Modélisation et Synthèse d'Image
TP1 : Prise en main d'*OpenGL*

Introduction

L'objectif du TP est de prendre en main les outils indispensables à la pratique de l'informatique graphique, à savoir :

- la gestion de fenête et la création de contexte *OpenGL* avec *GLFW*
- la synthèse d'image 2D et 3D en temps réel avec *OpenGL* (et sa librairie utilitaire associée : *GLEW*)
- la géométrie dans l'espace (calcul matriciel et vectoriel) avec *GLM*

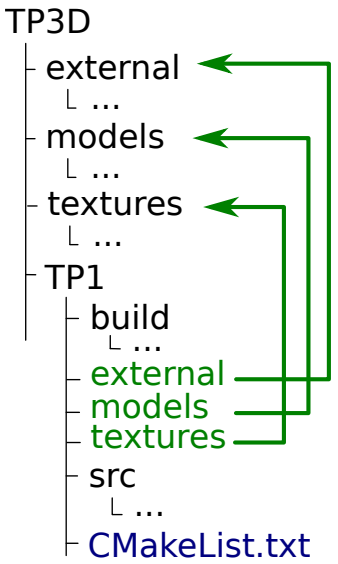
La base de code qui vous est fournie gère la création de la fenêtre et la création du contexte *OpenGL* (dans "main.cpp"). Pour la récupérer, allez à l'adresse suivate :
`http://romain.vergne.free.fr/blog/?page_id=228`.

Tous les TPs utilisent les mêmes bibliothèques et se basent sur la même hiérachie de fichiers. Suivez donc attentivement les instructions qui suivent pour éviter d'avoir à tout refaire lors des prochaines scéances :

- créez un répertoire dédié aux TPs de synthèse d'images (`mkdir ~/TP3D`)
- accédez à ce dossier (`cd ~/TP3D/`)
- récupérez/dézippez les fichiers externes et le TP1 :

```
wget http://maverick.inria.fr/~Romain.Vergne/SI-RICM4/external1.zip
wget http://maverick.inria.fr/~Romain.Vergne/SI-RICM4/models.zip
wget http://maverick.inria.fr/~Romain.Vergne/SI-RICM4/textures.zip
unzip external1.zip && unzip models.zip && unzip textures.zip
rm -f external1.zip models.zip textures.zip TP1.zip
wget http://maverick.inria.fr/~Romain.Vergne/SI-RICM4/TP1.zip
unzip TP1.zip && rm -f TP1.zip
```

- placez vous dans le répertoire TP1 (`cd TP1`)
- créez des liens symboliques vers les fichiers externes :
  - `ln -rs ../external/`
  - `ln -rs ../models/`
  - `ln -rs ../textures/`
- créez un dossier pour la compilation (`mkdir build`)
- accedez à ce dossier (`cd build`)
- lancez cmake (`cmake ..`)
- lancez la compilation (`make`)
- exécutez (`./polytech_ricm4_tp1`)



Voici la structure que devrait avoir votre arborescence de fichiers.

Dessiner un triangle en mode immédiat

Nous allons maintenant voir comment dessiner un triangle dans la zone de rendu. L'origine de l'espace est au milieu de la fenêtre, l'axe des abscisses (X) est horizontal, l'axe des ordonnées (Y) est vertical, et l'axe des cotes (Z) pointe en dehors de l'écran (vers vous).

Différentes méthodes existent pour dessiner un triangle avec *OpenGL*. La plus simple consiste à signaler qu'on souhaite dessiner un triangle, puis à transmettre les coordonnés de ses sommets un à un, puis à dire qu'on a fini. Ce code est à mettre dans la boucle de rendu :

```
glBegin(GL_TRIANGLES);
glVertex3f(-0.5f,-0.5f,0.0f);
glVertex3f( 0.5f,-0.5f,0.0f);
glVertex3f(0.0f, 0.5f,0.0f);
glEnd();
```

Observez le résultat de cette opération. Pour ajouter de la couleur, vous pouvez utiliser la commande `glColor3f()` avant le dessin (pour un triangle monochrome) ou bien avant la transmission de chaque sommet.

Utiliser les shaders

Les shaders : Un shader est un programme qu'*OpenGL* transmet directement à la carte graphique pour effectuer des opérations très rapidement. Dans la chaîne de processus d'*OpenGL* (*pipeline*), plusieurs shaders interviennent. Les deux principaux sont le *Vertex Shader* et le *Fragment Shader*. Le *Vertex Shader* est exécuté pour chaque sommet, il est chargé de lui affecter une position. Ensuite a automatiquement lieu la rasterisation, c'est à dire la transformation du contenu vectoriel (points, segments, triangle) en fragments (pixels). Le *Fragment Shader* est exécuté pour chaque fragment, il est chargé de lui affecter une couleur.

***GLSL* :** Les shaders sont programmés dans un langage appelé *GLSL* (*OpenGL* Shading Language) qui fait partie de *OpenGL* et qui ressemble au *C*. Les shaders sont compilés lors de l'exécution du programme principal. Leur extension est arbitraire ; nous utiliserons ".glsl" par clareté. La documentation de *GLSL* v.120 est disponible ici : `https://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf`

Le chargement des shaders se fait dans la fonction `LoadShaders`, que vous n'avez pas besoin de comprendre pour l'instant (sauf si ça vous fait plaisir). Le seul point à retenir est qu'une fois les shaders chargés, compilés et linkés on récupère un identifiant. Cet identifiant nous permet d'accéder à notre shader program (composé des deux shaders), et de dire à *OpenGL* de l'utiliser avant de procéder au dessin.

Vertex Shader : Créez un fichier "vertex.glsl" dans un nouveau dossier nommé "shader/" situé à la racine du projet. Ceui-ci devra contenir :

```
// Version d'OpenGL
#version 120

// Fonction appelee pour chaque sommet
void main() {
  // Affectation de la position du sommet
  gl_Position = ftransform();
}
```

Ce *Vertex Shader* affecte à chaque sommet la position définie dans "main.cpp", grâce à la fonction `ftransform()` qui fait partie de GLSL. `vec3` est un vecteur de 3 composants dans *GLSL*. Ce n'est pas un `glm::vec3` !

Fragment Shader : Créez un fichier "fragment.glsl" dans le dossier "shader/" contenant :

```
// Version d'OpenGL
#version 120

// Fonction appelee pour chaque fragment
```

```
void main() {
  // Affectation de la couleur du fragment
  gl_FragColor = vec4(1.,0.,0.,1.);
}
```

Ce Fragment Shader impose la couleur rouge à tout les fragments issus de la rasterisation de la géométrie. Les couleurs sont réprésentées par leur composantes *RGB* (red, green, blue). La quatrième composante est le canal alpha (utilisé pour la transparence).

Utilisation : Les shaders sont écrits, il faut désormais les utiliser dans le code *C++*. Pour les initialiser, on fait :

```
// Compilation du shader program et generation de l'ID du Shader
GLuint programID = LoadShaders("../shader/vertex.glsl", "../shader/fragment.glsl");
```

Maintenant, avant de dessiner, il faut dire à *OpenGL* d'utiliser notre shader program :

```
// On dit a OpenGL d'utiliser programID
glUseProgram(programID);
```

Compilez et exécutez. Vous pouvez changer des paramètres dans les shaders pour obtenir des réultats différents. Note: on peut créer et utiliser autant de shaders que l'on veut (il suffit de faire un appel à "glUseProgram" avec le bon identifiant). On peut aussi aussi revenir au mode classique avec "glUseProgram(0)".

Dessiner un triangle avec un VAO+VBO(s)

Nous allons maintenant voir une méthode alternative, plus moderne et plus efficace : l'utilisation de *Vertex Buffer Object* (ou *VBO*). Un tampon (*buffer* en Anglais) est une zone en mémoire (dans notre cas, mémoire = cache de la carte graphique) que l'on va remplir de nos données avant de lancer un processus qui va accéder à ces données : le shader. Ici le processus est le rendu de l'image, et les données sont les positions des sommets du triangle (avec éventuellement des normales, des couleurs, des coordonnées de texture, ...). Ces données ne sont transmises qu'une seule fois au début du programme, et non plus à chaque fois qu'on veut dessiner le triangle comme précédemment. Dans les version récente d'OpenGL, elles sont encapsulées dans ce qu'on appelle un *VAO* (pour *Vertex Array Object*). Cela permet de n'utiliser qu'un seul object pour manipuler tous les attributs (positions, normales, etc) des sommets d'un maillage.

Pour commencer, créez un nouveau *Vertex Shader* :

```
// Version d'OpenGL
#version 120

// Donnees d'entree
attribute vec3 vertex_position;

// Fonction appellee pour chaque sommet
void main() {
  // Affectation de la position du sommet
  gl_Position = vec4(vertex_position, 1.);
}
```

Ce shader prend en entrée un paramètre : `vertex_position`, que nous allons initialiser dans "main.cpp" avec un tableau de positions. Pour ce faire, on commence par créer ce tableau avec *GLM* dans l'initialisation de "main.cpp" :

```
// Definition d'un vecteur
vec3 v(-1.0f, -1.0f, 0.0f);

// Definition d'un tableau de vecteurs... a vous de le remplir!
vec3 vertex[3];
```

Ensuite on accède à un ID nous permettant de localiser `vertex_position` :

```
// Obtention de l'ID de l'attribut "vertex_position" dans programID
GLuint vertexPositionID = glGetAttribLocation( programID,
                                               "vertex_position");
```

Puis on copie les données sur la carte graphique. Ce code est à mettre dans l'initialisation :

```
// Creation d'un VAO et recuperation de son ID
GLuint vaoID;
glGenVertexArrays(1,&vaoID);

// Definition de notre VAO comme object courant
glBindVertexArray(vaoID);

// Creation d'un VBO et recuperation de son ID
GLuint vboID;
glGenBuffers(1, &vboID);

// Definition de notre VBO comme le buffer courant
// et lie automatiquement ce VBO au VAO actif (i.e. vaoID)
glBindBuffer(GL_ARRAY_BUFFER, vboID);

// Copie de donnees vers le VBO
glBufferData(GL_ARRAY_BUFFER, sizeof(vertex), vertex, GL_STATIC_DRAW);

// On indique a OpenGL comment lire les donnees
glEnableVertexAttribArray(vertexPositionID);
glVertexAttribPointer(
    vertexPositionID,    // ID de l'attribut a configurer
    3,                   // nombre de composante par position (x, y, z)
    GL_FLOAT,            // type des composantes
    GL_FALSE,            // normalisation des composantes
    0,                   // decalage des composantes
    (void*)0             // offset des composantes
);

glBindVertexArray(0); // on desactive le VAO
```

Ensuite dans la boucle de dessin on va dire à *OpenGL* de dessiner le contenu des tampons associés au VAO :

```
glBindVertexArray(vaoID); // On active le VAO

// on dessine le contenu de tous les VBOs (buffers) associes a ce VAO
glDrawArrays(GL_TRIANGLES, 0, 3);

glBindVertexArray(0); // On desactive le VAO
```

Enfin, à la fin du programme, on libère les buffers/objets :

```
glDeleteBuffers(1, &vboID);
glDeleteBuffers(1, &vaoID);
```

Prenez le temps de comprendre et de faire marcher le programme ci-dessus puis passez à a suite.

À vous de jouer

Faites une copie d'écran de chacun de vos résultats. Vous rassemblerez ces copies d'écran dans un document qui fera office de rapport. Chaque image devra être commentée.

- Tester la fonction :

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

  Que se passe t-il ?

- Changer la couleur de fond avec `glClearColor()`
- Changer la couleur du triangle dans le fragment shader.
- Rajouter les points nécessaires pour dessiner un carré à l'aide de `GL_TRIANGLES`. Vous pouvez également essayer un cube, ou autre.
- Essayer les différents types de primitives disponibles dans *OpenGL*, n'hésiter pas à modifier les sommets, à en rajouter ou en enlever pour mener à bien vos tests. La documentation se trouve sur le site `http://www.opengl.org/sdk/docs/man2/`.

  1. `GL_POINTS`
  2. `GL_LINES`
  3. `GL_LINE_STRIP`
  4. `GL_LINE_LOOP`
  5. `GL_TRIANGLE_STRIP`
  6. `GL_TRIANGLE_FAN`

  Quel est l'effet de chaque fonction ?

- Utiliser `glEnable(GL_VERTEX_PROGRAM_POINT_SIZE)` dans le .cpp et `gl_PointSize=2.0f` dans le vertex shader.
- Dans le vertex shader, rajouter à la fin : `gl_Position.x /= 2`. Expliquer le résultat. Essayez une transformation matricielle.

Information :

La dernière version d'*OpenGL* est *OpenGL* 4.5, qui va avec *GLSL*440. Dans les dernières version, il est obligatoire d'utiliser les shaders et les VAOs/VBOs, d'où l'importance de bien comprendre ces objets et la manière dont ils fonctionnent. Nous ferons encore quelques modifications dans les prochains TPs pour rendre votre code compatible avec les derniers standards.

Organisation

Rendre une archive *nom1_ nom2.zip* contenant :

- Le code commenté prêt à être compilé/executé et générant les sorties demandées.
- Un rapport contenant vos résultats (réponses, images, commentaires, ...).

À envoyer par mail à `romain.vergne@inria.fr` avec "[RICM4-TP1] Nom1 Nom2" en objet.