TSI : Synthèse d'images, OpenGL
CPE

Durée $\sim 4h$

Ce TP propose de prendre en main sous la forme d'un tutoriel la programmation de scènes virtuelles 3D en temps réelle à l'aide d'OpenGL. Dans un premier temps, le tutoriel abordera l'envoie de données sur la carte graphique. Dans un second temps, nous verrons la manipulation de shaders permettant de contrôler l'affichage. Nous verrons ensuite les différentes possibilités permettant d'améliorer son rendu et d'interagir avec celle-ci.

Ce TP se termine par un projet consistant à réaliser un jeu vidéo minimaliste en 3D afin de mettre en pratique les appels OpenGL, les shaders de rendus, et l'interaction avec l'utilisateur.

Contents

1 Lancement du programme

Question 1 *Exécutez le script* `main.py`.

Il s'agit d'un programme minimaliste utilisant le gestionnaire de fenêtres et d'événements GLFW et la bibliothèque OpenGL 3.3 pour l'affichage. Pour l'instant, seul un écran au fond bleu est affiché.

Conseil: Notez qu'à différents endroits du TP vous allez ajouter puis supprimer des lignes. Prenez l'habitude de sauvegarder vos fichiers intermédiaires avec de préférence une copie d'écran du résultat avant de supprimer des lignes que vous auriez écrite. Un gestionnaire de version, type `git` peut vous permettre de faire ces "copies". (Mettre tout en commentaire risque de rendre votre projet de moins en moins lisible).

Question 2 *Changez la couleur de fond en modifiant les paramètres de la fonction glClearColor().*

Remarque: Lorsque l'on développe un programme avec OpenGL, il arrive fréquemment que l'on ne voit pas un triangle blanc (resp. noir) sur fond blanc (resp. noir). Prenez l'habitude de prendre un fond ayant une couleur spécifique pour debugger plus facilement.

Question 3 *Observez la structure du code avec une partie d'initialisation de la fenêtre, du contexte, des données et des programmes GPU ainsi que la boucle d'affichage qui permet un affichage continu de la scène.*

Question 4 *Affichez dans le terminal le retour de la méthode* `glfw.get_time()` *dans la boucle d'affichage. Enlevez cet affichage pour la suite du TP.*

Question 5 *Combien d'images par seconde peut-on espérer?*

Question 6 *Changez la couleur du fond en fonction du temps.*

2 Affichage du premier triangle

Attention: Le triangle n'apparaîtra qu'à la fin de cette section.

2.1 Création et envoie d'un programme GPU

2.1.1 Principes de bases

La fonction `init_program()` est une fonction qui est appelée une fois en début de programme. Elle permet de créer le programme GPU qui sera utilisé dans notre code (on pourrait en avoir plus d'un).
Le programme GPU permet de programmer certaines étapes du pipeline graphique qui, résumé très rapidement, est :

1. vertex shader : programmable → positionne les sommets sur la fenêtre
2. fragmentation : non programme → divise les triangles en fragments
3. fragment shader : programmable → colorie chaque fragment pour colorer la fenêtre

Le processus pour créer le programme GPU est le suivant :

1. création et compilation d'un vertex shader

2. création et compilation d'un fragment shader
3. création d'un programme gpu lié aux précédents shader

Les fonctions suivantes permettent d'effectuer les différentes étapes en vérifiant le bon déroulement.

```
def compile_shader(shader_content, shader_type):
  # compilation d'un shader donné selon son type
  shader_id = GL.glCreateShader(shader_type)
  GL.glShaderSource(shader_id, shader_content)
  GL.glCompileShader(shader_id)
  success = GL.glGetShaderiv(shader_id, GL.GL_COMPILE_STATUS)
  if not success:
    log = GL.glGetShaderInfoLog(shader_id).decode('ascii')
    print(f'{25*"-"}\nError compiling shader: \n\
      {shader_content}\n{5*"-"}\n{log}\n{25*"-"}')
  return shader_id

def create_program( vertex_source, fragment_source):
    # creation d'un programme gpu
    vs_id = compile_shader(vertex_source, GL.GL_VERTEX_SHADER)
    fs_id = compile_shader(fragment_source, GL.GL_FRAGMENT_SHADER)
    if vs_id and fs_id:
        program_id = GL.glCreateProgram()
        GL.glAttachShader(program_id, vs_id)
        GL.glAttachShader(program_id, fs_id)
        GL.glLinkProgram(program_id)
        success = GL.glGetProgramiv(program_id, GL.GL_LINK_STATUS)
        if not success:
            log = GL.glGetProgramInfoLog(program_id).decode('ascii')
            print(f'{25*"-"}\nError linking program:\n{log}\n{25*"-"}')
        GL.glDeleteShader(vs_id)
        GL.glDeleteShader(fs_id)
    return program_id

    def create_program_from_file(vs_file, fs_file):
    # creation d'un programme gpu à partir de fichiers
    vs_content = open(vs_file, 'r').read() if os.path.exists(vs_file)\
     else print(f'{25*"-"}\nError reading file:\n{vs_file}\n{25*"-"}')
    fs_content = open(fs_file, 'r').read() if os.path.exists(fs_file)\
     else print(f'{25*"-"}\nError reading file:\n{fs_file}\n{25*"-"}')
    return create_program(vs_content,   fs_content)
```

Question 7 *Ajoutez ces fonctions à votre code, une compréhension de l'idée générale est un plus.*

2.1.2 Vertex Shader

Le vertex shader est un programme gpu qui est appelé, en parallèle, pour chaque sommet à afficher lors d'un appel à `glDraw...()`. Dans le cas d'un unique triangle, il est donc appelé 3 fois. Il a pour rôle premier d'affecter à la variable gl_Position représentant la position du sommet courant dans l'espace écran normalisé. Ce vecteur est en 4D pour permettre les projections perspectives (à voir en majeure Image).
Le code du vertex shader se trouve dans le fichier `shader.vert`. Il correspond à un nouveau langage: le GLSL (OpenGL Shading Language), ce n'est ni du C, ni du C++, mais il y ressemble

fortement et propose par défaut un ensemble de fonctions et types utiles (vecteurs, matrices, etc). Par exemple, un vecteur à `3` dimensions sera désigné par `vec3`, et un vecteur à `4` dimensions sera désigné par `vec4`.

Attention, le code de ces fichiers n'étant pas exécuté par le processeur mais par la carte graphique. Il n'est donc pas possible de réaliser d'affichage texte dans ces fichiers. Faites donc particulièrement attention, le debug de ces fichiers est généralement difficile.

Dans le cas du fichier fourni, la valeur de `position` est affecté à gl_Position. `position` est une variable d'entrée (`in`) récupérée dans *vertex shader*. Cette variable contient, dans notre cas, les coordonnées (dans l'espace 3D) de l'un des sommets du triangle affiché (`location=0` est utilisé pour faire le lien entre la variable et les données crées sur le CPU, voir sec.2.2). Ici, cette variable contient donc les coordonnées de l'un des trois sommets du triangle. Notez bien que le vertex shader est exécuté en parallèle sur de nombreux sommets. Dans le cas présent, votre carte graphique exécute donc en parallèle 3 vertex shaders. L'un ayant dans la variable `position` la valeur $(0, 0, 0)$, l'autre la valeur $(0, 0.8, 0)$, et le troisième $(0, 0, 0.8)$ -voir sec.2.2. Vous n'avez pas accès à la boucle réalisant ce parallélisme, et on ne peut pas prédire dans quel ordre les sommets vont être traités. Par contre, la synchronisation est réalisée pour la fragmentation lorsque les 3 vertex shaders associés à chaque sommet du triangle seront terminés.

Question 8 *Observez le vertex shader fournit.*

2.1.3 Fragment shader

La couleur de votre triangle (de chacun de ses pixels) est définie dans le fichier `shader.frag`. Le code de ce fichier dit de *fragment shader* est exécuté pour chaque pixel du triangle affiché. Il peut permettre de paramétrer finement la couleur de celui-ci. Notez que le code présent dans le fichier `shader.frag` est exécuté par la carte graphique (en parallèle pour de nombreux pixels).

Il faut obligatoirement une variable de sortie dans le *fragment shader* de type `vec4`, signalée par le mot clé `out`. Elle représentera la couleur du pixel. La variable possède 4 composantes, mais seules les 3 premières (r, g, b) nous serons utiles pour le moment. La dernière représente la transparence (α)

Question 9 *Observez le fragment shader fournit.*

2.1.4 Création et utilisation du programme GPU

Une fois le programme GPU crée, il faut indiquer que ce sera ce programme à utiliser pour afficher notre triangle. Pour cela on peut utiliser la ligne suivante :

```
GL.glUseProgram(program)
```

Question 10 *Dans le code python, modifiez le contenu de* `init_program()` *pour créer le programme GPU à partir des fonctions précédemment crées et des fichiers* `shader.vert` *et* `shader.frag` *puis indiquer que ce sera ce programme à utiliser. Rien ne s'affiche pour le moment.*

Notes Si nous avions plusieurs objets qui utilisent différents programmes GPU, il faudrait avant chaque demande d'affichage, repréciser le programme à utiliser à l'aide de la méthode glUseProgram().

2.2 Envoie des données sur la carte graphique

La fonction `init_data()` est une fonction qui est appelée une fois en début de programme. Elle permet de créer les données, de les transférer en mémoire vidéo et de spécifier au GPU comment les utiliser.
Le processus est le suivant :

1. création des données sur la RAM du CPU
2. création d'une liste d'état qui retiendra les buffers à utiliser et leurs organisations
3. envoi des données sur la VRAM
4. activation des attributs (types de donnée) à utiliser
5. configuration de l'organisation du buffer à utiliser

Pour stocker les données sur la RAM, nous utiliserons des tableaux `numpy` continus en mémoire. Ils s'utilisent de la manière suivante :

```
sommets = np.array(((0, 0, 0), (1, 0, 0), (0, -1, 0)), np.float32)
```

Question 11 *Construisez un tableau comme décrit contenant 3 sommets* $(0,0,0)$*,* $(1,0,0)$*, et* $(0,1,0)$*.*

Pour créer la liste d'état et les buffers, on utilisera ces lignes :

```
# attribution d'une liste d'état (1 indique la création d'une seule liste)
vao = GL.glGenVertexArrays(1)
# affectation de la liste d'état courante
GL.glBindVertexArray(vao)
# attribution d'un buffer de donnees (1 indique la création d'un seul buffer)
vbo = GL.glGenBuffers(1)
# affectation du buffer courant
GL.glBindBuffer(GL.GL_ARRAY_BUFFER, vbo)
```

Question 12 *Créez la liste d'état et les buffers.*

Pour envoyer les données dans le buffer courant, on utilisera cette ligne :

```
# copie des donnees des sommets sur la carte graphique
GL.glBufferData(GL.GL_ARRAY_BUFFER, sommets, GL.GL_STATIC_DRAW)
```

Question 13 *Envoyez les données de la RAM à la VRAM.*

Pour configurer la liste d'état (VAO), on utilisera ces lignes :

```
# Les deux commandes suivantes sont stockées dans l'état du vao courant
# Active l'utilisation des données de positions
# (le 0 correspond à la location dans le vertex shader)
GL.glEnableVertexAttribArray(0)
# Indique comment le buffer courant (dernier vbo "bindé")
# est utilisé pour les positions des sommets
GL.glVertexAttribPointer(0, 3, GL.GL_FLOAT, GL.GL_FALSE, 0, 0)
```

Ces dernières lignes indiquent que les données du buffer courant correspondent aux positions des sommets (expliqué plus tard pourquoi les positions) qui seront utilisés en cas de demande d'affichage.

Question 14 *Configurez la liste d'état.*

Question 15 *Envoyez les données sur la VRAM et configurez la liste d'état comme décrit précédemment.*

Notes

- Vous pouvez trouver aux liens suivants la documentation précise des fonctions glGenVertexArrays(), glBindVertexArray(), glGenBuffers(),glBindBuffer(), glBufferData(), glEnableVertexAttribArray(), glVertexAttribPointer().
- On crée deux variables `vao` et `vbo` qui sont des identifiant permettant de désigner de manière unique la liste d'état (ou **Vertex Array Object** / VAO) et le buffer de données (ou **Vertex Buffer Object** / VBO). On pourra définir plusieurs VBO et VAO dans le cas où l'on souhaite traiter plusieurs données séparément comme dans le cas où l'on a plusieurs objets. *Notez que les nom des variables sont quelconques, tout autre nom de variable conviendrait.*
- On notera que pour spécifier les données à afficher pour la suite, il suffira de choisir le VAO avec glBindVertexArray(). Dans notre cas, il n'y a qu'un objet donc il ne sera pas nécessaire de spécifier à chaque fois la liste d'état (VAO) à utiliser.
- Notons que les arguments de glVertexAttribPointer() sont les suivants:
  - `0` indique la location de la variable dans le vertex shader.
  - `3` indique la dimension des coordonnées (ici 3 pour x, y et z).
  - `GL_FLOAT` indique le type de données à lire, ici des nombres de type flottants.
  - `GL_FALSE` indique que les vecteurs ne pas normalisés
  - Le zéro suivant indique que l'on va lire les données les unes derrières les autres (il sera possible d'entrelacer des données de couleurs, normales plus tard).
  - Le dernier zéro indique le décalage à appliquer pour lire la première donnée, ici il n'y en a pas. (Plus tard, dans le cas de données entrelacées on décalera la lecture au premier élément correspondant).
- La fonction glVertexAttribPointer() ne fait que venir placer un pointeur de lecture. Elle n'envoie pas de données à la carte graphique.

Question 16 *Assurez vous que cette partie s'exécute sans erreurs (il n'y a toujours rien dans la fenêtre).*

2.3 Affichage

Intéressons nous désormais à la fonction `run()` qui lance la boucle d'affichage.

Question 17 *Copiez enfin la ligne suivante après l'effacement de l'écran (glClear()) et avant l'échange des buffers d'affichage (glfw.swap_buffers())*

```
GL.glDrawArrays(GL.GL_TRIANGLES, 0, 3)
```

glDrawArrays() réalise la demande d'affichage d'un triangle en partant du premier élément, désigné par glVertexAttribPointer() par l'intermédiaire du VAO courant, et pour 3 sommets (si nous avions 6 sommets, nous pourrions afficher 2 triangles).

Question 18 *Observez désormais l'affichage d'un triangle rouge sur l'écran lors de l'exécution.*

Remarque: Si votre triangle apparaît blanc, cela indique un problème lors de l'exécution. Il est probable que vos fichiers de shader ne soit pas lus (ex. mauvais chemin d'exécution).

Remarque: Le triangle est l'élément de base de tout affichage 3D avec OpenGL. Tous les autres objets seront formés en affichant un ensemble de triangles: un maillage.

Question 19 *Modifiez le paramètre* `GL_TRIANGLES` *de la fonction glDrawArrays() en* `GL_LINE_LOOP`.

Note: Il existe également le type `GL_LINES` qui vient lire les sommets deux à deux et trace un segment correspondant, et le type `GL_LINE_STRIP` qui vient lire les sommets à la manière de `GL_LINE_LOOP` mais sans lier le dernier élément avec le premier.
Remplacez désormais cet appel par les lignes suivantes pour obtenir une vue de votre triangle en fil de fer

```
glPointSize(5.0);
glDrawArrays(GL_POINTS, 0, 3);
glDrawArrays(GL_LINE_LOOP, 0, 3);
```

Remarque: Pour la suite du TP, on utilisera l'affichage du triangle plein (glDrawArrays(GL_TRIANGLES,...)), cependant, vous pourrez avoir intérêt à afficher par moment votre maillage en mode *fil de fer* ou *Wireframe* afin d'aider à visualiser l'organisation de vos triangles pour du debug.

3 Les Shaders

3.1 Fragment Shader

Question 20 *Écrivez une ligne quelconque dans le fichier* `shader.frag` *de manière à rendre le code incorrect. Relancez le programme, et observez à l'affichage que votre triangle s'affiche en blanc (ou autre, le résultat est indéfini) et qu'au moins une erreur s'affiche en ligne de commande.*

Ce type de comportement sera un signe d'erreur à corriger dans les fichiers de shaders. (Enlevez par la suite votre ligne générant l'erreur).

Question 21 *Changez la couleur du triangle en bleu en modifiant ce fichier.*

Le fragment shader dispose également d'une variable automatiquement mise à jour (build-in) pour chaque pixel: gl_FragCoord qui contient les coordonnées du pixel courant dans l'espace écran. Ainsi pour chaque pixel de votre triangle, un fragment shader est exécuté et sa variable gl_FragCoord contient sa position en coordonnées de pixels.

Ici l'écran étant de taille 800x800, les coordonnées x et y varient entre 0 et 800. Notez que cette variable possède 4 dimensions et non deux (explications au semestre prochain en IMI).

Question 22 *Écrivez les lignes suivantes dans votre shader*

```
void main (void)
{
  float r=gl_FragCoord.x/800.0;
  float g=gl_FragCoord.y/800.0;
  color = vec4(r,g,0.0,0.0);
}
```

Il est possible d'affecter des fonctions sur les couleurs plus complexes. Essayez par exemple ces fonctions

```
void main (void)
{
  float x=gl_FragCoord.x/800.0;
  float y=gl_FragCoord.y/800.0;
  float r=abs(cos(15.0*x+29.0*y));
  float g=0.0;
  if(abs(cos(25.0*x*x))>0.95){
    g=1.0;
  }
  else{
    g=0.0;
  }
  color = vec4(r,g,0.0,0.0);
}
```

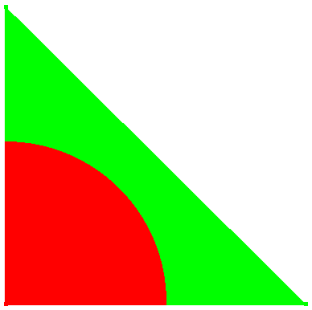

Figure 1: Triangle à afficher.

Question 23 *Affichez sur votre triangle une portion de disque rouge sur fond vert (voir Fig. 1). Cela ne doit pas vous prendre plus de 10min, sinon la solution est à la fin.*

Remarque: En affichant un carré couvrant l'écran en totalité, il est possible de créer tout type d'images en suivant cette approche, voir shadertoy.com.

La carte graphique possède de nombreux processeurs efficaces pour réaliser des opérations de calculs. Cette approche est l'une des plus performantes pour afficher et modifier une image. On a là une des portes d'entrée de la programmation dite à haute performance, ou HPC[1].

3.2 Vertex Shader

Question 24 *Il est possible de modifier la position et la forme de l'objet dans le vertex shader* `shader.vert`. *Attention, cette modification n'est que pour l'affichage, elle ne modifie pas les données envoyées à la carte graphique. Par exemple, ajoutez la ligne suivante en fin de shader:*

```
gl_Position.x/=2;
```

Question 25 *Expliquez le résultat obtenu à l'écran.*

Question 26 *Observez également ce que réalise le code suivant.*

[1]High Performance Computing

```
vec4 p=vec4(position, 1.0);
p.x=p.x*0.3;
p+=vec4(-0.7,-0.8,0.0,0.0);
gl_Position = p;
```

Remarque. Les coordonnées de gl_Position correspondent à une position normalisée dans l'espace écran visible entre les valeurs -1 et 1 (nous verrons plus tard que la coordonnée z de gl_Position doit également être comprise entre -1 et 1). Ainsi une position en (-1,-1) correspond au point inférieur gauche de l'écran, et (+1,+1) correspond à la position supérieure droite. (0,0) correspondant au centre de l'écran.

Question 27 *Quelle opération mathématique permet de passer des coordonnées normalisées de gl_Position du vertex shader aux coordonnées pixels de la variable gl_FragCoord du fragment shader rencontré précédemment ? (Ce travail est réalisé automatiquement dans le pipeline graphique, il n'y a rien à programmer)*

4 Passage de paramètres uniforms

Il est possible de passer des variables depuis le programme principal (depuis la RAM du CPU) vers les shaders (vers la VRAM du GPU) afin d'utiliser (en lecture seule) une valeur donnée dans le shader par le biais de paramètres qualifiés d'*uniform*.

Il faut pour cela modifier le shader voulu en ajoutant en dehors du main une variable qualifiée de `uniform`, qu'on appellera ici`translation` :

```
uniform vec4 translation;
```

Question 28 *Modifiez le vertex shader pour prendre en compte la translation issue de la variable uniforme dans l'affichage. Pour le moment, rien n'est transmis par le code python et donc l'execution ne permet par de déplacer le triangle.*

Notez que le vertex shader déclare désormais une variable vec4 qualifiée d'*uniform*. Cette variable est utilisée comme un paramètre de translation sur les coordonnées de l'objet. La valeur de ce paramètre est le même pour tous l'ensemble des sommets du triangle et est donné par le programme python exécuté sur le CPU.
Pour envoyer au programme gpu courant une variable uniforme, on peut utiliser le code suivant :

```
# Récupère l'identifiant du programme courant
prog = GL.glGetIntegerv(GL.GL_CURRENT_PROGRAM)
# Récupère l'identifiant de la variable translation dans le programme courant
loc = GL.glGetUniformLocation(prog, "translation")
# Vérifie que la variable existe
if loc == -1 :
  print("Pas de variable uniforme : translation")
# Modifie la variable pour le programme courant
GL.glUniform4f(loc, -0.5, 0, 0, 0)
```

Cet appel indique qu'une variable de type uniform (voir glUniform()) du shader va recevoir un paramètre depuis ce programme. glGetUniformLocation() permet de localiser la variable appelée textuellement translation dans le shader. Ensuite, les 4 valeurs flottantes sont envoyées dans le reste des paramètres. Cette variable uniforme n'est valable que pour le programme GPU courant.

Question 29 *Testez différentes valeurs de translation dans la fonction* `display_callback`. *Observez la translation résultante du triangle. Dans quelle plage de grandeur les coordonnées de translation en x/y peuvent varier tout en gardant le triangle dans l'écran? Est-ce que le paramètre* `translation_z` *modifie l'apparence de l'objet (dans quels plages d'intervalles) ? Avez-vous une explication par rapport aux effets observés?*

On souhaite maintenant déplacer le triangle automatiquement vers la droite de l'écran et le faire réapparaître automatiquement à gauche lorsqu'il a complètement disparu.

Question 30 *Modifiez les valeurs de* `translation_x/y/z` *automatiquement dans la boucle d'affichage en utilisant le temps. Est-ce que les données du triangle envoyées au GPU sont mise à jour? Comment savoir où se situe les sommets du triangle dans le repère monde? Dans le repère écran? Ajoutez la disparition/réapparition du triangle.*

5 Utilisation des touches du clavier

Question 31 *Observez la gestion des touches dans la fonction* `key_callback()` *et dans la boucle d'affichage. Puis, à l'aide de la documentation GLFW sur les noms des touches et sur les types d'actions, déplacez le triangle à l'aide des touches directionnelles. On utilisera pour le moment des variables globales (on préféra par la suite l'utilisation de classes).*

Pour pouvoir utiliser des touches simultanément, il faut maintenir une structure de données qui permet de savoir quelle touche est actuellement appuyé et lesquelles ont étés relâchées.

Question 32 *Modifiez la méthode précédent pour pouvoir utiliser deux touches simultanément.*

Question 33 *Modifiez le code pour que lors d'un appuie sur 'r', 'g' ou 'b', le triangle change d e couleur en fonction de la touche.*

6 Rotations

On souhaite maintenant appliquer une rotation à notre triangle. Nous utiliserons la librairie python pyrr.
Nous utiliserons des matrices de taille 4x4 pour être homogène dans le vertex shader. Pour créer une matrice 4x4 à partir d'une matrice de rotation 3x3 avec `pyrr`, on peut utiliser les l'api matrix :

```
rot3 =  pyrr.matrix33.create_from_z_rotation(np.pi/2)
rot4 = pyrr.matrix44.create_from_matrix33(rot3)
```

Pour transmettre une matrice uniforme aux shaders, on peut faire :

```
prog = GL.glGetIntegerv(GL.GL_CURRENT_PROGRAM)
loc = GL.glGetUniformLocation(prog, "rotation")
GL.glUniformMatrix4fv(loc, 1, GL.GL_FALSE, rot4)
```

Question 34 *Modifiez le code python et le vertex shader pour affecter des rotations suivant l'axe x et y à votre triangle lors de l'appuie sur les touches i,j,k et l.*

7 Projection

Une dernière notion non pris en compte jusqu'à présent concerne la projection du triangle de l'espace 3D vers l'espace (normalisé) de l'écran. Pour l'instant, les coordonnées 3D sont directement plaquée dans l'espace image en oubliant la coordonnée z si celle-ci est comprise entre -1 et 1. Ceci est équivalent à considérer que l'on réalise une projection orthogonale suivant l'axe z pour toute valeur de z comprise entre -1 et 1. Or une projection orthogonale ne permet pas de donner l'impression de distance à la caméra puisqu'un objet éloigné apparaîtra à la même taille qu'un objet proche.

Pour modéliser ce phénomène d'éloignement, il est nécessaire de considérer une autre matrice: la matrice de projection qui va modéliser l'effet d'une caméra. La description et l'utilisation d'espaces projectifs est vue en majeure image.

Pour l'instant, nous nous contenterons de considérer que ce phénomène de perspective peut être modélisé par une matrice de taille 4x4 qui est elle-même paramétrée par les variables suivantes: l'angle du champs de vision (FOV ou field of view) de la caméra, le rapport de dimension entre la largeur et hauteur, la distance la plus proche que peut afficher la caméra (> 0) et la distance la plus éloignée que peut afficher la caméra. Notez que pour obtenir un maximum de précision, il est important de limiter le rapport entre la distance la plus grande et la distance la plus faible.

Dans notre cas, on peut prendre : 50.0deg, avec un ratio de 1 et une distance entre 0.5 et 10.
`Pyrr` permet d'utiliser des matrices de projections perspectives.

Question 35 *Ajoutez une variable* `uniform` *dans le vertex shader pour la matrice de projection comme précédemment. On appliquera sur le point, dans l'ordre : la rotation puis la translation puis la projection.*

Question 36 *Utilisez les touches* `y` *et* `h` *pour déplacer votre triangle en profondeur. Observez l'effet de perspective (un triangle plus éloigné apparaît plus petit qu'un triangle proche).*

Notez que l'envoi d'une matrice de projection par le programme principal peu se faire dans la partie d'initialisation car les paramètres intrinsèques de la caméra sont constants tout au long de l'affichage.

8 Tableau de sommets et affichage indexé

Nous allons désormais ajouter un autre triangle à notre affichage. Pour cela, on considérera (dans la fonction `init_data()`) le vecteur de coordonnées tel que:

```
sommets = np.array(((0, 0, 0), (1, 0, 0), (0, 1, 0),
                    (0, 0, 0), (1, 0, 0), (0, 0, 1)), np.float32)
```

Question 37 *Dessinez sur une feuille de papier (avec un stylo) les deux triangles correspondants.*

Question 38 *Mettez à jour le programme et demandez l'affichage des 6 sommets dans l'appel à* `glDrawArrays`*.*

Notez que vous pouvez distinguer le second triangle en utilisant les rotations. Cependant, les couleurs des triangles ne dépendant que de la position des pixels dans la fenêtre d'affichage, il reste difficile de percevoir la séparation et la profondeur relative de ceux-ci. Notez également que le sommet (0, 0, 0) et (1, 0, 0) est dupliqué 2 fois sur la carte graphique. Cela engendre différentes limitations:

- Utilisation mémoire supérieur de par la duplication de sommet.
- Une modification sur un sommet demande la mise à jour à plusieurs endroits, avec un risque important d'oubli sur des maillages de grandes taille.

Pour répondre à ce problème, OpenGL dispose d'un affichage dit indexé. C'est à dire que l'on va séparer l'envoie des coordonnées des sommets (géométrie) de leur relation permettant de former un triangle (connectivité).

Question 39 *Remplacez la définition des sommets par la suivante*

```
sommets = np.array(((0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1)), np.float32)
```

Question 40 *Ajoutez également la définition d'un tableau d'indices:*

```
index=np.array(((0, 1, 2), (0, 1,3)), np.uint32)
```

Nous envoyons ensuite ce tableau d'entiers à OpenGL en indiquant qu'il s'agit d'indices:

```
# attribution d'un autre buffer de donnees
vboi = GL.glGenBuffers(1)
# affectation du buffer courant (buffer d'indice)
GL.glBindBuffer(GL.GL_ELEMENT_ARRAY_BUFFER,vboi)
# copie des indices sur la carte graphique
GL.glBufferData(GL.GL_ELEMENT_ARRAY_BUFFER,index,GL.GL_STATIC_DRAW)
```

Question 41 *Créez le buffer d'indices et copiez les données sur la carte graphique (dans la fonction* `init_data()`*). On utilise le nom de variable* `vboi` *pour "vbo index".*

Notez que cette fois le type d'élément est `GL_ELEMENT_ARRAY_BUFFER` qui indique qu'il s'agit d'indices.

Question 42 *Enfin, dans la fonction d'affichage, supprimez la ligne du glDrawArrays(), et faites appel à* `GL.glDrawElements(GL.GL_TRIANGLES, 2*3, GL.GL_UNSIGNED_INT, None)`

Regardez la doc de glDrawElements() :

- Le premier paramètre est identique à celui de glDrawArray() et indique le type d'élément affiché.
- Le second paramètre indique le nombre d'indices à lire, ici nous avons 2 triangles formés de 3 sommets, soit 6 valeurs.
- Le troisième indique le type de données, ici des entiers positifs.
- Le dernier paramètre indique l'offset à appliquer sur le tableau pour lire le premier indice (ici pas d'offset).

Question 43 *Exécutez le programme et assurez-vous que ayez le même résultat visuel que précédemment.*

9 Passage de paramètre interpolés entre shaders

Il est possible de passer des paramètres du vertex shader vers le fragment shader. Ces paramètres peuvent de plus varier en fonction de l'emplacement relative du fragment courant par rapport aux coordonnées des sommets du triangle. Pour cela, la carte graphique va pouvoir donner une valeur de paramètre au fragment shader obtenue à partir de l'interpolation linéaire (par défaut) des valeurs données par le vertex shader, en fonction de sa position dans le triangle.

On souhaite transmettre les coordonnées 3D, avant opération, des points des vertex shaders aux fragment shaders.

Question 44 *Créez une variable* `coordonnee_3d` *qualifiée de* `out` *de type* `vec3` *dans le vertex shader (en dehors du main). Donnez à cette variable, la position du sommet avant modification.*

Question 45 *Créez une variable* `coordonnee_3d` *qualifiée de* `in` *de type* `vec3` *dans le fragment shader (en dehors du main). Utilisez la composante en x, y et z pour respectivement définir la composante rouge, verte et bleue du fragment. Vous devez obtenir un triangle en degradé.*

Notez que contrairement à avant les couleurs ne dépendent que des coordonnées initiales du triangle et non plus de sa position relative sur la fenêtre. Ainsi déplacer le triangle ou lui affecter une rotation ne modifie plus la couleur. De plus, il est plus aisé de différencier le second triangle du premier puisque celui-ci se voit désormais affecté une couleur différente.

Question 46 *Modifiez la ligne* `glEnable(GL_DEPTH_TEST)` *en* `glDisable(GL_DEPTH_TEST)`. *Faites ensuite tourner le triangle sur lui même (sur un tour complet). Observez un phénomène visuellement perturbant: l'un des deux triangle est constamment affiché devant l'autre.*

Explication. Le *Depth Test* correspond au test de profondeur permettant d'assurer que l'on affiche bien les parties les plus proches de la caméra, indépendamment de l'ordre des triangles. Si celui-ci n'est pas activé, le dernier triangle envoyé est celui qui sera affiché devant tous les autres. Lors d'une animation cela perturbe notre perception de la 3D.
Réactivez le test de profondeur pour la suite du TP.

10 Illuminations et normales

La profondeur des triangles est difficilement perceptible car les couleurs présentent une illumination homogène. Pour obtenir une meilleur impression de profondeur, il est nécessaire d'illuminer la scène en supposant qu'il existe une lampe à un endroit. Pour obtenir un résultat correct, nous allons avoir besoin de définir les normales associées aux *vertex*.

Question 47 *Modifiez le tableau de sommets pour qu'il entrelace des coordonnées de sommets et des informations de normales (vous pouvez mettre dans un premier temps (0,0,0) pour les normales) .*

On doit donc avoir :

```
sommets = np.array(((0, 0, 0),(0, 0, 0),
  (1, 0, 0), (0, 0, 0),
  (0, 1, 0),(0, 0, 0),
  (0, -1, 0) ,(0, 0, 0)), np.float32)
```

Il faut donc modifier le pointeur de données `glVertexAttribPointer`. Le cinquième paramètre indique l'écart (appelé stride) entre deux données de coordonnées dans le tableau.

Question 48 *Modifiez le placement du pointeur des coordonnées de sorte à avoir un sommet tous les 2 x 3 x float32 (4 octets).*

On doit donc avoir (avec `from ctypes import *`) :

```
GL.glVertexAttribPointer(0, 3, GL.GL_FLOAT, GL.GL_FALSE, 2*3*sizeof(c_float()),
```

ou sans `ctypes` :

```
GL.glVertexAttribPointer(0, 3, GL.GL_FLOAT, GL.GL_FALSE, 2*3*4, 0)
```

Le sixième paramètre de glVertexAttribPointer() indique l'offset initial à appliquer au vecteur afin de tomber sur la première donnée de normale. Ici il faut se déplacer de

11 Solution

Résponse de la question 23

```
#version 330 core

// Variable de sortie (sera utilisé comme couleur)
out vec4 color;

//Un Fragment Shader minimaliste
void main (void)
{
  //Couleur du fragment
  float x=gl_FragCoord.x/800.0 - 0.5;
  float y=gl_FragCoord.y/800.0 - 0.5;

  if(x*x + y*y > 0.2*0.2)
    color = vec4(0.0,1.0,0.0,1.0);
  else
    color = vec4(1.0,0.0,0.0,1.0);
}
```