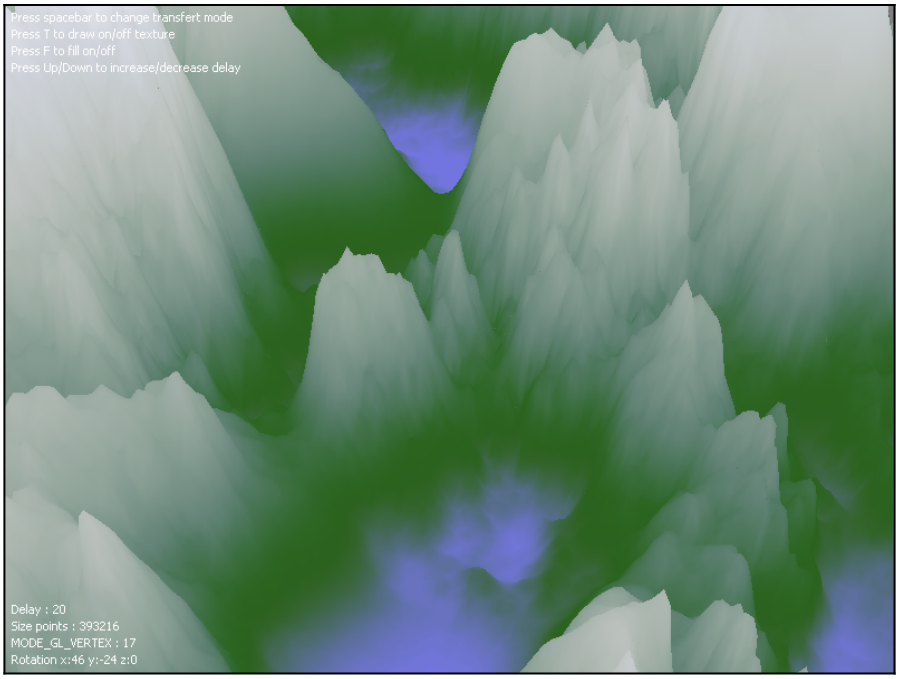
Qt OpenGL - Générer un terrain

Cette partie présente la structure de base utilisée pour réaliser notre application de génération de terrain. Après avoir rappelé les mécanismes de base pour réaliser du rendu 3D dans une application Qt, nous détaillerons la problématique du chargement de notre terrain. Enfin, un système simple de mesure des performances, utilisé tout au long du tutoriel, sera mis en place.



L'ensemble du code de ce chapitre se trouve sur GitHub, dans le projet appelé “OpenGL - code minimal”. Dans cet exemple, on n'affiche pas la heightmap mais un simple repère orthonormé.

La classe HeightmapWidget

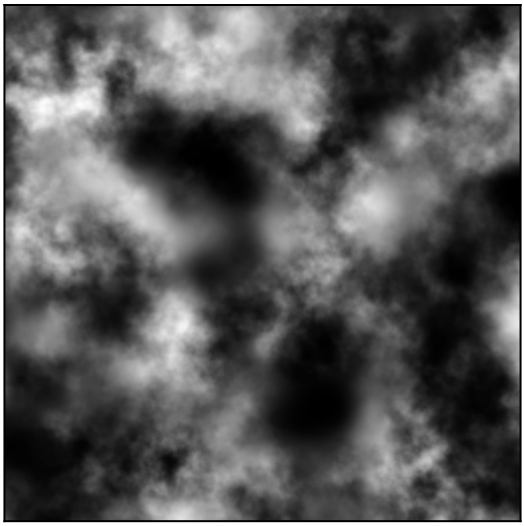
Qt fournit une classe gérant les rendus 3D avec OpenGL : QGLWidget. Cette classe hérite de QWidget et peut donc être manipulée de la même façon, en particulier être incluse dans d'autres widgets ou être directement affichée avec la fonction show(). QGLWidget bénéficie aussi des fonctions de gestion des évènements de QWidget, en particulier la gestion de la souris ou du clavier. Pour créer une nouvelle classe héritant de QGLWidget, il suffit de redéfinir au moins la fonction virtuelle paintGL, qui permet de réaliser le rendu. Habituellement, on redéfinit aussi les fonctions initializeGL, qui sera appelée une seule fois lors de l'initialisation du contexte OpenGL et resizeGL, qui sera appelée lors de chaque redimensionnement du widget.

```
class HeightmapWidget
{
Q_OBJECT

public:
    HeightmapWidget(QWIdget* parent = 0);
    ~HeightmapWidget();

    void initializeGL();
    void paintGL();
    void resizeGL();
};
```

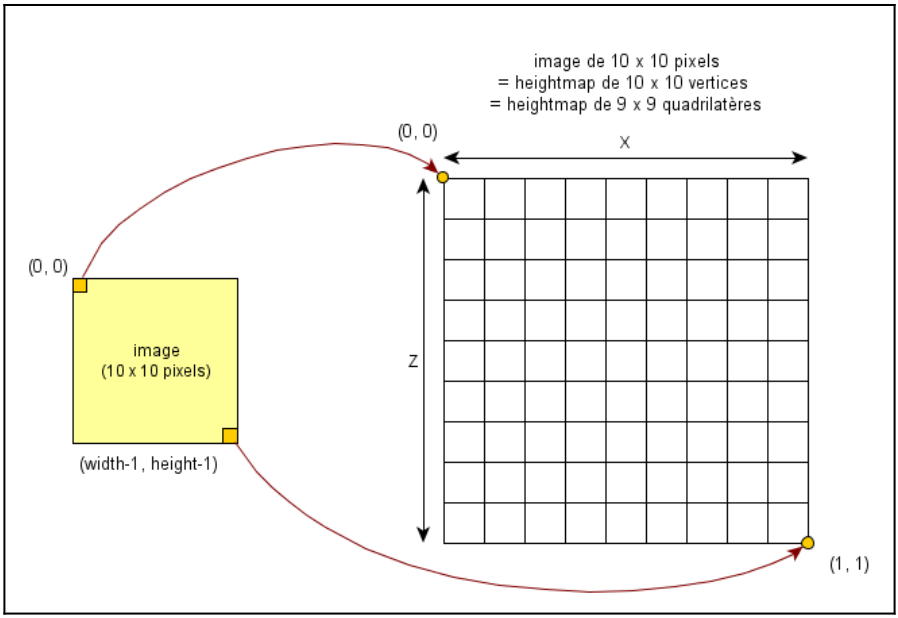
Pour plus de détails sur QGLWidget, le lecteur se reportera au tutoriel d'introduction : Intégration d'OpenGL dans une interface Qt.

Chargement des données du terrain

Le terrain est modélisé par une grille de points (x, y, z). Les données seront chargées depuis une image en niveaux de gris, appelée habituellement heightmap (que l'on peut traduire par carte des hauteurs). Chaque pixel (x, y) de l'image représente un point de notre terrain, l'altitude z du point (hauteur) correspond au niveau de gris de ce pixel. Voici l'image qui nous servira de heightmap tout au long du tutoriel.



Qt fournit la classe QVector3D pour stocker des coordonnées 3D. Cette classe gère les opérations mathématiques usuelles en 3D (addition et soustraction de vecteurs, multiplication par une constante, etc.) et peut être utilisée comme base pour le calcul matriciel (avec les classes QMatrix). Si on n'utilise pas Qt, il est possible de créer une structure similaire : struct vector3D { float x, y, z; };. Pour stocker l'ensemble des points formant le terrain, nous utiliserons un QVector de QVector3D. Un std::vector ou n'importe quel conteneur fera également l'affaire. L'avantage avec les vecteurs dans ce cas est que les données sont stockées dans des blocs mémoire contigus. Il sera donc possible d'envoyer directement au processeur graphique un bloc de données, sous forme de buffer (voir le chapitre sur les Vertex Buffer Object pour l'utilisation des buffers).



Le nombre de points de notre carte est conservé dans les variables vertices_by_x et vertices_by_z et le nombre de quadrilatères dans les variables quads_by_x et quads_by_z. En pratique, comme nous utilisons une grille uniforme, le nombre de quadrilatères par côté est le nombre de vertices par côté moins un. Nous créerons ces différentes variables pour la lisibilité. Certains algorithmes s'appliquent sur les vertices et d'autres sur les formes, il faut donc bien faire la distinction. Dans notre repère 3D, la hauteur est représentée grâce à l'axe y, les axes x et z sont utilisés pour représenter les coordonnées des points constituant le terrain. La raison de ce choix est que pour un repère orthonormé direct, si le plan horizontal correspond au plan xy, alors l'axe des z est orienté vers le bas. En prenant le plan xz comme plan horizontal, l'axe des y est correctement orienté vers le haut.

```
class HeightmapWidget
{
    ...
private:
    QVector<QVector3D>  m_vertices;
    int vertices_by_x;
```

```
    int vertices_by_z;
    int quads_by_x;
    int quads_by_z;
};
```

L'image est chargée en mémoire dans un objet de type QImage (QImage est optimisé pour l'accès aux pixels et est donc préféré à QPixmap, qui est optimisé pour l'affichage). Pour des raisons de simplicité, l'image est incluse dans un fichier de ressources Qt de type .qrc. Cela permet de ne pas se préoccuper des problèmes de répertoires dans lesquels sont stockées les données. Pour utiliser une ressource Qt, il faut tout d'abord ajouter un fichier de ressources à votre projet puis ajouter les fichiers de données dans ce fichier :

```
void HeightmapWidget::initializeGL()
{
    QImage img = QImage(QString(":/heightmap.png"));
```

Le nombre de vertices par dimension et le nombre de quadrilatères par dimension sont calculés à partir des dimensions de l'image.

```
    vertices_by_x = img.width();
    vertices_by_z = img.height();
    quads_by_x = vertices_by_x - 1;
    quads_by_z = vertices_by_z - 1;
```

Chaque pixel de l'image est parcouru à l'aide de deux boucles for imbriquées :

```
    QVector3D vertice;
    for(int z = 0; z < vertices_by_z; ++z)
    {
        for(int x = 0; x < vertices_by_x; ++x)
        {
```

La hauteur du point est calculée en fonction de la couleur (niveau de gris) du pixel. La fonction qGray est utilisée afin de récupérer le niveau de gris d'un pixel, ce niveau de gris variant de 0 à 255. Cette méthode a l'avantage de la simplicité mais n'est pas optimale. En particulier, le code des hauteurs sur 256 valeurs différentes peut être insuffisant en fonction

des besoins. De plus, les images sont codées sur 32 bits alors que 8 bits sont suffisants pour coder 256 valeurs. On pourra optimiser cela en lisant directement un tableau de float.

```
            QRgb color = img.pixel(x, z);
```

Pour terminer, le point 3D est stocké dans le vecteur. Les coordonnées x et z sont centrées et normalisées entre -(MAP_SIZE / 2) et (MAP_SIZE / 2) et la coordonnée y est normalisée entre 0 et 2. MAP_SIZE est une constante réelle valant 5.0.

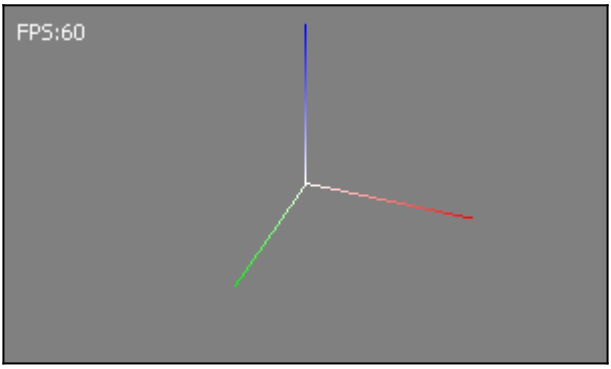
```
            vertice.setX((MAP_SIZE * x / vertices_by_x) -
MAP_SIZE / 2);
            vertice.setY(2.0 * qGray(color) / 255);
            vertice.setZ((MAP_SIZE * z / vertices_by_z) -
MAP_SIZE / 2);

            m_vertices.push_back(vertice);
        }
    }
```

Le code de chargement présenté ci-dessus peut être optimisé. En effet, l'ajout de contenu dans un vecteur grâce à la méthode push_back peut nécessiter un redimensionnement du vecteur à chaque ajout d'un point, entraînant des allocations et des copies inutiles. Une solution est de redimensionner le vecteur avant grâce à la méthode resize (avec pour taille le nombre de points constituant le terrain) puis d'utiliser un itérateur pour parcourir le vecteur.

Initialisation de la vue

Mentionnée plus haut, la transformation des coordonnées 3D en coordonnées 2D est une des étapes les plus importantes du Pipeline 3D. Pour effectuer cette transformation, le moteur de rendu 3D OpenGL a besoin d'informations, notamment la position de l'observateur dans la scène et des caractéristiques telles que l'angle de vision ou encore la distance de vue minimum et maximum afin de simuler l'effet de perspective. Ces informations sont stockées dans des matrices (matrice

de vue et matrice de projection). Le lecteur intéressé par les mécanismes mathématiques impliqués dans ces calculs se reportera à la FAQ.



Dans l'exemple de heightmap présenté, notre heightmap est fixe au centre de la vue, aux coordonnées (0,0,0) et c'est la position de la caméra qui se déplacera autour du centre. Pour définir la position de la caméra, nous utilisons trois paramètres de rotation (un pour chaque axe x, y et z : x_rot, y_rot et z_rot) et un paramètre pour la distance de la caméra au centre (distance) qui sont initialisés dans initializeGL() :

```
HeightmapWidget::initializeGL() {
    distance = -5.0;
    xRot = 0;
    yRot = 0;
    zRot = 0;
}
```

Lors de l'initialisation, il faut également définir la couleur de l'arrière-plan, qui sera utilisée à chaque mise à jour, avec la fonction Qt qglClearColor ou avec la fonction OpenGL glClearColor. La différence entre ces deux fonctions est que qglClearColor accepte des couleurs au format Qt (QColor). Il faut aussi activer le test de profondeur GL_DEPTH_TEST avec la fonction glEnable :

```
qglClearColor(Qt::darkGray); // ou avec glClearColor
glEnable(GL_DEPTH_TEST);
```

Lors de l'affichage de la vue 3D, il faut commencer par dessiner l'arrière-plan en le remplissant avec la couleur définie par qglClearColor en appelant la fonction glClear. Ici, on choisit d'effacer le tampon de couleur (GL_COLOR_BUFFER_BIT) et le tampon de profondeur (GL_DEPTH_BUFFER_BIT). OpenGl permet de choisir comment on souhaite afficher les polygones avec la fonction glPolygonMode. Cette fonction prend deux paramètres : le premier indique sur quelle face est appliqué le mode (la face antérieure avec GL_FRONT, la face postérieure avec GL_BACK ou les deux avec GL_FRONT_AND_BACK ; pour rappel, la face antérieure est celle pour laquelle les vertices sont dans le sens horaire) ; le second paramètre indique le mode d'affichage (des points avec GL_POINT, des lignes avec GL_LINE ou des surfaces pleines avec GL_FILL).

```
HeightmapWidget::paintGL() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

Il ne reste plus qu'à fournir à la carte graphique les paramètres de projection. Lorsqu'on n'utilise pas les shaders, les paramètres des matrices sont définis de la façon suivante :

- on sélectionne la matrice sur laquelle on souhaite travailler avec glMatrixMode ;
- on applique ensuite à cette matrice différentes opérations avec les fonctions glLoadIdentity (recharge la matrice identité), glRotate (rotation autour d'un axe), gluLookAt et gluPerspective.

```
    // Model view matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0, 0.0, -distance,
    0.0, 0.0, 0.0,
    0.0, 1.0, 0.0);

    glRotatef(x_rot / 16.0f, 1.0f, 0.0f, 0.0f);
    glRotatef(y_rot / 16.0f, 0.0f, 1.0f, 0.0f);
    glRotatef(z_rot / 16.0f, 0.0f, 0.0f, 1.0f);
```

```
    // Projection matrix
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0f, 1.0*width()/height(), 0.1f, 100.0f);
```

Par la suite, lorsque l'on utilisera les shaders, les matrices de projection seront envoyées directement aux shaders comme paramètres.

Il faut également mettre à jour les dimensions de la vue OpenGL lorsque le widget est redimensionné. Pour cela, on appelle simplement la fonction glViewport en donnant les dimensions du widget :

```
void HeightmapWidget::resizeGL(int width, int height)
{
    glViewport(0,0, width, height);
}
```

Gestion de la souris et du clavier

Puisque que QGLWidget hérite de QWidget, il possible de récupérer les évènements de la souris ou du clavier à l'aide des fonctions MouseEvent et KeyEvent. Ces fonctions étant appelées à chaque évènement, il suffit de les surcharger pour obtenir le comportement souhaité.

Par exemple, si nous souhaitons modifier l'affichage de notre vue 3D lorsque l'on appuie sur la barre d'espace, il suffit de surcharger la fonction keyPressEvent :

```
void HeightmapWidget::keyPressEvent(QKeyEvent *event)
{
    if(event->key() == Qt::Key_Space)
        // on modifie l'affichage
}
```

Si l'on souhaite donner à l'utilisateur la possibilité de s'approcher ou de s'éloigner de la heightmap en agissant sur la molette de la souris, on peut surcharger la fonction wheelEvent :

```
void HeightmapWidget::wheelEvent(QWheelEvent *event)
```

```
{
    distance *= 1.0 + (1.0 * event->delta() / 1200.0);
}
```

Pour finir, on peut ajouter la possibilité de tourner autour de la heightmap à l'aide de la souris. Pour cela, on calcule le déplacement de la souris entre deux évènements de déplacement (ou entre l'appui sur le bouton et le premier déplacement) et l'on modifie les variables x_rot, y_rot et z_rot en fonction :

```
void HeightmapWidget::mousePressEvent(QMouseEvent *event)
{
    last_pos = event->pos();
}

void HeightmapWidget::mouseMoveEvent(QMouseEvent *event)
{
    int dx = event->x() - last_pos.x();
    int dy = event->y() - last_pos.y();

    if (event->buttons() & Qt::RightButton)
    {
        rotateBy(dy*8, 0, 0);
        rotateBy(0, dx*8, 0);
    }
    last_pos = event->pos();
}

void HeightmapWidget::rotateBy(int x, int y, int z)
{
    x_rot += x;
    y_rot += y;
    z_rot += z;
}
```

Mesure des performances

Pour analyser les performances du rendu 3D, une solution serait de mesurer le temps mis par l'ordinateur pour exécuter la fonction de rendu

puis de réaliser une moyenne de cette valeur pour avoir un résultat stable. Dans notre application, nous allons utiliser une solution équivalente qui consiste à compter le nombre d'exécutions de la fonction de rendu pendant une seconde, ce qui nous donnera le nombre d'images par seconde affichées à l'écran (FPS : Frame Per Second).

Pour mettre à jour le rendu 3D, nous utilisons un timer qui appellera la fonction updateGL à intervalle régulier. Lorsque l'on souhaite mesurer le nombre de FPS (images par seconde), on lance ce timer avec un intervalle de 0 ms. En utilisation courante, on limite le nombre de FPS, par exemple en lançant le timer avec un intervalle de temps de 20 ms. La variable frame_count permet de compter le nombre d'images par seconde tandis que la variable last_time enregistre le temps de la dernière mise à jour de la vue.

Attention, pour mesurer le nombre de FPS maximal, il est nécessaire de désactiver la synchronisation verticale sous Windows. La méthode varie en fonction du modèle, le lecteur se reportera aux spécifications données par le constructeur de la carte graphique.

```
class HeightmapWidget
{
    ...
private:
    Qtimer timer;
    QTime last_time;
    int last_count;
    int frame_count;
};

HeightmapWidget::HeightmapWidget() {
    connect(&timer, SIGNAL(timeout()), this, SLOT(updateGL()));
    timer.start(20);
    frame_count = 0;
    last_count = 0;
    last_time = QTime::currentTime();
```

Le décompte du nombre d'images affichées par seconde est réalisé, dans la fonction de rendu paintGL, à l'aide d'une variable incrémentée à chaque passe. Ceci combiné à deux variables de type QTime qui nous

permettent de délimiter des intervalles de temps de une seconde. À chaque passe, on vérifie si au moins une seconde s'est écoulée depuis le dernier décompte. Si c'est le cas, le nombre de rendus effectués pendant l'intervalle est sauvegardé dans la variable last_count, le décompte est ensuite remis à zéro et le nouvel intervalle de temps démarre (fonction statique currentTime).

```
void HeightmapWidget::paintGL()
{
    ++frame_count;
    QTime new_time = QTime::currentTime();
    if (last_time.msecsTo(new_time) >= 1000)
    {
        // on sauvegarde le FPS dans last_count et on
réinitialise
        last_count = frame_count;
        frame_count = 0;
        last_time = QTime::currentTime();
    }
```

Le nombre de FPS est affiché avec la fonction renderText . À noter qu'il est possible d'afficher du texte en donnant les coordonnées 2D (dans notre cas) ou 3D.

```
    qglColor(Qt::white);
    renderText(20, 20, QString("FPS:%1").arg(last_count));
} // HeightmapWidget::paintGL()
```

À ce stade, le programme affiche une fenêtre simple avec le fond en noir, un repère orthonormé en couleur et le nombre de FPS en blanc.

Gestion des erreurs

OpenGL fonctionne sur le principe d'une machine à états. Les fonctions n'ont pas de boolean en retour de fonction pour indiquer qu'une erreur est survenue. En cas d'erreur, un flag interne à OpenGL est simplement activé. Il est donc possible (et même nécessaire) de vérifier qu'une erreur n'est pas survenue après un appel à une fonction OpenGL à l'aide de la fonction glGetError.

OpenGL, Qt