TSI : Synthèse d'images, OpenGL
CPE

Durée 12h

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 *Téléchargez l'ensemble des programmes du projet, décompressez les dans un répertoire spécifique. Lisez le fichier* `README.md` *pour obtenir les informations de compilation et d'exécution.*

Question 2 *Compilez et lancez le squelette de programme programme_1.*

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

Note: Il peut-être utile d'utiliser un gestionnaire de version type git pour le module.

Notez que le code doit s'exécuter sans erreurs. Dans le cas contraire, il est possible que votre code ne soit pas appelé depuis le répertoire principal. En particulier, il est nécessaire de paramétrer le répertoire d'exécution si vous utilisez un IDE.

Question 3 *Changez la couleur de fond en modifiant les paramètres de la fonction glClearColor() dans la fonction* `display_callback()`.

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.

Observez la fonction de réveil `glutTimerFunc(...)` qui permet un affichage continu de la scène (`glutPostRedisplay()` et `glutDisplayFunc(display_callback)`).

Question 4 *Affichez (*`printf()` *ou* `std::cout <<` *) un message sur la ligne de commande dans la fonction* `timer_callback()`*, observez ce qui se passe.*

Enlevez l'affichage pour la suite du TP.

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

Remarque: La méthode donnée en paramètre est appelé après un certain délai, il garantit un délai minimal mais pas maximal. Après un grand nombre d'itération le temps calculé (sommé) peut être faux. Dans une application industrielle, on préféra utiliser un chronomètre et pas une méthode par somme.

Question 6 *Changez la couleur de fond en fonction du temps, vous utiliserez une variable globale pour stocker le temps courant (en secondes) depuis le lancement du programme. On utilisera la méthode par somme.*

2 Création du premier triangle

2.1 Envoie des données sur la carte graphique

La fonction `init()` est une fonction qui est appelée une fois en début de programme. Nous allons créer les données et les transférer en mémoire vidéo (sur la carte graphique) dans cette fonction.

Question 7 *Construisez un tableau contenant 3 sommets* $(0,0,0)$, $(1,0,0)$, *et* $(0,1,0)$ *dont les coordonnées sont concaténées (dans la fonction* `init()`*)*

```
float sommets[]={ 0.0f,0.0f,0.0f,
0.8f,0.0f,0.0f,
0.0f,0.8f,0.0f};
```

Envoyons ces données sur la carte graphique en copiant les lignes suivantes à la fin de la fonction `init()`

```
//attribution d'une liste d'état (1 indique la création d'une seule liste)
glGenVertexArrays(1, &vao);
//affectation de la liste d'état courante
glBindVertexArray(vao);
//attribution d'un buffer de donnees (1 indique la création d'un seul buffer)
glGenBuffers(1,&vbo); CHECK_GL_ERROR();
//affectation du buffer courant
glBindBuffer(GL_ARRAY_BUFFER,vbo); CHECK_GL_ERROR();
//copie des donnees des sommets sur la carte graphique
glBufferData(GL_ARRAY_BUFFER,sizeof(sommets),sommets,GL_STATIC_DRAW);
CHECK_GL_ERROR();
```

Notes

- Vous pouvez trouver aux liens suivants la documentation précise des fonctions glGenVertexArrays(), glBindVertexArray(), glGenBuffers(),glBindBuffer(), glBufferData().
- On créera deux variables globales `vao` et `vbo` de type GLuint (généralement équivalent à un `unsigned int` sur la plupart des systèmes). Ces variables vbo 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.*
- Prenez l'habitude de terminer tous vos appels OpenGL en appelant la fonction `CHECK_GL_ERROR()` en cas d'erreur OpenGL, la ligne et l'erreur seront affichées pour faciliter le debug).

Question 8 *Assurez vous que cette partie compile et s'exécute sans erreurs sur la ligne de commande (il n'y a toujours rien dans la fenêtre).*

Ajoutez directement à la suite, les lignes suivantes:

```
// 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)
glEnableVertexAttribArray(0); CHECK_GL_ERROR();
// Indique comment le buffer courant (dernier vbo "bindé")
// est utilisé pour les positions des sommets
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0); CHECK_GL_ERROR();
```

Ces lignes indiquent que les données copiées précédemment sur la carte graphique par l'appel à glBufferData() correspondent aux positions des sommets courants qui seront utilisés en cas de demande d'affichage.

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).

Notons également que la fonction glVertexAttribPointer() ne fait que venir placer un pointeur de lecture. Cette fonction n'envoie pas de données à la carte graphique.

Question 9 *Vérifiez que votre programme compile et s'exécute sans erreurs (vous avez toujours une fenêtre vide).*

2.2 Affichage

Intéressons nous désormais à la fonction `display_callback()`. Cette fonction est appelée toute les 25 millisecondes (voir fonction `timer_callback()` qui paramètre cet appel).

Question 10 *Copiez enfin la ligne suivante après l'effacement de l'écran (glClear()) et avant l'échange des buffers d'affichage (glutSwapBuffers)*

```
glDrawArrays(GL_TRIANGLES, 0, 3); CHECK_GL_ERROR();
```

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

Question 11 *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 12 *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.

2.3 Affichage en fil de fer / wireframe

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.

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. Le gestionnaire de version peut vous permettre de faire ces "copies". (Mettre tout en commentaire risque de rendre votre projet de moins en moins lisible).

3 Les Shaders

3.1 Fragment Shader

(Attention le fragment shader a lieu après le vertex shader qui sera vu plus loin dans le TP)

La couleur de votre triangle 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).

Ce fichier de shader 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 pas possible de réaliser de `printf` ou `std::cout` dans ces fichiers. Faites donc particulièrement attention, le debug de ces fichiers est généralement difficile.

Question 13 *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).

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 14 *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 600x600, les coordonnées x et y varient entre 0 et 600. Notez que cette variable possède 4 dimensions et non deux (explications au semestre prochain en IMI).

Question 15 *Écrivez les lignes suivantes dans votre shader*

```
void main (void)
{
  float r=gl_FragCoord.x/600.0;
  float g=gl_FragCoord.y/600.0;
  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/600.0;
  float y=gl_FragCoord.y/600.0;
  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);
}
```

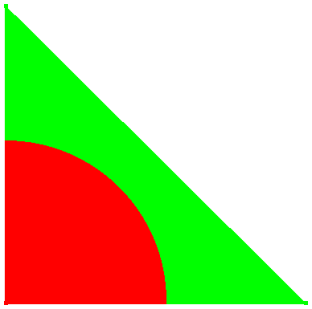

Figure 1: Triangle à afficher.

Question 16 *Affichez sur votre triangle une portion de disque rouge sur fond vert (voir Fig. 1).*

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].

[1]High Performance Computing

3.2 Vertex Shader

Il existe un autre shader: le vertex shader (exécuté avant le *fragment shader*). Celui-ci est appelé pour chaque sommet que l'on demande d'afficher lors d'un appel à `glDraw...()` (ici 3 fois car on a un triangle).

Le vertex shader à pour rôle premier d'affecter la variable gl_Position qui doit contenir la position du sommet courant dans l'espace écran normalisé dans un `vec4`.

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). 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 $(0, 0, 0.8)$. 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 17 *Il est possible de modifier la position et la forme de l'objet dans ce shader. Par exemple, ajoutez la ligne suivante en fin de shader:*

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

Question 18 *Expliquez le résultat obtenu à l'écran.*

Question 19 *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 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 20 *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 (lecture seule) une valeur donnée dans le shader par le biais de paramètres qualifiés d'*uniform*.

Considérez désormais le `programme_2`.

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 C exécuté sur le CPU (On peut également utilisé cette variable dans le *fragment shader*).

Les valeurs de la variable translation sont envoyées sur la carte graphique par l'appel suivant dans la fonction `display_callback`.

```
// Récupère l'identifiant de la variable pour le programme shader_progran_id
GLint loc_translation = glGetUniformLocation(shader_program_id, "translation"); CHECK_GL_ERROR();
// Vérifie que la variable existe
if (loc_translation == -1) std::cerr << "Pas de variable uniforme : translation" << std::endl;
// Modifie la variable pour le programme courant
glUniform4f(loc_translation,translation_x,translation_y,translation_z,0.0f); CHECK_GL_ERROR();
```

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 21 *Modifiez les valeurs de* `translation_x/y/z` *dans le programme principal et 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 22 *Modifiez les valeurs de* `translation_x/y/z` *dans le programme principal grâce à la boucle de temps. Est-ce que les paramètres du triangle sont mis à jour dans le code principal? 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

Considérez désormais le `programme_3`.

Ce programme utilise les même shaders, mais cette fois les variables `translation_x/y` sont des variables globales modifiées par les touches du clavier.

On utilisera les fonctions GLUTsuivantes glutKeyboardFunc pour la gestions des touches standards (caractères ascii), et la fonction glutSpecialFunc pour les caractères non ascii (tels que les touches haut, bas, gauche et droite).

Question 23 *Observez la gestion des touches dans la fonction* `main()`.

Remarque. Ces fonctions permettent de savoir lorsque l'utilisateur appuie sur une touche. Elles ne permettent cependant pas de savoir si l'utilisateur maintient la touche enfoncée (ou alors empêchent les appuis simultanés). Les fonctions glutKeyboardUpFunc et glutSpecialUpFunc permettent à l'inverse de savoir lorsque l'utilisateur relâche une touche. À l'aide d'une

variable globale de type booléen par touche d'intérêt, on peut savoir si la touche est appuyée ou non. Dans la fonction de temps, il suffit alors de regarder l'état de la touche.

Question 24 *Utilisez ce mécanisme pour afficher une translation régulière de votre triangle lors d'un appui continu sur une touche.*

On peut aussi savoir si une touche de modification (ex. ctrl) est appuyée avec glutGetModifiers (attention il faut utiliser des masques de bits)

Question 25 *Expliquez les couleurs observées lorsque vous déplacez votre triangle dans la fenêtre.*

6 Rotations

Considérez désormais le `programme_4`.

Ce programme incorpore cette fois la gestion d'une rotation. Une matrice de rotation (de taille 4x4) est calculée dans le programme principal (une struct de type mat4 qui possède le même nom que la désignation d'une matrice dans les shaders en GLSL). Cette matrice est paramétrée par l'axe autour duquel la rotation est appliquée ainsi que l'angle de rotation (le principe de la méthode de calcul de la matrice de rotation ainsi que les détails du code C++ correspondant seront expliqués dans les prochains semestres). La matrice est ensuite envoyée sur la carte graphique sous forme de paramètre uniform, puis est appliquée sur chaque sommet du triangle.

Question 26 *Utilisez les touches i,j,k et l pour affecter des rotations suivant l'axe x et y à votre triangle.*

- Observez l'application de la rotation dans le shader.
- Observez comment se déroule le calcul des matrices dans le programme principal dans la fonction `keyboard_callback`.
- Observez le passage de paramètre d'une matrice sous forme de uniform dans la fonction `display_callback`.

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 sera vue au semestre prochain.

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.

Considérez désormais le `programme_5` qui implémente la gestion d'une matrice de projection.

Question 27 *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 l'envoie d'une matrice de projection par le programme principal (ici uniquement envoyée dans la fonction `init()` car les paramètres sont constants tout au long de l'affichage) ainsi que l'application de celle-ci dans le shader.

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()`) le vecteur de coordonnées tel que:

```
float sommets[]={0.0f,0.0f,0.0f,
  1.0f,0.0f,0.0f,
  0.0f,1.0f,0.0f,
  0.0f,0.0f,0.0f,
  1.0f,0.0f,0.0f,
  0.0f,0.0f,1.0f
};
```

Question 28 *Dessinez sur une feuille de papier (avec un stylo) les deux triangles correspondants.*

Question 29 *Mettez à jour le programme (*`programme_5`*) et demandez l'affichage de 6 sommets dans l'appel 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 30 *Remplacez la définition des sommets par la suivante*

```
float sommets[]={0.0f,0.0f,0.0f,
  1.0f,0.0f,0.0f,
  0.0f,1.0f,0.0f,
  0.0f,0.0f,1.0f
};
```

Question 31 *Ajoutez également la définition d'un tableau d'indices:*

```
unsigned int index[]={0,1,2,
0,1,3};
```

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

Question 32 *Créez une variable globale vboi (symbolisant "vbo index") de type GLuint. Créez le buffer d'indices et copiez les données sur la carte graphique (dans la fonction* `init()` *) avec les appels suivants:*

```
//attribution d'un autre buffer de donnees
glGenBuffers(1,&vboi);
//affectation du buffer courant (buffer d'indice)
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,vboi);
//copie des indices sur la carte graphique
glBufferData(GL_ELEMENT_ARRAY_BUFFER,sizeof(index),
index,GL_STATIC_DRAW);
```

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

Question 33 *Enfin, dans la fonction d'affichage, supprimez la ligne du glDrawArrays(), et faites appel à* `glDrawElements(GL_TRIANGLES, 2*3, GL_UNSIGNED_INT, 0);`

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 34 *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.

Considérez le `programme_6`.

Notez cette fois la présence de la variable `coordonnee_3d` qualifiée de `out` dans le vertex shader. Cette variable est mise à jour avec la valeur des coordonnées 3D du sommet considéré (avant application des rotation, translation, et projection).
Ensuite, dans le fragment shader, la variable `coordonnee_3d`, qualifiée de `in`, contient la valeur des coordonnées interpolées linéairement en fonction de la position du fragment dans

le triangle. Chaque coordonnée de cette valeur est interprétée comme une couleur rouge, verte ou bleue. L'interpolation linéaire des coordonnées abouti à un dégradé linéaire dans l'espace des couleurs sur les triangles.

Notez que contrairement au programme précédent, cette fois 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 35 *Tentez désormais de réaliser (ou d'approximer) la figure 2 sur l'un des triangles (les couleurs doivent cette fois être indépendantes de la position et de l'orientation du triangle).*

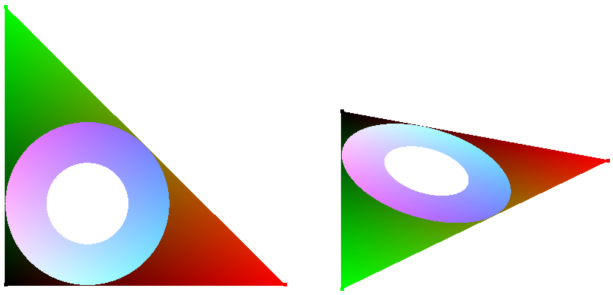

Figure 2: Figure à approximer telle que les couleurs soient indépendantes de la position et l'orientation du triangle.

Aide Le cercle inscrit à un triangle rectangle possède un rayon $r = (a + b - c)/2$, avec (a, b, c) les longueurs des cotés dont c la longueur de l'hypoténuse. Le centre du cercle inscrit est aux coordonnées (r, r) par rapport au sommet de l'angle droit.

- Que vaut a, b et c dans le cas présent?
- Notez que vous avez le droit à la fonction sqrt dans le shader. Prenez également l'habitude d'écrire vos nombres en flottants avec une virgule. Par exemple racine carré de 3 devra être écrit : sqrt(3.0) (et pas sqrt(3)).

Remarque Il est possible de réaliser un trou dans un triangle en utilisant la commande discard dans le fragment shader qui rend alors le fragment courant transparent (/ne l'affiche pas). Par exemple, le code

```
if( x>0.25)
  discard;
```

Permet de n'afficher que les pixels ayant une coordonnée x inférieure à 0.25.

Question 36 *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*.

Considérez désormais le `programme_7`.

Ce programme introduit de nouvelles structures qui simplifieront la manipulation des données. En particulier, elle introduit une classe de vecteur 2D et 3D (`vec2` et `vec3`) similaire aux vecteurs accessibles en GLSL.

Question 37 *Observez la nouvelle initialisation des données sous forme d'un tableau de* `vec3` *qui entrelace des coordonnées de sommets et des informations de normales.*

Question 38 *Observez le placement du pointeur des coordonnées:*

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 2*sizeof(vec3), 0);
```

Le cinquième paramètre indique que l'écart (appelé stride) entre deux données de coordonnées dans le tableau est de 2 fois la taille d'un vec3. Notez que l'on aurait également pu écrire de manière tout à fait équivalente: `2*3*sizeof(float)`.

Question 39 *Observez également la mise en place des normales pour à la location 1 (de la même manière):*

```
// Active l'utilisation des données de normales (le 1 correspond à la location dans le vertex shad
glEnableVertexAttribArray(1); CHECK_GL_ERROR();
// Indique comment le buffer courant (dernier vbo "bindé") est utilisé pour les normales des somme
glVertexAttribPointer(1, 3, GL_FLOAT, GL_TRUE, 2*sizeof(vec3), (void*)sizeof(vec3)); CHECK_GL_ERRO
```

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 `sizeof(vec3)` (ou de `3*sizeof(float)`). Les informations de normales sont ensuite utilisées dans les shaders respectifs afin d'afficher une illumination de Phong.

Question 40 *Observez que cette fois, le fait de faire tourner le triangle modifie l'illumination de celui-ci. La scène semble ainsi disposer d'une lumière éclairant le triangle.*

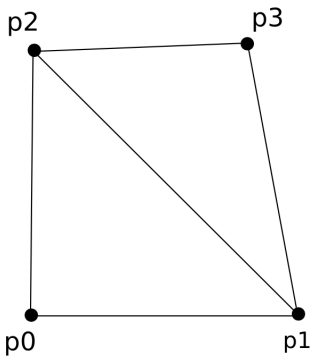

Figure 3: Maillage à construire.

Question 41 *Ajoutez un nouveau sommet p_3 et un nouveau triangle (p_1, p_3, p_2) afin d'obtenir le maillage montré en figure 3*

On considérera $p_3 = (0.8, 0.8, 0.5)$. 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.

Pour vous aider à réaliser cet ajout, notez que vous devez réaliser les actions suivantes sur le code

1. Créer un nouveau sommet p_3 sous forme de `vec3`.
2. Créer une nouvelle normale n_3 sous forme de `vec3` et modifier les normales n_1 et n_2.
3. Mettre à jour le tableau de géométrie en ajoutant p_3 et n_3.
4. Créer un nouveau `triangle_index(1, 3, 2)` et l'ajouter au tableau d'index.
5. Mettre à jour la demande d'affichage de 2 triangles (fonction `display_callback`).

Question 42 *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 les données sont les suivantes:

```
//coordonnees geometriques des sommets
vec3 p0=vec3(0.0f,0.0f,0.0f);
vec3 p1=vec3(1.0f,0.0f,0.0f);
vec3 p2=vec3(0.0f,1.0f,0.0f);
vec3 p3=vec3(0.8f,0.8f,0.5f);

//normales pour chaque sommet
vec3 n0=vec3(0.0f,0.0f,1.0f);
vec3 n1=vec3(-0.25f,-0.25f,0.8535f);
vec3 n2=vec3(-0.25f,-0.25f,0.8535f);
vec3 n3=vec3(-0.5f,-0.5f,0.707);

//couleur pour chaque sommet
vec3 c0=vec3(0.0f,0.0f,0.0f);
vec3 c1=vec3(1.0f,0.0f,0.0f);
vec3 c2=vec3(0.0f,1.0f,0.0f);
vec3 c3=vec3(1.0f,1.0f,0.0f);

//tableau entrelacant coordonnees-normales
vec3 geometrie[]={p0,n0,c0 , p1,n1,c1 , p2,n2,c2 , p3,n3,c3};
```

Les sommets p_0, p_1, p_2 et p_3 devraient ainsi être respectivement: noir, rouge, vert, et jaune.

Question 43 *Terminez la mise en place de la gestion des couleurs par sommets en interpolant celle-ci de manière linéaire sur les triangles.*

Pour cela, vous suivez les étapes suivantes:

- Mise à jour des décalages (*stride*) pour les coordonnées et normales dans glVertexAttrib-Pointer().
- 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).

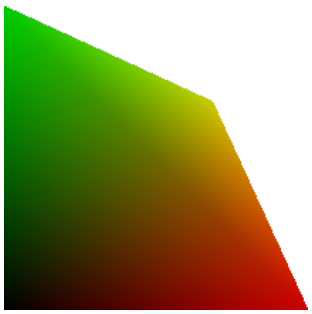

Figure 4: Triangle dont la couleur est indiqué pour chaque sommet, et interpolée sur la surface du triangle.

Pour information, l'image obtenue devrait être similaire à la figure 4 (sans erreurs au niveau des shaders lors de l'execution).

11 Texture

Considérez désormais le `programme_8`.

En plus des couleurs, ce programme intègre également la gestion des textures. De plus, cette fois, nous ajoutons une structure supplémentaire permettant d'organiser plus aisément les données: un `vertex_opengl` qui contient 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 5.

Question 44 *Observez qu'à l'exécution du programme, la couleur de la texture est mixée (multiplication) avec la couleur définie pour chaque sommet.*

Question 45 *Observez la mise en place du pointeur sur les coordonnées de texture dans la fonction d'initialisation.*

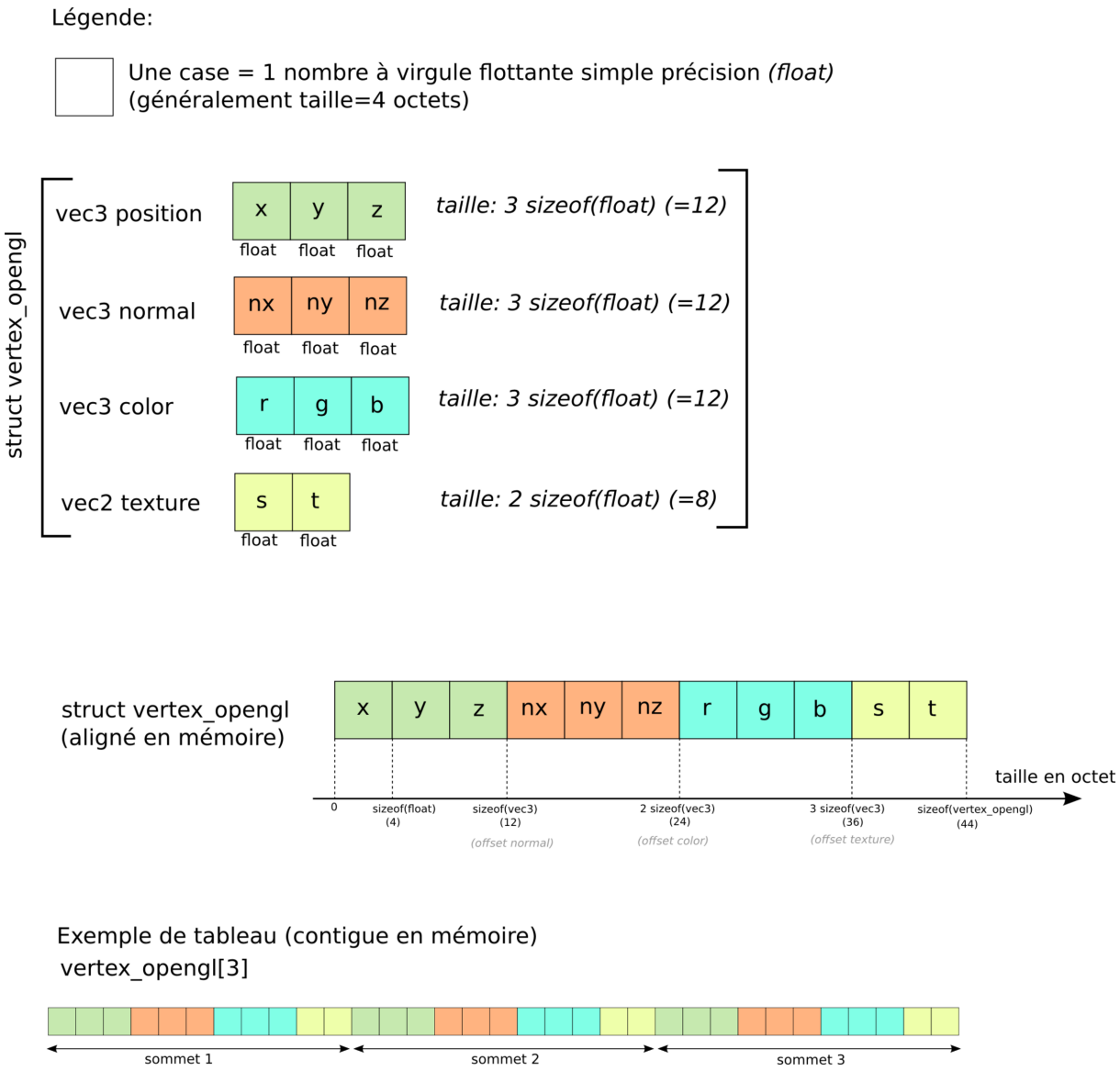

Figure 5: Organisation mémoire pour la structure vertex_opengl.

Question 46 *Observez le chargement de l'image et son envoie sur la carte graphique (seul un chargeur d'image tga est fournie, vous pouvez convertir une image en tga à l'aide de Gimp ou d'ImageMagick[2]). Notez que les images avec transparences ne sont pas acceptées.*

Question 47 *Modifiez les coordonnées de textures afin de comprendre leurs principes. Vous pouvez également modifier les couleurs et l'image utilisée.*

Question 48 *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 49 *Quels peuvent être les autres comportements ? (Recherchez dans la documentation sur internet). Testez certaines d'entre elles.*

[2]Commande convert en ligne de commande

12 Déplacement d'objets

Considérez le `programme_9`.

Ce programme affiche cette fois deux objets (identiques), mais permet de déplacer l'un alors que l'autre reste fixe. Pour déplacer un objet séparément d'un autre, il est possible d'utiliser l'approche suivante dans la fonction d'affichage:

- Rendre le VAO de l'objet 1 actif
- Envoie des rotations et translations spécifiques à l'objet 1 en tant que paramètre uniforme
- 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
- 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 50 *Observez la mise en place du déplacement d'un objet par rapport à l'autre dans la fonction d'affichage.*

Question 51 *Ajoutez un troisième objet sur la scène sous l'objet statique avec une autre image.*

Notez que 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 le `programme_10`.

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 (format `.off` ou `.obj`) qui peut être réalisé par un logiciel de modélisation (ex. Blender). Dans le cas de cette scène, les touches haut, bas, gauche et droite permette de translater le dinosaure suivant les coordonnées x et y. Les touches `o, l, k`, et `m` permettent d'appliquer une rotation sur ce même objet suivant les axes x et y. Les touches `e, d, s`, et `f`, ont par contre une influence sur l'ensemble des objets. Cela donne l'impression de déplacer la caméra dans la scène. `e` et `d` permettent d'approcher/reculer la caméra, alors que `s` et `f` permettent de tourner autour d'eux. 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 (matrice 4x4), 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 52 *Observez comment chaque objet est affiché avec des rotations et translations potentiellement différentes et comment celle-ci sont traités dans le shader.*

Question 53 *Faites en sorte de pouvoir déplacer le monstre indépendamment des autres objets.*

Question 54 *Comment faire en sorte que le dinosaure puisse avancer sur le sol dans la direction où il regarde. Vous pouvez soit appliquer la rotation au vecteur de direction du dinosaure, soit recalculer la direction à la main (la première méthode est plus simple).*

Question 55 *D'autres moyens de déplacer la caméra sont donnés dans la gestion des touches, observer leur fonctionnement. Comment sont gérés les centres de rotation, rotations, translations?*

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 6. On peut par exemple regarder la distance euclidienne entre le centre de notre objet et l'objet qu'on teste (principe des *Bounding sphere*). On peut aussi regarder la distance sur les axes (principe des *Axis Aligned Bounding Box*).
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
- afficher l'ensemble des vertex de votre maillage depuis le code principal
  `for(const auto& v : m.vertex) std::cout << v.position << std::endl;`
- regarder le fichier `.off` ou `.obj` (et utiliser python/matlab)

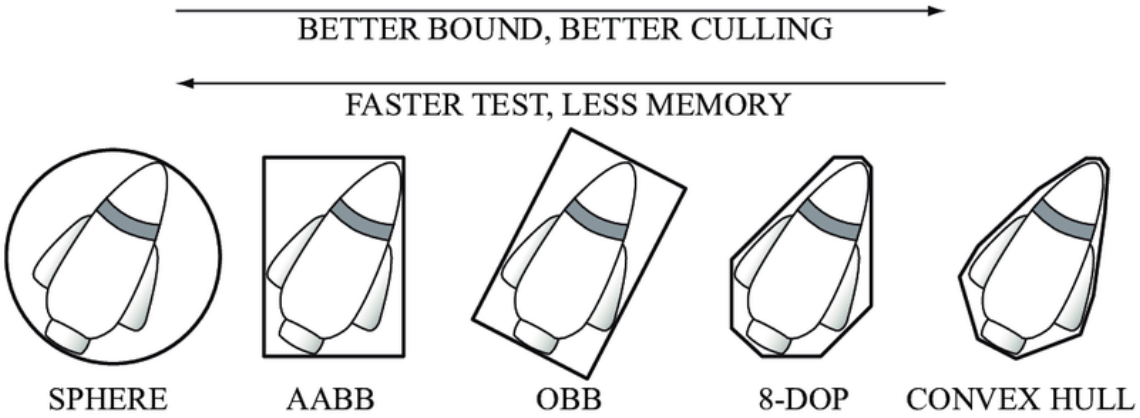

Figure 6: Volumes englobants usuels pour tester les collisions, image de **RealModel-a system for modeling and visualizing sedimentary rocks**, *Gang Mei 2014*

Question 56 *Regardez le chargement (toujours dans le* `programme_10`*) des objets (monstre et dinosaure), une transformation est appliqué (sous la forme d'une* `mat4`*). Le* `vec3` *formé par les trois premières composantes du* 4^e *vecteur colonne représente la translation appliqué au maillage. Où se situe les objets lors de l'envoi à la carte graphique? En quoi cela peut-être gênant pour gérer les collisions ? Faites en sorte que votre maillage envoyé au GPU soit centré sur* $(0, 0, 0)$ *mais que l'affichage ne change pas.*

Question 57 *Avec la méthode des sphères englobantes (il n'est pas nécessaire d'avoir le rayon minimal, une approximation suffit), empêchez la collision entre le dinosaure et le monstre.*

Question 58 *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.
La méthode est implémenté dans le `programme_11`. Pour plus d'informations, vous pouvez regarder : `https://gitlab.com/RemiCasado/glut_text_drawing/-/tree/master` dont cette partie est librement inspirée.

Question 59 *Comparez les différences avec le programme précédent (vous pouvez utiliser* `code --diff file1.cp`

16 Jeu

Le sujet vous sera donné en séance 2.