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 :

- Un gestionnaire de fenêtre : GLFW
- Une librairie graphique : OpenGL (GLEW)
- Une librairie de mathématiques pour l'informatique graphique : GLM

Un code vous est fournie. Dans le dossier src, utilise CMake dans un dossier temporaire pour générer un Makefile. Ensuite on compile normalement :

```
mkdir build
cd build
cmake..
make
```

Travail demandé

En partant du code fournie dans *main.cpp*, suivez les intructions :

Création d'une fenêtre avec GLFW

On commence par initialiser GLFW :

```
//Initialise GLFW
if( !glfwInit() )
{
        std::cout << "Failed to initialize GLFW" << std::endl;
    exit(EXIT_FAILURE);
}
```

Ensuite on peut créer la fenêtre :

```
glfwOpenWindowHint(GLFW_FSAA_SAMPLES, 4); // Anti Aliasing
glfwOpenWindowHint(GLFW_OPENGL_VERSION_MAJOR, 2); // OpenGL 2.1
glfwOpenWindowHint(GLFW_OPENGL_VERSION_MINOR, 1);

// Open a 1024x768 window and create its OpenGL context
if( !glfwOpenWindow( 1024, 768, 0,0,0,0, 32,0, GLFW_WINDOW ) )
{
        std::cout << "Failed to open GLFW window" << std::endl;
        glfwTerminate();
        exit(EXIT_FAILURE);
}

// Initialize GLEW
if (glewInit() != GLEW_OK)
{
    std::cout << "Failed to initialize GLEW" << std::endl;
    exit(EXIT_FAILURE);
}

// Set a title
glfwSetWindowTitle( "Polytech RICM 4" );
```

```
//Get OpenGL version informations
const GLubyte* renderer = glGetString (GL_RENDERER);
const GLubyte* version = glGetString (GL_VERSION);
std::cout << "Renderer : " << renderer << std::endl;
std::cout << "OpenGL version supported : " << version << std::endl;
```

Compiler et exécuter le code. Une fenêtre doit s'ouvrir et aussitôt se refermer. Il suffit d'attendre que l'utilisateur tape la touche *Echap* pour fermer la fenêtre.

```
//Enable GLFW to receive key press
glfwEnable( GLFW_STICKY_KEYS );
do {
        //Swap buffers
        glfwSwapBuffers();
}  //Check if ESC key is pressed or window is closed
while( glfwGetKey( GLFW_KEY_ESC ) != GLFW_PRESS
          && glfwGetWindowParam( GLFW_OPENED ) );
//Close OpenGL and terminate GLFW
glfwTerminate();
```

Compiler et exécuter le code. Maintenant la fenêtre reste ouverte.

Dessiner un triangle avec OpenGL

L'origine de l'écran est au milieu, l'axe des abscisses X est horizontal, l'axe des ordonnées Y est vertical. Définir un triangle à l'aide de la librairie glm. Un exemple d'utilisation de glm :

```
//To define a vertex
vec3 v(-1.0f, -1.0f, 0.0f);
//To define an array of 3 vertices
vec3 vertex[3];
```

Ensuite il s'agit d'envoyer le triangle à openGL. Pour cela on commence par créer un tampon (*buffer*). Ce buffer ou *vertex buffer object* permet d'envoyer des données (sommets, couleurs, normales, ...) sur la carte graphique via openGL et de garder un certain contrôle dessus. Ce code est à mettre juste avant la boucle de dessin :

```
// Get an id for our vertex buffer
GLuint vbo;
//Generate an empty buffer whose id is vbo
glGenBuffers(1, &vbo);
//Set the buffer vbo as the current buffer in openGL state machine
glBindBuffer(GL_ARRAY_BUFFER, vbo);
//Copy points into the current buffer
glBufferData(GL_ARRAY_BUFFER, sizeof(vertex), vertex, GL_STATIC_DRAW);
```

Ensuite dans la boucle de dessin on va dessiner le triangle :

```
//Clear the screen
glClear(GL_COLOR_BUFFER_BIT);
//Enable a vertex attribute with an id equal to 0
glEnableVertexAttribArray(0);
//Define a layout for the attribute 0 and associate it our vbo
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer(
        0, //attribute (as before)
        3, //vertex component's number (x,y,z)
        GL_FLOAT, GL_FALSE,      //type and normalized?
        0, (void*)0      // stride and array buffer offset
);
```

```
//Draw the triangle : 3 indices starting at index 0 -> 1 triangle
glDrawArrays(GL_TRIANGLES, 0, 3);
//Disable the layout
glDisableVertexAttribArray(0);
```

Enfin, juste avant *return 0*, on libère le buffer :

```
glDeleteBuffers(1, &vbo);
```

Compiler et exécuter le code. Désormais un triangle blanc est affiché. Sauf que vos ordinateurs sont un chouillat pas à jour, du coup il faudra attendre la prochaine étape pour vraiment dessiner.

Utiliser les shaders

Shaders : Un shader est un programme qu'openGL transmet directement à la carte graphique pour effectuer des opérations très rapidement. Dans la chaîne de processur d'OpenGL (*pipeline*), deux shaders interviennent, le **Vertex Shader** et le **Fragment Shader**. Le Vertex Shader est exécuté pour chaque sommet alors que le Fragment shader est exécuté pour chaque fragment. Etant donné que nous utilisons un 4x antialiasing, chaque pixel contient 4 fragment.

GLSL : Les shaders sont programmés dans un langage appelé GLSL (GL Shader Language) qui fait partie de GLSL. Les shaders sont compilés à l'exécution. L'extension n'a aucune importance (.glsl pour la clareté).

LoadShaders : Le chargement des shaders se fait dans la fonction LoadShaders que vous n'avez pas besoin de comprendre pour l'instant, à part si vous le souhaiter vraiment :). 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 programme shader composé des deux shaders.

Vertex Shader : Créer un fichier vertex.glsl dans le dossier src.

```
//Which GLSL version we are using
#version 120
//Input data
attribute vec3 vertex_position;
//Function main, called for each vertex
void main()
{
        //Set vertex position to what it was in the buffer we loaded
        //gl_Position is a built-in variables of GLSL
        gl_Position = vec4(vertex_position, 1.0);
}
```

vec3 est un vecteur de 3 composants dans GLSL. Ce n'est pas un glm::vec3 !

Fragment Shader : Créer un fichier fragment.glsl dans le dossier src.

```
#version 120

void main()
{
        gl_FragColor = vec4(1,0,0,1);
}
```

Ce Fragment Shader impose la couleur rouge aux 4 fragments de chaque pixel. Les couleurs sont réprésentées par l'encode RGB (red, green, blue). La quatrième composante est le canal alpha pour l'opacité.

Utilisation : Les shaders sont écrits, il faut désormais les appelés. Juste avant la définition des points :

```
//Dark blue background
glClearColor(0.0f, 0.0f, 0.4f, 0.0f);

//Get an id for our buffers
GLuint programID = LoadShaders("vertex.glsl", "fragment.glsl");
```

Juste avant la création du Vertex Buffer Object *vbo* :

```
//Get an id for our buffer
GLuint vertexPositionID = glGetAttribLocation( programID,
                                               "vertex_position");
```

Note : On vient de signaler au programme de shaders que l'on va utiliser un attribut "vertex_position" dans les shaders. Penser à répercuter cette information dans *glEnableVertexAttribArray*, *glVertexAttribPointer* et *glDisableVertexAttribArray*.

Maintenant dans la boucle de dessin :

```
//Use our shader
glUseProgram(programID);
//Draw triangle...
```

Compiler et exécuter. Désormais un triangle rouge est dessiné.

Note : On reviendra plus tard en détail sur le pipeline graphique et les shaders. Pour le moment il suffit de retenir que les shaders sont exécutés sur la carte graphique qui est hautement parallélisée. Le code OpenGL sert à transférer les données sur la carte graphique et à définir les outils pour les manipulers avec les shaders.

A vous de jouer

Faite 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.
- Changer la couleur dans le fragment shader.
- Dans la fonction de dessin remplacer GL_TRIANGLES par GL_POINTS.
- Utiliser glEnable(GL_VERTEX_PROGRAM_POINT_SIZE); dans le .cpp et *gl_PointSize=2.0f;* dans le vertex shader.
- Rajouter les points nécessaires pour dessiner un carré à l'aide de *GL_TRIANGLES*.
- 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 ?

– Dans le vertex shader, rajouter à la fin : *gl_Position.x /= 2.* Expliquer le résultat.

Information :

La dernière version d'OpenGL est OpenGL 4.4 et la version de GLSL correspondante est GLSL 4.30.6 (#430). On est donc bien loin des dernières versions et il y aura bien sûr quelques modifications à faire pour rendre votre code compatible avec les derniers standards. Si le matériel scolaire le permet je vous proposerais les prochains TP sur des versions plus récentes d'OpenGL. En attendant la version utilisée est tout à fait suffisante pour se familiariser avec l'esprit d'OpenGL.

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, ...).