Question 1:

Partie 1:

La classe `MainWidget` gère l'affichage de la fenêtre où s'exécute Opengl.
Elle gère aussi les interactions de l'utilisateur, mouvement de souris, clique. Elle s'occupe de la gestion de la vitesse de l'objet 3D. Elle initialise les shaders, le vertex shader et le fragment shader. Elle charge aussi la texture

La classe `GeometryEngine` s'occupent de la création et de l'affichage du cube de départ. Elle contient les buffers de vertices et d'indices du cube.

Partie 2:

Le fichier `fshader.glsl` correspond au fragment shader. Ce shader va gérer les textures sur les différents points. Il va appliquer la couleur à la coordonnée voulue à partir de la texture.
Le fichier `vshader.glsl` correspond au fichier décrivant le vertex shader de l'application. Le vertex shader calcule la position des vertex dans l'espace puis transmet les coordonnées de la texture utilisé sur le cube au fragment shader.

Question 2:

La fonction `void GeometryEngine::initCubeGeometry()` sert a initialiser les données du cube, c'est à dire la position et le nombre de vertices qui constituent le cube ainsi que les coordonnées des textures pour chaque vertice. Ces coordonnées sont utilisées pour se repérer sur l'image png qui sert de base pour la textures. Le point 0,0 se situe dans le coin bas gauche de l'image.
La fonction initialise aussi le tableau d'indices des vertices.
Pour finir la fonction transfert les données créées dans les VBO.

La fonction `void GeometryEngine::drawCubeGeometry(QOpenGLShaderProgram *program)` gère l'affichage du cube.
Elle va choisir les VBO à utiliser ainsi que dire à Opengl comment lire les données du VBO des vertices ainsi que l'offset a utiliser pour parcourir le buffer. Elle va indiquer les coordonnées des points dans l'espace dans un premier temps, donner le offset, donner un pas de 3 car 3 coordonnées. Puis nous allons indiquer les coordonnées de textures en utilisant un pas de 2. La fonction demande par la suite a Opengl d'afficher les VBO en utilisant `GL_TRIANGLE_STRIP`.

Question 3:

Pour créer la surface plane, on initialise un tableau de vertices de 16*16. On calcul la position de chaque vertice et on les ajoute dans le tableau.

```
// creation des vertices
int i = 0;
for(int y= 0; y < 16; ++y) {
	for(int x = 0; x < 16; ++x) {
		float r = 0.9 + static_cast <float> (rand()) /( static_cast <float> (RAND_MAX/(0.9-1.1))); // elevation random des points
		vert[i++] = {QVector3D((x / 8.0f), (y / 8.0f), r - 1.0f), QVector2D((x / 15.0f), (y / 15.0f))};
	}
}
```

On calcul la position des points de façon à ce que le plan ai un coté de taille 2. On donne a chaque vertice sa coordonnées de texture.

Pour les indices, on réalise 6 ajouts d'indices par vertices, sauf pour le dernier vertice d'une ligne et le dernier vertice en colonne que l'on ignore car sinon notre calcul d'indice va sortir du tableau. On ajoute nos indices dans le sens anti-horaire.

```
//init tableau indices
int j = 0;
for(int y= 0; y < 16 - 1; ++y) {
	for(int x = 0; x < 16 - 1; ++x) {
		indices[j++] = (y*16) + x;
		indices[j++] = (y*16) + x + 1;
		indices[j++] = (y*16) + x + 16;
		indices[j++] = (y*16) + x + 1;
		indices[j++] = (y*16) + x + 17;
		indices[j++] = (y*16) + x + 16;
	}
}
```

On ajoute 16 à l'indice quand on veut passer au vertices avec la même coordonnée x mais avec la coordonnée y supérieure.
La principale difficulté pour construire le tableau d'indice a été le nombre d'indices à donner ainsi que le bon ordre d'indices à insérer.

On alloue ensuite les VBO avec la bonne taille.

Pour afficher le plan, on reprend la même fonction que le cube mais on va modifier une seule ligne.

```
glDrawElements(GL_TRIANGLES, (15 * 15 * 6), GL_UNSIGNED_SHORT, 0);
```

On utilise le mode d'affiche `GL_TRIANGLES` au lieu de `GL_TRIANGLES_STRIP` et on donne la bonne taille du tableau d'indice.

Question 4:

Pour donner du relief au plan, j'ai donner une valeur au hasard a chaque vertices entre -0.1 et 0.1 sur l'axe Z. Cette valeur est peut être un peu trop élevée, cela ne donne pas un aspect naturel à la déformation. Concernant la caméra, j'ai récupérer les événements capturés par Qt pour les gérer. J'utilise les flèches pour faire bouger la caméra. La caméra se règle avec différents vecteur. Pour modifier sa direction, son emplacement, il nous suffit de manipuler ces vecteurs. Voici le code correspondant à la caméra:

```
void MainWidget::keyPressEvent(QKeyEvent *e){
   float sensibility = 0.1f;
   switch (e->key()) {
   case Qt::Key_Up:
       position += front * sensibility;
       break;
   case Qt::Key_Down:
       position -= front * sensibility;
       break;
   case Qt::Key_Left:
       position -= QVector3D::crossProduct(front,up).normalized() *
sensibility;
       break;
   case Qt::Key_Right:
       position += QVector3D::crossProduct(front,up).normalized() *
sensibility;
       break;
   }
   update();
}
```

```
MainWidget::MainWidget(QWidget *parent) :
   QOpenGLWidget(parent),
   geometries(0),
   texture(0),
   angularSpeed(0),
   position(0.0f, 0.0f, 5.0f),
   front(0.0f, 0.0f, -1.0f),
```

```
   up(0.0f, 1.0f, 0.0f)
{
   this->setFocusPolicy(Qt::ClickFocus);
   this->setMouseTracking(true);
}
```

```
   matrix.lookAt(position, position + front, up);
```