Introduction à OpenGL

J.M. ROBERT, 11/2019

TP synthèse de l'image et du son

Le but de ce TP est de prendre en main OpenGL 3.3 et supérieur. OpenGL est une bibliothèque graphique permettant (moyennant driver) de réaliser l'affichage en utilisant les ressources tels que cartes graphiques... L'utilisation de ces ressources permet le calcul des scènes en trois dimensions en parallèle (SIMD).

Ce TP est très fortement inspiré du tutoriel suivant : `https://www.opengl-tutorial.org/beginners-tutorials/tutorial-1-opening-a-window/`

Il s'en démarque cependant pour insister sur les aspects créations de l'image (matière, lumière...).

On peut trouver des informations utiles sur le site :

`https://learnopengl.com/`

Pour chaque étape `stepX`, vous pouvez sauvegarder la version de votre code dans le répertoire correspondant de la machine virtuelle : `/OGL/playground_steps/stepX`.

1 Step 0

Voici les étapes à accomplir :

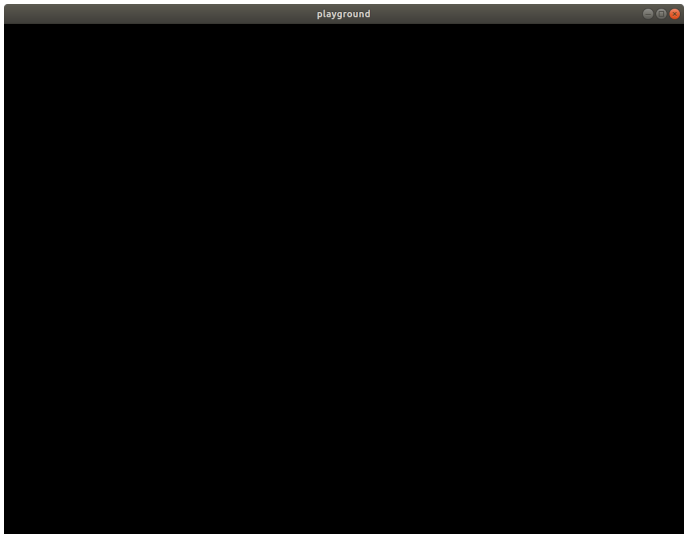
— télécharger la machine virtuelle (moodle) ;

— ouvrir un terminal et aller dans le répertoire `/OGL/playground/` ;

— exécuter le script : `$ make` ;

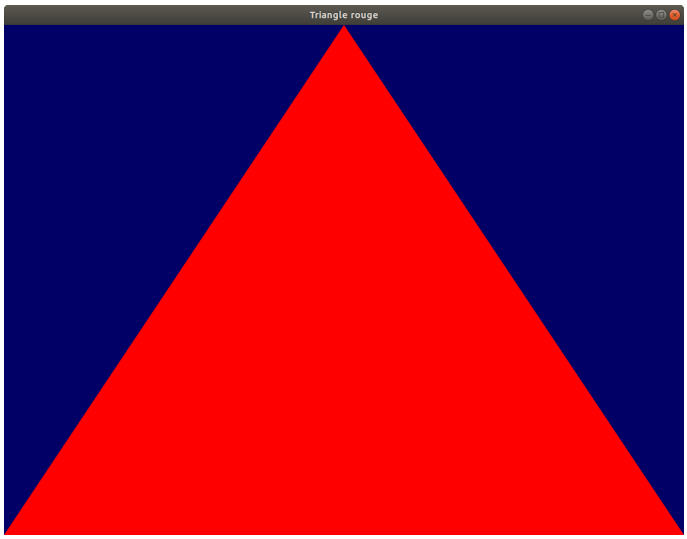
— la compilation de code source a généré un exécutable : `$ ./playground`.

Vous devez obtenir le résultat suivant :



2 Step1

L'objectif est d'afficher un triangle :



Par rapport à l'étape précédente, on a ajouté :

— la couleur de fond d'écran

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

— le buffer de *vertex* :

```
GLuint VertexArrayID;
glGenVertexArrays(1, &VertexArrayID);
glBindVertexArray(VertexArrayID);

// Create and compile our GLSL program from the shaders
GLuint programID = LoadShaders( "SimpleVertexShader.vertexshader",
                                "SimpleFragmentShader.fragmentshader" );

static const GLfloat g_vertex_buffer_data[] = {
-1.0f, -1.0f, 0.0f,
 1.0f, -1.0f, 0.0f,
 0.0f,  1.0f, 0.0f,
};

GLuint vertexbuffer;
glGenBuffers(1, &vertexbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data),
             g_vertex_buffer_data, GL_STATIC_DRAW);
```

la ligne «*Create and compile our GLSL program from the shaders*» correspond à l'utilisation du GLSL (*Shading Language*). Cela permet la gestion plus souple de certains paramètres et calculs pour les éléments ultérieurs (texture, couleurs, lumière, matière, ombres...). Ces fichers sont compilés à la volée, à chaque lancement de l'exécutable. Cette démarche peut sembler curieuse, mais c'est celle adoptée par OpenGL.

— et dans la boucle principale :

```
// Clear the screen
glClear( GL_COLOR_BUFFER_BIT );

// Use our shader
glUseProgram(programID);

// 1rst attribute buffer : vertices
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glVertexAttribPointer(
0,                  // attribute 0.
       // No particular reason for 0, but must match the layout in the shader.
3,                  // size
GL_FLOAT,           // type
GL_FALSE,           // normalized?
0,                  // stride
(void*)0            // array buffer offset
);

// Draw the triangle !
glDrawArrays(GL_TRIANGLES, 0, 3); // 3 indices starting at 0 -> 1 triangle

glDisableVertexAttribArray(0);
```

— et en fin de programme, on libère la mémoire allouée aux buffers et aux code GLSL compilé.

```
// Cleanup VBO
glDeleteBuffers(1, &vertexbuffer);
glDeleteVertexArrays(1, &VertexArrayID);
glDeleteProgram(programID);
```

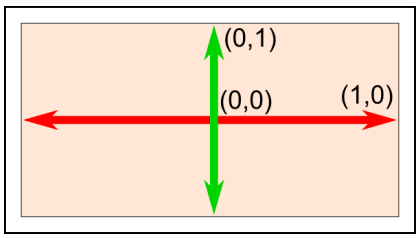
La ligne `glClear( GL_COLOR_BUFFER_BIT );` indique à la "machine à états" qu'elle remplit le buffer de pixels avec la couleur de fond prévue à cet effet.

Remarque 1. La ligne `glfwWindowHint(GLFW_SAMPLES, 4); // 4x antialiasing`, comme le commentaire l'indique, réalise l'antialiasing.

On note les coordonnées des points du triangle :

```
static const GLfloat g_vertex_buffer_data[] = {
-1.0f, -1.0f, 0.0f,
 1.0f, -1.0f, 0.0f,
 0.0f,  1.0f, 0.0f,
};
```

Ces coordonnées sont de type `float` et représentent l'écran de la façon suivante :



3 step2

On va dessiner plusieurs triangles en leur attribuant une couleur différente à chacun.

Dans les fichers de code GLSL, de nouvelles apparaissent. Par exemple, on remarque l'apparition de `layout` et les lignes suivantes :

```
    gl_Position.xyz = vertexPosition_modelspace;
    gl_Position.w = 2.0;

fragmentColor = vertexColor;
```

- La première ligne permet de préciser la position de nos triangles : les coordonnées que l'on indique seront celles du modèle (nous verrons plus tard comment on peut déplacer les objets dans la scène).
- la deuxième ligne indique un facteur d'échelle : `gl_Position.w = 2.0;`. Ceci permet de rentrer des coordonnées comprise donc entre $-2 < x, y, z < 2$ ce qui change les positions. Ainsi, les coordonnées des sommets du contours seront maintenant :

x	y	z
-1.0	-1.0	0.0
-1.0	1.0	0.0
1.0	1.0	0.0
1.0	-1.0	0.0
0.0	1.5	0.0

- la dernière ligne lit le buffer des couleurs (une par *vertex*).

Dans le code source `playground.cpp`, il faut donc un nouveau buffer pour les couleurs :

```
// One color for each vertex.
static const GLfloat g_color_buffer_data[] = {
1.0,  0.0f,  0.0f,
...
};

GLuint colorbuffer;
glGenBuffers(1, &colorbuffer);
glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_color_buffer_data), g_color_buffer_data, GL_STATIC_DRA
```

et dans la boucle principale, il faut tracer les triangles de la même manière que précédemment.
Le code des *shaders* permet d'affecter la couleur des *vertex* :

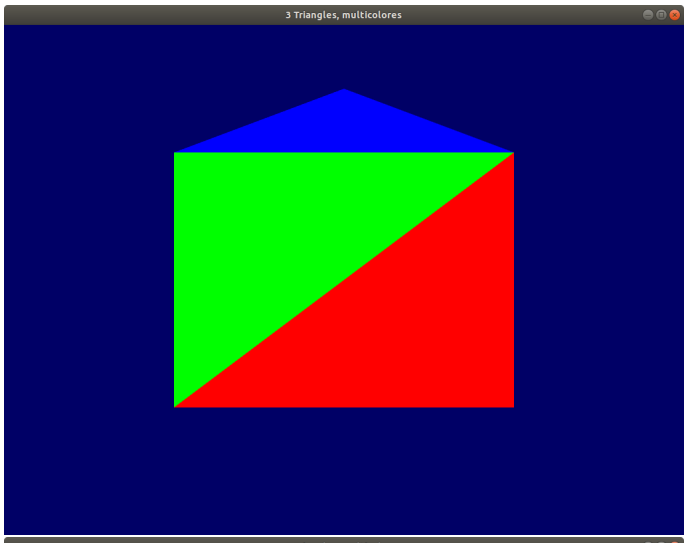
```
// 2nd attribute buffer : colors
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
glVertexAttribPointer(
1,                  // attribute 1.
              No particular reason for 1, but must match the layout in the shader.
3,                  // size
GL_FLOAT,           // type
GL_FALSE,           // normalized?
0,                  // stride
```

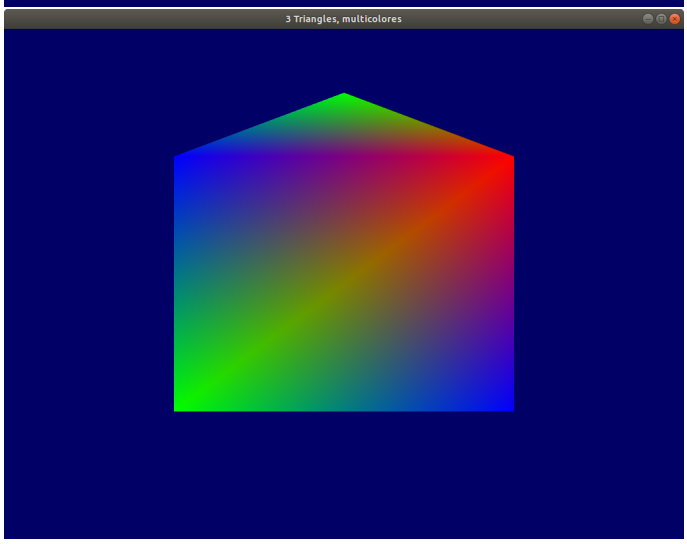
```
(void*)0            // array buffer offset
);
```

Les couleurs de chaque pixels seront reconstruites en fonctions des données de ce deuxième buffer.

Vous construirez les triangles, et le buffer des couleurs des sommets. Vous pourrez éventuellement jouer avec les couleurs des sommets : que constatez-vous ?

Exemple de ce que vous pouvez obtenir :





4 step3

On peut dessiner d'autres figures que les triangles précédents. Pour ce faire, on a d'autres possibilités que `glDrawArrays(GL_TRIANGLES, 0, 3); // 3 indices starting at 0 -> 1 triangle`

Voici les autres constantes à utiliser :

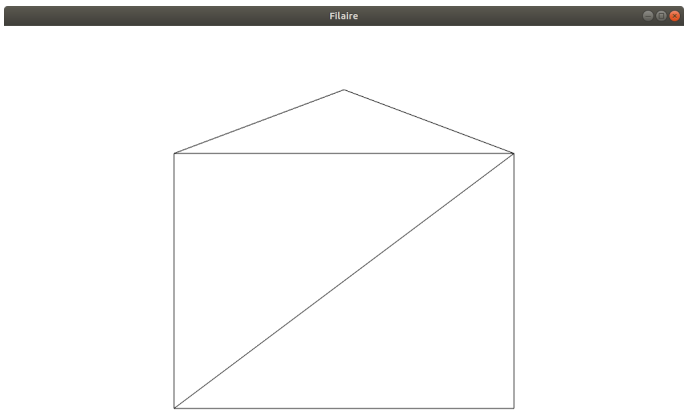
```
glDrawArrays(GL_LINES, 0, 10); // 10 indices starting at 0 -> 5 lines
glDrawArrays(GL_LINE_STRIP, 0, 10); // 10 indices starting at 0 -> 5 lines
glDrawArrays(GL_POINTS, 0, 10); // 10 indices starting at 0 -> 10 points
```

Par défaut, OpenGL trace des triangles pleins. On peut aussi demander un mode filaire. Voici les commandes :

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

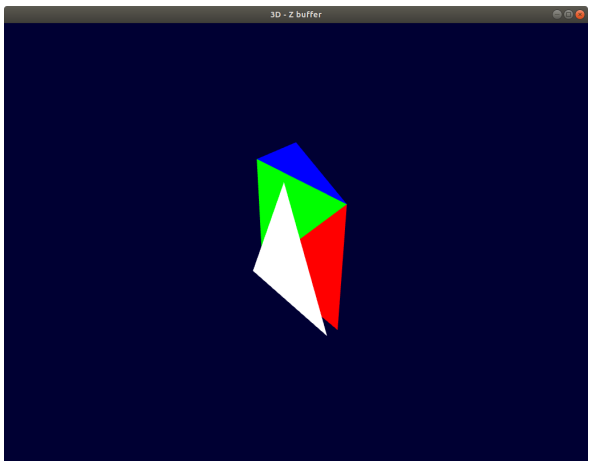
La première constante permet de dessiner en perspective sans parties cachées.

Modifier le code en deux versions pour obtenir le résultat suivant :



5 step4

On arrive enfin à la 3D ! On va dessiner un autre triangle, blanc, avec les coordonnées du premlier trinagle rouge excepté une coordonnée z des sommets positive, par exemple 0.2. En vous aidant du site `https://learnopengl.com/Advanced-OpenGL/Depth-testing`, modifiez votre code pour arriver à la vue suivante :



Il vous faut pour cela comprendre le système de visualisation et de projection sur l'écran. Voici un exemple de code :

```
// Get a handle for our "MVP" uniform
GLuint MatrixID = glGetUniformLocation(programID, "MVP");

// Projection matrix : 45 degrees Field of View, 4:3 ratio, display range : 0.1 unit <-> 100 units
glm::mat4 Projection = glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, 100.0f);
// Or, for an ortho camera :
//glm::mat4 Projection = glm::ortho(-10.0f,10.0f,-10.0f,10.0f,0.0f,100.0f); // In world coordinates

// Camera matrix
glm::mat4 View       = glm::lookAt(
glm::vec3(4,3,3), // Camera is at (4,3,3), in World Space
glm::vec3(0,0,0), // and looks at the origin
glm::vec3(0,1,0)  // Head is up (set to 0,-1,0 to look upside-down)
   );
// Model matrix : an identity matrix (model will be at the origin)
glm::mat4 Model      = glm::mat4(1.0f);
// Our ModelViewProjection : multiplication of our 3 matrices
glm::mat4 MVP        = Projection * View * Model;
    // Remember, matrix multiplication is the other way around
```

La matrice du modèle est l'identité (la position du modèle dans l'espace est inchangée). La matrice de vue est définie par la fonction `glm::lookAt()` avec les bons paramètres. La matrice de projection défini le plan de l'écran et l'ouverture de l'angle d'observation. Vous trouverez les indications sur les sites suivants :
`https://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/#the-model-matrix`
`https://learnopengl.com/Getting-started/Coordinate-Systems`

Dans la boucle principale, il faut transmettre la matrice de transformation au code du shader, afin d'en tenir compte :

```
// Send our transformation to the currently bound shader,
// in the "MVP" uniform
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
```

On pourra jouer avec les fonctions suivantes :

```
// Enable depth test
glEnable(GL_DEPTH_TEST);
// Accept fragment if it closer to the camera than the former one
glDepthFunc(GL_LESS);
```

et

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

En changeant les valeurs des constantes, en commentant les lignes, vous pourrez voir l'effets sur le comportement d'OpenGL.

6 step5

On va déplacer notre objet dans l'espace. On peut déplacer tout ou partie des triangles que l'on dessine. Pour ce faire :

Mettez en œuvre :

— une translation :

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ X & Y & Z & 1 \end{bmatrix}$$

— un changement d'échelle :

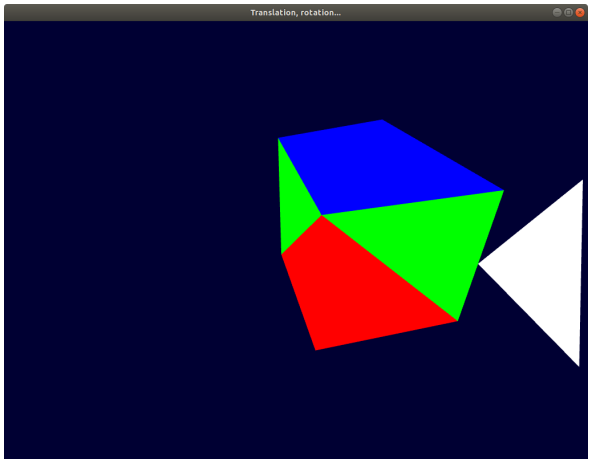
$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

— une rotation (ici, de 90 degrés d'axe Y) :

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

En conservant les matrices de vue et de projection précédentes, vous effectuez le produit $P \times V \times M$ et vous observez le résultat.

Par exemple, vous pourriez obtenir ceci (avec certains triangles précédents dessinés deux fois, ayant été déplacés une fois) :



7 step6

Dans tous les programmes précédents, on utilise une boucle principale qui redessine sans cesse la même scène, jusqu'à l'interruption par la touche [ESC] ou la fermeture de la fenêtre. On va maintenant animer la scène en modifiant la scène à dessiner à chaque tour de la boucle principale.

On procède de la façon suivante dans la boucle principale :

— on note l'instant précis d'entrée dans la boucle :

```
float currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
```

l'extrait de code suivant montre le calcul du temps écoulé entre les deux tours de boucles successifs, c'est à dire entre les deux dessins de la scène.

— On calcule l'angle de rotation à appliquer :

```
angle += 3.14159f/2.0f * deltaTime;
```

Quelle est la vitesse de rotation du modèle dans ce cas ?

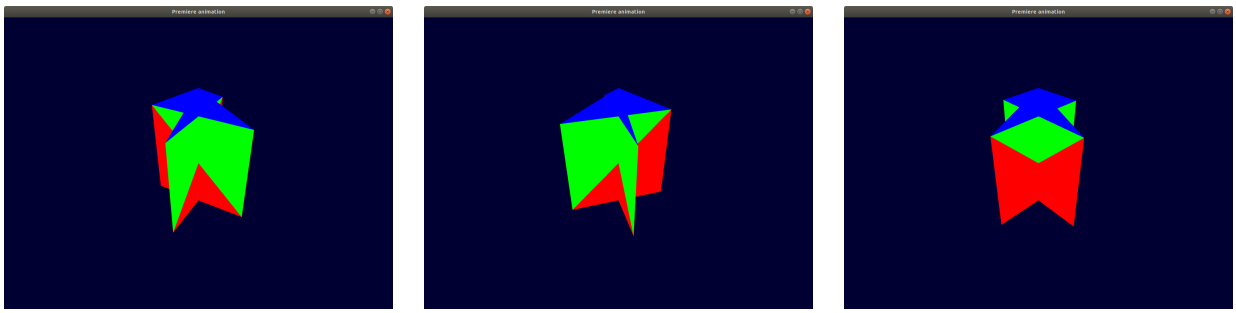
— On applique une rotation d'`angle` par rapport à la position initiale :

```
float c = (float) cos(angle);
float s = (float) sin(angle);
// Model matrix : an identity matrix (model will be at the origin)
//Model      = glm::mat4(1.0f);; // A bit to the right
Model      = {   c   ,0.0f,s,    0.0f,
0.0f,1.0f,0.0f, 0.0f,
-s  ,0.0f,c   , 0.0f,
0.0f,0.0f,0.0f, 1.0f};//glm::mat4(1.0f);*/
```

— Pour finir, on met à jour la matrice produit :

```
MVP        = Projection * View * Model;
// Remember, matrix multiplication is the other way around
```

Ici, on a appliqué ces transformations à nos triangles (qu'on a dupliqué en les tournant autour de Y à angle droit) :



8 step7

Dans notre première animation, la couleur est "plate", la lumière n'est pas définie. Pour rendre la scène plus réaliste, on va définir l'éclairement et le matériau.

Pour l'éclairement, les propriétés sont :	Pour le matériau, on peut définir :
la position de la source lumineuse ;	s'il est réfléchissant ou mat ;
la puissance d'éclairage ;	s'il est transparent ;
la couleur de la lumière ;	sa couleur naturelle
la présence d'un rayonnement diffus.	(ce que l'on voit par le rayonnement diffus).

Ces paramètres sont définis dans le fichier du *shader* : `StandardShading.fragmentshader`. Le code est suffisamment parlant et commenté.

Pour que le calcul de l'éclairement soit pris en compte, il faut définir les normales à chaque facettes pour chaque sommet (*vertex*). En conséquence, il faut rajouter un buffer supplémentaire et définir la position de la "lampe", avant la boucle principale :

```
static const GLfloat g_normal_buffer_data[] = {
 0.0f, 0.0f, 1.0f,
...
};

GLuint normalbuffer;
glGenBuffers(1, &normalbuffer);
glBindBuffer(GL_ARRAY_BUFFER, normalbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_normal_buffer_data),
             g_normal_buffer_data, GL_STATIC_DRAW);

// Get a handle for our "LightPosition" uniform
glUseProgram(programID);
GLuint LightID = glGetUniformLocation(programID, "LightPosition_worldspace");
```

Les valeurs contenues dans ce buffer correpsondent aux vecteurs normaux aux faces (triangles) pour chaque *vertex*. À vous de déterminer ces vecteurs. Et dans la boucle principale, on l'utilise de la façon suivante (attention au numéro du `layout`) :

```
// 3rd attribute buffer : normals
glEnableVertexAttribArray(2);
glBindBuffer(GL_ARRAY_BUFFER, normalbuffer);
glVertexAttribPointer(
2,                  // attribute 2.
3,                  // size
GL_FLOAT,           // type
GL_FALSE,           // normalized?
0,                  // stride
(void*)0            // array buffer offset
);
```

Il faut définir la position de la "lampe" :

```
// Get a handle for our "LightPosition" uniform
glUseProgram(programID);
GLuint LightID = glGetUniformLocation(programID, "LightPosition_worldspace");

glm::vec3 lightPos = glm::vec3(4,4,1);
glUniform3f(LightID, lightPos.x, lightPos.y, lightPos.z);
```

Ne pas oublier les `glEnableVertexAttribArray(X); //X = 0, 1 ou 2` pour chacun des buffers.

Pour le calcul de la lumière réfléchie, il faut transmettre les matrices de vues et du modèle au *shaders* :

avant la boucle principale, on identifie les matrices avec leurs homologues des *shaders* :

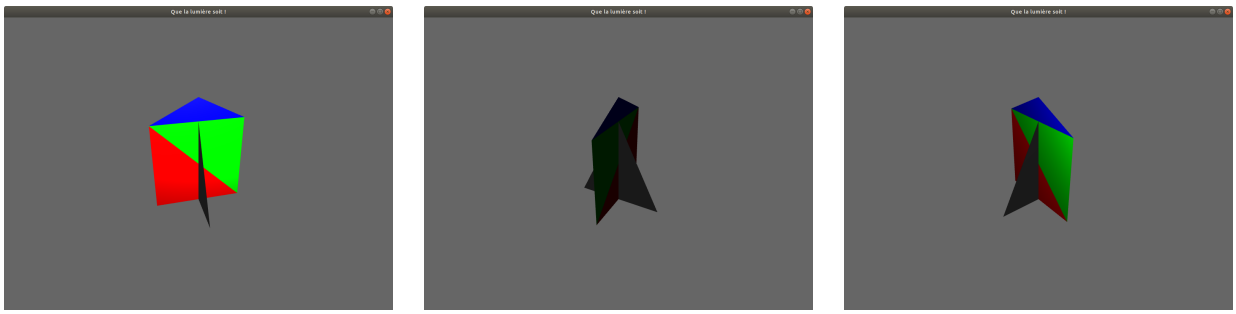
```
// Get a handle for our "MVP" uniform
GLuint MatrixID = glGetUniformLocation(programID, "MVP");
GLuint ViewMatrixID = glGetUniformLocation(programID, "V");
GLuint ModelMatrixID = glGetUniformLocation(programID, "M");
```

Puis, dans la boucle orincipale :

```
// Send our transformation to the currently bound shader,
// in the "MVP" uniform
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
glUniformMatrix4fv(ModelMatrixID, 1, GL_FALSE, &Model[0][0]);
glUniformMatrix4fv(ViewMatrixID, 1, GL_FALSE, &View[0][0]);
```

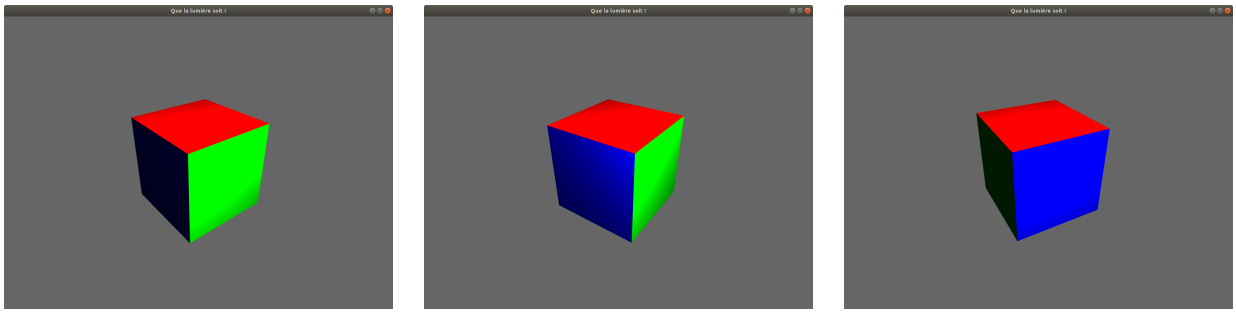
Si l'une de ces matrices est statiques, on peut faire cette transmission avant la boucle principae une fois pour toutes.

Ceci permet d'obtenir le comportement à l'éclairement souhaité :



9 step8

Dans l'étape précedente, on remarque que les faces des triangles ne sont pas rendues de la même manière. Ceci est dû à l'orientation des normales : l'éclairage est reçu selon l'orientation de ces dernières. Ici, on va juste transformer le modèle pour en faire un cube avec des normales toutes dirigées vers l'extérieur (et perpendiculairement aux faces visibles. Vous devriez savoir faire sans conseils supplémentaires :



On peut réaliser quelques expériences. Par exemple, pour vérifier l'effet miroir ou n'avoir que de la lumière diffuse.

Modifier le code de `StandardShading.fragmentshader` pour modifier les caractéristiques de la lumière (position, puissance, couleur...) et la position de la caméra et/ou de la lumière dans le code `playground.cpp`. Vous pourrez vérifier le rendu d'OpenGL en conditions extrêmes. Exemples :

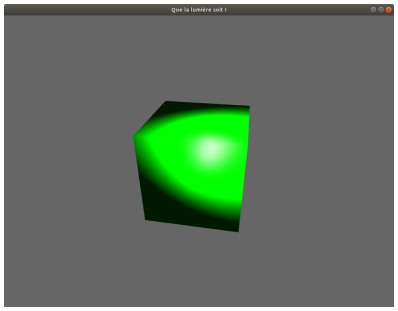
- version "miroir" : vous tracez un cube de couleur blanche et vous visualisez la "lampe" par reflet en ne rendant que la lumière spéculaire (miroir parfait) ;
- version "ambiante" : vous tracez un cube d'une couleur quelconque, et ne rendez que la lumière diffuse. Il faut forcer l'éclairage dans ce cas, mais en conditions extrêmes, vous devriez retrouver les figures primitives, sans définition de lumière.
- version "ambiante" : on peut rendre un cube mat, en supprimant la réflexion (lumière spéculaire) ;
- on peut combiner deux à deux les versions précédentes, pour avoir un aspect de surface spécifique.

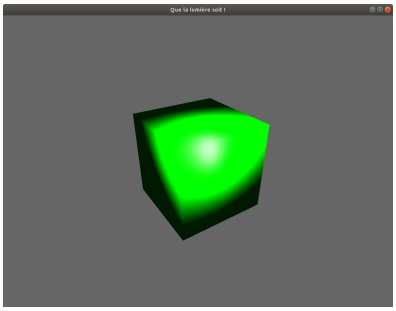
Ceci permet de comprendre expérimentalement la nature de la lumière et son effet sur les objets.

10 step9

Notre cube tourne dans l'espace. La question qui se pose : comment représente-t-on des surface avec courbure. En effet, jusqu'à maintenant, on n'a rendu que des facettes planes.

Dans cette étape, on va modifier les normales de notre cube pour voir l'effet sur le modèle. Notre cube étant centré sur l'origine, on va prendre des normales non plus perpendiculaires aux faces du cube, mais issues des sommets en partant de l'origine (le centre du cube). On trace un cube avec une couleur unique et on doit observer une visualisation "étrange" :

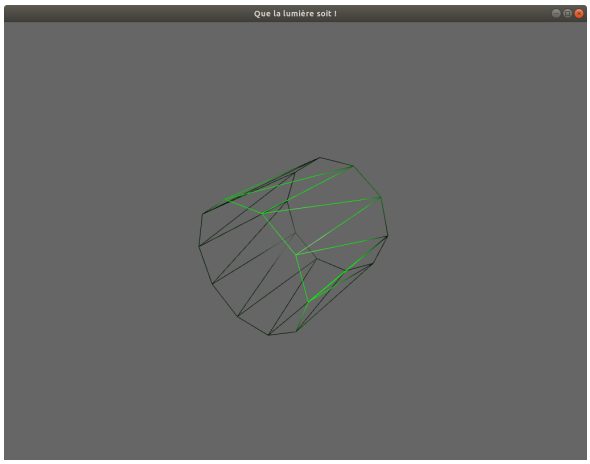




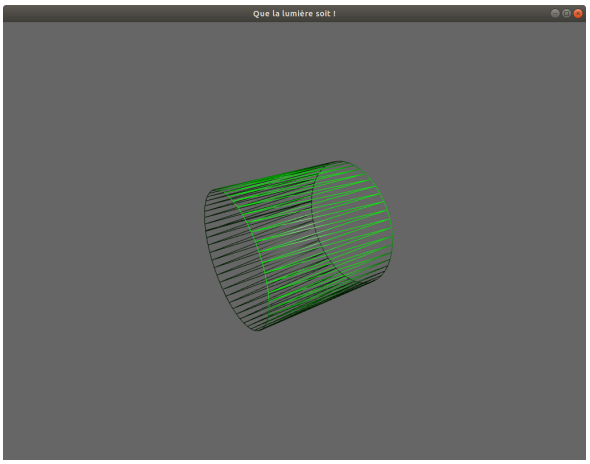
11 step10

L'étape 9 n'est qu'un intermédiaire pour tracer un objet courbe. On se propose de tracer un cylindre. Pour ce faire, il faut générer un buffer de *vertex* et de normales automatiquement.

Vous décidez d'un nombre de facettes pour discrétiser le cylindre. Voici deux exemples :



`#define N 10`



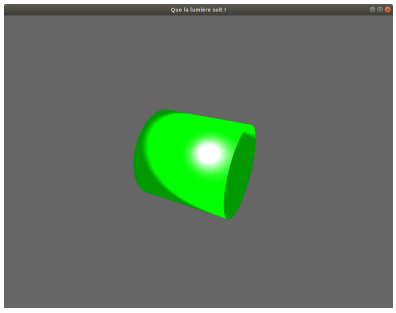
`#define N 50`

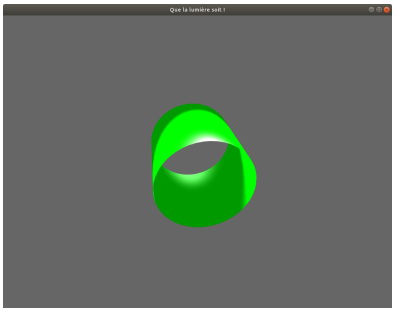
La macro `N` est le nombre de facettes. Avec un peu de trigonometrie et une boucle `for`, vous pouvez générer le buffer de *vertex* adéquat en calculant les sommets pour chaque valeur de l'angle $i * \pi/n$ avec $0 \leq i <$`N`. Les coordonnées des points sont alors :

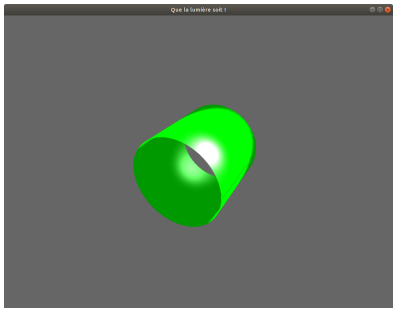
$$v_i = \begin{pmatrix} \pm 1 \\ \sin(i \cdot 2\pi/\texttt{N}) \\ \cos(i \cdot 2\pi/\texttt{N}) \end{pmatrix} ; v_{i+1} = \begin{pmatrix} \pm 1 \\ \sin((i+1) \cdot 2\pi/\texttt{N}) \\ \cos((i+1) \cdot 2\pi/\texttt{N}) \end{pmatrix}$$

Avec 4 sommets correspondant à deux valeurs de i consécutives, on peut placer dans le buffer les deux triangles formant la facette. Le buffer de normales est identique à celui de *vertex* excepté les coordonnées x qui sont à 0 (pour un cylindre d'axe Ox).

À partir de là, tracer le cylindre et l'animer se fait comme pour le cube des étapes précédentes. Vous pourrez faire varier `N` pour voir l'effet de facettisation inévitable, si ce paramètre est trop petit.







Pour finir, vous pouvez terminer cette étape en complétant le buffer de *vertex* avec les faces du cylindre (les normales sont triviales, parallèles à Ox).

Petit défi supplémentaire : saurez-vous représenter un cone, tronqué ou non ?

12 step11

L'élaboration de modèles 3D de formes simples, tels que le tube ou le cylindre, est possible quoique déjà fastidieuse. Il est donc nécessaire de penser à quelque de plus pratique ! Les modeleurs 3D de CAO (CATIA, Solidworks, AutoCAD, voire Openscad...) ou d'élaboration d'animation (Blender, Maya...) permettent l'exportation de modèles en format de fichiers (textes le plus souvent) tels que STL ou OBJ. Dans cette étape, on va utiliser un *objloader* et créer une scène utilisant les objets 3D exportés.

Le format que nous présentons ici est le format de fichier OBJ. Ce fichiers contient les différentes valeurs de façon lisibles. Le fichier comporte différentes sections avec des lignes préfixées. Ouvrons le fichier cube.obj dans un éditeur :

- ligne `v 1.000000 -1.000000 -1.000000` : `v` comme *vertex*, avec les trois coordonnées ;
- ligne `vt 0.748573 0.750412` : il s'agit de coordonnées de textures. Ce sont des coordonnées 2D, sur une image plane, à faire coïncider avec un *vertex*, comme nous le verrons plus tard.
- ligne `vn 0.000000 0.000000 -1.000000` : on indique ici une direction de normale ;
- ligne `f 5/1/1 1/2/1 4/3/1` : `f` indique une facette, c'est à dire un triangle, avec trois série de 3 nombres. Ici, `5/1/1` indique que la facette a pour sommet le $5^{\text{ème}}$ *vertex* de la liste, à utiliser avec le premier point de la liste de textures, et avec le premier vecteur dans la liste des normales. La facette se construit donc avec les trois *vertex* numéros 5,1,4, et avec la normale 1 pour les trois sommets (la facette étant plane, il est logique d'avoir le même vecteur normal pour les trois *vertex*).

L'interpréteur d'un tel type de fichier texte pour en reconstruire les buffers correspondants n'est pas d'une grande difficulté, mais d'un intérêt limité. On vous fournit donc dans le répertoire `/OGL/common/` l'*objloader* de `https://www.opengl-tutorial.org/`. Son utilisation est la suivante :

```
/**************************

Model (vertex, textures, normals)

**************************/

// Load the texture
GLuint Texture = loadDDS("uvmapRoom.DDS");

// Get a handle for our "myTextureSampler" uniform
GLuint TextureID  = glGetUniformLocation(programID, "myTextureSampler");

// Read our .obj file
std::vector<glm::vec3> vertices;
std::vector<glm::vec2> uvs;
std::vector<glm::vec3> normals;
bool res = loadOBJ("sphere2.obj", vertices, uvs, normals);

// Load it into a VBO

GLuint vertexbuffer;
glGenBuffers(1, &vertexbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
```

```
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3), &vertices[0], GL_STATIC_DR

GLuint uvbuffer;
glGenBuffers(1, &uvbuffer);
glBindBuffer(GL_ARRAY_BUFFER, uvbuffer);
glBufferData(GL_ARRAY_BUFFER, uvs.size() * sizeof(glm::vec2), &uvs[0], GL_STATIC_DRAW);

GLuint normalbuffer;
glGenBuffers(1, &normalbuffer);
glBindBuffer(GL_ARRAY_BUFFER, normalbuffer);
glBufferData(GL_ARRAY_BUFFER, normals.size() * sizeof(glm::vec3), &normals[0], GL_STATIC_DRAW
```

Comme vous pouvez les constater, on appelle simplement `loadOBJ("sphere2.obj", vertices, uvs, normals);` dans des vecteurs de vecteurs passés en paramètres, que l'on transfère directement dans les buffers correspondants `vertexbuffer`, `uvbuffer` `normalbuffer`, à utiliser ensuite de la façon habituelle.

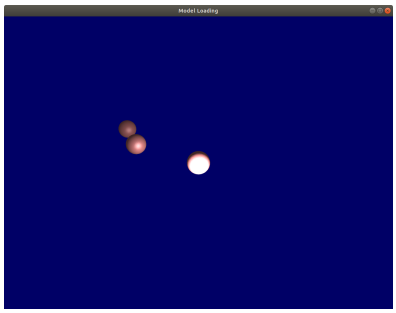
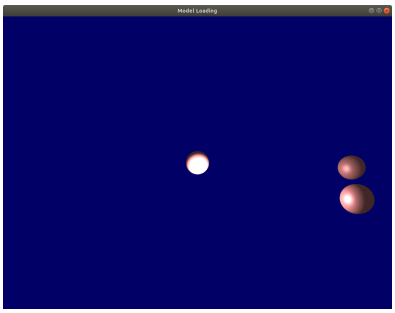
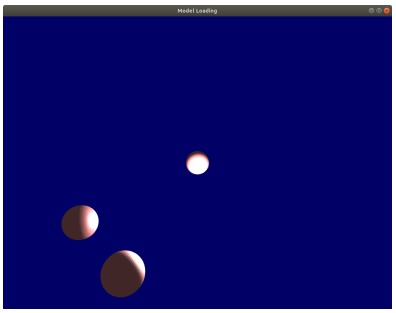
Vous pouvez donc modifier votre code et utiliser les différents modèles proposés pour les faire apparaître à l'écran et les animer comme précédemment. Il n'y a pas de photos d'exemple, cela vous laisse la surprise !

Si vous disposez déjà de modèles ou d'un modeleur capable d'exporter des modèles en format `.obj`, vous pouvez bien sûr exporter vos modèles ou en élaborer un pour la circonstances.

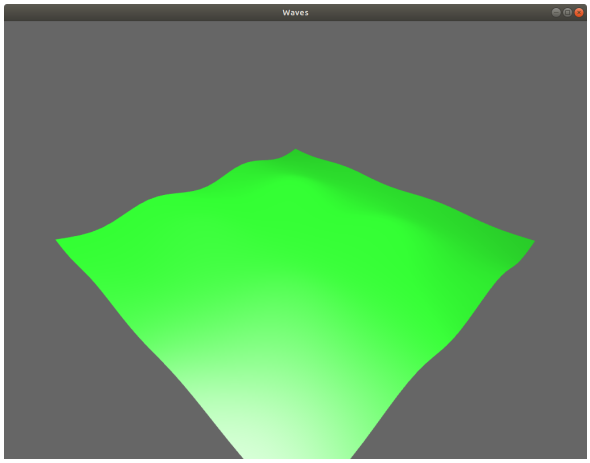
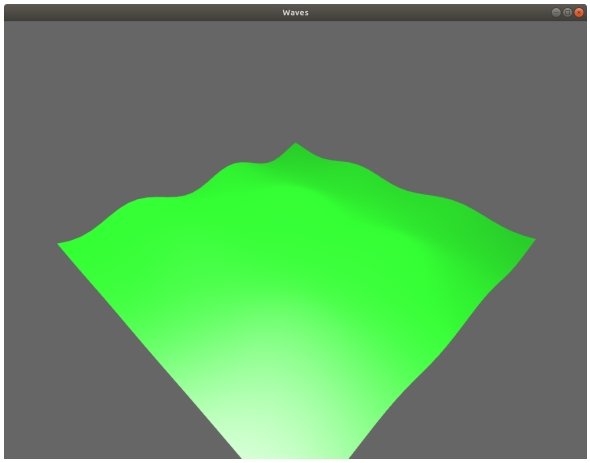
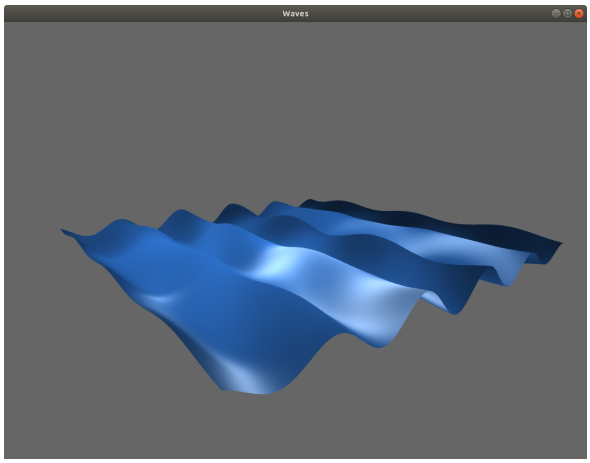
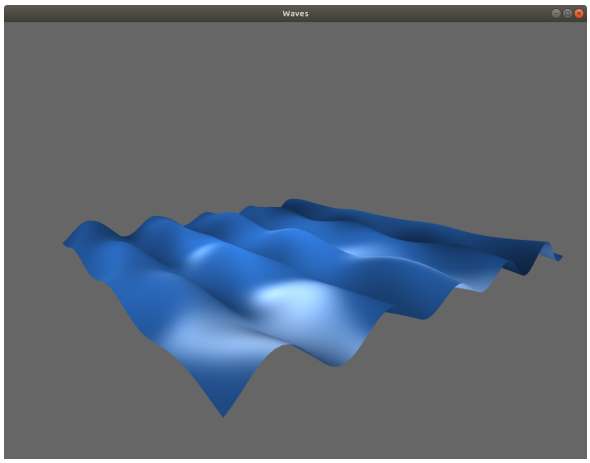
13 step12

Dans cette étape, on fait juste la synthèse de tout ce qui a été vu précédemment. On vous propose ici différentes animations que vous pouvez tenter de réaliser :

— un système planétaire (intéressant pour la combinaison des mouvements de chaque objet ; ici les trois objets sont les mêmes, mais on peut leur donner à chacun une taille différente, voire un modèle différent) ;

— faites des vagues (on modifie le buffer de vertex en temps réel) :

— en reprenant la sphère, vous pouvez la mouvoir pour simuler une trajectoire de chute libre (balistique) ou jouer au billard à trois bandes ;

— tout autre idée qui vous passe par la tête...

14 Pour aller plus loin

Il reste encore bien des choses à voir :

— OpenGL permet de simuler des *Light maps*, c'est à dire des éclairages perfectionnés (et non une simple lampe comme nous l'avons fait) ;

— OpenGL permet de calculer et donc de faire apparaître dans les scènes les ombres portées (avez-vous remarquer que dans notre système planétaire, il n'y a jamais d'éclipse de soleil ou de lune ?) ;

— OpenGL permet aussi de modéliser des matériaux aux propriétés optiques particulières (transparence notamment).

— La bibliothèque `GLFW` fournit aussi des fonctions pour les entrées clavier ou souris. Vous pouvez aussi rendre votre animation interactive.

— Nous avons utilisé OpenGL comme bibliothèque pour programmer en C/C++. OpenGL existe aussi sous la forme PyGL, WebGL...

Cette liste est loin d'être exhaustive. En espérant que ce petit tour vous aura donné envie d'approfondir.