

Utilisation avancée des processeurs graphiques avec Qt

par Rémi Achard Guillaume Belz

Date de publication : 28/04/2011

Dernière mise à jour : 30/07/2011

Ce tutoriel présente l'utilisation du processeur graphique (GPU) pour créer des applications de rendu 3D et de calculs parallèles multiplateformes sous Qt. Deux API libres et multiplateformes seront utilisées afin de communiquer avec le processeur graphique : OpenGL pour la partie rendu 3D et OpenCL pour le calcul. Un des points-clés de ce tutoriel est la recherche d'optimisation des performances. Différentes méthodes de programmation seront proposées : partant de versions qu'on pourrait qualifier de "naïves" c'est-à-dire simples mais non optimisées, le souci de performances nous poussera à l'utilisation de techniques d'optimisation permettant souvent d'améliorer considérablement les performances. Un exemple récurrent est utilisé tout au long du tutoriel : la génération de terrain.

N'hésitez pas à commenter cet article !

1 - Introduction.....	4
1-A - Contenu du tutoriel et objectifs.....	4
1-B - Environnement de développement.....	4
1-C - Prérequis.....	4
2 - Générer un terrain.....	5
2-A - La classe HeightmapWidget.....	5
2-B - Chargement des données du terrain.....	6
2-C - Initialisation de la vue.....	8
2-D - Gestion de la souris et du clavier.....	9
2-E - Mesure des performances.....	10
2-F - Gestion des erreurs.....	11
3 - Envoyer des données au processeur graphique.....	12
3-A - Envoi des vertex un par un.....	12
3-B - Envoi de tableau de vertices (Vertex Array).....	13
3-C - Utilisation des tableaux d'indices.....	14
3-D - Stockage des vertices dans la mémoire du processeur graphique (Vertex Buffer Objects).....	15
3-E - Comparaison des performances.....	17
4 - Utilisation du pipeline programmable.....	19
4-A - La manipulation de shaders avec Qt : QGLShaderProgram et QGLShader.....	20
4-B - Le code source des shaders.....	21
4-C - Manipulation des matrices à l'aide de Qt.....	22
4-D - Passage de paramètres aux shaders.....	23
5 - Ajouter des lumières et des textures.....	26
5-A - Le modèle de Phong.....	26
5-B - Chargement de la Normal Map.....	26
5-C - Afficher les lumières avec les shaders.....	27
5-D - Appliquer une texture.....	29
5-E - Calcul dans le Vertex Shader vs dans le Fragment Shader.....	31
6 - Réaliser un rendu off-screen.....	34
6-A - Utilisation de QGLPixelBuffer.....	34
6-B - Utilisation de QGLFrameBufferObject.....	36
6-C - Dessiner dans une texture dynamique avec QPainter.....	38
7 - Overpainting : dessiner en 2D avec QPainter sur une scène 3D.....	40
8 - Gestion des extensions avec QGLContext::getProcAddress().....	42
9 - Introduction au GPU computing.....	43
9-A - Le calcul parallèle.....	43
9-B - Les API OpenCL.....	45
9-C - Installer OpenCL.....	45
9-D - Installer QtOpenCL.....	46
11 - Architecture des GPU.....	47
11-A - Vue générale.....	47
11-B - Les mémoires.....	47
10 - Le platform layer API.....	48
10-A - Gérer les erreurs.....	48
10-B - La plateforme.....	49
10-C - Le device.....	50
10-D - Le contexte.....	50
10-E - Le programme.....	51
10-F - Le Command queue.....	51
10-F-a - Les streams.....	52
10-G - Le kernel.....	53
10-G-a - Créer d'un kernel.....	53
10-G-b - Passer des arguments au kernel.....	53
10-G-c - Exécuter le kernel.....	54
10-H - Les buffers.....	54
10-H-a - Allouer un buffer.....	54
10-H-b - Envoyer et recevoir des données depuis un buffer.....	55
10-h-c - Libérer un buffer.....	55

10-I - Les évènements.....	55
11-bis - Ajouter deux vecteurs.....	57
12-bis - Ajouter deux vecteurs.....	58
11-bis-A - Introduction.....	58
11-bis-B - Les tampons mémoire vidéo.....	58
14 - Exemples.....	60
12 - Calculer les vecteurs normaux de la heightmap.....	60
12-A - Principe du calcul.....	60
12-B - Calcul des normales à l'aide de QtOpenCL.....	62
12-C - Partager des données entre OpenCL et OpenGL.....	65
13 - Codes sources des exemples présentés.....	67
14-A - Application minimale QtOpenGL.....	67
14-B - Comparaison entre les différents modes de transfert de données au GPU.....	67
14-C - Utilisation des shaders.....	68
14-D - Rendu off-screen.....	69
15 - Références.....	70
15-A - Les tutoriels de Developpez.....	70
15-B - Site officiel OpenGL.....	70
15-C - OpenCL.....	70
16 - Copyright.....	71
17 - Remerciements.....	72

1 - Introduction

1-A - Contenu du tutoriel et objectifs

La recherche de performances dans les domaines du rendu 3D ou du calcul lourd a toujours été une préoccupation majeure pour les développeurs. Tout au long de ce tutoriel, nous allons étudier les différents moyens de réaliser des applications sous Qt utilisant au mieux la capacité de calcul mise à notre disposition par l'intermédiaire du processeur graphique. S'appuyant sur un problème classique de programmation 3D, le rendu de terrain, nous allons explorer les différentes techniques permettant d'optimiser l'utilisation de ces ressources de calcul.

L'objectif est de présenter en détail les fonctionnalités offertes par Qt permettant de travailler avec les GPU. Le module **QtOpenGL**, intégré dans le framework, nous permettra de réaliser des rendus 3D. Dans un premier temps, le rendu sera réalisé en utilisant le pipeline fixe et des fonctions qui sont maintenant dépréciées. Dans une deuxième étape, nous présenterons l'utilisation du pipeline programmable, des buffers et des shaders. Cela permettra au lecteur de pouvoir plus facilement adapter le code trouvé sur Internet (qui utilise souvent du code déprécié) en code moderne. Nous implémenterons différentes fonctionnalités classiques de la 3D : la gestion de l'éclairage, les ombres et les textures. Les techniques avancées de 3D, par exemple celles présentées sur le site de **NVIDIA**, n'utilisent pas de fonction de Qt spécifique et ne seront donc pas détaillées.

Dans la dernière partie, la bibliothèque **QtOpenCL**, provenant de Qt Labs, sera utilisée pour les calculs parallèles sur GPU. Le code présenté sera basé sur un exemple simple et didactique : le calcul des vecteurs normaux à des surfaces. Nous aborderons en particulier les techniques de profiling et de débogage et les benchmarks.

1-B - Environnement de développement

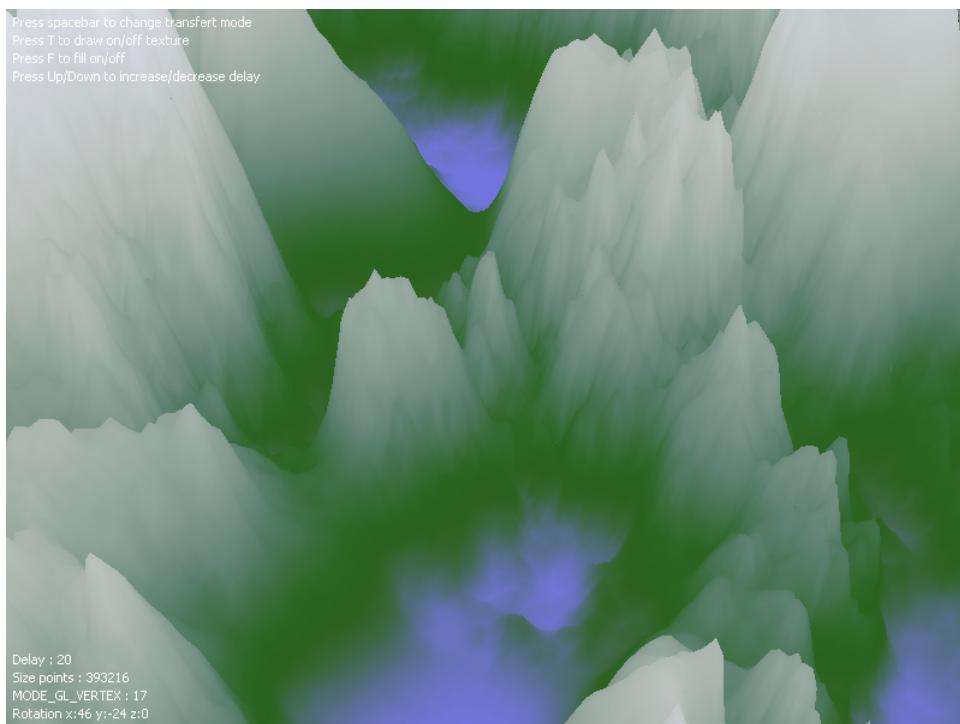
Pour pouvoir compiler le code présenté dans ce tutoriel, il faut pour commencer, disposer d'une version de Qt fonctionnelle. Certaines fonctionnalités, par exemple les buffers, nécessitent la version 4.7 de Qt. Pour la partie GPGPU, il faut également installer le module **QtOpenCL**, téléchargeable sur le site **Qt Labs**, ainsi que les pilotes de développement de la carte graphique, téléchargeable sur le site des constructeurs (par exemple, le **CUDA** pour les cartes NVIDIA). Le lecteur se référera aux différents manuels fournis pour leurs installations.

1-C - Prérequis

Ce tutoriel s'adresse en premier lieu aux développeurs ayant les bases de la programmation avec Qt. Avoir des notions en programmation 3D ou en GPGPU facilitera la compréhension mais n'est pas indispensable. Certaines notions de base ne seront pas présentées, le lecteur devra donc se référer aux autres tutoriels proposés par la rubrique **Qt de Developpez**, sur la **programmation Qt en général** et plus spécialement aux tutoriels **Intégration d'OpenGL dans une interface Qt**, **Qt Graphics et performance - Le moteur de rendu OpenGL** et **Utiliser OpenCL avec Qt**. Le lecteur pourra également consulter les tutoriels de la rubrique **Jeux de Developpez**, en particulier les tutoriels sur CUDA et sur la génération de terrain.

2 - 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 dans le projet appelé "OpenGL - code minimal" (fichier [Source zip](#)). Dans cet exemple, on n'affiche pas la heightmap mais un simple repère orthonormé.

2-A - 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

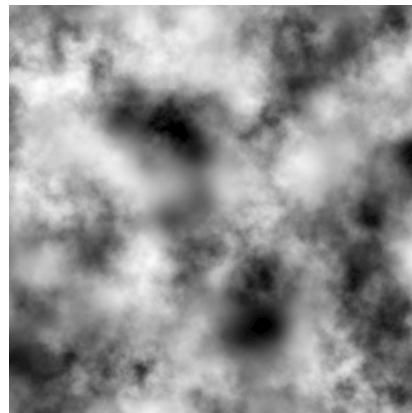
public:
    HeightmapWidget(QWidget* parent = 0);
    ~HeightmapWidget();

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

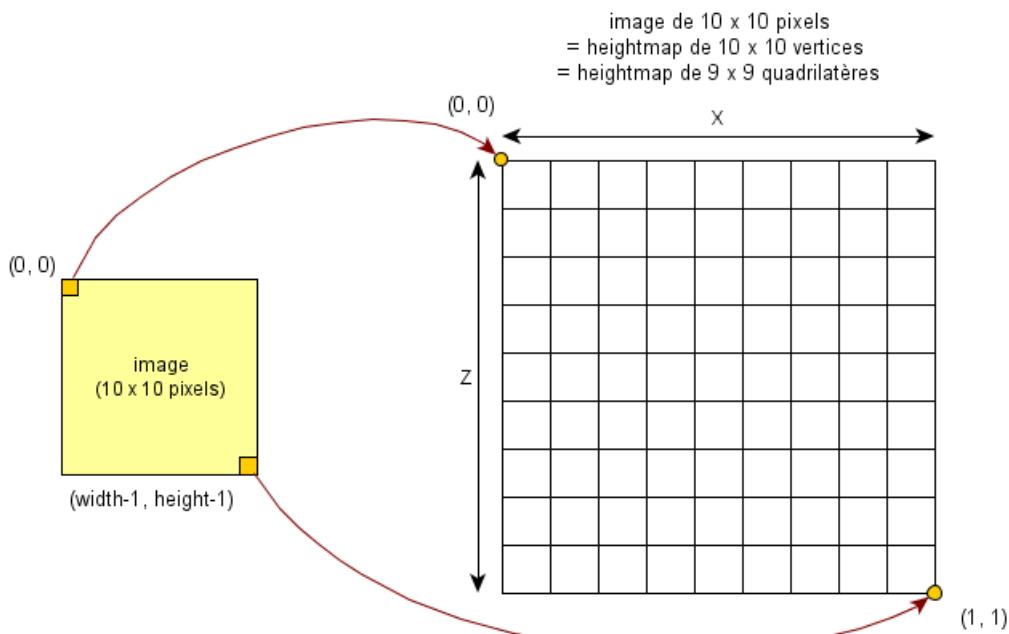
Pour plus de détails sur **QGLWidget**, le lecteur se reportera au tutoriel d'introduction : **Intégration d'OpenGL dans une interface Qt**.

2-B - 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 quadrillatères dans les variables `quads_by_x` et `quads_by_z`. En pratique, comme nous utilisons une grille uniforme, le nombre de quadrillatères 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 quadrillatères 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);

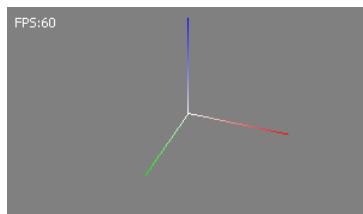
        m_vertices.push_back(vertice);
    }
}

```

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.

2-C - 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);
}
```

2-D - 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 événements 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;
}
```

2-E - 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

```
class HeightmapWidget
```

```
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()
```

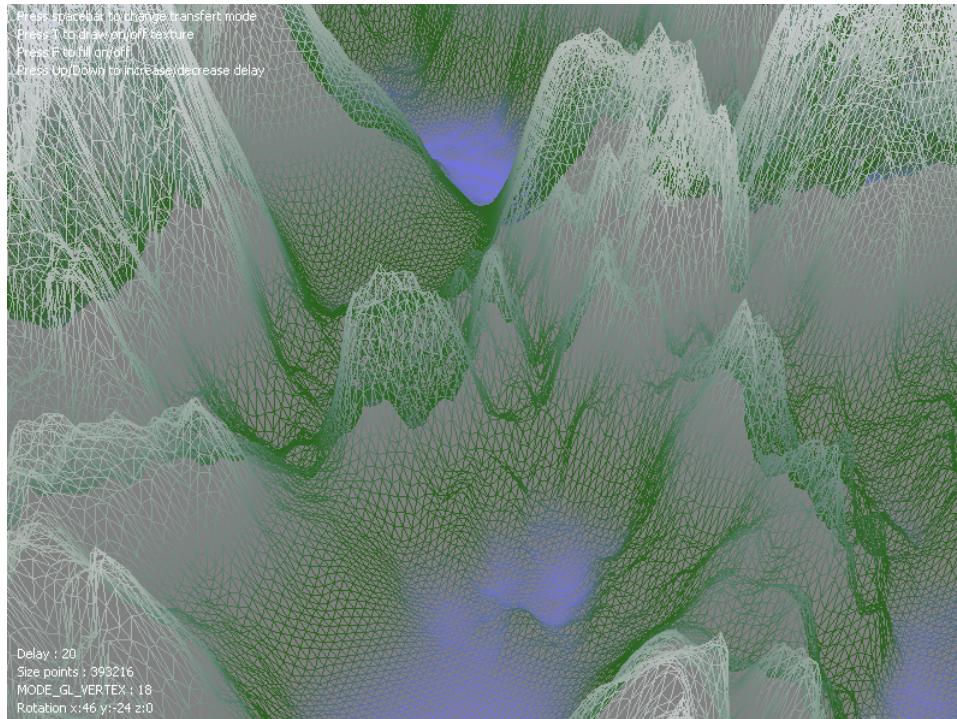
À ce stade, le programme affiche une fenêtre simple avec le fond en noir, un repère orthonormé en couleur et le nombre de FPS en blanc.

2-F - 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**.

3 - Envoyer des données au processeur graphique

Les sources du chapitre sont disponibles dans le fichier [Source zip](#). L'application permet de changer dynamiquement de mode de transfert de données, d'afficher ou non les textures, de modifier la limitation du FPS.



⚠ Il existe plusieurs méthodes pour transférer les données entre la mémoire centrale (RAM) et la mémoire du processeur graphique. Certaines des méthodes présentées ici sont obsolètes et sont donc déconseillées. Elles sont présentées à titre indicatif et de comparaison uniquement : beaucoup de tutoriels présents sur Internet utilisent des méthodes obsolètes ; il est donc intéressant d'avoir le code des différentes méthodes pour pouvoir adapter du code obsolète en code plus moderne.

Dans cette partie, nous allons utiliser le pipeline de rendu fixe de la carte graphique pour dessiner notre terrain. Le pipeline fixe fait partie des méthodes obsolètes et il n'est pas conseillé de l'utiliser. Il a été conservé sur les versions Desktop d'OpenGL uniquement, pour des raisons de compatibilité. Il est utilisé dans ce chapitre pour pouvoir se concentrer uniquement sur le transfert des données. L'utilisation du pipeline programmable est présentée dans le chapitre suivant.

3-A - Envoi des vertex un par un

La méthode la plus simple est d'envoyer chaque vertex au moment de l'affichage. Cette méthode a le désavantage de consommer de la bande passante inutilement, ce qui peut dégrader les performances. Dans la pratique, cette méthode se déroule en plusieurs étapes : dans un premier temps, il faut choisir le type de primitive à dessiner (ligne, triangle, carré, etc.) avec la fonction **glBegin** puis envoyer les caractéristiques de chaque vertex (position, couleur, vecteur normal, coordonnées de la texture, etc.) un par un. À la fin de l'envoi des vertices, il faut appeler la fonction **glEnd**. Il est possible de dessiner plusieurs primitives de même type dans un seul bloc **glBegin-glEnd**.

La difficulté dans cet exemple vient du fait que nous allons dessiner le terrain à l'aide de triangles, qui est la primitive de base pour le rendu 3D. Notre terrain est composé d'une grille carrée de N points, ce qui donne N-1 arêtes par dimension. La grille est donc composée de $(N-1)^2$ carrés. Pour un carré, il nous faut deux triangles côté à côté, ce qui correspond à 6 vertices. Les vertices composant chaque triangle devant être envoyés dans l'ordre à la carte

graphique, nous ne pourrons pas nous contenter de parcourir notre vecteur de vertices séquentiellement, deux boucles imbriquées seront nécessaires.

L'envoi des vertices composants chaque triangle du terrain est réalisé dans la fonction de rendu **paintGL** grâce à deux boucles for imbriquées parcourant chaque quadrilatère de la grille.

```
void HeightmapWidget::paintGL()
{
    glBegin(GL_TRIANGLES);
    for (int z = 0; z < quads_by_z; ++z)
    {
        for (int x = 0; x < quads_by_x; ++x)
        {

```

Nous parcourons le vecteur vertices contenant les positions des vertices avec une boucle pour chaque dimension x et z. Chaque couple (x, z) correspond à un seul carré de la grille. Il est nécessaire de calculer l'indice dans le tableau du premier point du carré correspondant à (x, z) :

```
    int i = z * vertices_by_x + x;
```

Les trois autres points définissant le carré se trouvent aux positions (x+1, z), (x, z+1) et (x+1, z+1). Les deux triangles composant chaque carré sont dessinés à l'aide de la fonction **glVertex3f**. La fonction **qglColor** permet de préciser la couleur qui sera appliquée à ces triangles.

```
    qglColor(Qt::green);

    glVertex3f(m_vertices[i].x(), m_vertices[i].y(),
               m_vertices[i].z());
    glVertex3f(m_vertices[i+vertices_by_x].x(), m_vertices[i+vertices_by_x].y(),
               m_vertices[i+vertices_by_x].z());
    glVertex3f(m_vertices[i+1].x(), m_vertices[i+1].y(),
               m_vertices[i+1].z());

    glVertex3f(m_vertices[i+1].x(), m_vertices[i+1].y(),
               m_vertices[i+1].z());
    glVertex3f(m_vertices[i+vertices_by_x].x(), m_vertices[i+vertices_by_x].y(),
               m_vertices[i+vertices_by_x].z());
    glVertex3f(m_vertices[i+1+vertices_by_x].x(), m_vertices[i+1+vertices_by_x].y(),
               m_vertices[i+1+vertices_by_x].z());
}

glEnd();
}
```

Une amélioration possible est de compiler les différentes instructions d'un bloc **glBegin-glEnd** dans une Display list avec les fonctions **glNewList** et **glEndList**.

3-B - Envoi de tableau de vertices (Vertex Array)

Au lieu d'envoyer les vertices un par un, il est possible d'envoyer directement un tableau contenant une liste de vertex à la carte graphique. Cette technique est communément appelée Vertex Array (littéralement "tableau de vertices"). Pour l'utiliser, nous ne pourrons pas nous contenter d'envoyer directement le tableau de vertex créé au chargement des données à partir de l'image heightmap. En effet, comme on a pu le voir dans le chapitre précédent, chaque carré est dessiné à l'aide de deux triangles ayant deux vertices en commun. Il est donc nécessaire de recopier les vertices dans un nouveau tableau (également un vecteur), en les ordonnant et en dupliquant certains vertex.

```
class HeightmapWidget : public QGLWidget
{
```

```

    ...
private:
    QVector<QVector3D>  m_vertexarray;
};

```

La méthode de remplissage du vecteur est similaire à celle présentée dans la partie précédente, sauf que les positions des vertices sont stockées dans un tableau créé lors de l'initialisation.

```

void HeightmapWidget::initializeGL()
{
    ...
    for (int z = 0; z < quads_by_z; ++z)
    {
        for (int x = 0; x < quads_by_x; ++x)
        {
            int i = z * vertices_by_x + x;

            m_vertexarray.push_back(m_vertices[i]);
            m_vertexarray.push_back(m_vertices[i+vertices_by_x]);
            m_vertexarray.push_back(m_vertices[i+1]);

            m_vertexarray.push_back(m_vertices[i+1]);
            m_vertexarray.push_back(m_vertices[i+vertices_by_x]);
            m_vertexarray.push_back(m_vertices[i+1+vertices_by_x]);
        }
    }
}

```

L'affichage de ce tableau de données nécessite plusieurs étapes. Dans un premier temps, il faut indiquer à la carte graphique que l'on va travailler avec des Vertex Array à l'aide de la fonction **glEnableClientState**. Il faut ensuite envoyer le tableau de données contenant les vertices à la carte graphique à l'aide de la fonction **glVertexPointer**. Cette fonction prend trois paramètres : le nombre de composantes dans un vertex (trois dans notre cas), le type de données (des GL_FLOAT dans l'exemple) et un pointeur constant vers le tableau de données. La fonction **glDrawArrays** permet de dessiner les triangles et prend comme paramètres le type de primitive (GL_TRIANGLES dans l'exemple) et le nombre de vertices.

```

void HeightmapWidget::paintGL()
{
    qglColor(Qt::white);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, m_vertexarray.constData());
    glDrawArrays(GL_TRIANGLES, 0, m_vertexarray.size());
    glDisableClientState(GL_VERTEX_ARRAY);
}

```

3-C - Utilisation des tableaux d'indices

Dans le paragraphe précédent, nous avons mis en évidence un problème de duplication des vertices qui conduit à une augmentation de la taille du tableau envoyé à la carte graphique. Pour éviter cette surcharge inutile, il est possible d'utiliser notre tableau de vertices créé lors du chargement de la heightmap et de fournir un tableau d'indices indiquant l'ordre des vertices à utiliser pour dessiner les triangles. Lorsqu'un vertex sera commun à plusieurs triangles, il suffira donc de dupliquer uniquement l'indice. Cette technique permet de réduire de façon non négligeable le volume de données à envoyer au processeur graphique et donc d'améliorer les performances.

```

class HeightmapWidget : public QGLWidget
{
    ...
private:
    QVector<GLuint> m_indices;
};

```

La méthode de création du tableau d'indices est équivalente aux méthodes présentées dans les chapitres précédents, avec pour seule différence que l'on remplit le tableau avec les indices.

```
void HeightmapWidget::initializeGL()
{
    ...
    for (int z = 0; z < quads_by_z; ++z)
    {
        for (int x = 0; x < quads_by_x; ++x)
        {
            int i = z * vertices_by_x + x;

            m_indices.push_back(i);
            m_indices.push_back(i + vertices_by_x);
            m_indices.push_back(i + 1);

            m_indices.push_back(i + 1);
            m_indices.push_back(i + vertices_by_x);
            m_indices.push_back(i + 1 + vertices_by_x);
        }
    }
}
```

Le rendu ne s'effectue plus avec la fonction **glDrawArrays** mais avec la fonction **glDrawElements**, qui prend comme paramètres le nombre d'indices dans le tableau, le type d'indice (GL_UNSIGNED_INT ici) et un pointeur constant vers le tableau d'indices.

```
void HeightmapWidget::paintGL()
{
    qglColor(Qt::green);

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, m_vertices.constData());
    glDrawElements(GL_TRIANGLES, m_indices.size(), GL_UNSIGNED_INT, m_indices.constData());
    glDisableClientState(GL_VERTEX_ARRAY);
    ...
}
```

3-D - Stockage des vertices dans la mémoire du processeur graphique (Vertex Buffer Objects)

Jusqu'à présent, les données sont envoyées à la carte graphique lors de chaque mise à jour de l'affichage. Lorsque le volume de données à envoyer est important, cela peut dégrader les performances. La solution consiste à envoyer une seule fois les données constantes et de mettre à jour uniquement les paramètres qui sont modifiés. Dans notre exemple de heightmap, les positions des vertices et les indices sont constants. On peut donc les envoyer lors de l'initialisation, ce qui diminue les transferts de données lors des mises à jour.

Les Vertex Buffer Objects sont des tampons de vertices stockés dans la carte graphique, à l'inverse des Vertex Array qui sont stockés dans la mémoire centrale et nécessitent donc un transfert vers la mémoire graphique à chaque rendu. L'économie en termes de transfert mémoire CPU/GPU sera importante.

Le module QtOpenGL fournit la classe **QGLBuffer** pour faciliter la manipulation des buffers. Il est possible d'utiliser les buffers avec ou sans indices. Nous ne présentons ici que la version avec indices. Deux types **QGLBuffer** seront donc utilisés, un pour le tableau de vertices et l'autre pour le tableau d'indices :

```
class HeightmapWidget : public QGLWidget
{
    ...
private:
    QGLBuffer    m_vertexbuffer;
    QGLBuffer    m_indexbuffer;
```

```
};
```

Il est nécessaire de préciser, lors de la construction de ces objets, le type de données qu'ils vont contenir :

```
HeightmapWidget::HeightmapWidget(QWidget *parent) :
    QGLWidget(parent),
    m_vertexbuffer(QGLBuffer::VertexBuffer),
    m_indicebuffer(QGLBuffer::IndexBuffer)
{
    ...
}
```

Pour créer les buffers, il faut les initialiser avec la fonction **create** puis préciser que l'on va travailler dessus avec la fonction **bind**. Il faut ensuite allouer un bloc mémoire *dans la carte graphique* de taille souhaitée avec la fonction **allocate**. Il est possible de remplir la mémoire graphique allouée à ce stade en fournissant un pointeur constant vers un tableau de données ou dans un second temps en fournissant un pointeur à la fonction **write**. On termine en précisant à la carte graphique que l'on a fini d'utiliser ce buffer avec la fonction **release**. L'allocation du buffer d'indices n'est pas détaillée mais ne présente aucune difficulté particulière.

Ces différentes fonctions sont équivalentes respectivement aux fonctions OpenGL suivantes : **glGenBuffers**, **glBindBuffer**, **glBufferData** et **glDeleteBuffers**.

```
void HeightmapWidget::initializeGL()
{
    // Vertex buffer init
    m_vertexbuffer.create();
    m_vertexbuffer.bind();
    m_vertexbuffer.allocate(m_vertices.constData(), m_vertices.size() * sizeof(QVector3D));
    m_vertexbuffer.release();

    // Indices buffer init
    m_indicebuffer.create();
    m_indicebuffer.bind();
    m_indicebuffer.allocate(m_indices.constData(), m_indices.size() * sizeof(GLuint));
    m_indicebuffer.release();
}
```

Le code du rendu est similaire au code du chapitre précédent. La principale différence vient du fait que l'on passe un pointeur nul aux fonctions, à la place du pointeur constant vers les données. Il faut dans un premier temps activer le buffer que l'on souhaite utiliser avec **bind** puis appeler les fonctions OpenGL en passant la valeur NULL à la place du pointeur de données. Il faut ensuite préciser à OpenGL que l'on a fini de travailler avec un buffer avec **release** avant de pouvoir utiliser un autre buffer. Si on souhaite ne pas utiliser de tableau d'indices, on utilisera **glDrawArrays** à la place de **glDrawElements**.

```
void HeightmapWidget::paintGL()
{
    qglColor(Qt::green);

    glEnableClientState(GL_VERTEX_ARRAY);

    m_vertexbuffer.bind();
    glVertexPointer(3, GL_FLOAT, 0, NULL);
    m_vertexbuffer.release();

    m_indicebuffer.bind();
    glDrawElements(GL_TRIANGLES, m_indices.size(), GL_UNSIGNED_INT, NULL);
    m_indicebuffer.release();

    glDisableClientState(GL_VERTEX_ARRAY);
    ...
}
```

Dans cet exemple, nous utilisons les VBO avec des tableaux d'indices, il est néanmoins tout à fait possible d'utiliser les VBO sans tableau d'indices et d'effectuer le rendu à l'aide de la fonction `glDrawArrays`.

3-E - Comparaison des performances

Nous allons maintenant comparer les performances des différentes techniques présentées tout au long de cette partie. Pour cela, le code de chaque méthode est implémenté dans un même programme. Le choix du rendu peut être changé dynamiquement.

```
class HeightmapWidget : public QGLWidget
{
    ...
private:
    enum MODE_RENDER { MODE_GL_VERTEX, MODE_VERTEXARRAY, MODE_VERTEXARRAY_INDICES,
        MODE_VERTEBUFFEROBJECT_INDICES };
    MODE_RENDER mode_rendu;
}
```

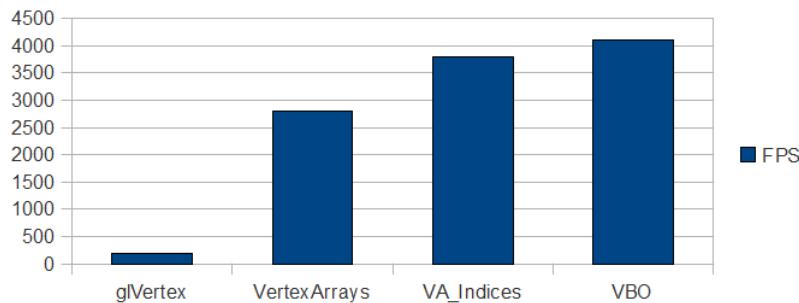
La sélection du mode de rendu s'effectue dans la fonction `paintGL` :

```
void HeightmapWidget::paintGL()
{
    ...
    switch(mode_rendu)
    {
        case MODE_GL_VERTEX:
            // implémentation de la première méthode
            break;
        case MODE_VERTEXARRAY:
            // implémentation de la deuxième méthode
            break;
        case MODE_VERTEXARRAY_INDICES:
            // implémentation de la troisième méthode
            break;
        case MODE_VERTEBUFFEROBJECT_INDICES:
            // implémentation de la quatrième méthode
            break;
    }
}
```

La fonction `keyPressEvent` est surchargée pour permettre à l'utilisateur de changer de mode de rendu grâce à la barre d'espace du clavier. Le modulo sert à conserver une valeur comprise entre 0 et 3, car il n'y a que quatre éléments dans notre énumération.

```
void HeightmapWidget::keyPressEvent(QKeyEvent *event)
{
    if (event->key() == Qt::Key_Space)
        mode_rendu = static_cast<MODE_RENDER>((mode_rendu + 1) % 4);
}
```

Les différentes versions du rendu du terrain sont implémentées ainsi que le mécanisme de changement du mode de rendu pendant l'exécution. Il est maintenant temps d'analyser les performances obtenues par les différentes méthodes de rendu. Voici un tableau récapitulant les performances obtenues sur la plateforme de test (Intel i5, Ubuntu 9.10, GPU NVIDIA GTX460 driver 3.2) utilisée lors de la rédaction de ce tutoriel.



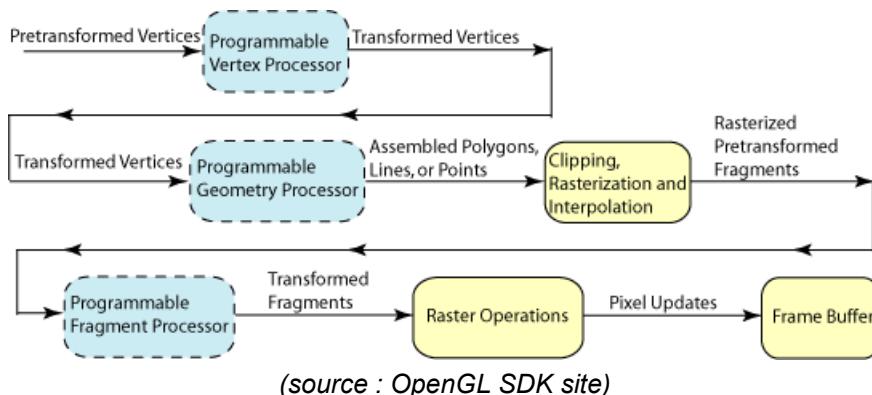
Sans surprise, les VBO fournissent les meilleures performances, le principal goulot d'étranglement dans notre exemple étant les transferts mémoire entre le client (CPU) et le serveur (GPU). En fonction du type de scène à dessiner et de la quantité de données constantes et non constantes utilisées, la différence de performances entre ces méthodes peut changer beaucoup. Il peut être intéressant d'effectuer des tests pour sélectionner la méthode la plus adaptée à ses besoins.

 *Il est possible que la synchronisation verticale bloque le nombre d'images par seconde (par exemple sous Windows). Pour débloquer le nombre d'images par seconde et donc mesurer les performances de l'application, il faut désactiver temporairement cette option dans le panneau de contrôle de la carte graphique.*

4 - Utilisation du pipeline programmable

Ce chapitre présente l'utilisation du pipeline programmable et des shaders avec OpenGL et Qt. Le lecteur intéressé par les détails se reportera au tutoriel de LittleWhite. Les codes sources peuvent être téléchargés dans le fichier **zip**.

Jusqu'à maintenant nous nous sommes limités à l'utilisation d'OpenGL avec le pipeline de rendu fixe. Cette technique s'est avérée trop restrictive pour la création d'environnements 3D réalistes : l'industrie du rendu 3D a donc poussé les fabricants de processeurs graphiques à revoir leur architecture de rendu 3D afin de la rendre plus souple. Un effort particulier a été réalisé sur les méthodes de personnalisation du rendu 3D : c'est la naissance du pipeline programmable et des fameux *shaders* permettant aux développeurs de modifier certaines étapes du pipeline 3D selon leurs besoins. Cette transition s'est effectuée en 2001-2002, avec l'arrivée de DirectX8 et des OpenGL Shader extensions. L'utilisation du pipeline fixe est donc déconseillée, la plupart des fonctions utilisées dans la partie précédente sont **dépréciées**. Ces fonctions ne sont plus supportées sur OpenGL ES 2.0 (version d'OpenGL destinée aux systèmes embarqués).



La principale nouveauté apportée par le pipeline programmable est l'utilisation de programmes remplaçant certaines étapes du rendu 3D qui étaient auparavant codées en "dur" dans le pipeline. Ces programmes sont fournis par le développeur 3D à la carte graphique et permettent de personnaliser des étapes du rendu OpenGL. On peut citer comme exemple d'utilisations répandues : la gestion de l'éclairage dynamique, le rendu d'eau réaliste ou encore la gestion des reflets. Ces programmes sont constitués de plusieurs shaders (qui provient de l'anglais *to shade* qui signifie *nuancer*). Les shaders sont écrits dans un langage proche du C et qui varie selon les API de rendu 3D utilisées. Dans le cas d'OpenGL, il s'agit du langage GLSL (OpenGL Shading Language). Direct3D utilise quant à lui le HLSL (High Level Shading Language). Heureusement, tous ces langages ont une syntaxe assez proche, ce qui permet aux développeurs de s'adapter à chaque langage rapidement.

⚠ Il existe plusieurs versions de GLSL et certaines pratiques sont dépréciées. Dans ce tutoriel, nous n'utiliserons pas les mots-clés et fonctions dépréciés. Le lecteur ayant déjà rencontré du code GLSL ne devra donc pas être surpris s'il ne retrouve pas exactement la même syntaxe dans les shaders qui seront présentés plus bas.

Il existe à l'heure actuelle quatre types de shaders, présentés dans l'ordre d'appel dans le pipeline :

- Vertex Shader : applique un traitement à chaque vertex (translation, couleur, etc.) ;
- Tessellation Shader : transforme une primitive (triangle, quadrilatère, etc.) en série de petits éléments (point, ligne, etc.) ;
- Geometry Shader : crée ou supprime les vertices d'une primitive 3D dynamiquement ;
- Fragment Shader (ou Pixel Shader dans la terminologie Direct3D) : applique un traitement à chaque pixel (modification de la couleur, transparence, etc.)

Les Vertex Shader et les Fragment Shader sont les plus anciens shaders pris en charge par les cartes graphiques et sont donc les plus utilisés. Les deux autres sont d'usage moins courant. Dans ce tutoriel, nous ne présenterons pas les Tessellation Shader, qui ne sont pas pris en charge par Qt. Les Vertex Shader et les Fragment Shader seront

présentés dans ce chapitre et un exemple de Geometry Shader (uniquement à partir de la version 4.7 de Qt) sera présenté dans le chapitre suivant.

Chaque shader s'applique à un élément de base (par exemple, un vertex pour les Vertex Shader et un pixel pour les Fragment Shader). Une particularité est que le code des shaders est exécuté à l'identique pour chaque élément de base, indépendamment les uns des autres (ce qui implique que le code d'un shader qui s'exécute pour deux éléments différents ne peut pas échanger de données entre eux ; par contre, deux shaders de types différents peuvent échanger des données entre eux, dans l'ordre d'exécution du pipeline, via l'intermédiaire des mémoires graphiques partagées). Ce concept est appelé "programmation parallèle". Les cartes graphiques sont spécialement conçues pour profiter de cette particularité : les processeurs graphiques sont en fait constitués de plusieurs cores, chacun pouvant exécuter un shader sur un élément. Plus le processeur contient de cores, plus le temps d'exécution globale sera diminué (bien sûr, d'autres éléments des processeurs graphiques interviennent sur les performances).

Dans cette partie, nous allons réécrire le programme de rendu de terrain pour qu'il utilise le pipeline programmable et présenter les outils de manipulation de shaders mis à notre disposition par Qt. Nous ne donnerons pas la version OpenGL du code, qui peut être trouvée dans le tutoriel de LittleWhite.

4-A - La manipulation de shaders avec Qt : QGLShaderProgram et QGLShader

Dans Qt, les shaders sont gérés principalement avec deux classes dans Qt : **QGLShaderProgram** et **QGLShader**.

- La classe **QGLShader** permet de manipuler les différents types de shaders, que ce soient les Vertex Shader, les Geometry Shader ou les Fragment Shader. Les shaders étant un code exécutable sur le processeur graphique, il faut fournir ce code sous forme de chaîne de caractères puis le compiler. Pour créer un shader, il suffit donc de créer un **QGLShader** en spécifiant le type de shader puis de compiler le code avec la fonction **compileSourceCode** (si l'on souhaite donner directement le code sous forme de chaîne de caractères) ou **compileSourceFile** (si le code est fourni dans un fichier séparé). On parle habituellement de compilation à la volée (Online compilation). Les dernières versions d'OpenGL permettent de sauvegarder et charger directement du code GLSL, sans avoir besoin de recompiler à chaque exécution (Offline compilation). Cette fonctionnalité n'est pas prise en charge nativement pour le moment dans Qt.
- La classe **QGLShaderProgram** permet de manipuler un programme OpenGL, c'est-à-dire plusieurs shaders liés entre eux pour former un pipeline programmable spécifique. Il ne peut y avoir au maximum qu'un seul shader de chaque type par programme et qu'un seul programme actif en même temps. Cette classe est responsable de l'activation des shaders dans le contexte OpenGL et de leurs manipulations (notamment le passage de paramètres). Il est possible créer directement des shaders à l'aide des fonctions **addShaderFromSourceCode** et **addShaderFromSourceFile** puis de récupérer les shaders d'un programme à l'aide de la fonction **shaders**.

La création de shaders est relativement simple, nous allons donc utiliser directement la méthode de création de shaders à partir d'un programme. Le **QGLShaderProgram** étant utilisé pour passer des paramètres aux shaders, nous le créons comme variable membre :

```
class HeightmapWidget : public QGLWidget
{
private:
    QGLShaderProgram m_program;
}
```

Pour le moment, nous utiliserons un seul **QGLShaderProgram** qui ne sera pas modifié au cours de son utilisation. Nous l'initialisons dans la fonction **initializeGL**. De plus, pour alléger le code C++, nous plaçons le code de chaque shader dans un fichier .glsl séparé (remarquez l'utilisation du mécanisme de ressources Qt que nous avons déjà rencontré au chargement de l'image heightmap).

```
void HeightmapWidget::initializeGL()
{
    m_program.addShaderFromSourceFile(QGLShader::Vertex,(":/shaders/vertex_shader.gl"));
```

```
m_program.addShaderFromSourceFile(QGLShader::Fragment,(":/shaders/fragment_shader.gl"));
```

Notre Shader Program contient simplement un Vertex Shader et un Fragment Shader. L'initialisation est presque terminée, la dernière étape consiste à lier les différents shaders composant le programme pour finaliser le pipeline de rendu. En cas d'erreur, le log peut être récupéré dans une QString grâce à la fonction **log** :

```
if (!m_program.link())
    QString error = m_program.log();
...
}
```

Si on avait voulu écrire directement le code dans le fichier .cpp sous forme de chaîne de caractères, il aurait fallu créer un objet de type **QGLShader**, l'initialiser en fournissant le type de shader et le code GLSL puis le compiler :

```
void HeightmapWidget::initializeGL()
{
    QGLShader vertex_shader(QGLShader::Vertex, this);
    const char* vertex_shader_source = "...";
    vertex_shader.compileSourceCode(vertex_shader_source);
    m_program.addShader(vertex_shader);
    m_program.link();
    ...
}
```

4-B - Le code source des shaders

Le contexte de rendu OpenGL est maintenant prêt pour utiliser le pipeline programmable, mais nous n'avons pas encore parlé du code des shaders. Pour débuter, un exemple de Vertex Shader et de Pixel Shader très simple va être présenté. Ces deux shaders se contentent de reproduire le comportement du pipeline fixe. Des exemples de shaders plus avancés seront présentés dans le chapitre suivant, sur la gestion des lumières, des ombres et des textures. Cependant, le code GLSL n'étant pas spécifique à Qt, le lecteur intéressé se reportera aux nombreux tutoriels présents sur la rubrique **Jeux** de Developpez et sur Internet.

Premier shader appelé dans le pipeline de rendu 3D, le Vertex Shader est exécuté pour chaque vertex composant le polygone à afficher. Il reçoit les paramètres d'un vertex en entrée (au minimum la position du vertex mais il peut également recevoir sa couleur, son vecteur normal, etc.) et renvoie des paramètres en sortie pour chaque vertex (également au minimum la position du vertex transformé).

Les variables d'entrées sont précédées des mots-clés `in` ou `uniform` (depuis la version 1.3 de GLSL, le mot-clé `in` remplace le mot-clé `attribute`). Les variables `in` sont les tableaux de données sur lesquels s'applique le shader. Comme un shader est exécuté en parallèle sur chaque élément, cela implique que l'ensemble du tableau n'est pas accessible mais uniquement les données correspondant au vertex courant. Au contraire, les variables `uniform` sont identiques pour tous les vertices. Elles seront donc utilisées pour passer par exemple la matrice de transformation Projection-Modèle-Vue ou la couleur d'ambiance.

Les variables de sortie sont spécifiées par le mot-clé `out`. En plus des variables définies par l'utilisateur, le langage GLSL fournit différentes variables "build-in", prêtes à l'emploi. Par exemple, la variable `gl_position` est utilisée pour envoyer la position du vertex, après transformation, aux étapes suivantes du pipeline.

En plus du mot-clé définissant si une variable est un paramètre d'entrée ou de sortie, il faut également indiquer le type de variable. En plus des types hérités du C (`int`, `float`, etc.), le langage GLSL fournit plusieurs autres types facilitant le calcul 3D. Par exemple, le type `vec4` représente un vecteur composé de quatre composantes et le type `mat4` définit une matrice de 4×4 . Il est courant de manipuler les coordonnées 3D avec quatre composantes au lieu de trois. La raison provient du fait que l'ensemble des transformations possibles en 3D (c'est-à-dire trois rotations et une translation) ne peuvent être représentées que par une matrice 4×4 . Puisqu'il n'est possible de multiplier une matrice

4x4 que par un vecteur de dimension 4 et que l'on préfère en général ne manipuler qu'un seul type de vecteur, on se limite habituellement aux vecteurs de dimension 4. Voir la [FAQ Mathématiques pour les jeux](#) pour plus de détails.

vertex_shader.gl

```
#version 130
in vec4 vertex;
uniform mat4 matrixpmv;
void main(void)
{
    gl_Position = matrixpmv * vertex;
}
```

Le Fragment Shader est appelé ensuite dans le pipeline pour chaque pixel apparaissant à l'écran. Il permet de modifier la couleur de chaque pixel dynamiquement, technique très utilisée entre autres pour l'éclairage. Le Fragment Shader présenté applique une couleur fixe (spécifiée dans la variable `fixed_color`) à chaque pixel. Nous verrons plus tard comment transmettre cette variable depuis notre application au shader. La couleur finale du pixel est transmise aux étapes suivantes du Pipeline 3D dans une variable de type `vec4` (représentant une couleur en RGBA : rouge, vert, bleu et transparence).

fragment_shader.gl

```
#version 130
out vec4 color;
uniform vec4 fixed_color;
void main(void)
{
    color = fixed_color;
}
```

4-C - Manipulation des matrices à l'aide de Qt

Jusqu'à maintenant, nous avons utilisé les fonctions fournies par OpenGL pour définir les paramètres de projection (`glMatrixMode`, `glLoadIdentity`, `glPerspective`, `glTranslate`, `gluLookAt`, etc.) Cependant, ces fonctions sont maintenant dépréciées et la méthode recommandée est d'envoyer directement aux shaders les matrices de projection. C'est le but de la variable `matrixpmv` utilisée dans le Vertex Shader précédent.

Le développeur doit donc maintenant implémenter ses propres fonctions de manipulation des matrices ou utiliser une librairie tierce. Dans ce but, Qt fournit ces outils de manipulation des matrices. Dans notre cas, nous utiliserons exclusivement la classe **QMatrix4x4** qui représente une matrice carrée 4x4. Cette classe est spécialement optimisée pour les calculs matriciels utilisés dans la programmation 3D. Voici un exemple de manipulation de matrices modèle-vue-projection utilisant la classe **QMatrix4x4**. Les fonctions **rotate**, **translate**, **perspective**, etc. sont similaires aux anciennes fonctions OpenGL utilisées dans le début du tutoriel.

Les paramètres utilisés sont les mêmes que ceux présentés dans le chapitre 2-C : `x_rot`, `y_rot` et `z_rot` représentent les rotations autour du point (0, 0, 0) et `distance` représente la distance entre ce point et l'observateur. Pour des raisons de lisibilité, les trois matrices (projection, modèle et vue) sont bien différencierées mais il est possible d'utiliser une seule matrice.

```
void HeightmapWidget::paintGL()
{
    QMatrix4x4 projection;
    projection.perspective(30.0, 1.0 * width() / height(), 0.1, 100.0);

    QMatrix4x4 model;
    model.rotate(x_rot / 16.0, 1.0, 0.0, 0.0);
    model.rotate(y_rot / 16.0, 0.0, 1.0, 0.0);
    model.rotate(z_rot / 16.0, 0.0, 0.0, 1.0);

    QMatrix4x4 view;
```

```

view.translate(0.0, 0.0, distance);
...
}

```

⚠ Le langage GLSL permet la manipulation des matrices et il serait possible de n'envoyer aux shaders que les paramètres qui varient au cours du temps et faire les calculs dans les shaders. Cela impliquerait que le même calcul soit refait dans les shaders pour chaque vertex à calculer, ce qui pourrait nuire aux performances.

4-D - Passage de paramètres aux shaders

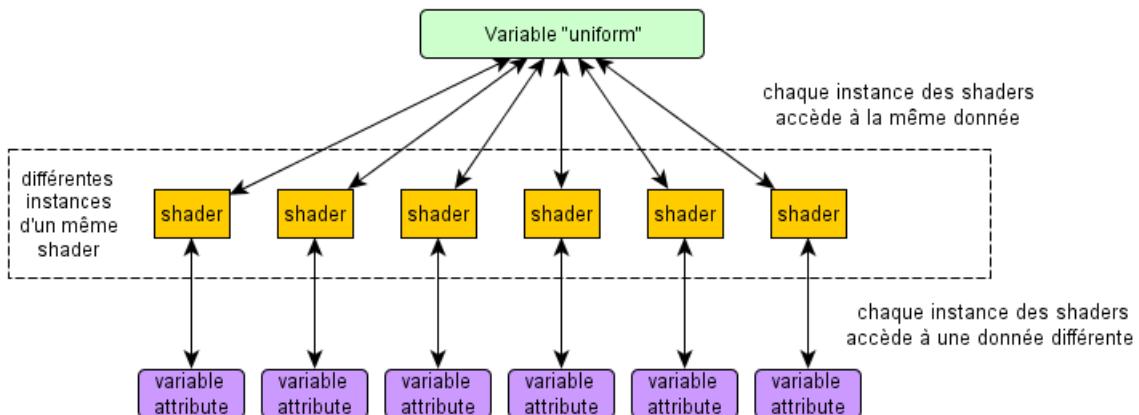
Les shaders sont maintenant initialisés, la matrice de projection est prête, il ne reste plus qu'à envoyer les données de la scène 3D au processeur graphique. Toujours situé dans la fonction **paintGL**, le code du rendu sera réalisé en utilisant la technique Vertex Buffer Object couplée à un tableau d'indices, comme présenté dans le chapitre précédent. Par défaut, OpenGL utilise le pipeline fixe (attention : OpenGL ES n'a pas de pipeline fixe et il est obligatoire de fournir un **QGLShaderProgram**), il faut donc dans un premier temps activer le **QGLShaderProgram**. Pour cela, on utilise la méthode **bind**. Il est conseillé de restaurer le contexte de rendu par défaut lorsque le rendu est fini avec la méthode **release**.

```

m_program.bind();
// utilisation du Program Shader
m_program.release();

```

Comme indiqué précédemment, il existe deux types de paramètres que l'on peut passer aux shaders : les Uniform, qui sont constants pour tous les vertex (par exemple la matrice de projection ou la couleur ambiante) et les Attribute, dont la valeur est différente pour chaque vertex. Qt fournit des fonctions pour chaque type : **setUniformValue** et **setAttributeValue** et leurs dérivées : les versions Array (pour les Uniform et les Attribute), qui permettent d'envoyer des tableaux de données, et la version Buffer (pour les Attribute uniquement), qui permet de lier un **QGLBuffer**.



Les paramètres sont identifiés dans le code GLSL par leur nom. Après compilation, le Shader program conserve un identifiant pour chaque variable du GLSL. Il existe trois méthodes pour passer des paramètres aux shaders : identifier la variable directement à partir de son nom, récupérer l'identifiant d'une variable puis utiliser cet identifiant ou imposer un identifiant constant au Shader Program (pour les Attribute uniquement).

```

// identifier la variable directement à partir de son nom
m_program.setUniformValue("param", value);

// récupération de l'identifiant
const int param_location = m_program.uniformLocation("param");
m_program.setUniformValue(param_location, value);

// imposer l'identifiant

```

```
const int param_location = 0;
m_program.bindAttributeLocation("param", param_location);
m_program.setAttributeValue(param_location, value);
```

On peut envoyer les données aux shaders à différents moments, à partir de l'instant où le programme est compilé : lors de l'initialisation (dans **initializeGL**) ou à chaque mise à jour (dans **paintGL**). Pour des raisons de performances, on enverra un maximum de paramètres lors de l'initialisation.

Pour simplifier les idées, on pourra adopter la démarche suivante : lors de l'initialisation, on récupère les identifiants des paramètres utilisés dans **paintGL**, par exemple la matrice de projection, qui sera modifiée par l'utilisateur pour se déplacer dans la vue 3D, ou les Vertex Buffer Object :

```
void HeightmapWidget::initializeGL()
{
    ...
    m_matrix_location = m_program.uniformLocation("matrixPmv");
    m_vertices_attribute = m_program.attributeLocation("vertex");
}
```

Lors de l'initialisation, on envoie également les paramètres constants en utilisant directement le nom des variables, par exemple la couleur ambiante et les buffers :

```
m_program.setUniformValue("fixed_color", QColor(Qt::red));
m_vertexbuffer.bind();
m_program.setAttributeBuffer(m_vertices_attribute, GL_FLOAT, 0, 3);
m_vertexbuffer.release();
...
}
```

Dans la fonction **paintGL()**, on envoie les variables en utilisant leur identifiant :

```
void HeightmapWidget::paintGL()
{
    ...
    m_program.setUniformValue(m_matrix_location, projection * view * model);
```

L'utilisation des Vertex Buffer Object est similaire à la méthode présentée dans les chapitres précédents. La fonction **glVertexPointer**, est remplacée par la fonction Qt **setAttributeBuffer**. Cette fonction prend simplement l'identifiant du buffer à la place d'un pointeur constant vers les données. Il faut également préciser à OpenGL que l'on utilise un buffer comme variable avec la fonction **enableVertexAttribArray**.

```
vertex_buffer.bind();
m_program.enableVertexAttribArray(m_vertices_attribute);
m_program.setAttributeBuffer(m_vertices_attribute, GL_FLOAT, 0, 3);
vertex_buffer.release();
```

Le rendu du terrain est réalisé avec la fonction **glDrawElements** étant donné que nous utilisons un tableau d'indices :

```
index_buffer.bind();
glDrawElements(GL_TRIANGLES, m_indices.size(), GL_UNSIGNED_INT, NULL);
index_buffer.release();
```

Pour terminer, une bonne pratique consiste à libérer les buffers précédemment utilisés.

```
} m_program.disableVertexAttribArray(PROGRAM_VERTEX_ATTRIBUTE);
```

Les performances obtenues sont similaires à celles de la version utilisant le pipeline fixe.

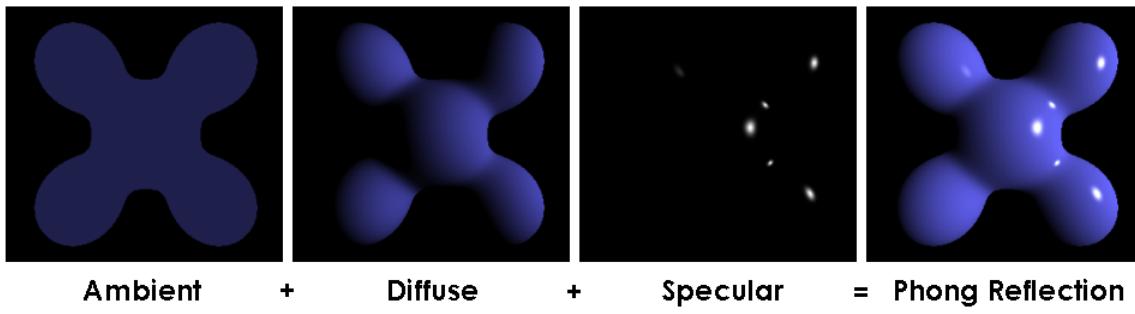
5 - Ajouter des lumières et des textures

Les shaders sont un formidable outil mis à disposition des développeurs pour personnaliser le rendu 3D. Le but de ce tutoriel n'est pas d'entrer en détail dans les techniques les plus avancées de rendu 3D mais d'illustrer l'utilisation des shaders. Nous allons présenter deux exemples simples de shaders : un **modèle d'illumination de Phong** et l'utilisation de textures. Ces exemples, bien que basiques au regard de la puissance des shaders, sont un bon moyen de se familiariser avec certains concepts courants en programmation 3D et avec l'utilisation des shaders. Le lecteur désirant approfondir ce domaine se reportera aux références données.

5-A - Le modèle de Phong

Le modèle d'illumination de Phong est un modèle empirique décomposant la lumière en trois composantes :

- la *composante ambiante* : correspond à l'éclairage ambiant de la scène, provenant de la réflexion multiple ; elle est constante pour chaque vertex ;
- la *composante diffuse* : correspond à la lumière incidente sur l'objet 3D et partant dans toutes les directions ; elle ne dépend que de l'angle d'incidence entre le vecteur lumière et le vecteur normal ;
- la *composante spéculaire* : correspond à la lumière réfléchie sur l'objet 3D ; elle dépend de l'angle d'incidence et de l'angle de réflexion entre le vecteur normal et le vecteur observateur.



(source : Wikimedia Commons)

Le détail des calculs mathématiques ne sera pas donné ici. Nous allons par contre expliquer à quoi correspondent les différents vecteurs utilisés. Pour chaque vertex, trois points dans l'espace 3D sont pris en compte : la position de l'observateur et la position de la lumière, qui sont constantes pour tous les vertices, et la position de chaque vertex. Le vecteur observateur correspond au vecteur allant du vertex à l'observateur. Le vecteur lumière correspond au vecteur allant de la lumière au vertex. Le vecteur normal correspond au vecteur partant du vertex et perpendiculaire à la surface. Le vecteur réfléchi correspond au vecteur symétrique du vecteur lumière par rapport au vecteur normal. Tous ces vecteurs doivent être normalisés avant utilisation. Le langage GLSL fournit la fonction normalize() dans ce but.

Il faut donc fournir de nombreuses informations aux shaders pour ce modèle d'illumination : les couleurs des lumières (ambiante, diffuse et spéculaire) et des matériaux, la position des lumières, la position de l'observateur et les vecteurs normaux à la surface pour chaque vertex. Pour les vecteurs normaux, nous utiliserons une Normal Map, c'est-à-dire une image pour laquelle chaque composante de la couleur (rouge, vert, bleu) correspond aux composantes du vecteur normal (x, y, z), chaque pixel correspondant à un vertex. Dans la troisième partie de ce tutoriel, sur le calcul GPGPU, nous présenterons en détail le calcul de vecteurs normaux et la génération de cette Normal Map.

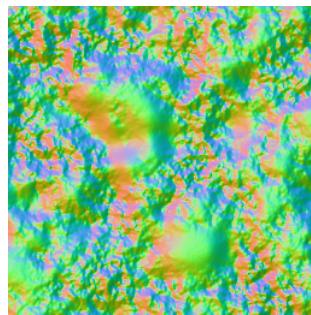
5-B - Chargement de la Normal Map

Chaque coordonnée étant stockée dans une image au format RGB donc normalisé entre 0 et 255, il suffit de normaliser ces coordonnées entre -1 et 1 pour avoir les coordonnées en 3D. Le code ne présente pas de difficulté particulière et est très similaire au code de chargement des vertices.

```
vertex_shader.gl
img = QImage(":/normals.png");
```

```
vertex_shader.gl
m_normals.reserve(m_indices.size());
for(int z = 0; z < vertices_by_z; ++z)
{
    for(int x = 0; x < vertices_by_x; ++x)
    {
        QVector3D normal;
        QRgb rgb = img.pixel(x, z);
        normal.setX(1.0 * qRed(rgb) / 125.0 - 1.0);
        normal.setY(1.0 * qGreen(rgb) / 125.0 - 1.0);
        normal.setZ(1.0 * qBlue(rgb) / 125.0 - 1.0);
        normal.normalize();
        m_normals.push_back(normal);
    }
}
```

La génération de la Normal Map utilisée sera présentée dans la partie GPGPU de ce tutoriel. Voici à quoi elle ressemble :



5-C - Afficher les lumières avec les shaders

Pour simplifier, nous n'allons présenter que le code des composantes ambiante et diffuse du modèle de Phong. La composante spéculaire permet d'ajouter un effet de reflet à la surface, ce qui est surtout intéressant pour des matériaux tels que l'eau ou le métal.

Pour calculer l'ombrage, il faut fournir au shader les informations suivantes : le vecteur normal pour chaque vertex et la direction de la lumière. Dans le Vertex Shader, on détermine le coefficient d'atténuation en calculant le produit scalaire entre le vecteur normal et le vecteur correspondant à la direction de la lumière avec la fonction **dot**. Les valeurs négatives du produit scalaire correspondent au cas où la lumière éclaire la face postérieure d'un triangle. On normalise donc entre 0 et 1 avec la fonction **max**. Le résultat est envoyé au Fragment Shader via la variable **color_factor** :

```
vertex_shader.gl
#version 130

in vec4 normal;
out float color_factor;
uniform vec4 light_direction;

void main(void)
{
    color_factor = max(dot(normal, light_direction), 0.0);
}
```

Dans le Fragment Shader, on récupère la variable **color_factor**. La couleur finale est le produit de la couleur ambiante et du coefficient d'atténuation. On affecte le résultat à la variable de sortie **color**.

```
fragment_shader.gl
#version 130

in float color_factor;
```

fragment_shader.gl

```

out vec4 color;
uniform vec4 ambiant_color;

void main(void)
{
    color = color_factor * ambiant_color;
}

```

Du côté des shaders, il ne reste plus qu'à calculer la position du vertex dans le repère de la caméra. Pour cela, il suffit de calculer le produit de la matrice de projection et la position du vertex et d'affecter le résultat dans la variable *build-in* `gl_Position`.

vertex_shader.gl

```

in vec4 vertex;
uniform mat4 matrixpmv;

void main(void)
{
    ...
    gl_Position = matrixpmv * vertex;
}

```

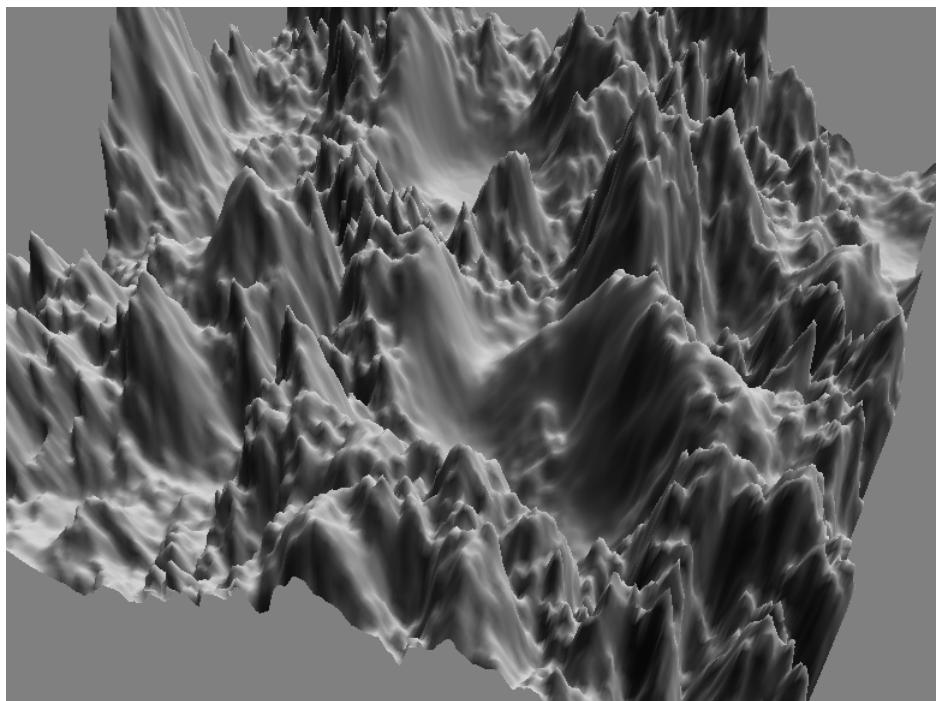
Du côté de l'application, il nous faut calculer les données et les envoyer au GPU. La méthode des Vertex Buffer Object est utilisée pour transmettre les positions des vertices et les vecteurs normaux. Les paramètres constants (direction de la lumière, la couleur ambiante, la matrice de transformation) sont envoyés comme Uniform :

```

m_program.setUniformValue("ambiant_color", QVector4D(0.4, 0.4, 0.4, 1.0));
m_program.setUniformValue("light_position", QVector4D(1.0, 1.0, 1.0, 1.0));
m_program.setUniformValue("matrixpmv", projection * view * model);

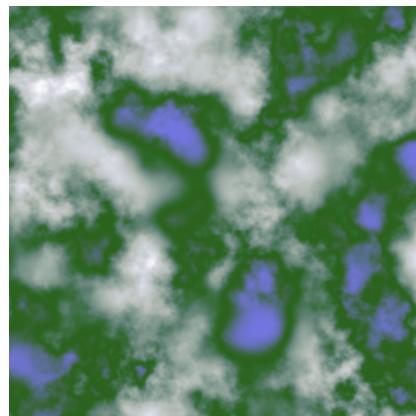
```

Voici le résultat de la heightmap obtenu avec les composantes ambiante et diffuse :



5-D - Appliquer une texture

L'ajout d'une texture sur notre exemple de heightmap nécessite de charger et lier la texture au contexte OpenGL, de définir pour chaque vertex une coordonnée dans le repère de la texture puis de modifier les shaders pour récupérer cette coordonnée et lire la couleur correspondante dans la texture. Voici la texture utilisée :



Pour lier une texture à un contexte OpenGL, on utilise habituellement les fonctions OpenGL **glGenTextures**, **glBindTexture** et **glTexImage2D** (voir le [tutoriel d'introduction à OpenGL et Qt](#) pour le détail de la méthode). Ici, nous allons utiliser une autre méthode. En effet, Qt fournit différentes fonctions pour manipuler les textures, en particulier une fonction pour lire directement un fichier image et le charger dans un contexte OpenGL : **bindTexture()**. Cette fonction retourne un identifiant de type GLuint pour cette texture, qui sera utilisé lors de l'affichage.

```
initializeGL()
{
    m_texture_location = bindTexture(":/texture.png");
}
```

Dans la fonction de rendu, un appel à la fonction OpenGL **glBindTexture** permet d'activer la texture à utiliser :

```
paintGL()
{
    glBindTexture(GL_TEXTURE_2D, m_texture_location);
}
```

Pour appliquer une texture, OpenGL a également besoin de pouvoir faire la correspondance entre un vertex et un point sur la texture. Pour cela, il faut fournir un tableau contenant les coordonnées (x, y) pour chaque vertex. Nous utiliserons un tableau de **QVector2D** pour les stocker :

```
QVector<Qvector2D> m_textures;
```

Les coordonnées de texture sont normalisées entre 0 et 1, c'est-à-dire que le coin en haut à gauche de la texture correspond au point (0, 0) et le point en bas à droite correspond au point (1, 1). Dans notre exemple de heightmap, le calcul des coordonnées de texture est relativement simple : chaque pixel de l'image de la texture correspond à un vertex, on modifie donc simplement la boucle de calcul des positions des vertices :

```
void HeightmapWidget::initializeGL()
{
    QVector2D coordonnees;
    for(int z = 0; y < vertices_by_z; ++z)
    {
        for(int x = 0; x < vertices_by_x; ++x)
        {
            // calcul de la position des vertices
            coordonnees.setX(1.0 * x / quads_by_x);
            coordonnees.setY(1.0 - 1.0 * z / quads_by_z);
            ...
        }
    }
}
```

```
    m_textures.push_back(coordonnees);
}
}
```

Pour passer ces coordonnées au processeur graphique, on utilise encore un Vertex Buffer Object chargé au moment de l'initialisation.

```
m_texturebuffer.create();
m_texturebuffer.bind();
m_texturebuffer.allocate(m_textures.constData(), sizeof(QVector2D) * m_textures.size());
m_texturebuffer.release();
```

Dans le Vertex Shader, on prend en entrée les coordonnées de texture puis on récupère la couleur correspondante dans la texture avec la fonction **texture**. La texture elle-même est passée comme paramètre de type sampler2D.

vertex_shader.gl

```
in vec2 texture_coordonnees;
out vec4 texture_color;
...
void main(void)
{
    texture_color = texture(texture2d, texture_coordonnees.st);
}
```

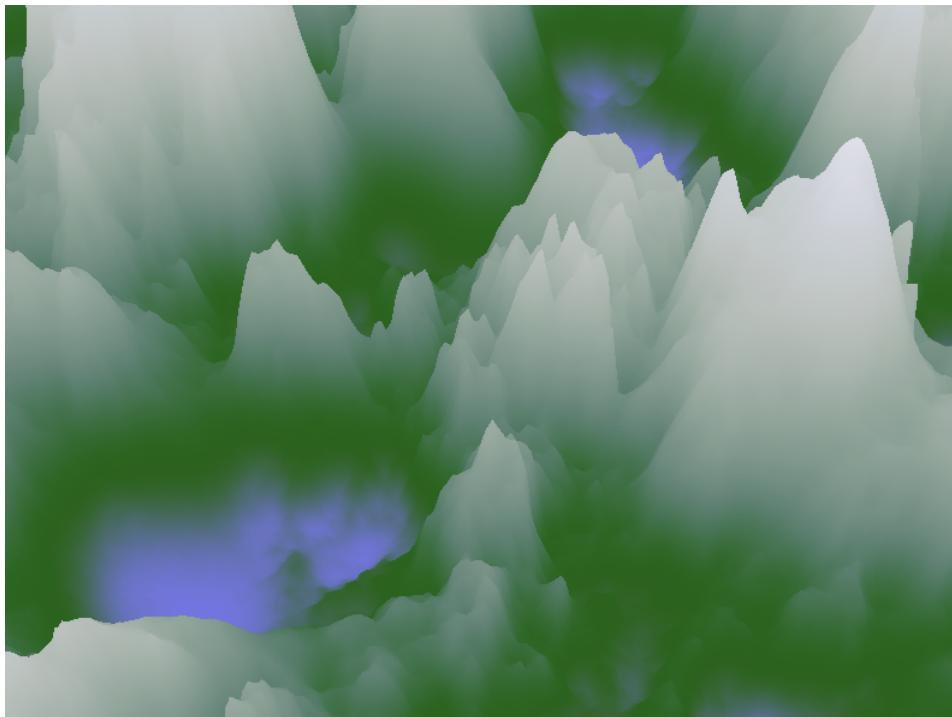
Le Fragment Shader reçoit en entrée la couleur interpolée à partir des couleurs de la texture envoyées par le Vertex Shader.

fragment_shader.gl

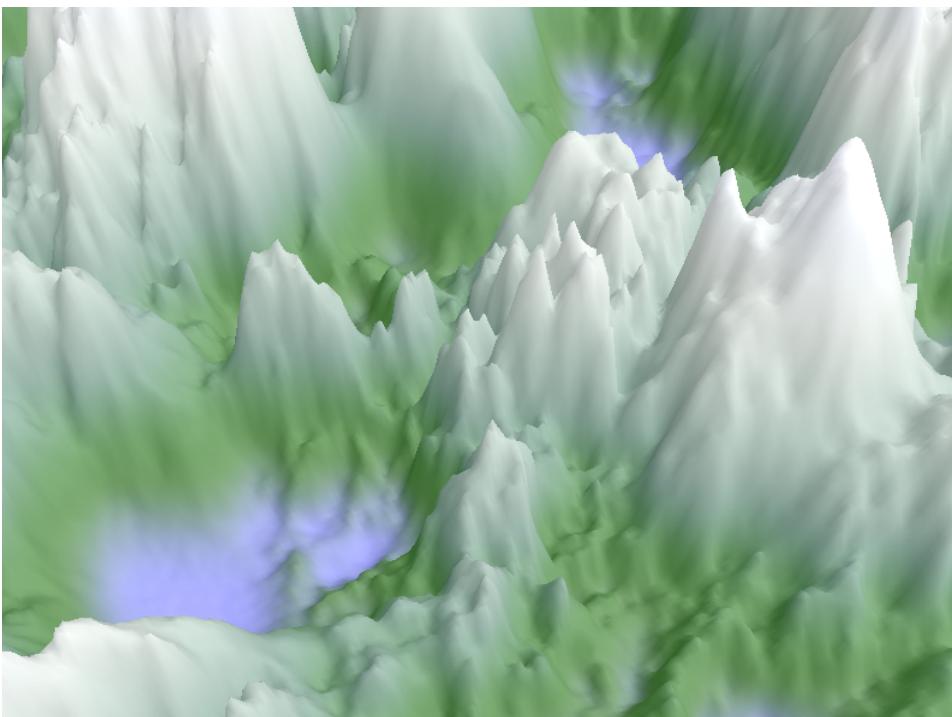
```
#version 130
in vec4 texture_color;
out vec4 color;

void main(void)
{
    color = color_ambiant * texture_color;
}
```

Voici le rendu généré avec la texture choisie :

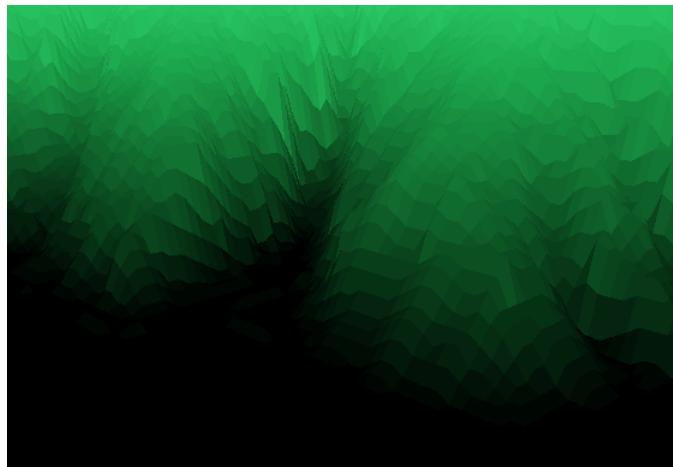


En appliquant la texture et les ombres, on obtient un effet 3D intéressant :

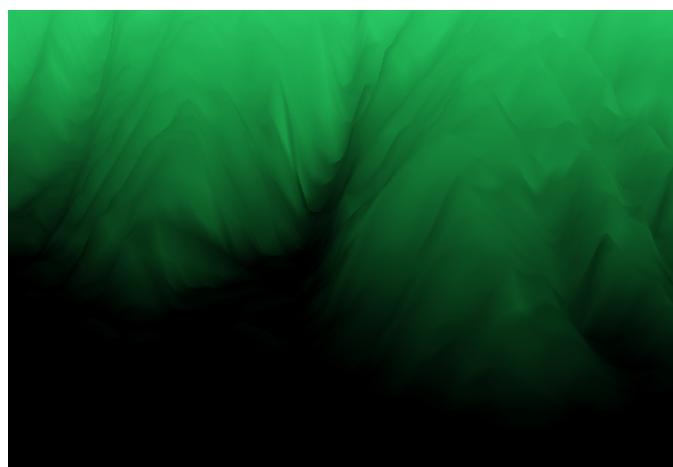


5-E - Calcul dans le Vertex Shader vs dans le Fragment Shader

En fait, pour calculer la couleur de la texture pour un vertex, on a deux possibilités : soit on transmet les coordonnées de la texture entre le Vertex Shader et le Fragment Shader et on calcule dans ce dernier la couleur correspondante de la texture, soit on calcule la couleur de la texture dans le Vertex Shader et on transmet la couleur entre le Vertex Shader et le Fragment Shader. La première version donne le rendu suivant (détail après agrandissement) :



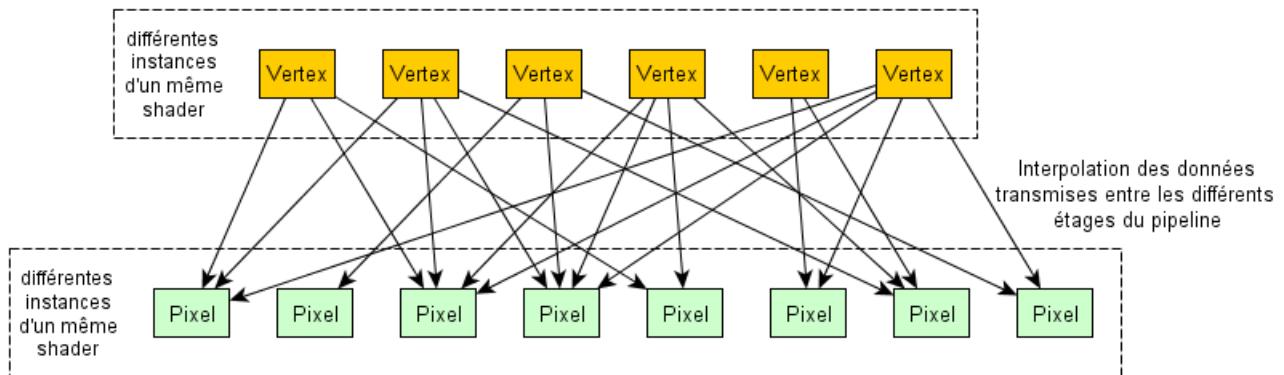
La seconde version donne le rendu suivant (même détail affiché) :



On observe que le résultat n'est pas identique. Il faut bien comprendre ce qui se passe lorsque l'on transmet des données entre le Vertex Shader et le Fragment Shader pour comprendre le rendu obtenu.

Pour chaque vertex que l'on affiche, le GPU crée une instance du Vertex Shader. Dans notre exemple, cela veut dire que l'on a 955206 vertices et donc 955206 threads pour le Vertex Shader. Le Fragment Shader est instancié pour chaque pixel de la fenêtre de rendu. Par exemple, pour une fenêtre de rendu de 800 par 600, on aura donc 480000 pixels et 480000 threads pour le Fragment Shader.

On comprend alors aisément que les données ne sont pas envoyées directement entre les shaders. Elles sont en fait interpolées : la valeur d'une variable `in` du Fragment Shader est en fait la combinaison de la variable `out` correspondante, provenant de plusieurs instances différentes du Vertex Shader. Le passage d'une primitive définie par des vertex à un ensemble de pixels visibles à l'écran est effectué par le moteur de rastérisation.



En fonction du traitement que l'on souhaite faire, le rendu sera meilleur s'il est fait dans le Vertex Shader ou dans le Fragment Shader. De même, les performances obtenues seront différentes selon le shader dans lequel on fait les calculs. Il sera parfois nécessaire de faire un compromis entre qualité du rendu et performances.

6 - Réaliser un rendu off-screen

OpenGL dispose d'un mécanisme permettant de rediriger le rendu d'une scène 3D dans un buffer au lieu de l'afficher directement à l'écran. L'intérêt principal est de pouvoir profiter de l'accélération matérielle pour générer une image (ou pour récupérer le résultat d'un calcul effectué sur un processeur graphique sous la forme d'une image).

Nous allons présenter les pBuffer et les Frame Buffer Object (FBO) à travers un exemple simple : nous allons dessiner dans le Fragment Shader des cercles concentriques de différentes couleurs.

6-A - Utilisation de QGLPixelBuffer

La classe **QGLPixelBuffer** permet de manipuler les pBuffer avec Qt.

```
QGLPixelBuffer m_pbuffer;
```

L'initialisation prend en paramètre la taille du buffer et le format utilisé. Le format correspond au format d'affichage de contexte OpenGL (il permet notamment l'activation du double buffering). La fonction **format** permet de récupérer le **QGLFormat** du contexte openGL courant.

```
heightmapWidget::heightmapWidget(QWidget *parent) :
    m_pbuffer(QSize(512, 512), format(), this)
{
```

Cependant, dans notre cas, nous souhaitons créer un Pixel Buffer Object avec une taille identique au Frame Buffer Object, pour pouvoir comparer la différence de performance. Pour cela, on va créer Pixel Buffer Object sur le tas :

```
// QGLPixelBuffer* m_pbuffer;
m_pbuffer = new QGLPixelBuffer(QSize(512, 512), format(), this)
{
```

Nous allons maintenant générer une nouvelle texture dans laquelle sera stockée l'image produite. L'appel à la méthode **generateDynamicTexture** permet de générer une texture OpenGL de même taille que celle du pBuffer. L'identifiant de la texture est récupéré sous forme d'entier non signé dans la variable **m_pbuffer_location**.

La méthode **bindToDynamicTexture** permet de lier le contenu du pBuffer à une texture : dès que le pBuffer sera modifié, la texture liée sera mise à jour automatiquement. Malheureusement, cette fonctionnalité n'est pas disponible sur toutes les plateformes, notamment Linux (serveur X11). Cette fonction retourne un boolean indiquant si cette fonctionnalité est prise en charge par le système. Si ce n'est pas le cas, il faudra effectuer manuellement la mise à jour de la texture après modification.

```
m_pbuffer_location = m_pbuffer->generateDynamicTexture();
has_pbuffer = m_pbuffer->bindToDynamicTexture(m_pbuffer_location);
```

La génération de la texture sera réalisée dans un Fragment Shader. Nous créons donc un Program Shader chargé d'appliquer ce Fragment Shader sur chaque pixel de notre texture de rendu. Nous passons les dimensions de la texture pour permettre de centrer notre rendu.

```
m_program_pbuffer.addShaderFromSourceFile(QGLShader::Fragment,(":/pbuffer.glsl");
m_program_pbuffer.link();
m_program_pbuffer.bind();
m_program_pbuffer.setUniformValue("pbuffer_size", QVector2D(m_pbuffer->width(), m_pbuffer-
>height()));
m_program_pbuffer.release();
```

Le code du shader n'introduit pas de nouveau concept, il produit simplement des cercles concentriques à l'aide la fonction GLSL **sin** avec une couleur dépendant de la position du pixel par rapport au centre.

pbuffer.glsl

```
#version 130

uniform vec2 pbuffer_size;
out vec4 color;

void main(void)
{
    vec2 FragCoord = vec2(gl_FragCoord.x / pbuffer_size.x - 0.5, gl_FragCoord.y / pbuffer_size.y - 0.5);
    float radius = sqrt(FragCoord.x * FragCoord.x + FragCoord.y * FragCoord.y);
    float gray = sin(radius * 200.0);

    color = vec4(FragCoord.x * gray, FragCoord.y * gray, 0.0, 1.0);
}
```

Il ne reste plus qu'à modifier la fonction de rendu **paintGL** pour ajouter le rendu off-screen avant le code du rendu à l'écran. L'appel à la méthode **makeCurrent** permet d'utiliser le contexte OpenGL du pBuffer : l'image générée par notre Fragment Shader sera donc stockée dans le pBuffer et non à l'écran.

```
void HeightmapWidget::paintGL()
{
    m_pbuffer->makeCurrent();
    m_program_pbuffer.bind();
```

Nous traçons simplement un carré dans lequel sera dessinée la texture. Pour des raisons de simplicité, nous utilisons ici les primitives **glVertex**. Dans le cas où la liaison dynamique de texture n'est pas supportée, nous recopions manuellement le contenu du Vertex Buffer Object dans la texture avec la méthode **updateDynamixTexture**.

```
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_QUADS);
    glVertex2f(-1.0, -1.0);
    glVertex2f(-1.0, 1.0);
    glVertex2f(1.0, 1.0);
    glVertex2f(1.0, -1.0);
glEnd();

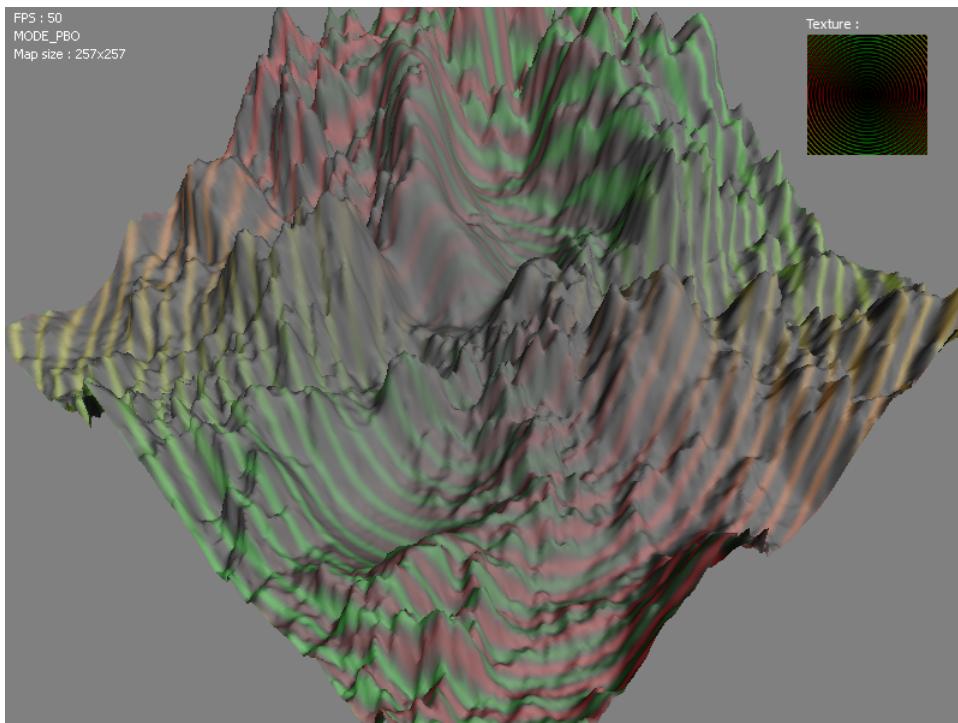
if (!has_pbuffer)
    m_pbuffer->updateDynamicTexture(m_pbuffer_location);

m_program_pbuffer.release();
```

On réactive le contexte OpenGL du widget avec **makeCurrent**. La texture est ensuite liée au contexte à l'aide la fonction OpenGL **glBindTexture**.

```
makeCurrent();
glBindTexture(GL_TEXTURE_2D, m_pbuffer_location);
```

Voici le rendu final de notre exemple :



6-B - Utilisation de QGLFrameBufferObject

Les Frame Buffer Object permettent également de réaliser du rendu off-screen et de générer des images à l'aide d'OpenGL. De plus, ils présentent un certain nombre d'avantages par rapport aux Pixel Buffer Object, par exemple :

- des performances légèrement plus élevées en général, dues à l'absence de changement de contexte OpenGL ;
- les Frame Buffer Object sont une extension OpenGL pure, ne dépendant pas de bibliothèque spécifique au système utilisé, ce qui les rend plus portables que les Pixel Buffer Object.

Les Frame Buffer Object sont gérés dans Qt à l'aide de la classe **QGLFrameBufferObject**. À la différence des Pixel Buffer Object, il est nécessaire d'avoir un contexte OpenGL actif au moment de l'initialisation. Il n'est donc pas possible de les initialiser dans la liste d'initialisation dans le constructeur :

```
class HeightmapWidget : public QGLWidget
{
private:
    QGLFramebufferObject m_fbo;
public:
    HeightmapWidget(QWidget *parent) :
        QGLWidget(parent),
        m_fbo(512,512) // erreur : pas de contexte OpenGL valide
    {
    }
};
```

De plus, l'opérateur d'affectation étant privé, nous ne pouvons pas créer notre objet **QGLFrameBufferObject** dans la pile :

```
class HeightmapWidget : public QGLWidget
{
private:
    QGLFramebufferObject m_fbo;
public:
    HeightmapWidget(QWidget *parent) :
```

```

        QGLWidget(parent)
    {
        makeCurrent(); // OK : un contexte OpenGL est actif
        m_fbo = QGLFramebufferObject(512, 512); // erreur : l'affectation est impossible
    }
};

```

Il est donc indispensable de créer notre objet dans le tas sous forme de pointeur. On prendra les précautions nécessaires pour une gestion correcte de la mémoire, en n'oubliant pas d'appeler `delete` dans le destructeur.

```

class HeightmapWidget : public QGLWidget
{
private:
    QGLFramebufferObject* m_fbo;
public:
    HeightmapWidget(QWidget *parent) :
        QGLWidget(parent)
    {
        makeCurrent(); // OK : un contexte OpenGL est actif
        m_fbo = new QGLFramebufferObject(512, 512); // OK : on affecte un pointeur
    }
    ~HeightmapWidget()
    {
        if (m_fbo) delete m_fbo;
    }
};

```

Le Program Shader étant similaire à celui du chapitre précédent (les couleurs sont changées, pour voir la différence avec le Pixel Buffer Object), nous ne détaillerons que le code du rendu. L'utilisation des Frame Buffer Object ne requiert pas de changement de contexte OpenGL : nous appelons directement les méthodes **bind** et **release** pour activer le rendu off-screen.

```

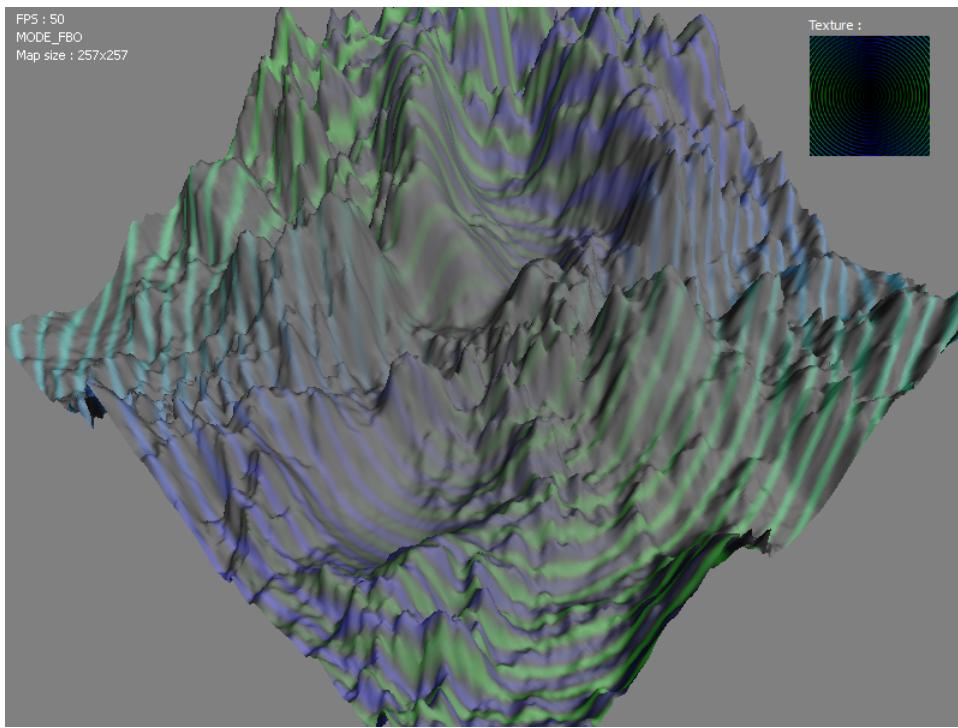
void HeightmapWidget::paintGL()
{
    m_program_fbo.bind();
    m_fbo->bind();
    ...
    m_fbo->release();
    m_program_fbo.release();
}

```

La texture générée est ensuite liée au contexte OpenGL pour l'affichage du terrain :

```
glBindTexture(GL_TEXTURE_2D, m_fbo>texture());
```

Le résultat final :

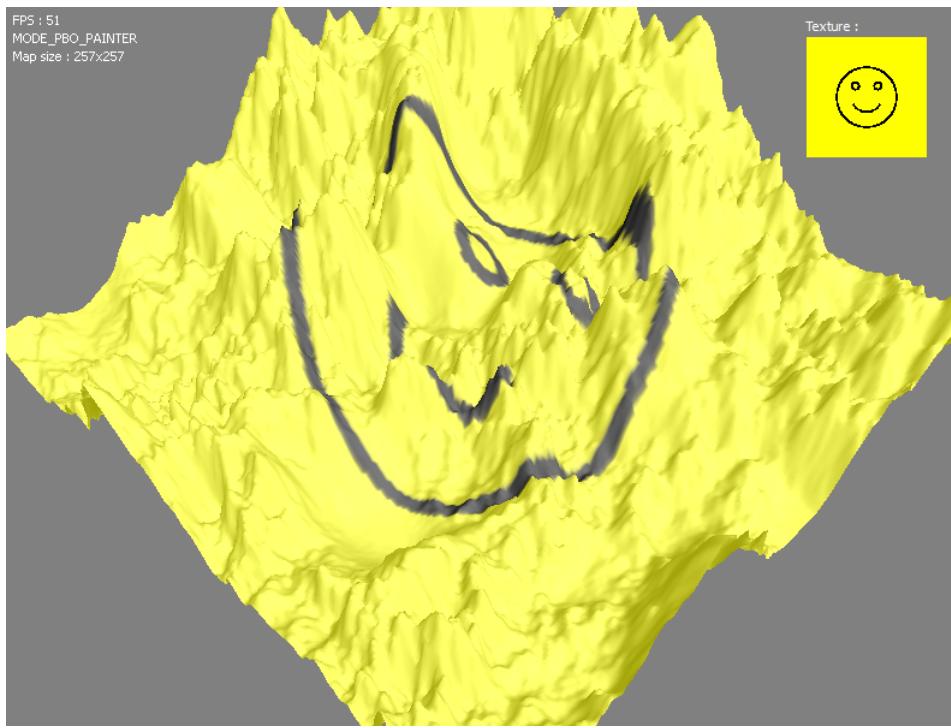


6-C - Dessiner dans une texture dynamique avec QPainter

Les classes **QGLPixelBuffer** et **QGLFramebufferObject** héritent de la classe **QPaintDevice**, il est donc possible de dessiner dedans directement avec **QPainter**.

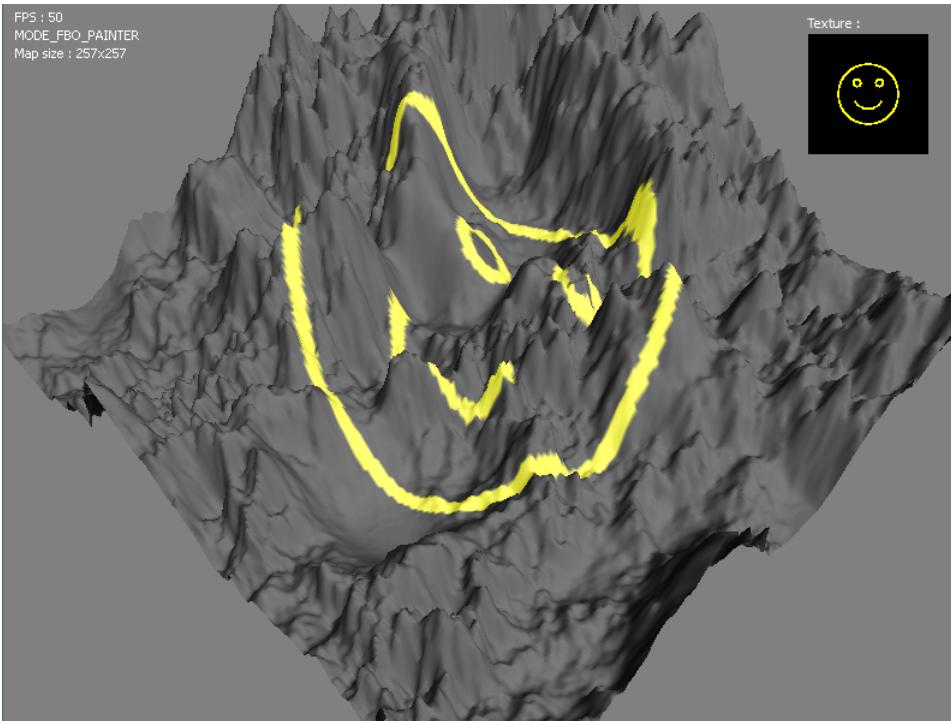
Pour cela, on crée un QPainter avec le **QGLPixelBuffer** comme paramètre et on peut l'utiliser dans la foulée. Lorsque l'on a fini d'utiliser **QPainter**, on réactive le contexte du **QGLWidget** et on active la texture dynamique avec la fonction **glBindTexture** :

```
QPainter pbuffer_painter(m_pbuffer);
// On dessine avec le QPainter
pbuffer_painter.end();
makeCurrent();
glBindTexture(GL_TEXTURE_2D, m_pbuffer_location);
```



Le code est similaire pour le Frame Buffer Object : l'identifiant GLuint de la texture générée par le Frame Buffer Object est obtenu avec la fonction `texture`. De plus, comme `QPainter` modifie le contexte OpenGL courant, il est nécessaire de restaurer les paramètres du contexte après utilisation de `QPainter`. En revanche, il n'est pas nécessaire d'appeler la fonction `makeCurrent` puisque Frame Buffer Object travaille avec le même contexte OpenGL que le widget. Dans notre cas, puisque la plupart des paramètres du contexte sont envoyés directement dans le shader, il faut simplement rétablir les dimensions de la vue :

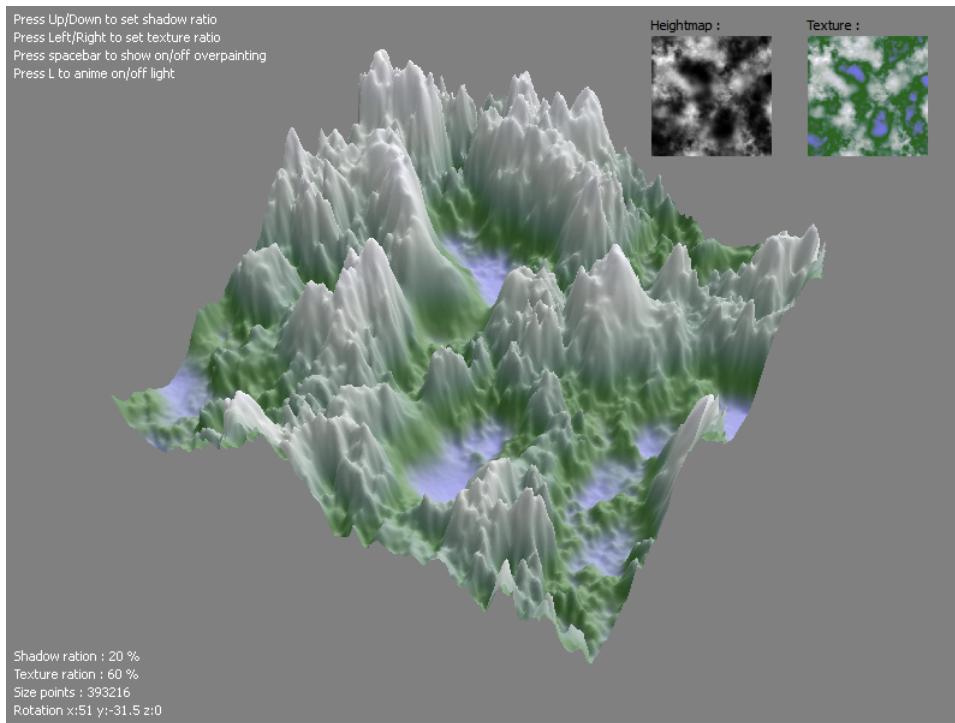
```
glViewport(0, 0, width(), height());
```



Cette technique pourra être utilisée, par exemple, pour afficher un texte défilant sur un panneau lumineux d'une scène représentant une rue ou afficher les images d'une télévision.

7 - Overpainting : dessiner en 2D avec QPainter sur une scène 3D

Dans certaines applications, il peut être utile de dessiner des objets en premier plan d'une fenêtre de rendu OpenGL (par exemple un affichage tête haute pour les jeux vidéo). Dans cette partie, nous allons afficher en premier plan de la vue 3D les images de la heightmap et de la texture utilisées pour la génération du terrain.



Qt dispose d'une classe dédiée au dessin 2D très complète : **QPainter**. Cette classe permet de dessiner toutes sortes de formes, de la simple ligne aux polygones complexes. Elle est aussi capable de dessiner des images à l'aide des fonctions **drawImage** et **drawPixmap**. Dans cet exemple, nous allons utiliser la fonction **drawPixmap** pour afficher les images et la fonction **drawText** pour afficher des informations. Pour rappel, la classe **QPixmap** est optimisée pour l'affichage, contrairement à **QImage** qui est optimisée pour la manipulation de pixels.

Habituellement, pour réaliser un rendu 3D, on n'a pas besoin d'implémenter la fonction **paintEvent**. Celle-ci est appelée automatiquement lors de la mise à jour du rendu et appelle la fonction **paintGL**. Pour réaliser l'overpainting, nous allons devoir modifier cette fonction **paintEvent** et appeler manuellement la fonction **paintGL**. Il faut donc dans un premier temps modifier le code de notre timer pour que celui-ci appelle la fonction **update** à la place de la fonction **updateGL** :

```
connect(&timer, SIGNAL(timeout()), this, SLOT(update()));}
```

Ensuite, il faut spécifier à **QWidget** que l'on prend en charge la mise à jour du background et qu'il n'est pas nécessaire de le faire automatiquement. Pour cela, on ajoute les deux lignes suivantes dans le constructeur :

```
setAttribute(Qt::WA_OpaquePaintEvent);
setAutoFillBackground(false);}
```

Dans la fonction **paintEvent**, on commence par appeler la fonction de rendu 3D OpenGL **paintGL**. Ensuite, on crée un **QPainter** pour dessiner sur le widget courant. Lorsque le widget courant est un **QGLWidget**, **QPainter** utilise automatiquement OpenGL pour faire le rendu 2D. Lors de la création du contexte de **QPainter**, les paramètres 3D sont donc modifiés : il est donc nécessaire de les réinitialiser dans **paintGL** (par exemple d'activer le test de profondeur) :

```
glEnable(GL_DEPTH_TEST);
```

Normalement, il faudrait aussi redonner les matrices de transformation 3D. Dans notre cas, les matrices sont envoyées directement aux shaders, sans passer par les fonctions **glMatrixMode** et autres. Il n'est donc pas nécessaire de les initialiser.

Pour le dessin 2D, on utilise les méthodes **drawText** et **drawPixmap** pour afficher une image et un texte. Pour terminer, il faut appeler la méthode **end** pour indiquer que nous en avons fini avec le dessin 2D, sous peine d'avoir des bogues graphiques.

```
QPainter painter(this);
painter.drawText(width() - 260, 20, "Heightmap :");
painter.drawPixmap(width() - 260, 25, 100, 100, QPixmap(":/heightmaps/secondlife.jpg"));
painter.drawText(width() - 130, 20, "Texture :");
painter.drawPixmap(width() - 130, 25, 100, 100, QPixmap(":/textures/side1.png"));
painter.end();
```

 *Il faut noter que cet exemple n'est pas du tout optimisé. En effet, le chargement des images dans les **QPixmap** avec la fonction **drawPixmap** est effectué à chaque mise à jour du rendu. Pour optimiser, il faudrait charger l'image une seule fois puis la stocker.*

8 - Gestion des extensions avec QGLContext::getProcAddress()

Les fonctionnalités offertes par les cartes graphiques sont très variables en fonction du constructeur et évoluent à un rythme différent d'OpenGL. Pour gérer cette grande diversité et pour faciliter l'intégration de nouvelles fonctionnalités, OpenGL utilise un système d'extensions : chaque nouvelle fonction est ajoutée dans une extension. Pour pouvoir l'utiliser, il faut donc : vérifier que le matériel supporte la fonctionnalité puis charger la fonction. Le lecteur se reportera au tutoriel [Les extensions OpenGL](#) pour avoir des explications détaillées sur la procédure générale à suivre, en particulier sur la syntaxe à utiliser pour déclarer les pointeurs de fonctions.

Il est possible d'utiliser une bibliothèque tierce qui permet de prendre en charge les extensions OpenGL, par exemple [GLEW](#). Dans le cas où l'on souhaite utiliser peu d'extensions et que l'ajout d'une bibliothèque est trop lourd, Qt fournit la fonction [getProcAddress](#) pour récupérer une extension OpenGL :

```
PFNGLGENBUFFERSARBPROC glGenBuffers = (PFNGLGENBUFFERSARBPROC) getProcAddress("glGenBuffersARB");
if (glGenBuffers)
    // la fonction est disponible
else
    // la fonction n'est pas disponible
```

9 - Introduction au GPU computing

! La programmation sur GPU présente des difficultés spécifiques qui nécessite en général des connaissances avancées en programmation parallèle. Son utilisation n'est pas forcement triviale et elle n'est pas intéressante pour tous types de problématique. Une mauvaise utilisation peut aboutir à des performances médiocres, voir plus mauvaises que la version CPU.

On observe régulièrement des personnes qui souhaitent utiliser le calcul sur GPU pour accélérer un algorithme qui est relativement couteux en terme de temps. Or, il apparaît souvent que les problèmes de performances proviennent avant tout d'un problème de choix conceptuel et non d'une limitation des capacités du CPU. Et, il apparaît souvent aussi que le passage au calcul GPU n'améliore pas les performances dans ce cas : le problème se situe bien en amont de l'implémentation et le recourt au GPU ne corrigera pas le problème.

Il est donc nécessaire de bien comprendre les spécificités du GPU Computing, les contraintes en terme de transfert de données, d'accès mémoire, de parallélisation des tâches, etc. pour espérer obtenir une amélioration des performances.

Le calcul sur cartes graphiques (*GPU Computing* ou anciennement *GPGPU* pour *General-Purpose Processing on Graphics Processing Units*) est une technique permettant de réaliser des calculs parallèles à l'aide de la carte graphique. Partant du constat qu'un grand nombre de calculs habituellement réservés au CPU étaient facilement parallélisables, les fabricants de carte graphiques ont modifié l'architecture de leur puces et développé des API permettant aux développeurs de réaliser des calculs génériques sur carte graphique.

Plusieurs API sont nées, la plus connue étant probablement celle de NVIDIA : **CUDA** (pour *Compute Unified Device Architecture*). AMD propose également une API dédiée au GPU computing : ATI Stream SDK. Dans ce tutoriel nous allons utiliser une autre API proposée par le groupe **Khronos** (qui propose également OpenGL) : **OpenCL** (pour *Open Computing Language*). Le travail du groupe Khronos consiste à proposer une API standard pour OpenCL, multiplateforme et open-source, qui sera ensuite implémentée par les différents constructeurs de puces graphiques sur différents types de processeurs parallèles (CPU multicores, GPU, Digital Signal Processor, Cell, etc). Cette API dérive du C (version C89) mais les spécifications pour le portage C++ sont données dans OpenCL 1.1. Il est également possible de regarder les sources de la version C++ pour voir comment est pris en charge la version C.

OpenCL définit un langage équivalent au C permettant d'exécuter du code sur des périphériques de calcul (CPU, GPU, accélérateur, etc.) Le code à exécuter est envoyé au runtime OpenCL sous forme de chaîne de caractères char*. Ce code est donc compilé à l'exécution (contrairement à CUDA par exemple, qui est compilé avant l'exécution). Cela permet de gagner en flexibilité et en portabilité mais implique de prendre les précautions nécessaires pour que le programme reste performant et robuste sur différentes plateformes. OpenCL accepte les modèles de parallélisation des données et des tâches.

Pour présenter les notions de base du GPU Computing et d'OpenCL, nous allons introduire un nouvel exemple : l'addition de deux vecteurs, qui sera repris tout au long de cette partie sur le GPU Computing.

9-A - Le calcul parallèle

Observons le travail effectué par le processeur lorsque l'on additionne deux vecteurs. La méthode consiste à parcourir les deux tableaux, d'ajouter deux éléments entre eux puis de mettre le résultat dans un tableau résultat. Le code C++ est relativement simple :

```
#include <vector>

int main(int, char**)
{
    // declarations
    const int N = 100000;
    std::vector<float> v1(N);
```

```

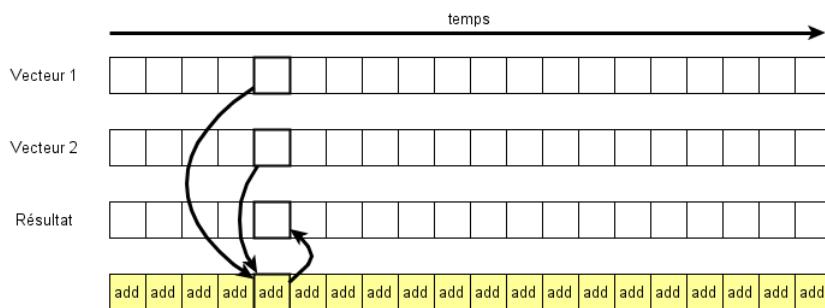
std::vector<float> v2(N);
std::vector<float> result(N);

// initialisation du contenu des vecteurs

// addition
for (int i=0; i<N; ++i)
    result[i] = v1[i] + v2[i];

// affichage du résultat
}
    
```

L'algorithme est résumé sur la figure suivante :

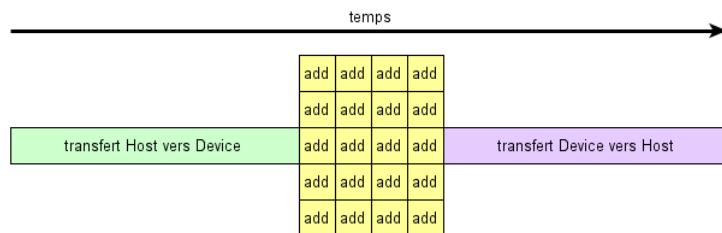


Cet algorithme présente une particularité très intéressante : l'addition de deux éléments pour un indice i donné est indépendante des autres additions. En d'autres termes, lorsque l'on réalise l'addition $result[1] = v1[1] + v2[1]$, on n'a pas besoin d'attendre que l'addition $result[0] = v1[0] + v2[0]$ soit terminée. Les données et les calculs peuvent être découpés en sous-partie qui peuvent être réalisées indépendamment les unes des autres.

Cette particularité va être intéressante lorsque l'on utilise un processeur multicore. Par exemple, avec un processeur dual-cores, un premier core va prendre en charge une partie des données à traiter (par exemple les éléments du tableau avec les indices pairs ; ou les $N/2$ premiers éléments), l'autre core prendra en charge l'autre partie des données (par exemple les éléments d'indice impairs ou les $N/2$ derniers éléments). Le travail de ces deux cores sera totalement indépendant et donc le temps total pour réaliser l'addition des deux vecteurs prendra deux fois moins de temps qu'en utilisant qu'un seul core. De même, si on utilise un processeur quad-cores, le temps de travail sera divisé par quatre. Le terme "parallélisme" vient donc du fait que l'on travaillera avec des tâches s'exécutant en même temps, indépendamment les unes des autres.

Dans le domaine de la 3D, de nombreux calculs présentent cette particularité d'être indépendants. Par exemple, la transformation d'un point par une matrice dans le Vertex Shader ou le traitement de chaque pixel dans le Fragment Shader. Les GPU se sont donc progressivement adaptés pour profiter au maximum des possibilités de parallélisation et proposent maintenant plusieurs centaines à plusieurs milliers de core pouvant traiter les données en parallèle. Le temps de traitement des calculs est alors divisé par le nombre de core pouvant fonctionner en parallèle. Et on va justement profiter de cette énorme puissance disponible dans les GPU pour y faire certains calculs lourds qui prendraient beaucoup plus de temps sur le CPU.

A ce niveau, on remarque une contrainte du GPU computing. En plus du temps de calcul, il est nécessaire d'avoir une étape pour transférer les données depuis la RAM vers la carte graphique. Ce temps de transfert passe par les ports AGP ou PCI selon l'ancienneté du système. La bande passante, c'est-à-dire la quantité de données qui peuvent transiter par seconde, est limité par les caractéristiques techniques de ces ports. Malgré des taux actuels de plusieurs Go/s, ces transferts représentent un véritable goulot d'étranglement qui peuvent diminuer fortement les performances, voir même rendre le GPU Computing moins performant que la version CPU.



On a donc deux contraintes importantes à ce niveau avec l'approche GPU Computing : il faut que l'algorithme utilisé soit parallélisable et il faut que le rapport temps de calcul / temps de transfert soit maximum. On comprend donc bien l'importance de bien choisir quelle partie du code optimiser et quel algorithme utiliser. Un algorithme réalisant beaucoup de calculs sur peu de données donnera les performances les plus intéressantes en GPU Computing. On voit également qu'il est important de systématiquement mesurer les performances pour n'optimiser au mieux que ce qui est nécessaire.

i Pourquoi présenter l'utilisation d'OpenCL pour le GPU Computing plutôt que CUDA ou d'autres solutions ? La principale raison est que OpenCL est une API portable, indépendante d'un constructeur en particulier. Il sera possible de garder le même code quelque soit la machine sur laquelle s'exécute le programme. De plus, les utilisateurs habitué à OpenGL se retrouveront dans un environnement familier et il sera plus facile de le prendre en main. Par exemple, les fonctionnalités spécifiques à un constructeur de GPU seront disponibles, comme dans OpenGL, sous forme d'extensions.

9-B - Les API OpenCL

Les API d'OpenCL se divisent en trois :

- Le Platform layer API est appelé depuis le code du programme hôte. Cette API permet de gérer les ressources offertes par OpenCL (rechercher et initialiser les périphériques compatibles OpenCL, créer des contextes de travail, etc.)
- L'OpenCL Language est utilisé pour écrire le code des fonctions exécutées sur les périphériques compatibles OpenCL (kernels). Ce langage est basé sur le C version 99 avec des extensions de langage. Ce langage fournit en particulier un certain nombre de fonctions et variables accessibles dans les kernels (build-in).
- Le Runtime API permet de compiler et d'exécuter les kernels et de gérer la mémoire vidéo. Il prend également en charge la configuration de l'exécution des kernels en créant le nombre de kernels nécessaires et en leur attribuant un index à 1D, 2D ou 3D. Chaque kernel (work item) est identifié de façon unique par un index global et exécute le même code sur des données différentes. Le runtime se charge aussi de regrouper les kernels dans des work group (les work group sont alors identifiés par un index unique et chaque work item dans un work group est identifié par un index unique dans ce work group).

9-C - Installer OpenCL

Pour exécuter un programme contenant du code OpenCL, un seul fichier suffit généralement : libopencl (.dll sous Windows et .so sous Linux). Depuis quelques temps, les constructeurs de GPU incluent ce fichier directement dans les pilotes et il n'est donc pas nécessaire de le fournir. Donc, si la machine sur laquelle vous souhaitez exécuter votre programme est à jour et correctement configurée, vous n'avez pas de fichier supplémentaire à fournir.

Pour compiler un programme contenant du code OpenCL, il faut donc simplement ajouter les fichiers d'en-têtes. Ces fichiers étant spécifiques du constructeur, il est nécessaire de se reporter aux pages de téléchargement des constructeurs et à la documentation fournie (par exemple **NVIDIA** et **AMD**). VS2010 et CUDA 3.2 : <http://www.codeproject.com/Tips/186655/CUDA-3-2-on-VS2010-in-9-steps.aspx>

9-D - Installer QtOpenCL

QtOpenCL est un projet proposé par QtLabs. Vous pouvez télécharger les différentes versions sur [qt-git](#). Les instructions de compilation sont données dans la [documentation](#) de QtOpenCL. La compilation de QtOpenCL nécessite normalement de compiler également Qt. Pour vous faciliter le travail, Developpez vous propose cette bibliothèque à télécharger.

11 - Architecture des GPU

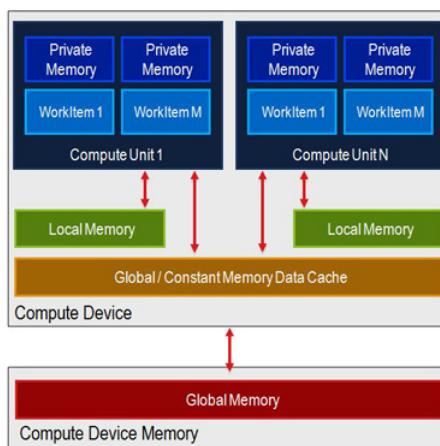
Pour les détails, voir le tutoriel CUDA

11-A - Vue générale

SM, CU, bus, mémoires, etc.

11-B - Les mémoires

Les données sont accessibles dans les kernels uniquement après transfert des données. Plusieurs mémoires sont utilisables dans les kernels : la mémoire globale, la mémoire privée, la mémoire locale et la mémoire constante.



10 - Le platform layer API

Le fonctionnement d'une application utilisant OpenCL peut se résumer en trois étapes de base : le transfert des données entre l'hôte (CPU, qui sera nommé *host* dans le code) et le périphérique (GPU, qui sera nommé *device* dans le code), le calcul sur la carte graphique proprement dit puis le transfert du résultat entre le périphérique et l'hôte. À ces trois étapes, il faut ajouter des étapes d'initialisation et d'affichage. Au final, il faudra donc suivre le processus suivant :

- initialisation des données ;
- initialisation du contexte OpenCL ;
- transfert des données de l'hôte vers le périphérique ;
- calcul proprement dit ;
- transfert des données du périphérique vers l'hôte ;
- affichage du résultat.

10-A - Gérer les erreurs

Pour utiliser les routines OpenCL, il faut inclure dans les en-tête correspondant à la librairie utilisée (à adapter en fonction de l'installation, voir les manuels d'installation spécifique pour chaque implémentation d'OpenCL).

```
cpp
#include <CL/cl.h>
```

```
cpp
#include <CL/cl.hpp>
```

```
cpp
#include <QtOpenCL>
```

La gestion des erreurs dans OpenCL utilise un paramètre de type `cl_int`, soit en paramètre de retour de fonction, soit en passant une variable en référence non constante comme paramètre, selon la fonction utilisée. Par exemple pour les fonctions `clGetPlatforms` et `clCreateContext` (ces fonctions sont expliquées juste après) :

```
cl_int err;
err = clGetPlatformIDs(1, &cpPlatform, NULL); // code d'erreur en paramètre de retour
gpuContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &err); // code d'erreur passé par référence
```

Lorsqu'aucune erreur n'a été trouvée, le code de retour vaut `CL_SUCCESS`. Pour connaître la liste des codes d'erreur et le nom des constantes correspondantes, il est possible de regarder dans le fichier `CL/cl.h`. Pour faciliter le débogage du code, on utilise souvent une fonction qui affichera le code d'erreur uniquement s'il est différent de `CL_SUCCESS` :

```
inline void checkError(const char* msg, cl_int err)
{
    if (err != CL_SUCCESS)
    {
        std::cerr << "ERROR: " << msg << " (" << err << ")" << std::endl;
        exit(EXIT_FAILURE);
    }
}
```

Il est également possible d'utiliser les exceptions dans la version C++ d'OpenCL mais cela sera traité dans un chapitre spécifique.

Avec QtOpenCL, la gestion des erreurs est un peu différentes : les fonctions retourne une variable de type binaire. L'erreur peut être trouvée en utilisant le fonction `log()`.

10-B - La plateforme

Une plateforme OpenCL regroupe un CPU et un ou plusieurs devices et permet de gérer la mémoire et d'exécuter les kernels.

Version C

```
cl_platform_id platforms[5];
size_t size = 0;
err = clGetPlatformIDs(5, platforms, &size);
```

Version C++

```
cl::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);
```

QtOpenCL

```
QList<QCLPlatform> platforms = QCLPlatform::platforms();
```

Il est possible de récupérer différentes informations sur la plateforme :

- **CL_PLATFORM_PROFILE** : nom du profil supporté par l'implémentation. Peut prendre les valeurs **FULL_PROFILE** (toutes les spécifications d'OpenCL sont supportées et il n'est nécessaire d'utiliser les extensions) ou **EMBEDDED_PROFILE** (toutes les spécifications ne sont pas prises en charge et il faut tester les extensions présentes) ;
- **CL_PLATFORM_VERSION** : version d'OpenCL supportée. Le format est le suivant : **OpenCL#espace>#version_majeur.version_mineur>#espace>#informations plateforme-spécifique>** ;
- **CL_PLATFORM_NAME** : nom de la plateforme ;
- **CL_PLATFORM_VENDOR** : nom du vendeur de la plateforme ;
- **CL_PLATFORM_EXTENSIONS** : retourne la liste des extensions supportées par la plateforme. Chaque nom d'extension (sans espace) est séparé par un espace. Pour tester si une extension est présente, il suffit de rechercher si cette chaîne de caractères contient le nom de l'extension ;

Version C

```
char value[256] = "";
err = clGetPlatformInfo(clPlatform, CL_PLATFORM_PROFILE, 256, value, NULL);
err = clGetPlatformInfo(clPlatform, CL_PLATFORM_VERSION, 256, value, NULL);
err = clGetPlatformInfo(clPlatform, CL_PLATFORM_NAME, 256, value, NULL);
err = clGetPlatformInfo(clPlatform, CL_PLATFORM_VENDOR, 256, value, NULL);
err = clGetPlatformInfo(clPlatform, CL_PLATFORM_EXTENSIONS, 256, value, NULL);

// TODO : code spécifique C ?
std::string extensions(value);
bool as_fp64 = (extensions.find("cl_khr_fp64") != string::npos);
```

Version C++

```
std::string platformVendor;
platformList[0].getInfo(CL_PLATFORM_VENDOR, &platformVendor);
std::cerr << "Platform is by: " << platformVendor << "\n";
// TODO : code spécifique C++
```

QtOpenCL

```
QString profile = platform.profile();
QString vendor = platform.vendor();
QString version = platform.version();

QCLPlatform::VersionFlags flags = platform.versionFlags();
bool is_compatible_1_0 = flags.testFlag(QCLPlatform::Version_1_0);
```

QtOpenCL

```
bool is_compatible_1_1 = flags.testFlag(QCLPlatform::Version_1_1);

bool as_fp64 = platform.hasExtension("cl_khr_fp64");
```

Avec QtOpenCL, il est possible de tester si le profil est de type FULL_PROFILE ou EMBEDDED_PROFILE à l'aide des fonctions isFullProfile et isEmbeddedProfile.

QtOpenCL

```
bool is_full_profil = platform.isFullProfile();
bool is_embedded_profil = platform.isEmbeddedProfile();
```

TODO : qu'est qu'une extensions ? Quelles sont les extensions d'OpenCL ?

10-C - Le device

Le device est l'entité matériel correspondant à un périphérique d'exécution sur lequel est exécuté le code OpenCL. Les devices peuvent être des processeurs (CL_DEVICE_TYPE_CPU), des carte graphiques (CL_DEVICE_TYPE_GPU) ou des accélérateurs (c'est-à-dire des cartes possédant une puce GPU mais pas de sortie vidéo, CL_DEVICE_TYPE_ACCELERATOR). Une plateforme OpenCL peut contenir plusieurs devices, de types différents en général (ou moins un CPU). Certaines opérations sont spécifiques au contexte : compilation et exécution des kernels.

Il est possible de récupérer la liste des devices appartenant à une plateforme avec la fonction clGetDeviceIds en spécifiant le type de device (ou CL_DEVICE_TYPE_ALL pour avoir tous les devices compatibles openCL) :

Version C

```
cl_device_id devices[5];
size_t size = 0;
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 5, &devices, &size);
```

Version C++

```
cl::vector<cl::Device> devices;
devices = context.getInfo<CL_CONTEXT_DEVICES>();
```

Version QtOpenCL

```
QList<QCLDevice> devices;
devices = QCLDevice::allDevices();
devices = QCLDevice::devices(QCLDevice::GPU, platform);
```

10-D - Le contexte

Le contexte est l'entité logique sur laquelle s'exécute le code OpenCL. Un contexte peut être rattachée à plusieurs entités matérielles (device) en même temps. Certaines opérations sont spécifiques au contexte : création de buffer (tampon en mémoire vidéo indexé sous forme d'un tableau 1D) et d'images (tampon en mémoire vidéo indexé sous forme de tableaux 2D et 3D).

Version C

```
cl_context gpuContext;
gpuContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &err);
```

Version C++

Version C++

```
cl_context_properties cprops[3] = {CL_CONTEXT_PLATFORM, (cl_context_properties)(platformList[0])(), 0};
cl::Context context(CL_DEVICE_TYPE_CPU, cprops, NULL, NULL, &err);
```

Version QtOpenCL

```
QCLContext context;
bool fail = context.create();
```

TODO : création de contexte et device directement

10-E - Le programme

TODO : charger un fichier texte, créer un programme

Version C

```
TODO : à faire
char* src = "...";
program = clCreateProgramWithSource(context, 1, &src, NULL, &err);
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

Version C++

```
std::ifstream file("filename.cl");
std::string str(std::istreambuf_iterator<char>(file), (std::istreambuf_iterator<char>()));
cl::Program::Sources src(1, std::make_pair(str.c_str(), str.length() + 1));
cl::Program program(context, src);
```

Version QtOpenCL

```
// A partir d'une chaîne de caractères
QString src = "...";
QCLProgram program = QCLContext::createProgramFromSourceCode(src);

// A partir d'un fichier
QCLProgram program = QCLContext::createProgramFromSourceFile("filename.cl");
```

Compilation d'un programme

Version C

```
cl::Program program(context, source);
err = program.build(devices, "");
```

Version C++

```
cl::Program program(context, source);
err = program.build(devices, "");
```

Version QtOpenCL

```
cl::Program program(context, source);
err = program.build(devices, "");
```

10-F - Le Command queue

Les calculs sont envoyés à la carte graphique à l'aide de d'une queue d'instructions *cl_command_queue*. Chaque device a une commande queue en utilisant un contexte donné.

Version C

```
commands = clCreateCommandQueue(gpuContext, cdDevice, 0, &clErrcode);
```

Version C++

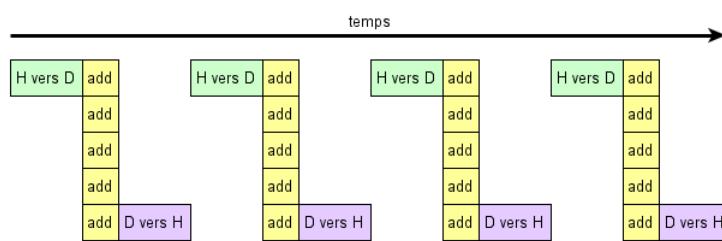
```
cl::CommandQueue commands(context, devices[0], 0, &err);
```

Version QtOpenCL

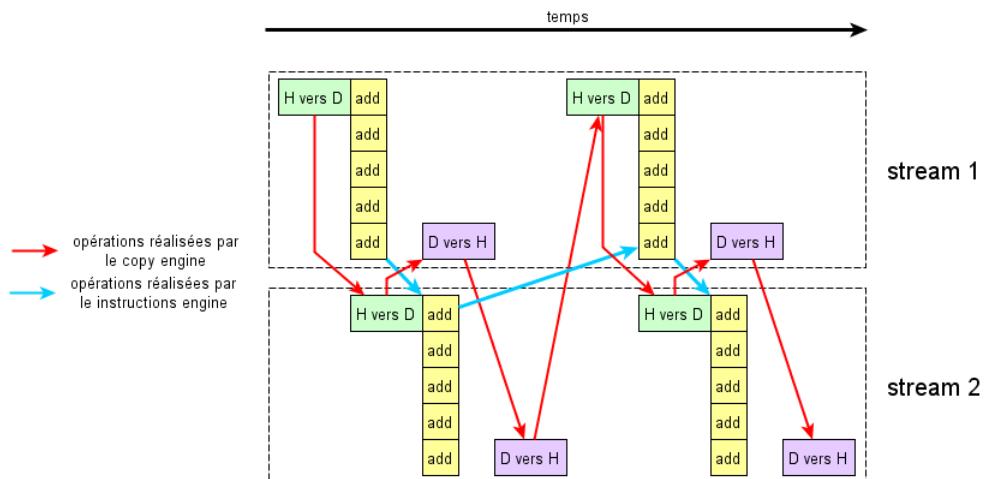
```
QCLCommandQueue
```

10-F-a - Les streams

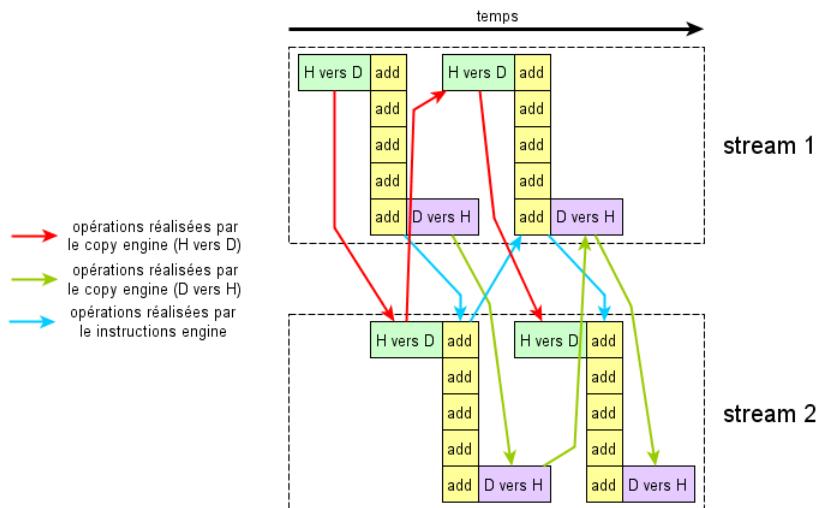
Utiliser 1 stream



Utiliser 2 streams avec overlap des transferts et des calculs



Utiliser 2 streams avec overlap des transferts in et out et des calculs



10-G - Le kernel

10-G-a - Créer d'un kernel

Un kernel est une fonction déclarée dans un programme et exécuté sur un périphérique compatible OpenCL.

Version C

```
cl_kernel kernel = clCreateKernel(program, "kernel", &err);
```

Version C++

```
cl::Kernel kernel(program, "kernel", &err);
```

Version QtOpenCL

```
QCLKernel kernel = program.createKernel("kernel");
```

Le kernel est une fonction précédée du mot clé `__kernel` avec un langage proche du C.

10-G-b - Passer des arguments au kernel

Les kernels peuvent prendre des arguments

Version C

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&dev_A);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&dev_B);
clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*)&dev_C);
```

Version C++

```
err = kernel.setArg(0, A);
```

Version QtOpenCL

```
// Utilisation de la fonction setArg
kernel.setArg(0, A);
```

10-G-c - Exécuter le kernel

Le kernel est exécuté grâce à la fonction `clEnqueueNDRangeKernel()`. `Size threads` exécuteront simultanément le kernel, chacun se chargeant d'additionner une composante. Les kernels sont lancés en utilisant des index 1D, 2D ou 3D (en pratique, c'est du 3D mais avec $y=1$ et $z=1$ pour le 1D et $z=1$ pour le 2D). Le nombre total d'éléments lancés (kernels ou work item) est appelé le global work size. Les éléments (work item) peuvent être regroupés en work group. Chaque work group est défini par un index local local work size et chaque éléments dans un groupe est défini par un index dans ce groupe. TODO : dessins représentant des éléments 1d, 2D, 3D, des work groupes, etc.

Version C

```
size_t GlobalWorkSize[1] = { size };
clEnqueueNDRangeKernel(commands, kernel, 1, 0, GlobalWorkSize, 0, 0, 0);
```

Version C++

```
err = commands.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(hw.length() + 1), cl::NDRange(1, 1), NULL, NULL);
```

Version QtOpenCL

```
kernel.setGlobalWorkSize(100, 100);
kernel.run();

// Passer des arguments et lancer le kernel
kernel(A, B);
```

10-H - Les buffers

10-H-a - Allouer un buffer

En général, les cartes graphiques ont une mémoire dédiée, différentes de la mémoire centrale utilisée par le CPU. Il faut donc copier les données à traiter dans la mémoire vidéo avant de lancer le calcul sur GPU.

Pour envoyer des données au GPU, il est possible d'utiliser deux méthodes. Soit d'allouer un tampon mémoire (buffer) et de copier les données, soit de passer des données en paramètres lors du lancement du kernel. Cette seconde méthode sera expliquée dans le chapitre suivant, en même temps que le lancement des kernels.

Pour allouer un bloc de données en mémoire vidéo, il faut juste connaître la taille du bloc à allouer en octets. Lorsque l'on travaille sur un tableau de données, la taille est obtenue simplement en multiplier la taille du tableau par la taille des éléments du tableau en octets.

TODO : buffer = 1D, image = 2D et 3D.

TODO : bloquante ou non ?

TODO : paramètre de `clCreateBuffer` + transfert des données dans `clCreateBuffer` ou séparément

Version C

```
// Version C
cl_mem dev_A = clCreateBuffer(gpuContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, size *
    sizeof(cl_float), A, &err);
cl_mem dev_B = clCreateBuffer(gpuContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, size *
    sizeof(cl_float), B, &err);
cl_mem dev_C = clCreateBuffer(gpuContext, CL_MEM_WRITE_ONLY, size * sizeof(cl_float), 0, &err);
```

Version C++

```
cl::Buffer gpuA(gpuContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, size * sizeof(cl_float), A,
&err);
cl::Buffer gpuB(gpuContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, size * sizeof(cl_float), B,
&err);
cl::Buffer gpuC(gpuContext, CL_MEM_WRITE_ONLY, size * sizeof(cl_float), 0, 0);
```

Version QtOpenCL

```
QCLBuffer -> création d'un buffer à partir d'un vector
```

QtOpenCL fournit la classe QCLVector pour simplifier la gestion mémoire d'un vecteur de données. QCLVector alloue le vecteur dans les deux mémoires (centrale et vidéo) et se charge de synchroniser les données.

Version QtOpenCL

```
QCLVector<int> input1 = context.createVector<int>(2048);
```

10-H-b - Envoyer et recevoir des données depuis un buffer

Le résultat étant stocké sur le GPU, on recopie son contenu dans le pointeur C à l'aide de la fonction `clEnqueueReadBuffer()`.

Version C

```
err = clEnqueueReadBuffer(commands, C, CL_TRUE, 0, size * sizeof(cl_float), C, NULL, NULL, NULL);
```

Version C++

```
err = queue.enqueueReadBuffer(C, CL_TRUE, 0, size, outh);
```

Version QtOpenCL

```
// Fonctions bloquantes
event = buffer.write(C, size);
event = buffer.read(C, size);

// Fonctions non bloquante
event = buffer.writeAsync(0, C, size);
event = buffer.readAsync(0, C, size);
```

10-h-c - Libérer un buffer

Enfin, les buffers alloués sur le GPU sont libérés. Pour les version C++ et QtOpenCL, la libération des buffers est automatique lors de la destruction des objets.

Version C

```
clReleaseMemObject(dev_A);
clReleaseMemObject(dev_B);
clReleaseMemObject(dev_C);
```

10-l - Les évènements

Les évènements permettent de connaître le status de la tâche auxquels ils sont associés. Cela permet de connaître en particulier si une tâche est terminée et de mesurer le temps d'exécution de cette tâche.

Version C

```
cl_event event;
clEnqueueNDRangeKernel(commands, kernel, 1, 0, GlobalWorkSize, NULL, NULL, &event);
```

Version C++

```
cl::Event event;
err = commands.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(size), cl::NDRange(1, 1), NULL,
&event);
```

Version QtOpenCL

```
QCLEvent event = kernel.run();
```

Les évènements possède la fonction wait qui permet de bloquer la fonction en attendant que la tâche soit terminée. Cela permet par exemple d'être sur que le kernel est finit de s'exécuter avant de lire les données.

QtOpenCL permet également d'utiliser les fonctions de QtConcurrent pour lancer un kernel. Celui-ci est alors lancé dans un thread séparé et de suivre les évènements avec QFutur. Il est alors possible de connecter des signaux et slots en passant par QFutureWatcher. voir <http://doc.qt.nokia.com/opencl-snapshot/concurrent.html>

Version QtOpenCL

```
// Avec QFutur
kernel.setGlobalWorkSize(100, 100);
QFuture<void> future = QtConcurrent::run(kernel, A, B);
future.waitForFinished();

// Avec QFutureWatcher
QFutureWatcher<void>* watcher = new QFutureWatcher<void>(this);
watcher->setFuture(QtConcurrent::run(kernel, a1, b1));
connect(watcher, SIGNAL(finished()), this, SLOT(eventFinished()));
```

11-bis - Additionner de deux vecteurs

12-bis - Additioner de deux vecteurs

11-bis-A - Introduction

TODO: ajouter les explications sur le but de l'exemple Pour vérifier que le calcul est correcte, on réaliser dans un premier temps le calcul sur le CPU pour vérifier que les version OpenCL donnent de résultats identiques :

main.cpp

```
// TODO : extraire que le code C++ et virer le reste
#include <vector>
#include <iostream>

int main(int, char**)
{
    // declarations
    const int N = 100000;
    std::vector<float> v1(N);
    std::vector<float> v2(N);
    std::vector<float> result_cpu(N, 0.0);
    std::vector<float> result_gpu(N, 0.0);

    // initialisation du contenu des vecteurs
    for (int i=0; i<N; ++i)
    {
        v1[i] = 2.0 * i;
        v2[i] = i * i;
    }

    // addition CPU
    for (int i=0; i<N; ++i)
        result_cpu[i] = v1[i] + v2[i];

    // test du résultat
    int error = 0;
    for (int i=0; i<N; ++i)
        if (result_cpu[i] != result_gpu[i])
            ++error;
    std::cout << "Il y a " << error << " éléments différents entre les versions CPU et GPU" << std::endl;
}
```

11-bis-B - Les tampons mémoire vidéo

L'addition des vecteurs avec OpenCL est réalisée par la méthode *AddVector()* de la classe *CLVectorAdd*. Les résultats obtenus sont enfin vérifiés côté CPU pour s'assurer du bon fonctionnement de notre programme.

```
// Add vectors using GPU
CLVectorAdd v;
v.AddVector(A, B, C, size);

// Compare results
bool error = false;
for(int i = 0; i < size; i++)
{
    error = !(C[i] == i + (i * i));
    if(error)
        break;
}
if(error)
    cout << "Erreur dans l'addition de vecteurs" << endl;
else
    cout << "Addition réussie" << endl;
```

Les vecteurs sont ensuite stockés dans la mémoire globale (DRAM) du GPU et initialisés dans le cas des deux vecteurs d'entrée.

cpp

```
// Version C
cl_mem dev_A = clCreateBuffer(gpuContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, size *
    sizeof(cl_float), A, &err);
cl_mem dev_B = clCreateBuffer(gpuContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, size *
    sizeof(cl_float), B, &err);
cl_mem dev_C = clCreateBuffer(gpuContext, CL_MEM_WRITE_ONLY , size * sizeof(cl_float), 0 , &err);

// Version C++
char * outH = new char[hw.length()+1];
cl::Buffer outCL(context, CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR, hw.length()+1, outH, &err);

// Version QtOpenCL
QCLBuffer -> création d'un buffer à partir d'un vector
QCLVector -> création d'un vecteur et d'un buffer et transfère automatique CPU<->GPU
QCLVector<int> input1 = context.createVector<int>(2048);
```

Le résultat étant stocké sur le GPU, on recopie son contenu dans le pointeur C à l'aide de la fonction `clEnqueueReadBuffer()`.

cpp

```
clEnqueueReadBuffer(clCommandQueue, dev_C, CL_TRUE, 0, size * sizeof(cl_float), C, 0, 0, 0);
```

Enfin, les buffers alloués sur le GPU sont libérés.

cpp

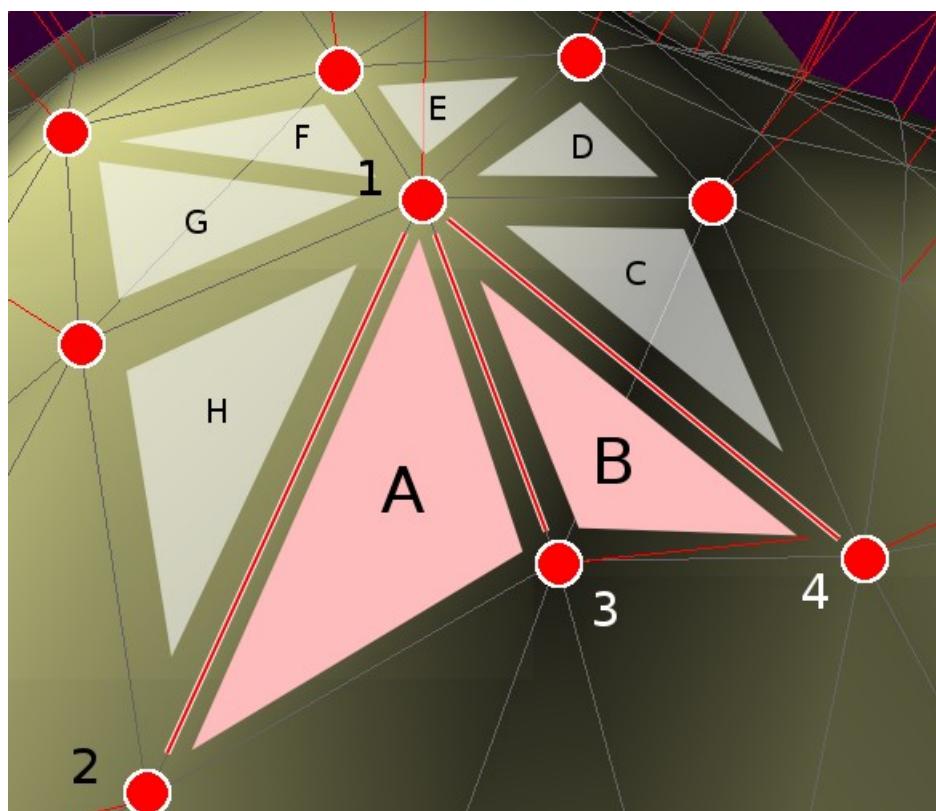
```
clReleaseMemObject(dev_A);
clReleaseMemObject(dev_B);
clReleaseMemObject(dev_C);
```

14 - Exemples

12 - Calculer les vecteurs normaux de la heightmap

12-A - Principe du calcul

Prenons un vertex quelconque dans notre terrain (excepté en bordure). Comme le montre l'image, ce vertex est entouré de 8 voisins, il y a donc 8 plans (de A à H) entourant chaque vertice. Pour obtenir un seul vecteur normal par vertex il est donc nécessaire d'effectuer une moyenne sur le voisinage.



Le vecteur normal à un plan est calculé en effectuant le produit vectoriel (cross product) entre les deux vecteurs définissant le plan (chaque triangle sur l'image représente un plan).

pseudo-code

```

Pour chaque point de la carte
    Pour chacun des 8 triangles définis par un point du voisinage
        Calculer le vecteur normal d'un triangle
    Calculer le vecteur moyen

```

Les normales seront stockées dans un vecteur de vecteur 3D, *m_normals*. L'implémentation de l'algorithme de calcul des normales se fait de la façon suivante : une première fonction *makeNormals* se charge de parcourir chaque vertex du terrain et appelle la fonction de calcul de normales, *computeNormal* pour chacun des 8 plans entourant le vertex. Chaque normale définissant un des 8 plans voisins et récupérée et additionnée dans un vecteur (dit d'accumulation). Une fois les 8 normales additionnées, le vecteur d'accumulation est normalisé nous permettant d'obtenir une normale moyenne.

Les coordonnées 3D du vertex ainsi que celles de ses deux voisins définissant le plan *index* sont récupérées à l'aide d'une fonction nous permettant de passer d'une coordonnée 2D (image) à 1D (buffer) : *indexOfPoint()*.

```
    inline int indexOfPoint(const int x, const int y) { return (y * vertices_by_x +
x); }
```

```
// h
 QVector<QVector3D> m_normals;

// .cpp
void HeightmapWidget::makeNormals()
{
    const int num_neighbor = 8;

    m_normals.resize(num_vertices);
    for(int z = 0; z < vertices_by_z; ++z)
    {
        for(int x = 0; x < vertices_by_x; ++x)
        {
            QVector3D normal;

            for(int i = 0; i < num_neighbor; i++)
                normal += computeNormal(x, z, i);

            normal.normalize();
            m_normals[indexOfPoint(x, z)] = normal;
        }
    }
}
```

La fonction *computeNormal* se charge de calculer le vecteur normal d'un plan voisin du vertex en cours de traitement. Elle prend en paramètre les coordonnées du vertex dont la normale est en cours de calcul, (*x*,*z*) et le numéro du plan voisin (compris entre 0 et 7) *index* (A à H sur l'image). Deux tableaux constants permettent d'obtenir la position relative d'un des huit points voisins en fonction du numéro de plan donné.

```
QVector3D HeightmapWidget::computeNormal(const int x, const int z, const int
index)
{
    const int x_offset[] = { -1, 0, 1, 1, 1, 0, -1, -1, -1 };
    const int z_offset[] = { -1, -1, -1, 0, 1, 1, 0, -1 };
```

La position des deux points définissant le plan *index* est calculée, en tournant dans le sens anti-horaire ces deux points sont aux indices *index*, *index+1*. Une vérification systématique de non débordement du cadre de l'image est effectuée, en cas de débordement un vecteur nul est renvoyé.

```
int x_B = x + x_offset[index];
if (x_B < 0 || x_B >= vertices_by_x) return QVector3D();
int z_B = z + z_offset[index];
if (z_B < 0 || z_B >= vertices_by_z) return QVector3D();

int x_C = x + x_offset[index+1];
if (x_C < 0 || x_C >= vertices_by_x) return QVector3D();
int z_C = z + z_offset[index+1];
if (z_C < 0 || z_C >= vertices_by_z) return QVector3D();
```

Enfin, le produit vectoriel entre les deux vecteurs définissant le plan *index* est calculé. Pour cela, les coordonnées 3D du vertex en cours de traitement et des deux points représentant le sous plan étudié sont récupérées.

heightmapwidget.cpp

```

QVector3D A = m_vertices[indexOfPoint(x, z)];
QVector3D B = m_vertices[indexOfPoint(x_B, z_B)];
QVector3D C = m_vertices[indexOfPoint(x_C, z_C)];

QVector3D AB = B - A;
QVector3D AC = C - A;

QVector3D cross = QVector3D::crossProduct(AC, AB);
cross.normalize();

return cross;
}

```

12-B - Calcul des normales à l'aide de QtOpenCL

Dans cette partie nous allons utiliser OpenCL pour accélérer le calcul des normales, qui était auparavant réalisé coté CPU. La mise à jour des normales peut être utile dans le cas où nous modifions dynamiquement les positions des vertices du terrain.

Pour ce faire nous utiliserons le module QtOpenCL qui se présente sous la forme d'un plugin à télécharger et compiler. (voir article « Utiliser OpenCL avec Qt » <http://qt-labs.developpez.com/bibliotheque/qtopencl/>).

 *QtOpenCL gère plusieurs aspects d'openCL implicitement, dans le cadre d'une utilisation standard, le développeur n'aura pas à manipuler les mécanismes OpenCL « bas niveau » (platform, command queue,).*

La première chose à faire, au niveau du constructeur, est d'initialiser OpenCL et de lui préciser sur quel type de processeur parallèle les calculs vont s'effectuer. Le programme est ensuite compilé et en pointeur vers le kernel utilisé est récupéré.

C++

```

if (!GPUContext.create(QCLDevice::GPU))
qFatal("Could not create OpenCL context");
UpdateNormalsProgram = GPUContext.buildProgramFromSourceFile(QLatin1String(":/clprograms/update-
normals.cl"));

UpdateNormalsKernel = UpdateNormalsProgram.createKernel("NormalsUpdate");

```

Le calcul est implémenté dans une fonction : *makeNormalsGPU()*. Comme pour l'exemple d'addition de vecteurs, il est nécessaire d'envoyer les données aux GPU, dans notre cas, les données en entrée sont les vertices composants notre terrain (*gpu_in_vector*) et les données en sortie sont un vecteur contenant pour chaque vertice, la normale qui lui est associée (*gpu_out_vector*).

QtOpenCL dispose d'une classe de base pour les buffers stockés sur le serveur (GPU) : *QCLBuffer*. Cette classe définit plusieurs opérations de bases telle que l'allocation, l'écriture, la lecture. Nous utilisons une classe héritant de *QCLBuffer* : *QCLVector*. Cette classe est l'équivalent d'un *Qvector* mais stocké sur la mémoire serveur.

Le vecteur de sortie est allouer, ainsi que le vecteur d'entrée qui est ensuite remplis avec les vertices du terrain.

cpp

```

// Buffer setup
QCLVector<QVector4D> gpu_out_vector = GPUContext.createVector<QVector4D>(num_normals,
QCLMemoryObject::ReadWrite);

```

cpp

```
QCLVector< QVector4D > gpu_in_vector = GPUContext.createVector< QVector4D >(num_vertices,
QCLMemoryObject::ReadOnly);
gpu_in_vector.write(m_vertices.constData(), num_vertices);
```

Le kernel est ensuite appelé, *num_vertices* threads exécuteront en parallèle ce kernel, c'est à dire un thread par vertice. Ce nombre est indiqué à l'aide de la fonction *setGlobalWorkSize* de la classe *QCLKernel*.

Le kernel est lancé directement sans appel de fonction, grâce à une surcharge de l'opérateur () de la classe *QCLKernel*. Notre kernel prend 4 arguments :

- Pointeur vers le buffer de vertices (entrée)
- Pointeur vers le buffer de normales (sortie)
- Nombre de vertices
- Largeur du terrain (pour trouver les positions des vertices voisins dans le buffer)

cpp

```
// Kernel launch
UpdateNormalsKernel.setGlobalWorkSize(num_vertices);

QCLEvent event = UpdateNormalsKernel(gpu_in_vector, gpu_out_vector, num_vertices, vertices_by_x);
event.waitForFinished();
```

N'oublions pas de libérer les buffers alloués sur le GPU.

cpp

```
// Release buffers
GPUContextGL.release(gpu_in_vector);
GPUContextGL.release(gpu_out_vector);
```

Le calcul est effectué à chaque passage dans la fonction *paintGL()*. Les normales étant stockées à l'aide VBO, le contenu du VBO est mis à jour.

cpp

```
// Update normals
if(normal_on_gpu)
makeNormalsGPU();
else
makeNormalsCPU();
normal_buffer.bind();
normal_buffer.write(0, m_normals.constData(), sizeof(QVector4D) * m_normals.size());
normal_buffer.release();
```

Voici le code du kernel :

cl

```
//-----
//-----
bool nearXmin(const int index, const int width)
{
    return ((index % width) == 0);
}
bool nearXmax(const int index, const int width)
{
    return ((index % width) == (width - 1));
}
bool nearYmin(const int index, const int width)
{
    return ((index / width) == 0);
```

```

cl
}

bool nearYmax(const int index, const int width)
{
return ((index / width) == (width - 1));
}
//-----
//-----
bool nearBorder(const int index, const int pos, const int width)
{
bool near = false;
if (pos == 0 || pos == 1 || pos == 7)
    near |= nearXmin(index, width);
else if (pos == 3 || pos == 4 || pos == 5)
    near |= nearXmax(index, width);
if (pos == 5 || pos == 6 || pos == 7)
    near |= nearYmin(index, width);
else if (pos == 1 || pos == 2 || pos == 3)
    near |= nearYmax(index, width);
return near;
}
//-----
//-----
int indexOfPosition(const int index, const int pos, const int width)
{
int ii = index;
if (pos == 0 || pos == 1 || pos == 7)
    --ii;
else if (pos == 3 || pos == 4 || pos == 5)
    ++ii;
if (pos == 5 || pos == 6 || pos == 7)
    ii -= width;
else if (pos == 1 || pos == 2 || pos == 3)
    ii += width;
return ii;
}
//-----
//-----
float4 getVector(__global const float4* in, const int index, const int pos, const int width)
{
int ii = indexOfPosition(index, pos, width);
float4 O = in[index];
float4 A = in[ii];
return (A - O);
}
//-----
//-----
void updateVectors(__global const float4* in, float4* vector_plan, const int index, const int width)
{
for (int pos = 0; pos < 8; ++pos)
{
    vector_plan[pos] = getVector(in, index, pos, width);
}
}
//-----
//-----
float4 getNormal(__global const float4* in, float4* vector_plan, const int pos)
{
float4 u = vector_plan[pos];
float4 v = vector_plan[(pos + 1) % 8];
return normalize(cross(v, u));
}
//-----
//-----
float4 getSumNormal(__global const float4* in, const int index, const int width)
{
float4 vector_plan[8];
float4 sum_normal = { 0.0, 0.0, 0.0, 0.0 };
updateVectors(in, vector_plan, index, width);
for (int pos = 0; pos < 8; ++pos)
{
    if (!nearBorder(index, pos, width) && !nearBorder(index, (pos+1)%8, width))

```

```

cl
        sum_normal += getNormal(in, vector_plan, pos);
    }
    sum_normal = normalize(sum_normal);
    //sum_normal.w = 1.0;
    return sum_normal;
}
//-----
//-----
__kernel void NormalsUpdate(
__global const float4* in,           // int points vector
__global float4* out,                // out normals vector
const int size,                     // size of vectors
const int width)                   // map width
{
const int global_id = get_global_id(0);
if (global_id < size)
{
    out[global_id] = getSumNormal(in, global_id, width);
}
}

```

12-C - Partager des données entre OpenCL et OpenGL

Dans la partie précédente, nous avons accéléré le calcul des normales en utilisant le GPU. Néanmoins, il est dommage de devoir récupérer le buffer de résultat d'OpenCL, depuis le GPU vers le CPU, pour le renvoyer ensuite au GPU dans un VBO.

OpenCL dispose d'une fonctionnalité intéressante dans notre cas : le partage de contexte entre OpenGL et OpenCL. En utilisant cette fonctionnalité, il est possible de stocker directement les résultats d'OpenCL dans le VBO contenant les normales, nous évitant un allé-retour vers la mémoire système inutile.

Le partage OpenCL / OpenGL est implémenté à l'aide de la classe *QCLContextGL*. Un contexte OpenGL doit être actif à la création du contexte CL/GL. Ce partage de données entre les deux contextes nécessite la présence d'une extension OpenCL : *cl_khr_gl_sharing*. Sa présence peut être testée avec la fonction *supportsObjectSharing()* de la classe *QCLContextGL*.

```

cpp
QCLContextGL GPUContextGL ;
makeCurrent();
if(!GPUContextGL.create())
qFatal("Could not create OpenCL / OpenGL context");

```

Outre le changement de contexte (*QCLContext* vers *QCLContextGL*), les différences par rapport au programme de la partie précédente apparaissent au niveau de la création et libération des buffers. Existant déjà côté GPU sous la forme de VBO, seul des pointeurs vers ces buffers seront récupérés.

La fonction *createGLBuffer()* de la classe *QCLContextGL* récupère le pointeur vers un buffer OpenGL grâce à son identifiant (obtenu avec la fonction *bufferId()* de la classe *QGLBuffer*).

```

cpp
QCLBuffer gpu_in_vector = GPUContextGL.createGLBuffer(vertex_buffer.bufferId(),
QCLMemoryObject::ReadOnly);

QCLBuffer gpu_out_vector = GPUContextGL.createGLBuffer(normal_buffer.bufferId(),
QCLMemoryObject::ReadWrite);

```

Avant d'effectuer tout type d'opération à l'aide d'OpenCL sur ces buffers, il est impératif d'appeler la méthode *acquire()*.

cpp

```
GPUContextGL.acquire(gpu_in_vector);
GPUContextGL.acquire(gpu_out_vector);
```

Le kernel est ensuite executé (même code que la partie précédente).

cpp

```
// Kernel launch
UpdateNormalsKernel.setGlobalWorkSize(num_vertices);
UpdateNormalsKernel(gpu_in_vector, gpu_out_vector, num_vertices, vertices_by_x);
```

Et les buffers OpenGL sont libérés.

cpp

```
// Release buffers
GPUContextGL.release(gpu_in_vector);
GPUContextGL.release(gpu_out_vector);
```

Il n'y a bien entendu plus de raisons de lire le contenu du buffer de résultat d'OpenCL car il s'agit du VBO utilisé pour le rendu.

Le code de la fonction de rendu est modifié pour supprimer la mise à jour du VBO dans le cas où les normales sont calculées côté GPU.

cpp

```
// Update normals
if(normal_on_gpu)
makeNormalsGPU();
else {
```

```
}
```

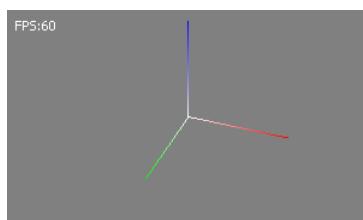
13 - Codes sources des exemples présentés

Voici les codes sources des exemples abordés dans ce tutoriel. Pour les tester, il suffit d'ouvrir le fichier .pro avec Qt Creator et de lancer l'application. Il est intéressant aussi d'étudier ces codes sources pour bien comprendre le fonctionnement et l'implémentation d'OpenGL/OpenCL dans Qt.

14-A - Application minimale QtOpenGL

[Source](#) [Sources](#)

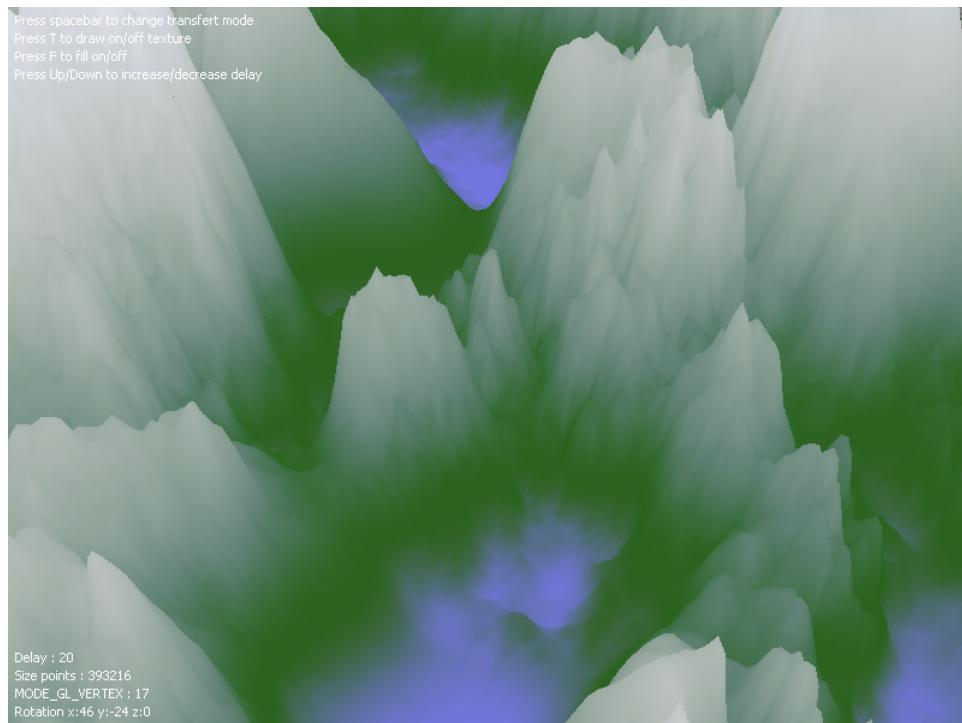
Cette application permet de présenter le code minimal nécessaire pour initialiser un contexte OpenGL avec Qt. Elle affiche un simple repère orthonormé et le nombre d'images par seconde (FPS). Le bouton droit de la souris permet de tourner autour du repère et la molette permet de s'approcher et de s'éloigner.



14-B - Comparaison entre les différents modes de transfert de données au GPU

[Source](#) [Sources](#)

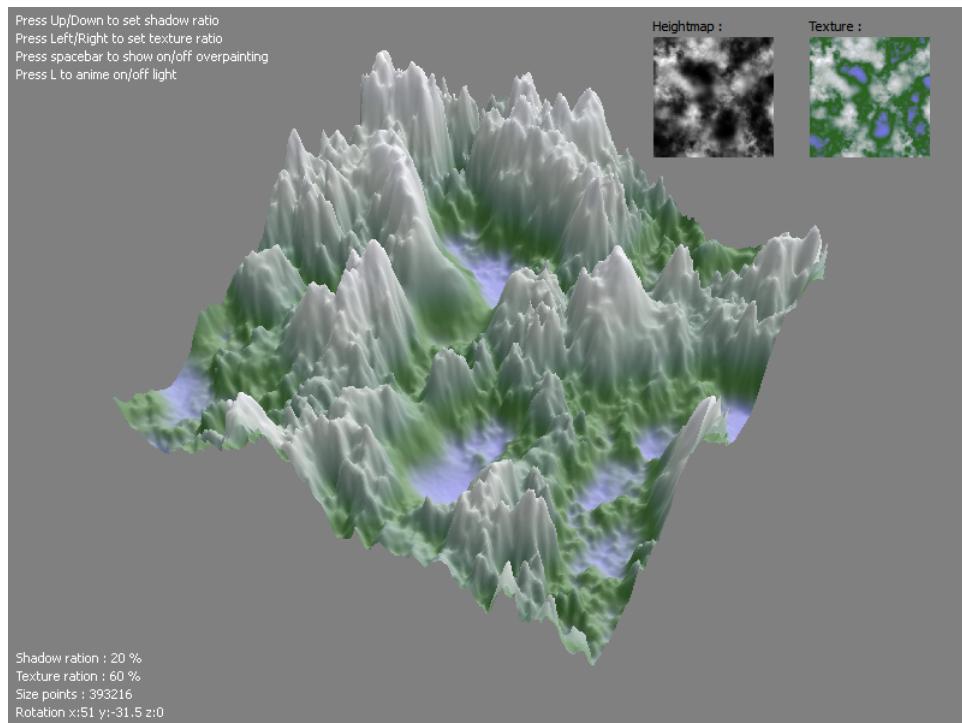
Cette application permet de présenter les différentes méthodes de transfert des données entre le CPU et le GPU (directe, vertex array, vertex buffer, indices). Elle affiche une heightmap avec texture. Le bouton droit de la souris permet de tourner autour du repère et la molette permet de s'approcher et de s'éloigner. La touche "T" permet d'afficher ou non la texture. La touche "F" permet d'afficher la heightmap sous forme de fil de fer ou non. La barre d'espace permet de changer de mode de transfert. Les flèches haut et bas permettent de modifier le délai d'affichage entre deux images. Pour déterminer le taux de FPS maximal, il suffit de mettre le délai à 0.



14-C - Utilisation des shaders

Source Sources

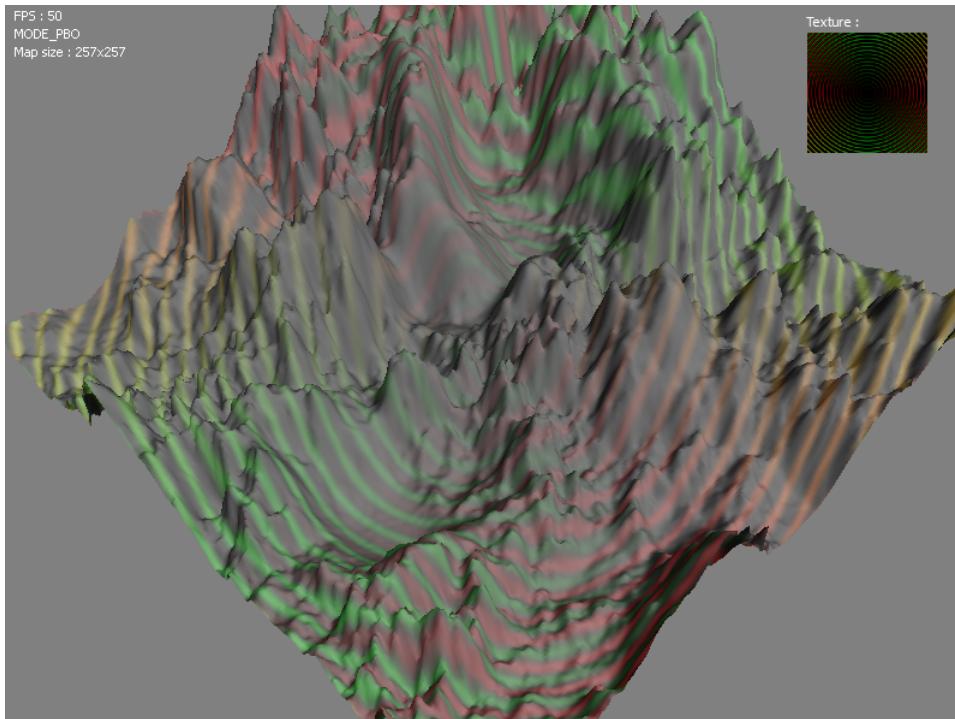
Cette application permet de présenter l'utilisation des shaders avec Qt. Elle affiche une heightmap avec texture et ombrage de Phong. La source de lumière est de type parallèle, c'est-à-dire que la source de lumière est à l'infini. L'angle d'incidence varie au cours du temps mais il est possible de stopper l'animation en appuyant sur la touche "L". Le bouton droit de la souris permet de tourner autour du repère et la molette permet de s'approcher et de s'éloigner. La barre d'espace permet d'afficher ou non le texte et les images 2D. Les flèches droite et gauche permettent de contrôler l'intensité de la texture. Les flèches haut et bas permettent de contrôler l'intensité des ombrages.



14-D - Rendu off-screen

Source Sources

Cette application permet de tester différents modes de rendu off-screen. Elle affiche une heightmap avec texture et ombrage de Phong. Le bouton droit de la souris permet de tourner autour du repère et la molette permet de s'approcher et de s'éloigner. La barre d'espace permet de changer de mode de rendu off-screen.



15 - Références

15-A - Les tutoriels de Developpez

Les autres tutoriels proposés par la rubrique **Qt** de Developpez :

- [Intégration d'OpenGL dans une interface Qt](#) ;
- [Qt Graphics et performance - Le moteur de rendu OpenGL](#) ;
- [Utiliser OpenCL avec Qt](#) ;
- [Intégrer Ogre à Qt](#) ;
- [Présentation de GLC_lib](#).

Les autres tutoriels proposés par la rubrique **Jeux** de Developpez :

- [Afficher une heightmap avec OpenGL](#) ;
- [Génération de terrain par l'algorithme de Perlin](#) ;
- [Génération de terrain et triangulation de Delaunay](#) ;
- [Génération de textures de terrain](#) ;
- [Utilisation de l'algorithme DiamondSquare pour la génération de terrain](#) ;
- [Création d'une Skybox en OpenGL](#) ;
- [Une introduction à CUDA](#) ;
- [CUDA approfondi](#).

15-B - Site officiel OpenGL

[Liste de toutes les documentations officielles d'OpenGL \(toutes versions\)](#). Voir en particulier :

- [Documentation de référence pour OpenGL 2.1](#) ;
- [Documentation de référence pour OpenGL 3.3](#) ;
- [Tableau résumant les principales fonctions d'OpenGL 3.2](#) ;
- [Documentation de référence pour OpenGL 4.1](#) ;
- [Tableau résumant les principales fonctions d'OpenGL 4.1](#) ;
- [Documentation de référence du GLSL](#).

15-C - OpenCL

- [Documentation de référence pour OpenCL \(Khronos\)](#) ;
- [Forum technique OpenCL \(Khronos\)](#) ;
- [Binding Python](#) ;
- [OpenTK : Binding C#](#) ;

16 - Copyright

Certaines images utilisées dans ce tutoriel sont issues de Wikipédia. La licence et la liste des auteurs peuvent être consultées sur les pages suivantes :

- **L'image de la heightmap.** Les images de la texture et de la normal map ont été créées à partir de cette image ;
- **L'image des composantes de l'ombrage de Phong.**

La représentation du pipeline graphique est issue du tutoriel **OpenGL Shading Language Overview** sur le site OpenGL.

17 - Remerciements

Merci à **LittleWhite**, **dourouc05**, **yan**, **bafman** et **AuraHxC** pour leur relecture et leurs conseils. Merci à **ClaudeLELOUP** et à **jacques_jean** pour leur relecture orthographique très attentive.