Synthèse d'Image et Animation
Aide de cours sur l'implémentation d'une application OpenGL en Qt

lionel.reveret@inria

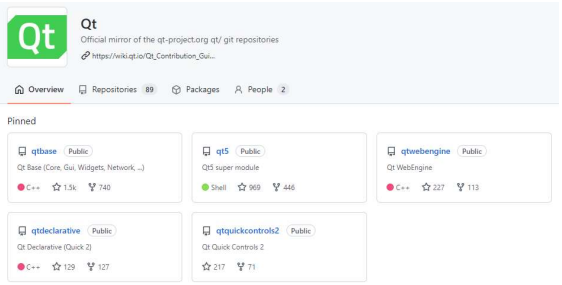
2021-22

Les TP et projets du cours SIA reposent sur une base de code C++ d'une application OpenGL en Qt qui vous est fournie.
OpenGL est la librairie graphique pour la partie 3D qui est spécifiée en C par le groupe de normalisation Khronos.
Qt est un environnement de développement C++ multiplateforme d'interface très répandu. Il comprend, entre autres, une implémentation OpenGL pour afficher du contenu 3D dans une fenêtre.

La partie Rendu du cours se focalise sur les shaders GLSL en ne requérant donc que la programmation de ces shaders. La partie Animation nécessite de considérer tout le pipeline graphique, en particulier les interactions entre des calculs qui ne peuvent se faire qu'en CPU et un affichage des éléments en GPU. Le code C++ de l'application OpenGL en Qt peut paraître complexe avant de pouvoir y intervernir. Le but de ce document est de vous présenter les aspects principaux d'une application OpenGL en Qt et de vous aider à identifier les éléments importants où insérer votre contribution.

Avant de présenter l'architecture du code du projet SIA, il est abordé un premier exemple OpenGL simple issu de la distribution open source de Qt. Ce premier code est accessible depuis le dépôt github de Qt, dans les exemples OpenGL du module qtbase. Il s'agit de l'exemple cube qui affiche simplement un cube texturé, manipulable avec la souris.

https://github.com/qt

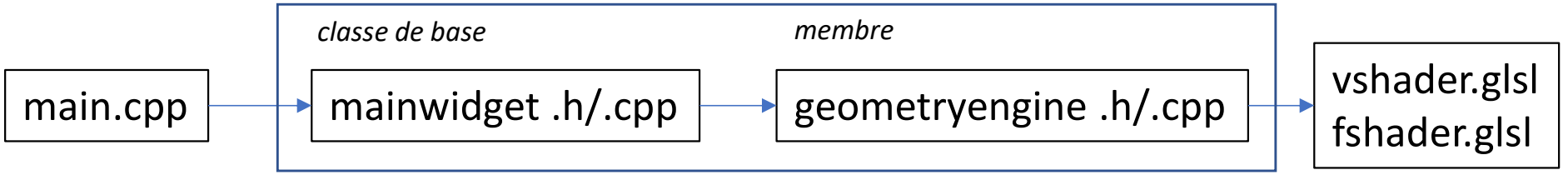


https://github.com/qt/qtbase/tree/dev/examples/opengl/cube

Principes d'une application Qt

Le but principal de Qt est de proposer la possibilité de générer des applications reposant sur des fenêtres graphiques et des interactions utilisateurs, typiquement via le clavier et la souris. Qt repose sur deux mécanismes d'interaction : les messages entre des émetteurs (signals) et des receveurs (slots), ainsi que des événements. L'élément de base des fenêtres est une classe QWidget, qui se spécialise par dérivation en toute sorte d'éléments graphiques, du bouton à une espace de rendu via un contexte OpenGL. L'application repose sur une classe QApplication qui lance la fenêtre principale.

Architecture de l'exemple Qt-base / OpenGL « cube »



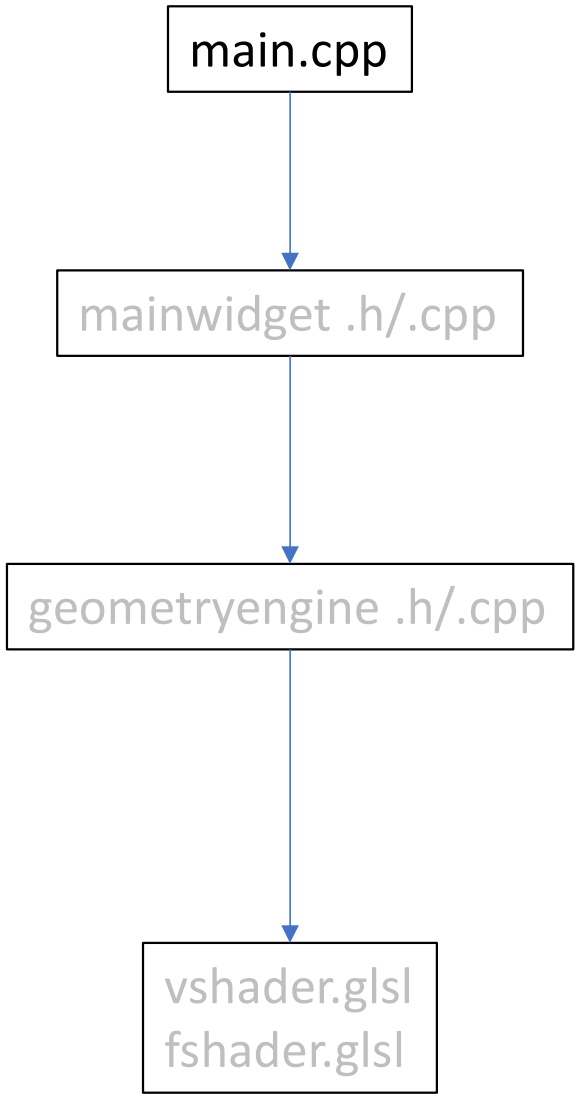
Déclaration de l'application et lancement de la fenêtre principale de type *MainWidget*.

Classe *MainWidget* dérivant d'une *Widget* / fenêtre et implémentant les étapes principales OpenGL (initialisation, animation, rendu).

Ressources spécifiques pour les données 3D de géométrie et texture : création, transfert GPU et rendu.

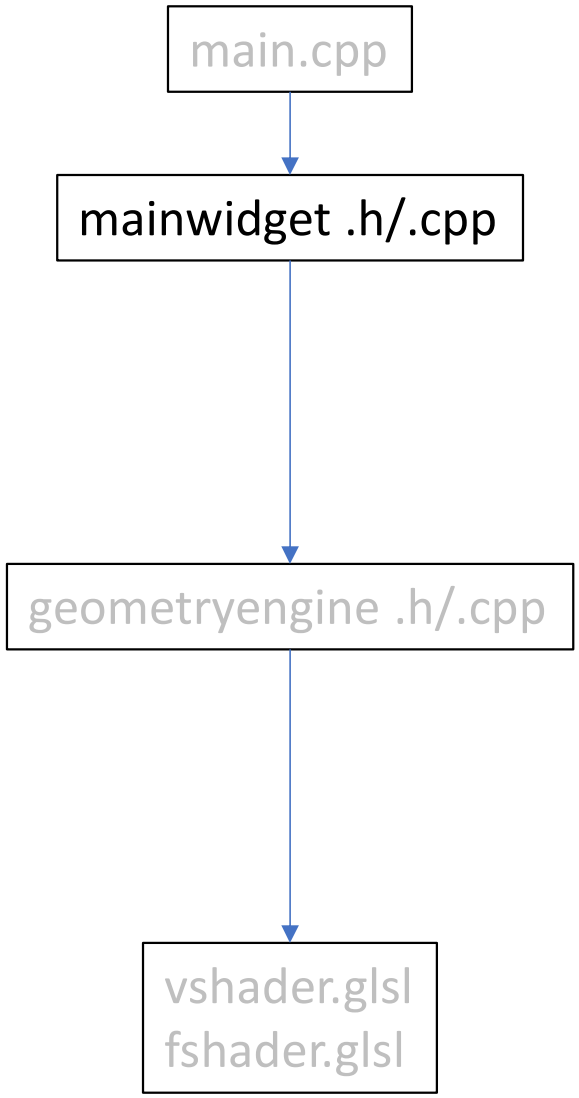
Shaders simples : projection 3D, coloration par texture

Détails de l'exemple OpenGL / Qt-base « cube »



Déclaration d'une *QApplication* et association d'une fenêtre *MainWidget*

Détails de l'exemple OpenGL / Qt-base « cube »

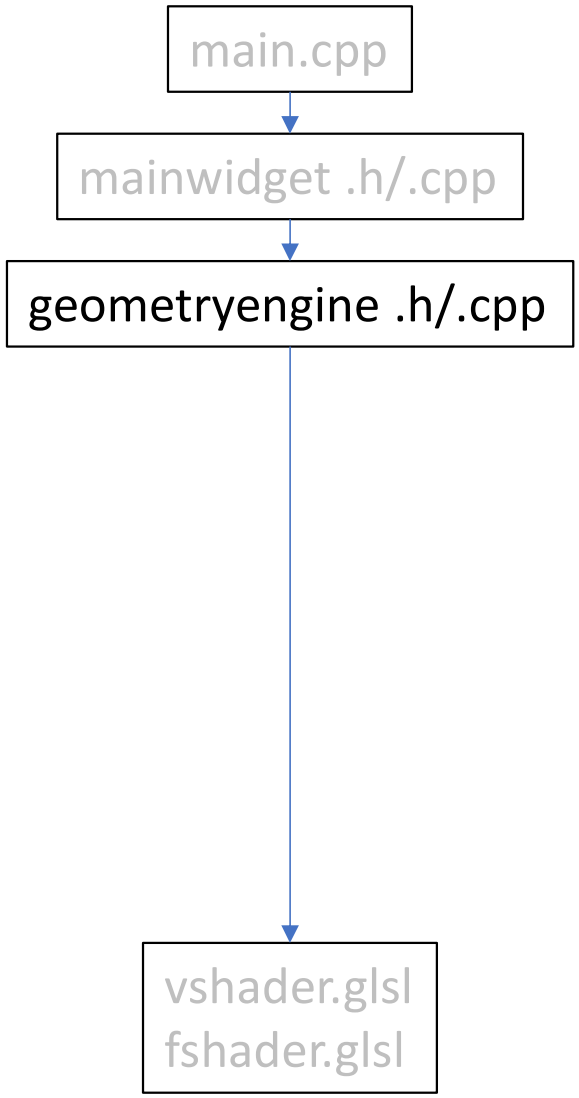


Classe *MainWidget* dérivant d'une classe *QOpenGLWidget* (fenêtre avec des événements spécifiques OpenGL) et *QOpenGLFunctions* (buffers, program shaders, etc)

Les fonctions évènements sont appelées automatiquement. On trouve les évènements liés à l'implémentation OpenGL (**initializeGL**, **resizeGL**, **paintGL**) et ceux liés à l'interaction utilisateur (**mousePressEvent**, **mouseReleaseEvent**, timerEvent) :

- **initializeGL** : appelé au lancement du contexte OpenGL, on y retrouve ici le chargement des shaders et textures, la création de la géométrie (classe *GeometryEngine*) et le lancement du timer pour l'animation
- **resizeGL** : appelé dès qu'on change la taille de la fenêtre pour ajuster la matrice de projection
- **paintGL** : appelé dès qu'un rendu est nécessaire, activation des textures (par ***bind***), calcul des matrices et transfert dans les shaders, appel du dessin
- **mousePressEvent** / **mouseReleaseEvent** : appelé dès que le bouton de la souris est activé, calcul pour le transformer en rotation
- **timerEvent** : appelé à intervalles réguliers dès que tous les autres évènements sont traités, typiquement pour implémenter une boucle d'animation. Un rendu est déclenché par l'appel de la fonction **update()**

Détails de l'exemple OpenGL / Qt-base « cube »



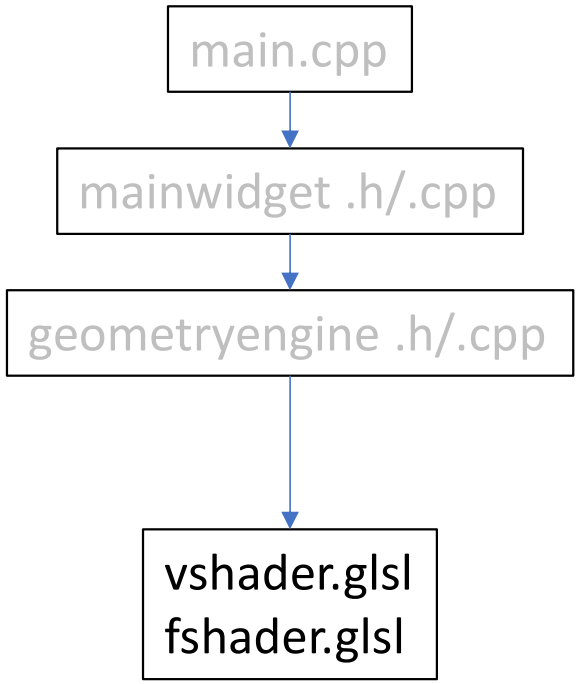
La classe *GeometryEngine* crée les données de géométrie et de texture (vertexes, coordonnées de textures et topologie de triangle). Elle dérive de *QOpenGLFunctions* pour récupérer des mécanismes OpenGL. Les données sont d'abord créées dans des tableaux en mémoire CPU, puis transférées dans des buffers OpenGL via les objets d'interface *QOpenGLBuffer* pour les VBO (Vertex Buffer Object).

Les VBO sont créés dans le contexte OpenGL avec un appel à **create()**. On les rend actif par un appel à **bind()**, d'une part pour le transfert des données CPU avec **allocate()** et d'autre part pour le rendu.

Le rendu se retrouve dans la procédure **drawGeometry()** :

- activation des VBO avec **bind()**, le contexte OpenGL reconnait avoir un buffer de données et un buffer d'indices qui seront utilisés en streaming par les shaders
- description de l'alignement des données dans les buffers
- lancement du rendu avec **glDrawElements()** selon une primitive GL_TRIANGLE_STRIP

Détails de l’exemple OpenGL / Qt-base « cube »



Shaders basics de projection 3D et de coloration par texture

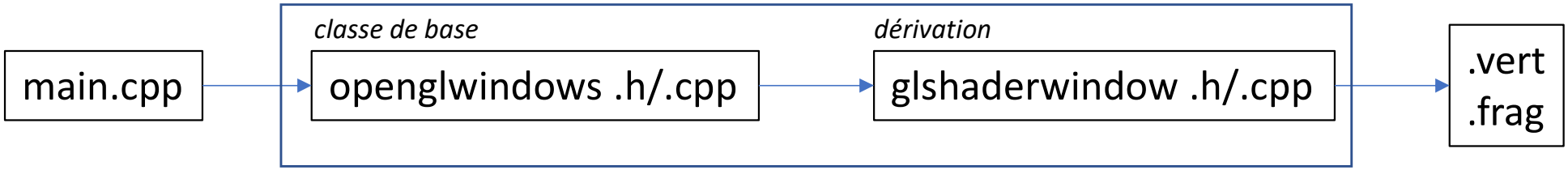
Repères dans le code fourni pour le projet SIA

Pour la partie animation du projet, vous pouvez soit reprendre l'exemple de base « cube » que l'on vient de décrire, soit continuer l'application fournie comme support de code pour les projets SIA. Elle suit exactement les mêmes principes que l'exemple de base « cube » et vous offre un lien avec la bibliothèque trimesh pour le chargement de modèles 3D et bénéficiera des shaders que vous avez déjà implémentés.

La difficulté est que beaucoup de code existe et les fonctionnalités de base sont difficiles à identifier. Pour l'affichage demandé du squelette d'animation, il est conseillé de repartir de la structure utilisée pour le sol (ground), qui est un exemple de création procédurale de géométrie. On peut s'affranchir de tout ce qui concerne les ombrages et le GPGPU.

L'organisation de l'architecture du support de code diffère de l'exemple « cube » en reprenant cependant les mêmes concepts.

Architecture du support de code fourni en SIA

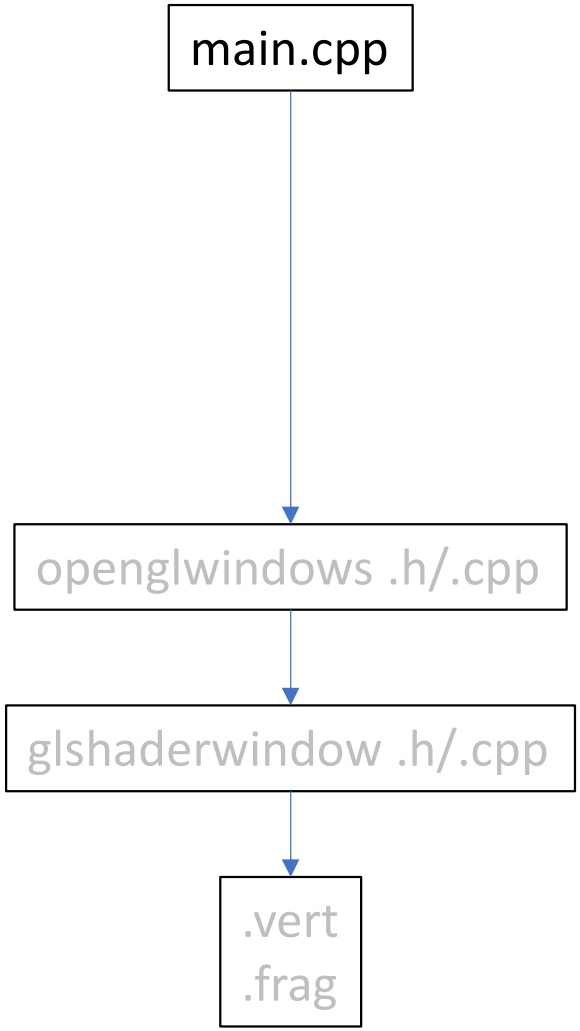


Déclaration de l'application et lancement de la fenêtre principale de type *glShaderWindow*

Evènements de rendu et d'animation avec une implémentation non-encapsulée d'OpenGL contrairement à l'exemple « cube » : gestion explicite du contexte et du canevas

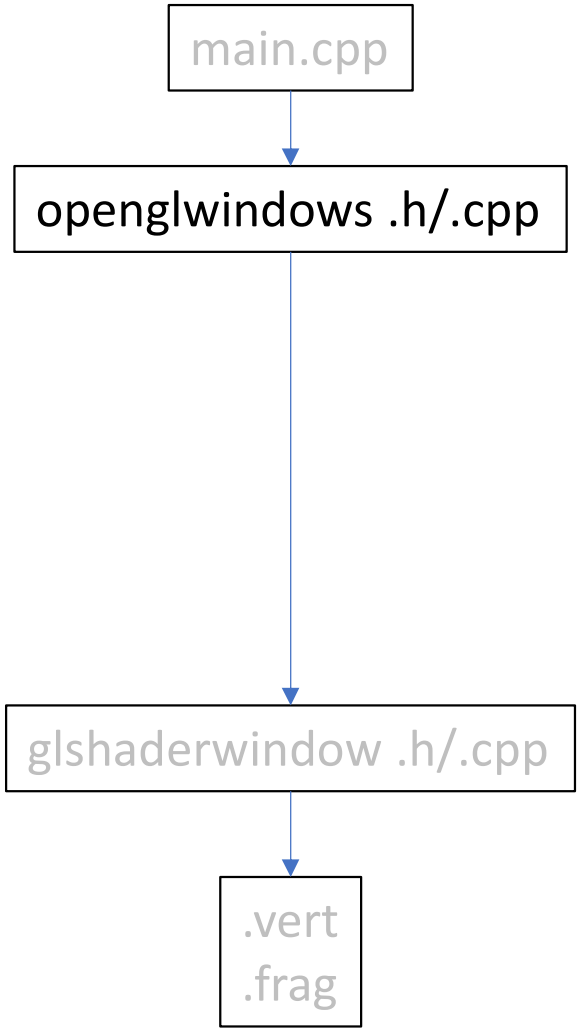
Evènements utilisateurs et ensemble des objets 3D, OpenGL et programs, certains évènements de rendu sont repris ici via une fonction virtuelle

Repères dans le code fourni pour le projet SIA



On retrouve le lancement d'une application par *QApplication* et une fenêtre de rendu via la classe *glShaderWindow* qui dérive de la classe *OpenGLWindow* créée. On commencera donc la description par cette dernière classe. On trouve aussi dans le fichier main.cpp la création de menu

Repères dans le code fourni pour le projet SIA



Cette classe implémente une *QWindow* et non pas une *QWidget* ni une *QOpenGLWidget*. Cela signifie que les mécanismes classiques **InitializeGL** / **PaintGL** ne seront pas présents. Cette classe utilise juste l'évènement **exposeEvent** qui signale automatiquement que l'affichage doit être modifié et donc déclenche un appel au rendu. Elle crée aussi un contexte OpenGL et un canevas d'affichage (DevicePainter). Il est suggéré dans le complément de code pour l'animation d'ajouter la gestion d'un flag pour l'animation. Il sera utilisé dans la classe dérivée *glShaderWindow*. De même, l'implémentation de l'initialisation et du rendu sont relayés à cette classe *glShaderWindow* via des fonctions virtuelles.

Repères dans le code fourni pour le projet SIA

Cette classe contient tous les éléments du projet. Les fonctions principales sont :

initialize()
Est appelé une seule fois à l'issue du tout premier rendu. L'état OpenGL est initialisé, les objets pour les programs shader sont créés et les VBO sont aussi créés. A noter qu'il est fait aussi usage de VAO (Vertex Array Object). Ils permettent d'encapsuler l'activation et l'alignement de plusieurs VBO. Ils sont utiles quand on a plusieurs objets 3D à dessiner et donc plusieurs VBO à gérer. Dans le cas précédent, il n'y avait qu'un seul couple de VBO vertex/texcoord et indices à gérer, donc pas de nécessité de VAO. Via les VAO, au moment du rendu, il suffit d'activer par **bind()** un VAO sans avoir à activer les différents VBO.

bindSceneToProgram()
Cette procédure fait le transfert des données 3D stockées en CPU vers les VBO et spécifie l'alignement des données vis-à-vis des program shaders. Comme il a déjà été dit, l'objet associé au sol est un bon exemple de construction procédurale de géométrie à transférer au VBO.

render()
C'est la procédure principale de rendu, appelé via la classe OpenGLWindow. Dans le cas simple du sol, il y a juste une activation du shader program, un positionnement de ses paramètres, une activation du VAO (plus besoin de spécifier l'alignement des VBO) et un appel à **glDrawElements()**.

timerEvent()
Cette classe peut typiquement être utilisée pour gérer l'avancement de l'indice de trame dans le déroulement de l'animation et lancer le calcul des nouvelles positions à transférer au rendu.