Pas *stepHeightMap* :

```
stepHeightmap =  qi->size().width() / nbrSommets;
```

Pour calculer la hauteur Z d'un point, on fait la moyenne des composantes RGBA de la couleur du pixel de la heightmap, et on divise par 255 pour avoir un float entre 0 et 1. On multiplie ensuite cela par un coefficient *coef* (plus il est grand, plus le terrain sera « élancé ») afin d'avoir notre Z final.

```
QColor col = qi->pixel(j * stepHeightmap, i * stepHeightmap);
int r, g, b, a;
col.getRgb(&r, &g, &b, &a);

float coef = 2.0;

float z = (r + g + b + a) / 255. / 4. * coef;
```



Le terrain (128x128) généré par une heightmap

Question 2

Plutôt que de faire tourner à chaque fois tous les points du terrain, j'ai fait tourner la caméra sur elle-même.

On met la position et la rotation initiales dans la définition de la classe *MainWidget* :

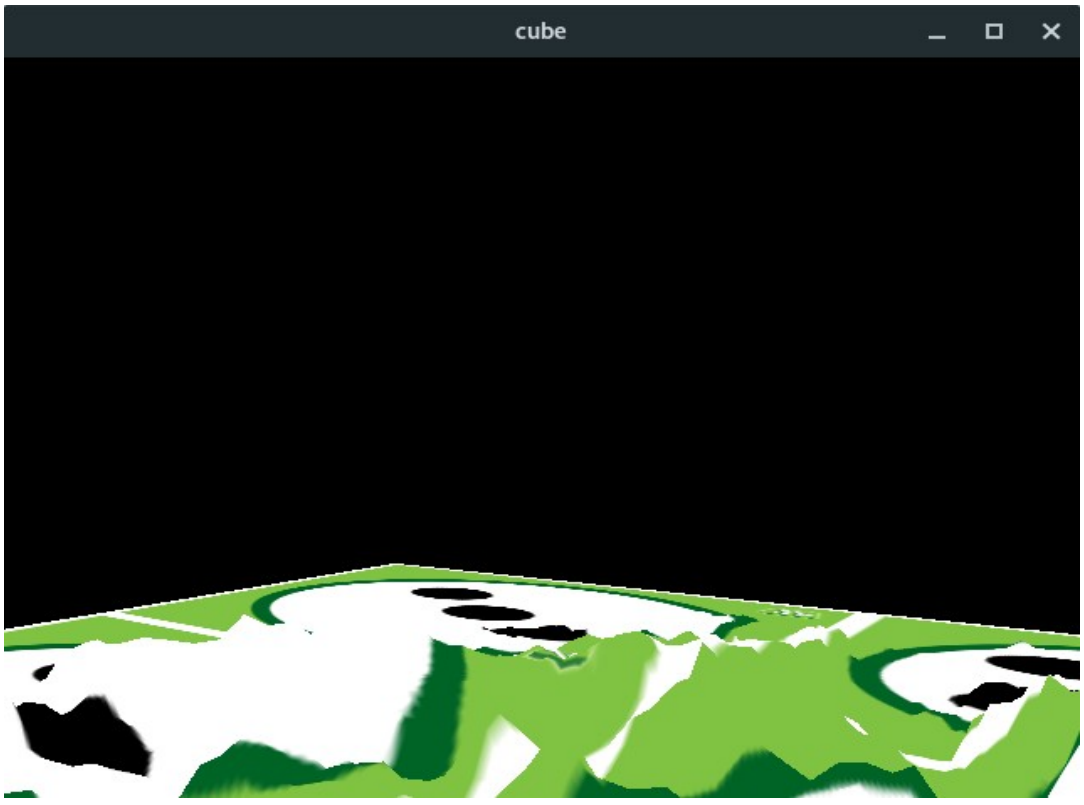
```
QQuaternion cameraRotation = QQuaternion::fromEulerAngles(-90, 0, 0);
QVector3D cameraPosition = QVector3D(-8, -8, -3);
```

On tourne sur l'axe Y à intervalles fixes dans la méthode *timerEvent()* :

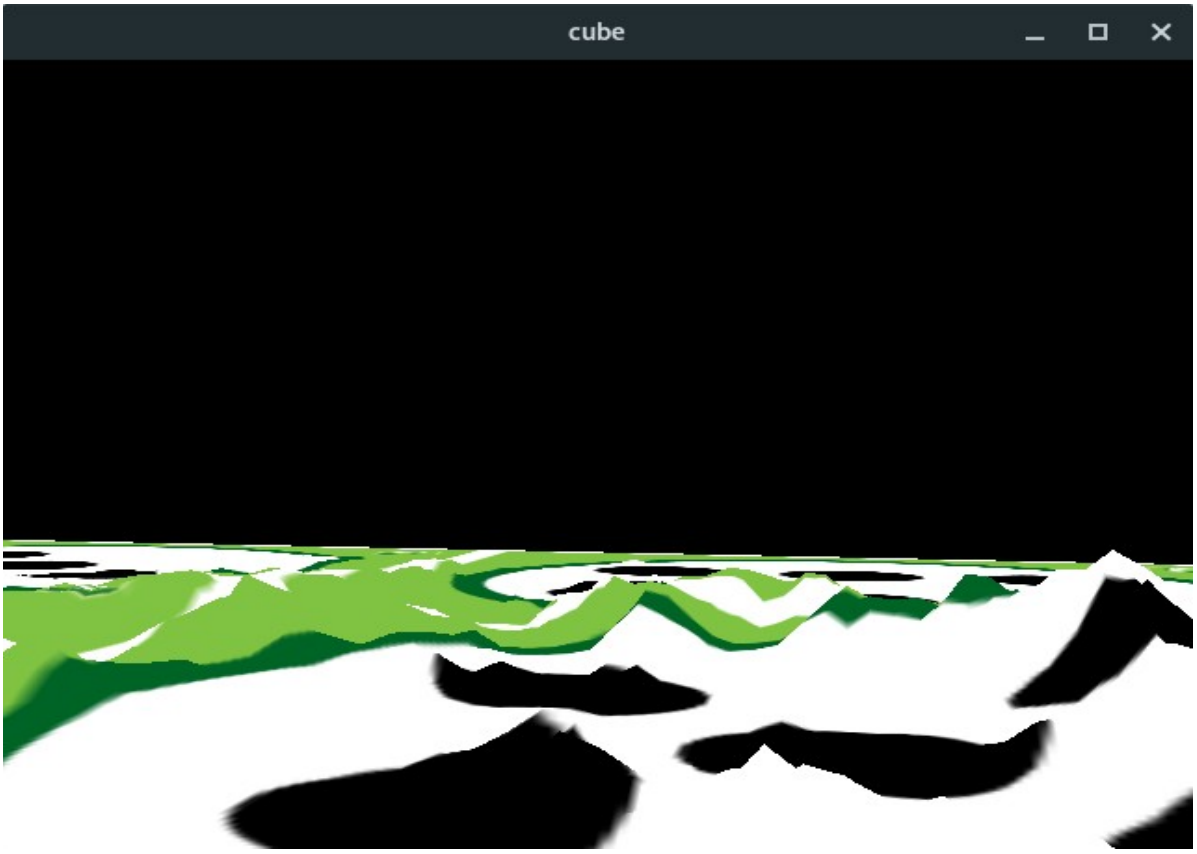
```
cameraRotation = QQuaternion::fromEulerAngles(0, rotationSpeed, 0) *
                    cameraRotation;
```

Problème : avec le code existant, la caméra tournait autour de l'origine du repère, et non de son propre centre. J'ai donc inversé la rotation et la translation dans *paintGL()*, afin que l'on effectue en premier la rotation de la caméra (le centre de la caméra coincidant alors à l'origine du repère), puis sa translation :

```
// on inverse R et T si on veut faire tourner la camera sur elle même (tp2)
// (sinon elle tourne autour de l'origine)
matrix.rotate(cameraRotation);
matrix.translate(cameraPosition);
```



La caméra tourne sur elle-même, quelle que soit sa position



Le même terrain, la même heightmap, avec le même nombre de sommets, mais scalé 2x plus grand (argument size de initTerrainGeometry())

NOTE : lorsque la rotation automatique est activée, le déplacement avec les touches ZQSD se fait toujours, mais dans le repère monde plutôt que le repère caméra. Je n'ai pas encore réussi à résoudre ce problème.

Question 3

La classe *QTimer* permet de lancer un événement à intervalles fixes, toutes les x ms. L'instance de *QTimer* associée à notre *MainWidget* lance sa méthode *timerEvent().*
On peut le stopper, le relancer, l'initialiser,... en appelant les méthodes adéquates.
Plus particulièrement, sa méthode *start()* prend en argument un entier (ainsi qu'un pointeur sur un objet contenant une méthode *timerEvent()* à appeler), qui définit l'intervalle d'appel en ms.

Dans *initializeGL()*, on initialise ce timer avec l'intervalle en ms correspondant au nombre de FPS voulu :

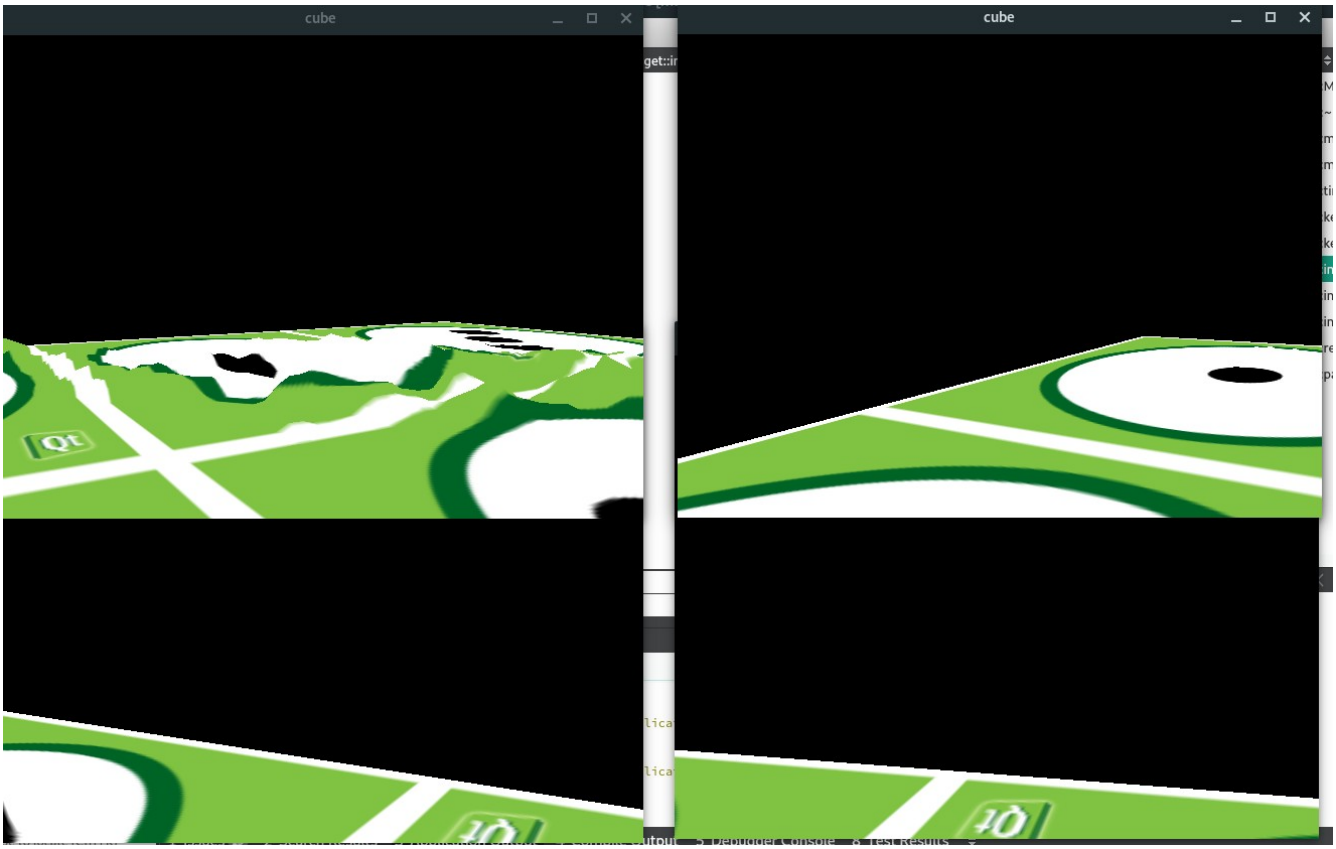
```
    timer.start(1000 / fps, this);
```

Par exemple, si on passe 100fps au constructeur de *MainWindow*, l'événement *timerEvent()* sera appelé toutes les 1000 / 100 = 10ms.

Il est ensuite facile, à partir de ça, de créer plusieurs fenêtres avec des fps différents :

```
MainWidget widget1(1), widget10(10), widget100(100), widget1000(1000);
widget1.show();
widget10.show();
widget100.show();
widget1000.show();
```

Ce qui nous donne ceci :



Nous pouvons remarquer que la caméra de chaque fenêtre tourne à une vitesse différente : celle ayant une fréquence de rafraîchissement de 1fps très lentement, celle à 10fps un peu plus rapidement, etc. Ce qui est normal, puisque le code tournant la caméra est appelé, sur une même durée t, plus de fois lorque le framerate fixé est élevé.
En revanche, on se rend assez clairement compte que la fenêtre à 1000 fps ne va pas « 10 fois plus vite » que celle à 100 fps. Peut-être cela est-ce dû à la limite de précision du *QTimer* , ou bien au temps d'exécution de la méthode *timerEvent()* ?

Avec les touches UP et DOWN (qui augmentent ou diminuent l'angle de chaque rotation, pour compenser le framerate), on peut rapprocher les vitesses de rotation de chaque fenêtre pour les « synchroniser » :

```
        case Qt::Key_Up:
            if(rotationSpeed < 10.0) rotationSpeed += 0.1;
        break;

        case Qt::Key_Down:
            if(rotationSpeed > 0.2) rotationSpeed -= 0.1;
        break;
```

Néanmoins, lorsque le framerate est bas, cela créée des «à-coups».

Questions Bonus

Texturation du terrain avec des couleurs

Plutôt que d'utiliser une image de texture, on va utiliser des couleurs en fonction de l'altitude du terrain en un point donné.

Pour ce faire, on crée deux nouveaux shaders : *vshader_color* et *fshader_color*, que l'on charge dans *initShaders()* à la place des anciens.

Dans l'un comme dans l'autre, un vecteur3 *terrain_color* (représentant une couleur RGB) remplace les coordonnées de texture.

Il est récupéré « interpolé » dans le fragment shader de cette manière :

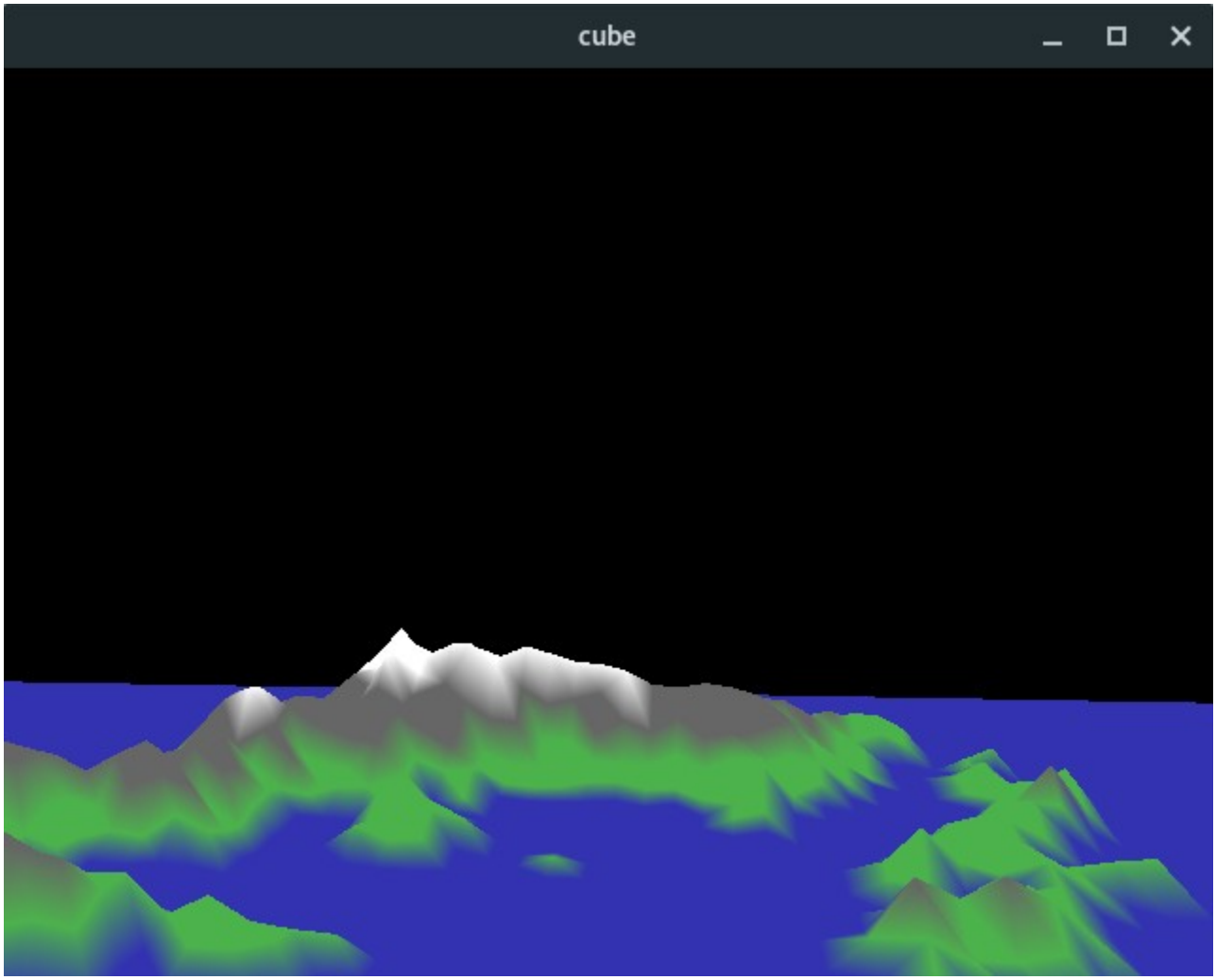
```
    gl_FragColor = vec4(terrain_color, 1.0);
```

Dans le vertex shader, on lui donne sa valeur, en fonction de sa coordonnée Z :

```
    if (a_position.z > 1.5) // neige
    {
       terrain_color = vec3(1.0, 1.0, 1.0);
    }

    else if (a_position.z > 1.0) // montagne haut, cailloux
    {
       terrain_color = vec3(0.4, 0.4, 0.4);
    }

    else if (a_position.z < 0.65) // eau
    {
       terrain_color = vec3(0.2, 0.2, 0.7);
    }

    else // montagne bas, végétation
```

```
    {
        terrain_color = vec3(0.3, 0.7, 0.3);
    }
```

Ce qui donne le résultat suivant :



Lumières

- La lumière par défaut est une lumière de type « ambiante », sans origine, sans direction, où tout est éclairé de manière uniforme.
Pour simuler un soleil, nous pourrions par exemple implémenter des shaders de lumière « diffuse », où la luminosité d'un point donné prend en compte la direction de la lumière. De cette manière, les faces exposées au soleil seraient plus lumineuses que les faces cachées.

- Une source de lumière localisée (lampe, torche, ampoule,…), avec une certaine puissance et une position dans l'espace 3D, pourrait être implémentée avec des shaders de « lumière ponctuelle ». L'idée est que contrairement à une lumière ambiante ou diffuse, celle-ci perd petit à petit de son intensité en fonction de la distance des objets éclairés.

- Une lumière de type « speculaire » permettrait de simuler certains matériaux, par exemple les métaux qui ont des reflets. En jouant sur la propriété de « puissance spéculaire » P d'un objet 3D, on pourrait ainsi jouer sur la manière dont il est rendu.