OpenGL
et les textures.

Avant de mettre des textures sur les faces du cube, nous allons construire des classes pour les objets 3D qui chacun encapsuleront le code OpenGL (VAO, shaders et commandes de rendu) et une classe **Camera** qui encapsulera les matrices de projection et de vue.

; une classe de base **DrawableOpenGL** qui servira de prototype à toutes les autres qui en hériteront : classe **Cube** pour le cube coloré du TP1, classe **CubeTex** pour un cube avec des textures sur chaque face, classe **De**, classe **SkyBox** qui en héritent.

1. Préparation : copier la totalité du répertoire du **TP1** en **TP2** et changez le nom de TP1.pro en TP2.pro. Lancez **QtCreator** avec TP2.pro. dans TP2.pro modifiez le nom du target.

2. Classe **Camera** : ajoutez une classe Camera au projet TP2 ; ses deux méthodes principales sont *view*() et *projection*() qui renvoient toutes les deux des **glm::mat4**
   a. Pour définir la matrice *projection* les variables membres sont
      - m_fov, m_aspect_ratio, m_zmin, m_zmax, quatre floats
      - ajouter leurs accesseurs et mutateurs un clic-droit puis Refactor…
      - écrire la méthode *projection*() en utilisant *glm::perspective*(…) comme dans le TP1
   b. Pour définir la matrice *view*() on va utiliser la méthode *glm::lookAt*(…) qui prend comme arguments trois vecteurs qui représentent la position de l'observateur, la position du point ciblé par la caméra et enfin la direction qui indique le « haut »
      - Pour cela ajouter 3 nouvelles variables membres de type glm::vec3 appelées m_position, m_target et m_up avec leur accesseurs et mutateurs
      - Ecrire la méthode *view()*
   c. Le constructeur de Camera initialiser toutes ces variables membres : les floats comme dans le TP1 ; m_position sera l'origine, m_target sera (0, 0, -1) et m_up sera (0, 1, 0).

3. Classe **DrawableOpenGL** : servira de prototype à toutes les autres qui en hériteront, comme la classe **Cube** pour le cube coloré du TP1. Avec un clic droit sur le projet, choisissez **Ajouter nouveau …** puis **C++ Class C++.** Donnez le nom **DrawableOpenGL** à la classe (sans classe de base) ;
   a. La classe DrawableOpenGL doit hériter de QOpenGLFunctions_3_3_Core
   b. Variables membres (ajouter les #include nécessaires)
      - Pour encapsuler les objets OpenGL

```
QOpenGLVertexArrayObject* m_vao;
QOpenGLBuffer* m_vbo;
QOpenGLShaderProgram* m_programme;
```

      - Pour avoir accès à la caméra à utiliser pour le rendu

```
Camera* m_camera;
```

      ainsi qu'un mutateur **setCamera**()

- Pour gérer la position et l'orientation de l'objet les variables

```
glm::vec3 m_position;
GLfloat m_roulis, m_tangage, m_lacet;
```

c. Méthode

```
void initializeShader(QString vs, QString fs);
```

dans lequel il faut recopier le code qui dans le TP1 était dans la méthode *initializeShader* de **WidgetOpenGL** en utilisant vs et fs à la place des noms des fichiers contenant les shaders

d. Deux méthodes virtuelles qui seront à surcharger dans les classes qui en hériteront :
   - virtual void *initializeVAO*(){} dans lequel les classes filles mettront le code pour initialiser VBO et VAO (et appeler initializeShader)
   - virtual void *draw*(){} dans lequel il y aura le code du rendu

4. Classe **Cube** : ajoutez de la même façon que plus haut une classe **Cube** qui cette fois hérite de **DrawableOGL**.
   a. Dans cube.h ajoutez
      - **l'include** de drawableogl.h
      - la déclaration de la méthode **initializeVAO** et de la méthode **draw**
   b. Définition du constructeur : on appelle le constructeur de la classe parent
   c. Définition de **initializeVAO**()
      - Appel de **initializeOpenGLFunctions()**
      - Appel de initializeShader(":/shaders/simple.vert", ":/shaders/simple.frag");
      - Copie du code à partir de la méthode **initializeVAO** de **WidgetOpenGL**
   d. Définition de **draw**()
      - Dans la définition de **draw** de la classe **Cube** on doit déplacer, à partir de WidgetOpenGL.cpp toute la partie de **paintGL** qui fait le rendu du VAO, sauf que les matrices *vue* et *proj* sont obtenues à partir de *m_camera*
      - Pour la matrice *model* :

```
model = glm::translate(glm::mat4(1.0f), m_position);
model = glm::rotate(model, glm::radians(m_tangage),
                                        glm::vec3(1.0f,0.0f,0.0f));
model = glm::rotate(model, glm::radians(m_lacet),
                                        glm::vec3(0.0f,1.0f,0.0f));
model = glm::rotate(model, glm::radians(m_roulis),
                                        glm::vec3(0.0f,0.0f,1.0f));
model = glm::translate(model, glm::vec3(-0.5f, -0.5f, -0.5f));
```

   e. Dans **WidgetOpenGL.h** il faut
      - inclure cube.h et camera.h
      - ajouter une variable membre privée de type **Cube*** appelée **m_cube**
      - ajouter une variable membre privée de type **Camera*** appelée **m_camera**
   f. Dans la méthode **initializeGL** de **WidgetOpenGL.cpp** nettoyé :
      - Initialiser la variable m_camera
      - Initialiser la variable m_cube
      - Appeler la méthode initializeVAO de m_cube
      - Passer la camera à m_cube
   g. Dans **paintGL**, après le glClear il n'ya plus qu'à appeler **m_cube.draw()** pour faire le rendu

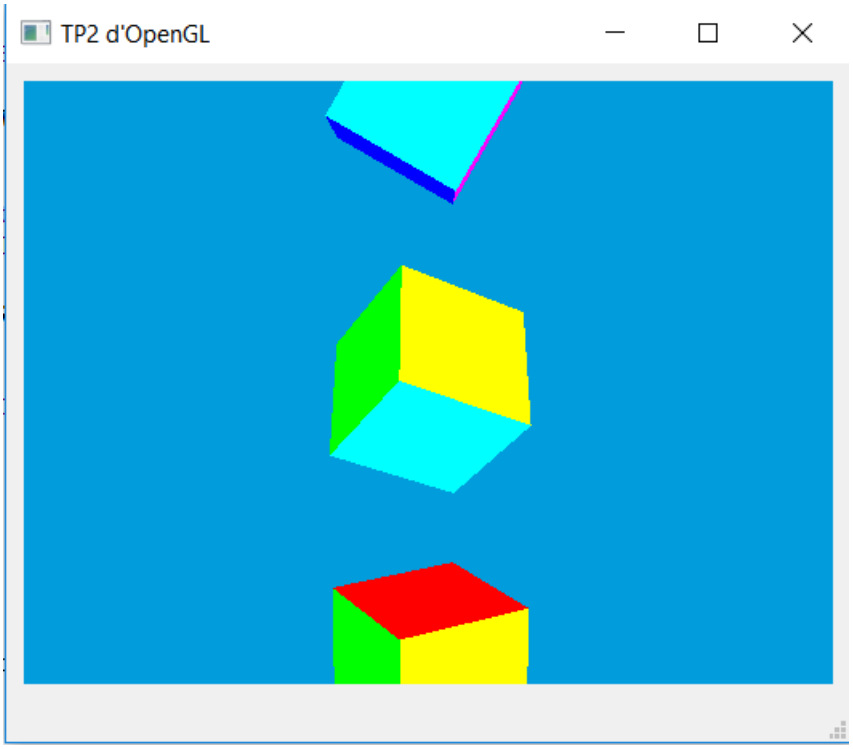
5. Grâce à la classe **Cube** on peut maintenant facilement placer plusieurs cubes dans la scène. Mais pour l'instant tous les cubes vont être placés au même endroit ; on doit donc

ajouter à **DrawableOpenGL** des accesseurs à déclarer et définir :

- void setPosition(GLfloat x, GLfloat y, Glfloat z)
- void setOrientation(GLfloat roulis, GLfloat tangage, GLfloat lacet)

a. Dans la méthode *initializeGL*() de **WidgetOpenGL** après avoir appelé **initializeVAO** de m_cube il faut le placer et l'orienter avec **setPosition** et **setOrientation**

b. Vous pouvez placer 3 cubes ou plus dans votre scène (la meilleure solution étant de définir un tableau – ou une liste – de DrawableOpenGL*) qu'il faut se rappeler de définir, initialiser, placer/orienter et rendre.



c. Pour les faire tourner il faut ajouter une méthode **tourne**(d1, d2, d3) dans **drawableOpenGL** qui incrémente les angles d'Euler correspondant de d1, d2 et d3. Cette méthode sera appelée dans **update**().

6. Cube texturé : copiez le fichier **cube.cpp** et **cube.h** en **cubetex.cpp** et **cubetex.h**. Ajoutez ces deux fichiers au projet et modifier le nom de la classe de **Cube** en **CubeTex** dans les deux fichiers. Nous allons modifier **CubeTex** pour obtenir un cube coloré aux faces texturées.

a. Dans **cubetex.h**

- Ajoutez une variable membre privée de type **QOpenGLTexture*** appelée **m_texture** qui encapsule la texture OpenGL à coller sur les faces ;
- Ajoutez une méthode **setTexture(QString textureFilename)** qui initialise **m_texture** à partir d'une **QImage** lue dans le fichier;

```
m_texture = new QOpenGLTexture(QImage(textureFilename).mirrored());
```

b. Dans la méthode **initializeVAO** de **CubeTex**, il faut :

- Définir quatre **glm**::**vec3** de coordonnées texture appelés **NW, NE, SW, SE** qui donnent les coordonnées texture des coins d'un carré (la 3ème coordonnée à 0)
- Ajouter à la fin du tableau **vertex_data** des coordonnées texture pour chacun des 24 sommets du cube ;

- Appeler **initializeShader** avec deux nouveaux programmes de shader qui seront dans les fichiers « texture.vert » et « texture.frag » (définis plus loin).
- Les coordonnées textures mises dans le VBO seront un 3ème attribut à utiliser par le vertex shader qui portera l'identifiant 2, donc il faut en informer le programme de shader avec un nouveau **setAttributeArray** et **enableAttributeArray**.

c. Vertex shader : copier simple.vert en texture.vert et l'ajouter en tant que ressource au projet sous le préfixe *shaders* ;

- Nouvelle variable *in* pour dire que les coordonnées-texture des sommets se trouvent dans le VBO et que ce sont des vec3 :

```
layout(location=2) in vec3 coord_texture;
```

- Nouvelle variable out : qui contiendra les coordonnées textures reçues par le fragment shader qui les traitera :

```
out vec2 tex_coord;
```

- Copie du *in* dans le *out* (avec une coordonnée en moins) :

```
tex_coord = coord_texture.st;
```

d. Fragment shader : copier simple.frag en texture.frag et l'ajouter en tant que ressource au projet sous le préfixe *shaders* ;

- Nouvelle variable in pour récupérer le out du vertex shader
- Nouvelle variable uniforme qui représente la texture à échantillonner :

```
uniform sampler2D texture0;
```

- Production de la couleur du fragment en multipliant (par composante) la couleur originale par la couleur lue dans la texture :

```
fragColor = out_color * texture(texture0, tex_coord);
```

e. Dans la méthode **draw** de **CubeTex**

- Lier et activer la texture

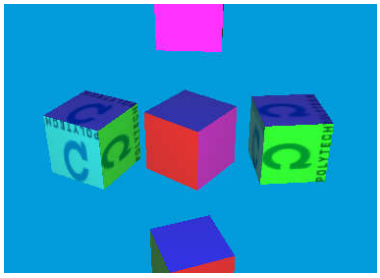
```
m_texture->setMinificationFilter(QOpenGLTexture::LinearMipMapLinear);
m_texture->setMagnificationFilter(QOpenGLTexture::Linear);

//Liaison de la texture
m_texture->bind();
//Activation de la tetxure
glEnable(GL_TEXTURE_2D);
```

- Pour passer au fragment shader la nouvelle variable uniforme

```
m_programme->setUniformValue("texture0",0);
```

En fait la texture 0 est la texture qui est actuellement bondée dans le contexte OpenGL, donc cette ligne doit apparaître après les précédentes…

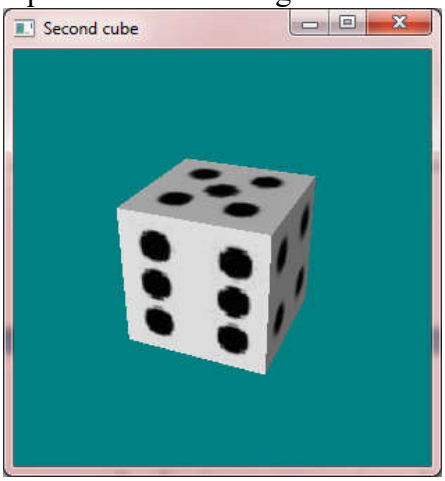
f. on ajoute aux ressources un nouveau préfixe « /images » et on ajoute le fichier contenant l'image à utiliser comme texture (par exemple **POLYTECH_RVB.jpg**). On peut donner un alias à cette ressource (par exemple « logo » que l'on pourra utiliser (avec le préfixe) comme nom de fichier (« :/images/logo ») dans **setTexture**.

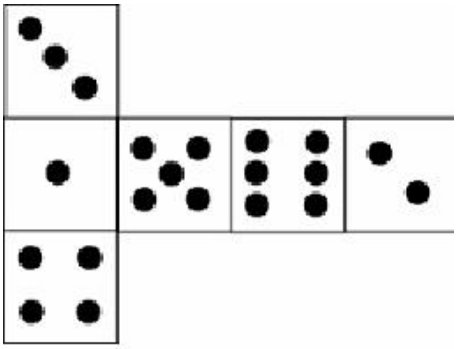


g. dans la classe **WidgetOpenGL** on ajoute au tableau de DrawableOpenGL une (ou plusieurs) instances de **CubeTex** que l'on crée et on initialise dans initializeGL comme pour les cubes avec en plus l'appel à setTexture(":/images/logo").

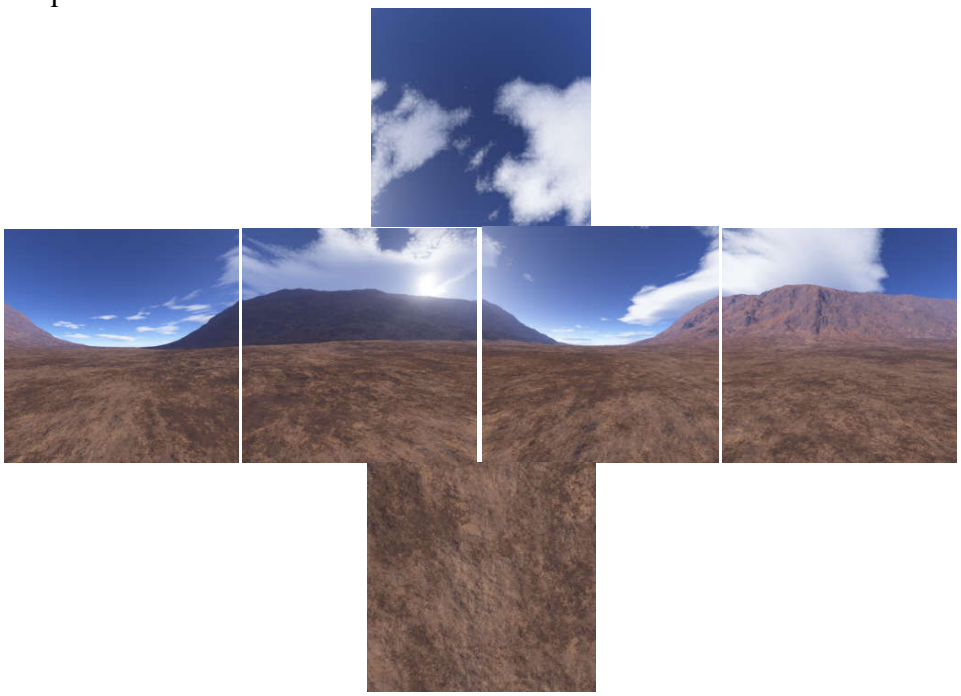
7. Toujours en copiant la classe **CubeTex** en **DeAJouer**, on veut maintenant appliquer la

texture « texture_de.JPG » qui représente les six faces d'un dé à jouer. Le code est exactement le même que dans la question 5 sauf que les coordonnées textures des points du cube doivent être modifiées (par exemple pour avoir sur la face avant le 3 les coordonnées textures seront entre 0.75 et 1 pour s et entre 0 et 0.25 pour s). Vous pouvez enlever l'information de couleur dans le VBO pour avoir une image comme celle-ci.





8. Utilisez la classe **DeAJouer** pour faire une **Skybox** : il s'agit d'un très grand cube sur les faces desquelles on met les six faces d'un décor.



Il faut utiliser **glScale**() pour appliquer une homothétie à ce cube pour qu'il soit très grand et centré en 0 comme l'observateur. Il faut aussi désactiver l'éclairage et le **culling** (**glDisable(GL_CULL_FACE)**) pour pouvoir voir les faces internes du cube.
Au final on veut donc plusieurs cubes de type différents qui tournent à l'intérieur de la skybox avec la possibilité de « regarder autour de soi » à la souris.