TSI : Synthèse d'images, OpenGL
CPE

Durée $\sim 4h$

Ce TP permet de prendre en main la programmation de scènes virtuelles 3D en temps réelle à l'aide d'OpenGL. Pour cela, un petit tutoriel vous est proposé afin de bien comprendre le fonctionnement d'OpenGL et de mettre en oeuvre les concepts simples de l'informatique graphique. Dans un premier temps, le tutoriel abordera l'envoie des 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 envoi 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 programmable → 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 shaders

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):
  # création 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):
  # création 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. Le rôle premier du vertex shader est d'affecter à la variable gl_Position 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`.

Remarque: Le code de ces fichiers n'étant pas exécuté par le processeur mais par la carte graphique, il n'est pas possible de réaliser d'affichage texte dans ces fichiers. Faites donc particulièrement attention lors de la modification de ces fichiers, le debug 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 les coordonnées de l'un des trois sommets du triangle. Notez bien que les vertex shaders sont exécutés en parallèle sur les sommets. Dans le cas présent, votre carte graphique exécute donc en parallèle 3 vertex shaders. L'un aura dans la variable `position` la valeur $(0, 0, 0)$, l'autre la valeur $(1, 0, 0)$, et le troisième $(0, 1, 0)$ (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 permet de paramétrer finement la couleur de chacun de ceux-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 Envoi 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 par le 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 données (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 le fonctionnement des données et le stocker dans 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, None)
```

Ces lignes indiquent que les données du buffer courant correspondent aux positions des sommets qui seront utilisés lors de la demande d'affichage.

Question 14 *Configurez la liste d'état.*

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 `None` 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 15 *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 16 *Copiez 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 de triangles en partant du premier élément (le 0), désigné par glVertexAttribPointer() par l'intermédiaire du VAO courant et pour 3 sommets (si nous avions 6 sommets, nous pourrions afficher 2 triangles).

Question 17 *Observez désormais l'affichage d'un triangle rouge sur l'écran lors de l'exécution.*

Remarque: Si votre triangle apparaît blanc ou noir, cela indique sûrement 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 18 *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
Remarque: Pour la suite du TP, on utilisera l'affichage du triangle plein. Vous pourrez afficher par moment votre maillage en mode *fil de fer* ou *Wireframe* afin de visualiser l'organisation de vos triangles pour du debug.

3 Les Shaders

3.1 Fragment Shader

Question 19 *Écrivez une ligne quelconque dans le fichier* `shader.frag` *de manière à rendre le code incorrect. Regardez le message d'erreur afin de pouvoir détecter ce type de problème par la suite.*

Enlevez la ligne générant l'erreur.

Question 20 *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 pixels.

Ici la fenêtre 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 21 *Remplacez le* `main()` *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);
}
```

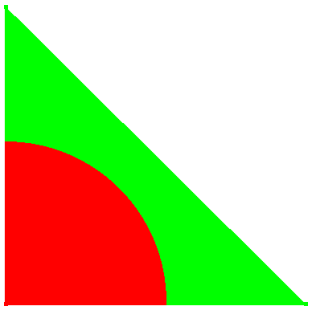

Figure 1: Triangle à afficher.

Question 22 *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

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.

Question 23 *Ajoutez la ligne suivante à la fin du* `main()` *du shader:*

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

Question 24 *Expliquez le résultat obtenu à l'écran. Vous pourrez la supprimer ensuite.*

Question 25 *Observez également ce que réalise le code suivant.*

```
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. Les points sont visibles 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.

[1] High Performance Computing

Question 26 *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 rencontrée précédemment ? (Ce travail est réalisé automatiquement dans le pipeline graphique, il n'y a rien à programmer).*

4 Passage de paramètres uniformes

Il est possible de passer des variables depuis le programme principal (RAM et CPU) vers les shaders (VRAM et 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 27 *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.*

Remarque: 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 28 *Testez différentes valeurs de translation dans la boucle d'affichage de la fonction* `run()`. *Observez la translation résultante du triangle.*

Question 29 *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 de translation en 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 en x 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.*

Question 31 *Modifiez le code python et le fragment shader pour que lors d'un appuie sur 'r', 'g' ou 'b', le triangle change de couleur en fonction de la touche.*

5 Utilisation des touches du clavier

Question 32 *Observez la gestion des touches dans la fonction* `key_callback()` *puis à l'aide de la documentation GLFW sur les noms des touches et sur les types d'actions, déplacez le triangle avec les touches directionnelles. On utilisera pour le moment des variables globales (on préférera par la suite l'utilisation de classes).*

Pour utiliser des touches simultanément, il faut maintenir une structure de données (telle qu'un dictionnaire) qui permet de savoir quelle touche est actuellement appuyé et lesquelles ont étés relâchées. L'incrémentation des variables globales se fait alors dans la boucle d'affichage.

Question 33 *Modifiez le code pour pouvoir utiliser deux touches simultanément.*

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)
```

Pour multiplier des matrices avec `pyrr` on peut utiliser `pyrr.matrix44.multiply()`.

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). On utilisera une translation sur z au démarrage de -5.*

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 `GL_ELEMENT_ARRAY_BUFFER` 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 des indices, 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 paramètres interpolés entre shaders

Il est possible de passer des paramètres du vertex shader vers le fragment shader. Ces paramètres varient 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 bilinéaire (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 de déplacement, 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 dégradé.*

Note: 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, 0, 1) ,(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.
On doit donc avoir (avec `from ctypes import *`) :

```
GL.glVertexAttribPointer(0, 3, GL.GL_FLOAT, GL.GL_FALSE, 2*3*sizeof(c_float()), None)
```

ou sans `ctypes` :

```
GL.glVertexAttribPointer(0, 3, GL.GL_FLOAT, GL.GL_FALSE, 2*3*4, None)
```

Le sixième paramètre de glVertexAttribPointer() indique l'offset initial à appliquer au vecteur. Pour ajouter un offset, il faut utiliser le code suivant:

```
sizefloat = sizeof(c_float())
GL.glVertexAttribPointer(..., ..., ..., ..., ..., c_void_p(3*sizefloat))
```

Question 48 *Modifiez le placement du pointeur des coordonnées de sorte à avoir un sommet tous les $2 \times 3 \times float32$ (4 octets) sans offset pour les positions. L'affichage ne devrait pas changer.*

Question 49 *Ajoutez l'attribut de normale à la location 1. Les normales commencent après 3 flottants de position. L'affichage ne devrait pas changer.*

Les informations de normales sont ensuite utilisées dans les shaders respectifs afin d'afficher une illumination de Phong.
Vous pourrez utilisez les shader `phong.vert` et `phong.frag`.

Question 50 *Modifiez le code pour utiliser les shaders permettant d'avoir une illumination de Phong.*

Afin de visualiser une illumination correcte (les normales sont actuellement fausses), on utilisera les données de sommets suivantes :
On considérera $p_0 = (0, 0, 0)$, $p_1 = (1, 0, 0)$, $p_2 = (0, 1, 0)$ et $p_3 = (0.8, 0.8, 0.5)$. Et les différentes normales: $n_0 = (0, 0, 1)$, $n_1 = (-0.25, -0.25, 0.85)$, $n_2 = n_1$, $n_3 = (-0.5, -0.5, 0.707)$ associées respectivement à p_0, p_1, p_2 , et $p3$.
Notez que les normales pour p_0 et p_3 sont les normales approximatives de leurs triangles respectifs.
Notez également que les normales associées à p_1 et p_2 sont les moyennes des normales des deux triangles auxquelles ils appartiennent.

Question 51 *Modifiez les sommets afin d'obtenir le maillage montré en figure 2*

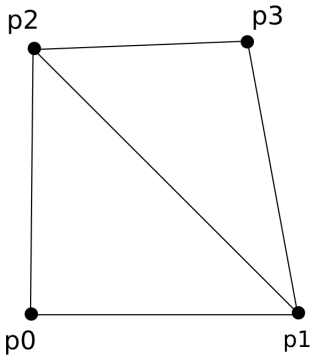

Figure 2: Maillage à construire.

Question 52 *Observez que les deux triangles donnent l'impression de former une surface lisse (la séparation entre les deux triangles n'apparaît pas clairement du fait de l'interpolation des normales à l'intérieur des triangles).*

Nous souhaitons désormais ajouter une composante supplémentaire traitée par la carte graphique:
Une couleur définie par sommet depuis le programme principal.
Cette fois, supposons que nous ajoutons des couleurs à chaque sommet.
Les sommets p_0, p_1, p_2 et p_3 devraient ainsi être respectivement: noir, rouge, vert, et jaune.

Question 53 *Ajoutez des données de couleur par sommets en entrelaçant dans l'ordre, position, normale et couleur. Modifiez les attributs de vertex pour prendre en compte la couleur et pour récupérer les couleurs dans le fragment shader.*

Pour cela, vous suivez les étapes suivantes:

- Ajout des couleurs dans le tableau de sommet.
- Mise à jour des décalages (*stride*) pour les coordonnées et normales dans glVertexAttribPointer().

- Ajout d'un nouveau type de données de couleurs envoyée sur la carte graphique: Activer l'utilisation des couleurs à la bonne location (similairement aux sommets et normales)
- Mise en place du pointeur de couleur à l'aide de la fonction glVertexAttribPointer(). Réfléchissez à l'écart (/stride) et à l'offset à appliquer.
- Dans le vertex shader, récupérez le contenu de la variable `color` contenant la couleur du sommet courant (sous forme de `vec3`) et passez-le au fragment shader par le biais d'une variable interpolée (*varying*).
- Dans le fragment shader, utilisez cette variable en tant que couleur (à la place de la couleur écrite en dure dans le shader actuel).

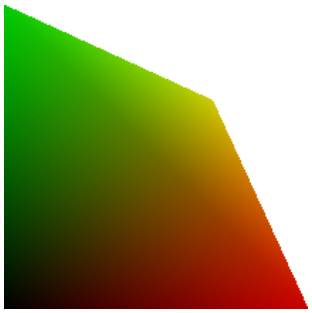

Figure 3: Triangle dont la couleur est indiqué pour chaque sommet, et interpolée sur la surface du triangle.

Pour information, l'image obtenue devrait être similaire à la figure 3 (sans erreurs au niveau des shaders lors de l'execution).

11 Texture

En plus des couleurs, on souhaite ajouter des textures ("image à plaquer"). Au final maintenant chaque sommet doit contenir : des coordonnées 3D, une normale, une couleur, et une coordonnée de texture à 2 composantes. Un schéma explicatif de l'organisation en mémoire d'une telle structure est fournie en figure 4.

Question 54 *Ajoutez des coordonnées de textures et modifiez les pointeurs d'attributs en fonction. Les paramètres uv de textures à utiliser sont les suivantes :*

```
uv0, uv1, uv2, uv3 = np.array((0, 0), np.float32), np.array((1, 0), np.float32), np.array((0, 1), np.float32), np.array((1, 1), np.float32),
```

Pour lire une image et la charger sur la VRAM, on peut utiliser le code suivant :

```
def load_texture(filename):
  if not os.path.exists(filename):
      print(f'{25*"-"}\nError reading file:\n{filename}\n{25*"-"}')
  im = Image.open(filename).transpose(Image.Transpose.FLIP_TOP_BOTTOM).convert('RGBA')
  texture_id = GL.glGenTextures(1)
  # sélection de la texture courante à partir de son identifiant
  GL.glBindTexture(GL.GL_TEXTURE_2D, texture_id)
  # paramétrisation de la texture
  GL.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_WRAP_S, GL.GL_REPEAT)
  GL.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_WRAP_T, GL.GL_REPEAT)
  GL.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAG_FILTER, GL.GL_LINEAR)
  GL.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MIN_FILTER, GL.GL_LINEAR)
  GL.glTexImage2D(GL.GL_TEXTURE_2D, 0, GL.GL_RGBA, im.width, im.height, 0, GL.GL_RGBA, G
  return texture_id
```

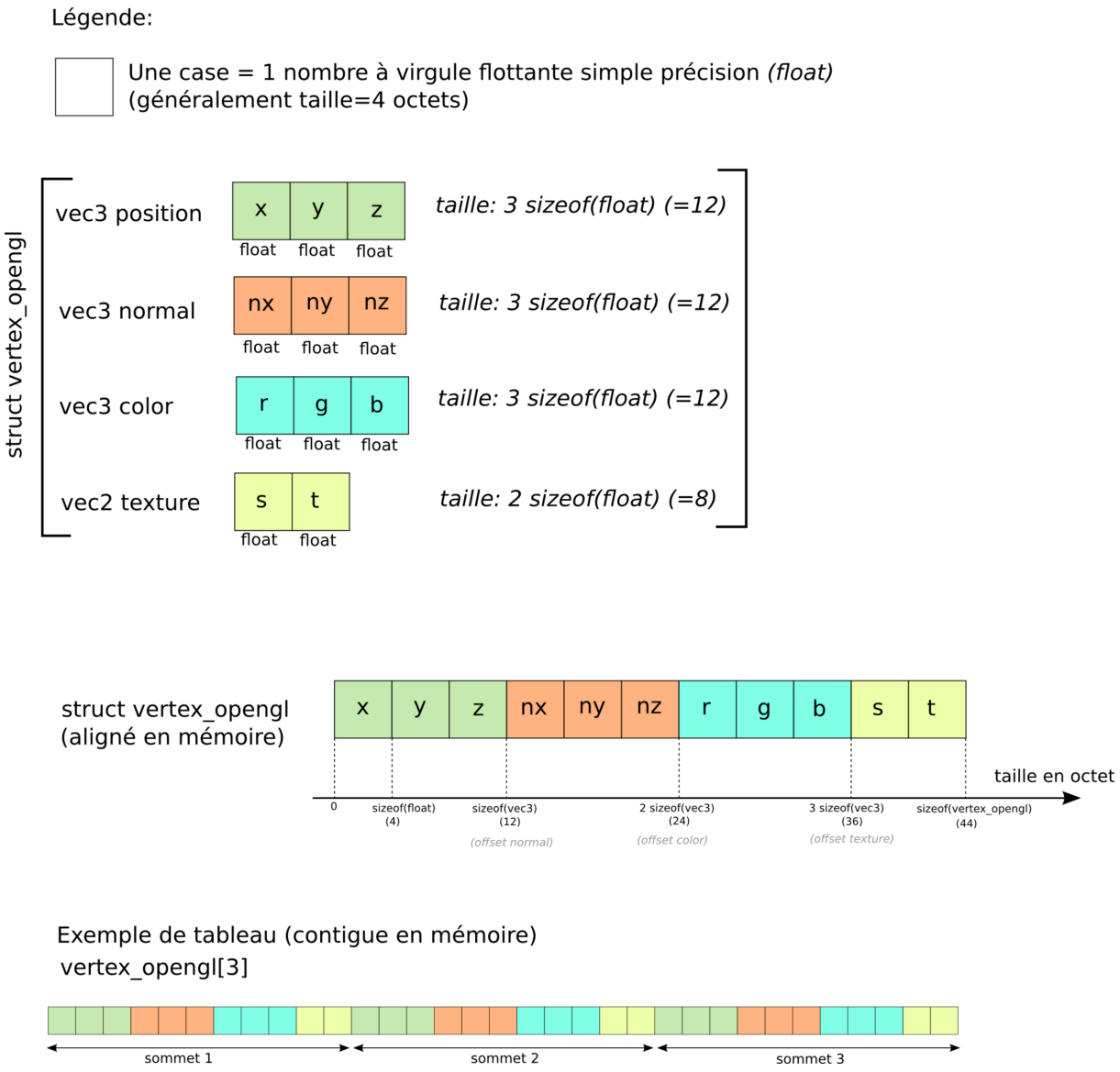

Figure 4: Organisation mémoire pour la structure vertex_opengl.

Ce code permet de récupérer l'identifiant de la texture chargée sur la VRAM et le GPU. On peut ensuite choisir la texture à utiliser avec :

```
GL.glBindTexture(GL.GL_TEXTURE_2D, texture_id)
```

Question 55 *À l'aide du code précédent, ajoutez l'envoie de la texture sur la VRAM.*

Pour utiliser une texture dans le fragment shader, il faut utiliser une variable `uniform` :

```
uniform sampler2D texture;
```

puis pour récupérer la couleur de la texture dans le fragment shader, on peut utiliser la fonction suivante :

```
vec4 color_texture = texture2D(texture, vtex);
```

avec `vtex` la coordonnée de texture interpolée.
On utilisera une multiplication entre la couleur interpolée et la couleur de la texture.

Question 56 *Ajoutez la couleur de la texture dans le fragment shader et observez le résultat.*

Question 57 *Modifiez les coordonnées de textures afin de comprendre leurs principes. Vous pouvez également modifier les couleurs et l'image utilisée.*

Question 58 *Que ce passe t-il lorsque les coordonnées de textures sont inférieures à 0 ou supérieures à 1?*

Remarque. Ce comportement est dû au mot clé `GL_REPEAT` passé à la fonction glTexParameter() lors de l'envoie de la texture sur la carte graphique.

Question 59 *Optionnel. Quels peuvent être les autres comportements ? (Recherchez dans la documentation sur internet). Testez certaines d'entre elles.*

12 Déplacement d'objets

On souhaite maintenant afficher deux objets identique. Pour cela, on peut faire deux appels à `glDraw...` avec des translations différentes.

- Rendre le VAO de l'objet 1 actif
- Envoie des rotations et translations spécifiques à l'objet 1 en tant que paramètre uniforme (+texture)
- Demande d'affichage de l'objet 1
- Rendre le VAO de l'objet 2 actif
- Envoie des rotations et translations spécifiques à l'objet 2 en tant que paramètre uniforme (+texture)
- Demande d'affichage de l'objet 2

On peut faire cela pour un nombre quelconque d'objets. Il suffit alors de stocker pour chaque objet de la scène : d'identifiant du vao et son nombre de triangle, sa transformation (translation et rotation), l'identifiant de la texture. Dans le cas où un objet n'est pas déplacé, on placera la rotation et la translation à une valeur fixe (ex. identité pour la rotation, et translation constante).

Question 60 *Mettez en place la visualisation de deux objets avec déplacement d'un objet par rapport à l'autre dans la fonction d'affichage.*

Question 61 *Ajoutez un troisième objet sur la scène sous l'objet statique avec une autre image.*

Remarque: Si l'ensemble des objets de la scène se déplacent de la même manière, cela donne l'impression que c'est la caméra qui se déplace.

13 Scène complète

Considérez l'archive qui servira de base pour le projet.

Cette fois une scène plus complexe contenant plusieurs objets est affichée. Une structure maillage est également fournie. Il est possible de charger un maillage à partir d'un fichier `.obj` qui peut être réalisé par un logiciel de modélisation (ex. Blender). Dans le cas de cette scène, on peut bouger le dinosaure et la caméra. Afin de séparer les transformations appliquées sur

un objet spécifique des transformations appliquées sur l'ensemble des objets, nous définissons les déformations dites de *Model* spécifiques à un objet, des déformations dites de *View* qui s'appliquent à tous les objets. La transformation finale du sommet est obtenue après avoir appliqué dans l'ordre

- La déformation du modèle (spécifique à l'objet)
- La déformation de la vue (commune à tous les objets)
- La projection (commune à tous les objets)

Afin de faciliter les déformations, nous séparons également les composantes suivantes : la rotation (sous forme d'angles d'Euler puis de matrice4x4), le centre de rotation (vec3), la translation (vec3). Pour chaque objet, nous définissons ainsi :

- une translation du modèle (vec3), t.
- une rotation du modèle (matrice 4x4), R.
- un centre de rotation3 du modèle (vec3), c.

La transformation d'un sommet original p en p' suite à cette déformation est donnée par

$$p' = R(p - c) + c + t$$

De manière similaire, nous définissons ces mêmes déformations communes à tous les objets (view transformation) donnant l'impression du déplacement de la caméra:

- une translation de la vue (vec3), t_v
- une rotation de la vue (matrice 4x4), R_v
- un centre de rotation de la vue (vec3), c_v .

Question 62 *Observez comment chaque objet est affiché avec des rotations et translations potentiellement différentes et comment celle-ci sont traités dans le shader.*

Question 63 *Faites en sorte de pouvoir déplacer le monstre indépendamment des autres objets.*

14 Collisions

Pour gérer les collisions, regarder si les triangles s'intersectent n'est pas optimal. Cela nécessite beaucoup de calculs alors que souvent une approximation suffit. On utilise alors des volumes englobants dont les plus utilisés sont visible figure 5. On peut par exemple regarder la distance euclidienne entre le centre de notre objet et l'objet qu'on teste (principe des *Bounding sphere*).
D'autres méthodes ad-hoc peuvent facilement être imaginées notamment dans le cas d'un monde discret.
Pour regarder les positions des vertex d'un maillage vous pouvez :

- utiliser `blender`, dans modeling, sélectionner un vertex et faire 'n'
- `meshlab`, option : render/show box
- regarder le fichier `.obj`

Question 64 *Regardez le chargement des objets (dinosaure).*

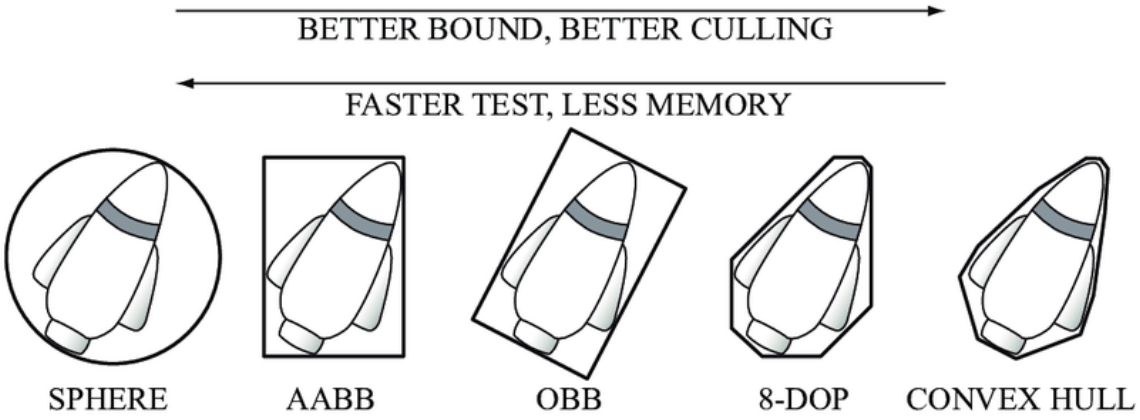

Figure 5: Volumes englobants usuels pour tester les collisions, image de **RealModel-a system for modeling and visualizing sedimentary rocks**, *Gang Mei 2014*

Question 65 *Avec la méthode des sphères englobantes (il n'est pas nécessaire d'avoir le rayon minimal, une approximation suffit), empêchez la collision entre deux dinosaures (ne pas charger deux fois le même maillage).*

Question 66 *Avec une méthode qui dérive des boites englobantes alignées sur les axes, empêchez la collision entre le dinosaure et le plan.*

15 Doubles programmes GPU

On a vu que pour afficher du contenu sur un écran avec OpenGL, il faut entre autre :

- spécifier un programme GPU (Comment?)
- spécifier les données (Quoi?)

Dans la partie précédente, bien que l'on affichait plusieurs objets, tous étaient affichés de la même manière (transformation MVP dans le vertex shader et illumination de Phong et texture dans le fragment shader).
On souhaite maintenant afficher du texte sur l'écran (interface utilisateur ou GUI). La méthode la plus courante est d'afficher des morceau de texture représentant les caractères sur l'écran. Si l'on souhaite afficher des objets en 3d en plus du texte de l'interface utilisateur, il faut gérer deux programmes différents.

16 Jeu

Un énoncé indépendant est disponible.

17 Solution

Réponse de la question 22

```
#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);
}
```