TP Synthèse d'images:

- Partie 1 -

Lancement du programme.

➔ **Compilez** et **lancez** le squelette de programme fourni (`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.
Pour l'instant, seul un écran au fond bleu est affiché.

Note: Vous êtes libre d'utiliser l'éditeur de code que vous préférez. (Mais n"utilisez pas gedit qui n'est pas un éditeur). Le fichier CMakeLists.txt est fourni dans le cas où vous souhaitez utiliser QtCreator (voir fiche annexe sur les IDE).

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

Remaque: 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 debuger plus facilement.

Création du premier triangle.

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.

➔ **Construisez** un tableau contenant 3 sommets (0,0,0), (1,0,0), et (0,1,0) dont les coordonnées sont concaténés (dans la fonction `init()`):

```
float sommets[]={0,0,0,
                 1,0,0,
                 0,1,0};
```

Envoyons ces données sur la carte graphique en copiant les lignes suivantes à la suite:

```
//attribution d'un buffer de donnees (1 indique la création d'un buffer)
glGenBuffers(1,&vbo); PRINT_OPENGL_ERROR();
//affectation du buffer courant
glBindBuffer(GL_ARRAY_BUFFER,vbo); PRINT_OPENGL_ERROR();
//copie des donnees des sommets sur la carte graphique
glBufferData(GL_ARRAY_BUFFER,sizeof(sommets),sommets,GL_STATIC_DRAW);
PRINT_OPENGL_ERROR();
```

Note:
- On créera une variable globale `vbo` de type `GLuint` (généralement équivalent à un *unsigned int* sur la plupart des systèmes).
Cette variable `vbo` est un identifiant permettant de distinguer plusieurs buffers de données au besoin.

- Prenez l'habitude de terminer tous vos appels OpenGL en appelant la macro `PRINT_OPENGL_ERROR(.` En cas d'erreur OpenGL, la ligne et l'erreur seront affichées pour faciliter le debug.

➔ **Assurez vous** que cette partie compile et s'exécute (il n'y a toujours rien dans la fenêtre).

Indiquons ensuite que les données copiées sur la carte graphique correspondent aux positions des sommets en copiant les deux lignes suivantes:

```
    // Active l'utilisation des données de positions
    glEnableClientState(GL_VERTEX_ARRAY); PRINT_OPENGL_ERROR();
    // Indique que le buffer courant (désigné par la variable vbo) est utilisé
pour les positions de sommets
    glVertexPointer(3, GL_FLOAT, 0, 0); PRINT_OPENGL_ERROR();
```

Notons que les arguments de `glVertexPointer()` sont les suivants:

- 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 "`float`".
- 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).

➔ **Vérifiez** que votre programme compile et s'exécute sans erreurs.

Demandons ensuite l'affichage du triangle dans la boucle d'affichage.
Pour cela, nous nous intéressons à la fonction `display_callback()`. Cette fonction est appelée toute les 25 millisecondes (voir fonction `timer_callback()` qui paramètre cet appel).

➔ Affichez (`printf`) un message sur la ligne de commande dans la fonction `display_callback()`, observez ce qui se passe.

(Enlevez le printf pour la suite du TP.)

Après l'effacement de l'ecran (`glClear`) et avant l'échange des buffers d'affichage (`glutSwapBuffers`) copiez la ligne suivante:

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

Cette ligne réalise la demande d'un triangle en partant du premier élément, et pour 3 sommets.

➔ Observez l'affichage d'un triangle sur l'écran.

Note: 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.

➔ **Utilisez** les flèches du clavier pour appliquer des rotations sur la caméra.

Notez qu'il peut être difficile de visualiser l'orientation du triangle car celui-ci contient une couleur unique et monotone qui rend la perception 3D difficile.

Note: Les rotations sont paramétrés par les variables `rotation_x` et `rotation_y` indiquant les angles de rotations (en degrés) qui sont mises à jours dans la fonction `special_callback()`. Ces angles sont convertis en rotation de caméra par les fonction `glRotate()` dans la fonction `display_callback()`.

➔ Modifiez les paramètres de l'une des fonctions `glRotate()` pour faire en sorte que les flèches de gauche/droite réalisent une rotation autour de l'axe z à la place de l'axe y.

(Par la suite, vous pouvez suivant les exemples et préférences changer le type de rotations)

➔ 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_STIP` 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); PRINT_OPENGL_ERROR();
glDrawArrays(GL_LINE_LOOP, 0, 3); PRINT_OPENGL_ERROR();
```

(Pour la suite du TP, on utilisera l'affichage du triangle plein)

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.
(mettre tout en commentaire risque de rendre votre projet de moins en moins lisible).

Les Shaders.

Fragment Shader.

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. 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`". Faites donc particulièrement attention, le debug de ces fichiers est difficile.

La variable `gl_FragColor` est une variable connue par et défaut devant être remplie par le *fragment shader*. La variable possède 4 composantes, mais seules les 3 premières (r,g,b) nous serons utile pour le moment.

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

Le *fragment shader* dispose également d'une variable automatiquement mise à jour pour chaque pixel: `gl_FragCoord` qui contient les coordonnées du pixel courant dans l'espace écran.
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 (explication au semestre prochain).

Ecrivez les lignes suivantes dans votre shader:

```
void main (void)
{
  float r=gl_FragCoord.x/800;
  float g=gl_FragCoord.y/800;
  gl_FragColor = vec4(r,g,0,0);
}
```

➔ **Expliquez** ce que vous observez. Pourquoi le triangle change de couleur lorsque l'on applique une rotation? Entre quelles valeurs varient r et g?

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;
  float y=gl_FragCoord.y/800;

  float r=abs(cos(15*x+29*y));
  float g=0;
  if(abs(cos(25*x*x))>0.95)
      g=1;
  else
      g=0;

  gl_FragColor = vec4(r,g,0,0.0);
}
```

➔ Tentez d'afficher sur votre triangle une portion de disque rouge sur fond vert (la partie du disque affichée dépend de la position du triangle), voir Fig. 1.

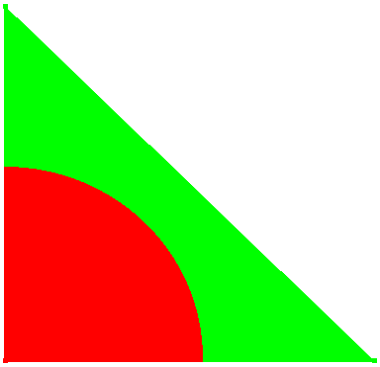


Fig. 1. *Exemple d'affichage obtenue*

Remarque*:* En affichant un carré couvrant l'écran en totalité, il est possible de créer tout type d'images en suivant cette approche.
La carte graphique possédant de nombreux processeurs efficace pour réaliser des opérations de calculs, il s'agit d'ailleurs de l'une des approche les plus performantes pour afficher et modifier une image. (Plus rapide que l'écriture dans un tableau en C de manière itérative, et bien plus rapide que l'écriture dans une matrice sous Matlab). Il s'agit de l'une des porte d'entrée de la programmation dite "à haute performance" (voir cours GPGPU de la dernière année).

Vertex Shader.

Il existe un autre shader: le *vertex shader*.
Celui-ci est appelé pour chaque sommet que l'on demande d'afficher (ici 3 fois pour un triangle).

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

Ici, la fonction `ftransform()` est appelée. Cette fonction retourne la projection du sommet 3D dans l'espace écran. Cette étape fait intervenir deux matrices: La matrice de projection qui correspond à la perspective, et la matrice dite de "*ModelView*" qui correspond aux transformations globales appliquées à la scène. Ces deux matrices sont définies au début de la fonction `display_callback()` du code C. Notez qu'il est possible de manipuler séparément la partie translation de la partie rotation à l'aide des appels `glTranslate` et `glRotate`.
Ces matrices (*Projection* et *ModelView*) sont en réalité de matrices 4x4 (explication au semestre suivant en géométrie projective).

Il est possible de modifier l'apparence de l'objet dans ce shader. Par exemple, ajoutez la ligne suivante en fin de shader:
`gl_Position.x/=2;`

➔ Expliquez le résultat obtenu à l'écran.

La variable `gl_Vertex` est disponible par défaut dans le vertex shader. Cette variable contient les coordonnées 3D du sommet courant (avant toute projection sur l'espace écran).
De plus, il est possible de faire communiquer des variables entre le vertex shader et le fragment shader. Ecrivez le code suivant dans le vertex shader:

```
varying vec4 position3d;
void main (void)
{
  gl_Position = ftransform();
  position3d=gl_Vertex;
}
```

Et le suivant pour le fragment shader

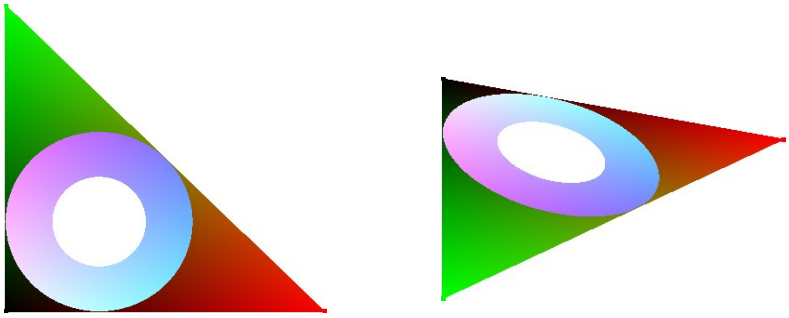
```
varying vec4 position3d;
void main (void)
{
  gl_FragColor = position3d;
}
```

➔ Expliquez le résultat obtenu à l'écran.

Explication: La variable `position3d` est mise à jour avec les coordonnées du sommet courant dans le vertex shader.
Cette valeur est non seulement accessible dans le fragment shader, mais en plus cette valeur a été interpolée automatiquement pour tous les pixels du triangle (conformément à la position relative du pixel par rapport aux sommets).
En appliquant cette "*position*" en tant que couleur, le premier sommet de coordonnée (0,0,0) correspond à du noir, le second aux coordonnées (1,0,0) à du rouge, et le troisième (0,1,0) à du vert.
Enfin, ces coordonnées étant interpolées pour tous les pixels, nous obtenons un **dégradé** de couleur sur le triangle.
C'est ce qu'indique le mot clé **`varying`** = une valeur interpolée en passant du vertex shader au fragment shader.

Notez que cette fois la couleur est indépendante de l'orientation du triangle, celle-ci ne dépend que des coordonnées initiales.

➔ Tentez désormais de réaliser (ou d'approximer) cette figure sur le triangle (les couleurs doivent cette fois être indépendamment de l'orientation du triangle).



Aide: Le cercle inscrit à un triangle rectangle possède un rayon r=(a+b-c)/2, où (a,b,c) sont les longueurs des cotés, et c est la longueur de l'hypoténuse. Le centre du cercle inscrit est aux coordonnées (r,r) par rapport au sommet de l'angle droit.
Les couleurs sont ici du blanc, les couleurs correspondantes aux positions 3d et leurs complémentaires.

Passage de paramètres depuis le programme principal.

Il est possible de passer d'autres paramètres depuis le programme principal du processeur vers les shaders sur la carte graphiques. Ces paramètres sont qualifiés de **`uniform`** car ils sont constant et uniformément appliqués à l'ensemble des shaders.

Nous allons passer un paramètre indiquant le temps au shader ce qui va nous permettre de réaliser des animations.

Premièrement, initialisez à zéro une variable globale entière `counter_time` dans le programme principale.
Incrémentez cette variable dans la fonction `timer_callback()`.
Ajoutez également la ligne suivante dans la fonction d'affichage `display_callback()`:

```
glUniform1i (get_uni_loc(shader_program_id, "time"), counter_time);
PRINT_OPENGL_ERROR();
```

Note:
La fonction `get_uni_loc` (qui fait elle même appel à la fonction OpenGL `glGetUniformLocation`) permet de localiser la variable `time` dans le shader (et indique une erreur si il ne la trouve pas).
`glUniform1i` indique que l'on va passer en paramètre une valeur entière. Celle-ci aura pour valeur de `counter_time`.

Ajoutez dans les deux shaders la ligne (dans la définition des variables en début de fichier).

```
uniform int time;
```

Vous avez à présent accès au temps s'incrémentant dans vos shaders.
Ajoutez par exemple les deux lignes suivantes dans le vertex shader:

```
gl_Position.x+=cos(time/10.0);
gl_Position.y+=sin(time/10.0);
```

➔ **Observez** l'animation résultante

➔ **Utilisez** cette variable temporelle pour obtenir des couleurs variants au cours du temps.

Profondeur.

Ajoutez un autre triangle à la forme.
Pour cela, on considérera désormais (dans la fonction `init()`) le vecteur de coordonnées tel que:

```
float sommets[]={0,0,0 ,
                 1,0,0 ,
                 0,1,0 ,
                 0,0,0 ,
                 1,0,0 ,
                 0,0,1};
```

- ➔ **Dessinez** sur une feuille de papier (avec un stylo) les deux triangles correspondants.
- ➔ **Mettez à jour** le programme et demandez l'affichage de 6 sommets dans l'appel `glDrawArrays`.

(supprimez l'animation pour permettre d'étudier plus aisément le résultat visuel)

- ➔ **Modifiez** la ligne `glEnable(GL_DEPTH_TEST)` en `glDisable(GL_DEPTH_TEST).` Faite 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)

La profondeur des triangles est difficilement perceptible car la couleur est 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*".

- ➔ Quels sont les normales (unitaires) des deux triangles affichés?

Nous allons cette fois envoyez les positions des sommets suivit des normales à la carte graphique.
Premièrement, nous déclarons les données dans la fonction `init()`.
A la place du tableau `sommet[]`, nous avons désormais le tableau suivant:

```
float vertex[]={0,0,0 , 1,0,0 , 0,1,0 , 0,0,0 , 1,0,0 , 0,0,1,
                0,0,1 , 0,0,1 , 0,0,1 , 0,1,0 , 0,1,0 , 0,1,0};
```

Il contient les 6 sommets, suivit des 6 normales.

Envoyons les données sur la carte graphique:

```
glGenBuffers(1,&vbo); PRINT_OPENGL_ERROR();
glBindBuffer(GL_ARRAY_BUFFER,vbo); PRINT_OPENGL_ERROR();
glBufferData(GL_ARRAY_BUFFER,sizeof(vertex),vertex,GL_STATIC_DRAW);
PRINT_OPENGL_ERROR();
```

Définissons le pointeur sur les coordonnées des sommets:

```
glEnableClientState(GL_VERTEX_ARRAY); PRINT_OPENGL_ERROR();
glVertexPointer(3, GL_FLOAT, 0, 0); PRINT_OPENGL_ERROR();
```

Définissons également le pointeur sur les normales:

```
glEnableClientState(GL_NORMAL_ARRAY); PRINT_OPENGL_ERROR();
long int offset_normal=18*sizeof(float);
glNormalPointer(GL_FLOAT,0,buffer_offset(offset_normal));
PRINT_OPENGL_ERROR();
```

Notez cette fois les deux arguments 0, et `offset_normal`.
- Le 0 indique que les normales seront lues les unes derrière les autres sans entrelacement d'autres données.
- La variable `offset_normal` indique l'offset à appliquer à partir du début du tableau pour accéder à la première valeur de la normale. L'offset est attendue sous forme de pointeur, c'est pour cela que l'on fait appel à la fonction `buffer_offset()`.

Note supplémentaire: La fonction `buffer_offset()` vient en fait convertit la valeur entière recu en paramètre en GLvoid*. (GLvoid* est similaire au void*, l'ajout de GL permet d'assurer que la taille de la variable est compatible avec les pointeurs attendus sous OpenGL).

L'offset pour le buffer des normales doit indiquer que l'on doit se déplacer de 6 sommets avant d'accéder à la première normale. Chaque sommet étant composé de 3 coordonnées, cela fait 18 valeurs. Enfin, chaque valeur entière est stockée sur `sizeof(float)` octets (qui vaut généralement 4). L'offset est donc de `18*sizeof(float)` (qui vaut 72 dans notre cas).

- ➔ **Sauvegardez** vos shaders précédent quelque part, et utilisez les shaders fournis (`shader_illumination`) qui vont venir prendre en compte les valeurs de normales.
- ➔ **Executez** le programme et observez que cette fois l'effet de profondeur en 3D apparaît plus distinctement. Ce shader vient rendre compte d'un phénomène d'illumination faisant croire qu'une lampe a été placée aux coordonnées (2,2,0).

➔ **Changez** désormais les données d'entrées par ceci:

```
float s=1.0f/sqrt(2.0f);
float vertex[]={0,0,0 , 1,0,0 , 0,1,0 , 0,0,0 , 1,0,0 , 0,0,1,
                0,s,s , 0,s,s , 0,0,1 , 0,s,s , 0,s,s , 0,1,0};
```

➔ **Réalisez** un croquis des triangles et des normales qui leur sont associées (un vecteur à chaque sommet).

➔ **Observez** que cette fois les deux triangles ne semble plus clairement séparés lorsqu'ils sont vues de face (faites des rotations pour voir l'effet de la lumière), ils semblent former une surface presque continue.

En appliquant des normales différentes sur les 3 sommets d'un même triangle, nous pouvons ainsi donner l'impression d'une surface lisse courbée alors que celle-ci n'est formée géométriquement que de triangles planaires.

Déplacement des normales.

(A faire uniquement si il vous reste du temps pour le TP 1)

➔ **Remplacez** dans le fragment shader la ligne `n=normalize(normal);` par celles ci

```
float x=vertex_3d_original.x;
float z=vertex_3d_original.z;
vec3 n=normal+vec3(0,0.1*sin((x+z)*50),0.1*cos((x+z)*50));
n=normalize(n);
```

➔ **Observez** l'effet lorsque vous lancez le programme. La surface semble être ondulée sous certains angles alors qu'il ne s'agit que d'un triangle géométrique.

Note: Cette approche consistant à déformer artificiellement les normales permet de simuler des détails artificiels à haute résolution tout en gardant de grands triangles. Il s'agit d'une technique très classique pour simuler des détails dans les jeux vidéos.

- Partie 2 -

Modélisation

Considérez le nouveau programme fourni (`programme_2`).
Cette fois, vous disposez des `struct vec3` modélisant des vecteurs ou positions 3D de l'espace.
Les `vec3` du programme sont relativement similaires aux `vec3` de GLSL.
Ce sont des *enregistrements* similaires au struct vues en C, mais disposant de quelques spécificités liées au C++: constructeurs et opérateurs +,-,*,/ afin de faciliter leurs manipulations.

Affichage indexé

- ➔ **Observez** que les 4 sommets distincts (et 4 normales) sont cette fois crées sous forme de `vec3` dans la fonction `init()`.
- ➔ **Re-créez** l'objet géométrique précédent formé de 2 triangles en utilisant la ligne suivante avant la création des `vbo` dans la fonction `init()`.

```
vec3 vertex[]={p0,p1,p2,p0,p1,p3,
               n0,n0,n1,n0,n0,n2};
```

Remarque: Cette ligne est similaire à la création d'un tableau de flottants concaténés. Les éléments étant organisés ici par groupe de 3.
`vertex[0]` permet d'adresser le premier `vec3`. Sa coordonnée x sera accessible avec la syntaxe suivante: `vertex[0].x`

- ➔ **Exécutez** le programme et assurez vous que vous obteniez un résultat similaire au cas précédent.

Notez que les positions **p0** et **p1** (ainsi que les normales **n0** associées) sont dupliquées dans le tableau de coordonnées. C'est une situation très courante lorsque l'on manipule un maillage où les triangles partagent des arêtes.

Si l'on souhaite modifier la position **p0**, nous somme ici obligés de la modifier à deux endroits dans la structure de données. Par exemple:

```
vertex[0].x += 0.5;
vertex[3].x += 0.5;
```

Oublier l'une des deux opérations revient à scinder à nouveau les triangles.
En plus de consommer de la mémoire inutilement (duplication de coordonnées identiques), il est préférable de ne pas avoir à faire ce travail: Pour modifier un sommet, il faut modifier une seule entrée du tableau.

Pour cela, OpenGL dispose d'une autre méthode de rendu dite **indexée**.

Pour cela, nous stockons désormais les sommets uniquement une seule fois, peut importe l'ordre. Par exemple

```
vec3 vertex[]={p0,p1,p2,p3,
               n0,n0,n1,n2};
```

(à la place du tableau vertex précédent)

➔ **Modifiez** l'offset pour le buffer de normales envoyé à OpenGL en accord avec la nouvelle dimension du tableau.

Ensuite, nous allons indiquer à OpenGL l'indexation des triangle (également appelé connectivité). Il faut indiquer que le premier triangle sera formé par les sommets d'indices (0,1,2), et le second triangle par les sommets d'indices (0,1,3).

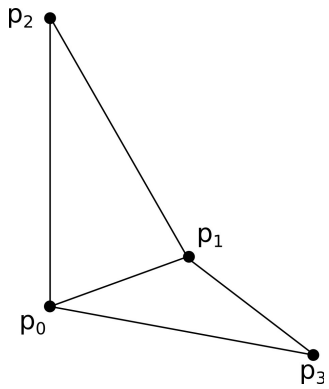


Fig. 2. Indexation des sommets

➔ **Définissez** pour cela le tableau

Soit directement en tableau d'`unsigned int`:

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

Soit en utilisant une structure `triangle_index` (contenant 3 `unsigned int`)

```
triangle_index triangle0(0,1,2);
triangle_index triangle1(0,1,3);
triangle_index index[]={triangle0,triangle1};
```

Remarque: Les deux notations reviennent au même au niveau de la mémoire et des syntaxes ultérieurs (demandez si vous ne comprenez pas l'une d'entre elle).

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

➔ **Créez** une variable globale `vboi` (signifiant "*vbo index*") de type `GLuint`.
➔ **Créez** le buffer d'indices et copiez les données sur la carte graphique avec les appels

suivants:

```
glGenBuffers(1,&vboi); PRINT_OPENGL_ERROR();
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,vboi); PRINT_OPENGL_ERROR();
glBufferData(GL_ELEMENT_ARRAY_BUFFER,sizeof(index),index,GL_STATIC_DRAW);
PRINT_OPENGL_ERROR();
```

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

Enfin, dans la fonction d'affichage, supprimez la ligne du `glDrawArray`, et faites appel à:

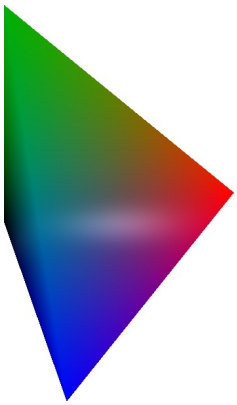
```
glDrawElements(GL_TRIANGLES, 2*3, GL_UNSIGNED_INT, 0);
PRINT_OPENGL_ERROR()
```

Note:
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).

➔ **Executez** le programme et assurez vous que ayez le même résultat visuel que précédemment.

➔ **Créez** un tétraèdre liant les différents sommets uniquement en modifiant les indices (et le nombre d'éléments à afficher dans `glDrawElements`).

Remarque: Les normales ne prenant pas en compte l'ajout de ces triangles, et donnant une apparence lisse peuvent rendre l'objet difficile à appréhender.



Modélisation paramétrique

(Si le temps le permet: avant 10h au TP 2)

Considérez désormais le programme de modélisation paramétrique (`programme_3`).

Cette fois, un maillage particulier est généré dans la fonction `init()`.
Ce maillage correspond à la fonction paramétrique:
x(u,v)=u
y(u,v)=v
z(u,v)=0
avec, (u,v) variant dans [0,1].

Il s'agit d'un plan, et les paramètres (u,v) sont échantillonés sur une grille régulière de 25x25.
La première double boucle vient remplir le tableau de coordonnées, et la seconde boucle vient remplir les indices de la connectivité des triangles (notez que l'on utilise la structure triangle_index pour simplifier la gestion).
L'organisation de la connectivité suit le schéma suivant (Fig. 3):

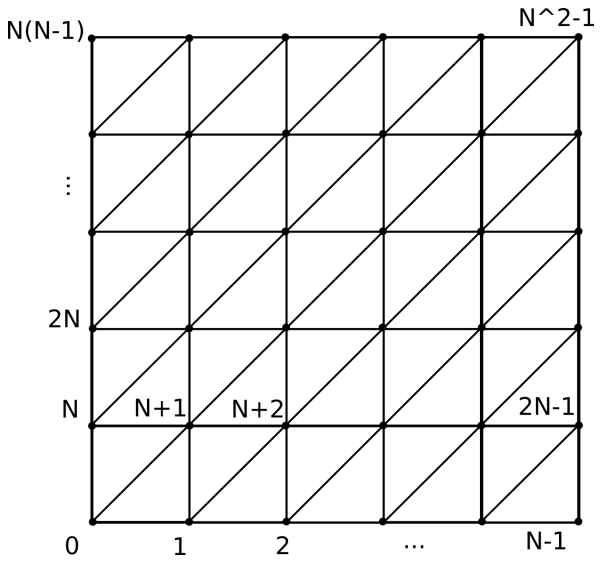


Fig. 3. Indexation du maillage

➔ **Testez** que le programme s'exécute et affiche bien un plan.

➔ **Ajoutez** la ligne suivante dans la fonction d'`init` et visualisez le résultat.

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

Cette ligne permet de n'afficher que les lignes du bord des triangles. Vous pouvez ainsi visualiser l'organisation du maillage. Cette commande peut être utile pour debuger certaines configurations. (Commentez cette ligne pour la suite du TP).

➔ **Remplacez** dans le vertex shader la ligne `vec4 p=gl_Vertex` par celles-ci

```
float a=0.1;
float b=8*8;
float x0=0.5;
float x=gl_Vertex.x;
float y=gl_Vertex.y;
float z=a*exp(-((x-x0)*(x-x0))*b);
vec4 p=vec4(x,y,z,1.0);
```

Notez que la surface n'est désormais plus plane et qu'elle contient une bosse. Cependant les normales correspondent toujours au cas du plan et ne mettent pas en avant la forme de la bosse. L'équation de la surface étant connue analytiquement, il est cependant possible de connaître le vecteur normal exact en tout point.

➔ **Calculez** et mettez à jour le vecteur normal dans le vertex shader. Observez que cette fois votre surface est illuminée correctement.

Aide possible: Le language GLSL dispose des fonctions de produits vectoriels par défaut. On appellera `cross(a,b)` pour réaliser le produit vectoriel entre a et b.

➔ **Passez** la variable de temps à votre shader et faites en sorte que la bosse se déplace continûment de gauche à droite de manière périodique.

Cela pourrait par exemple modéliser l'effet des vagues se déplaçant au cours du temps (voir Fig. 5).
(N'oubliez pas de mettre à jour le calcul de la normale au besoin) .

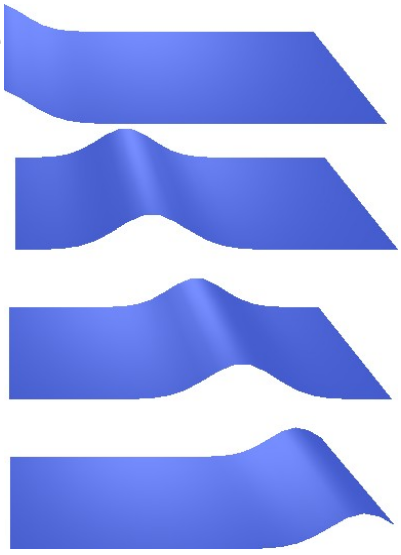


Fig. 5. Un effet de vagues possible avec les shaders

Texture

Chargez le programme suivant (`programme_4`).

➔ **Compilez** et **executez** le programme, observez que le triangle possède des propriétés de couleurs et de textures.

Cette fois, les données associées aux normales, couleurs, et textures sont définies dans les paramètres associés aux sommets envoyés sur la carte graphique.

Pour cela, nous avons définit la structure `vertex_opengl` qui contient respectivement:

- Une position (`vec3`): *x,y,z*
- Une normale (`vec3`): *nx,ny,nz*
- Une couleur (`vec3`): *r,g,b*
- Une coordonnée de texture (`vec2`): *s,t*

Chaque sommet du triangle est formé de ces 4 attributs qui sont définis dans la fonction `init()`.
Les données des sommets et indices sont ensuite copiés sur la carte graphique.
Enfin, une image utilisée pour la texture est lue depuis le disque dur et envoyée sur la carte graphique.

Dans la fonction d'affichage, cette fois nous retrouvons la gestion des différents pointeurs: positions, couleurs, normales.

Note: Nous placions auparavant la gestion de ces pointeurs dans la fonction `init` car nous n'avons toujours affichée qu'une seule entité sur un seul vbo. Si on souhaite par contre afficher plusieurs maillages dépendant de plusieurs vbo, cette opération devra se faire avant chaque affichage, nous montrons donc cette seconde solution dans ce cas.

Contrairement aux cas précédents, les positions de sommets ne sont plus stockés de manière contigus. Le tableau de `vertex_opengl` est constitué d'un entrelacement entre position, normale, couleur, et coordonnées de textures. Le tableau des 3 sommets contient donc des données entrelacées. Reportez vous à l'image suivante (Fig. 6.) pour une illustration de l'organisation en mémoire.

➔ **Retrouvez** les différents offsets placés pour définir les buffers sur les normales, couleurs, et textures (dans la fonction d'affichage).

➔ **Modifiez** les coordonnées de textures de la fonction `init()`. Pour observer leurs

comportements.

➔ **Que ce passe t-il** lorsque les coordonnées de textures sont inférieurs à 0 ou supérieur à 1?

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

➔ Quels peuvent être les autres comportements ? (recherchez dans la documentation sur internet). **Testez** certaines d'entres elles.

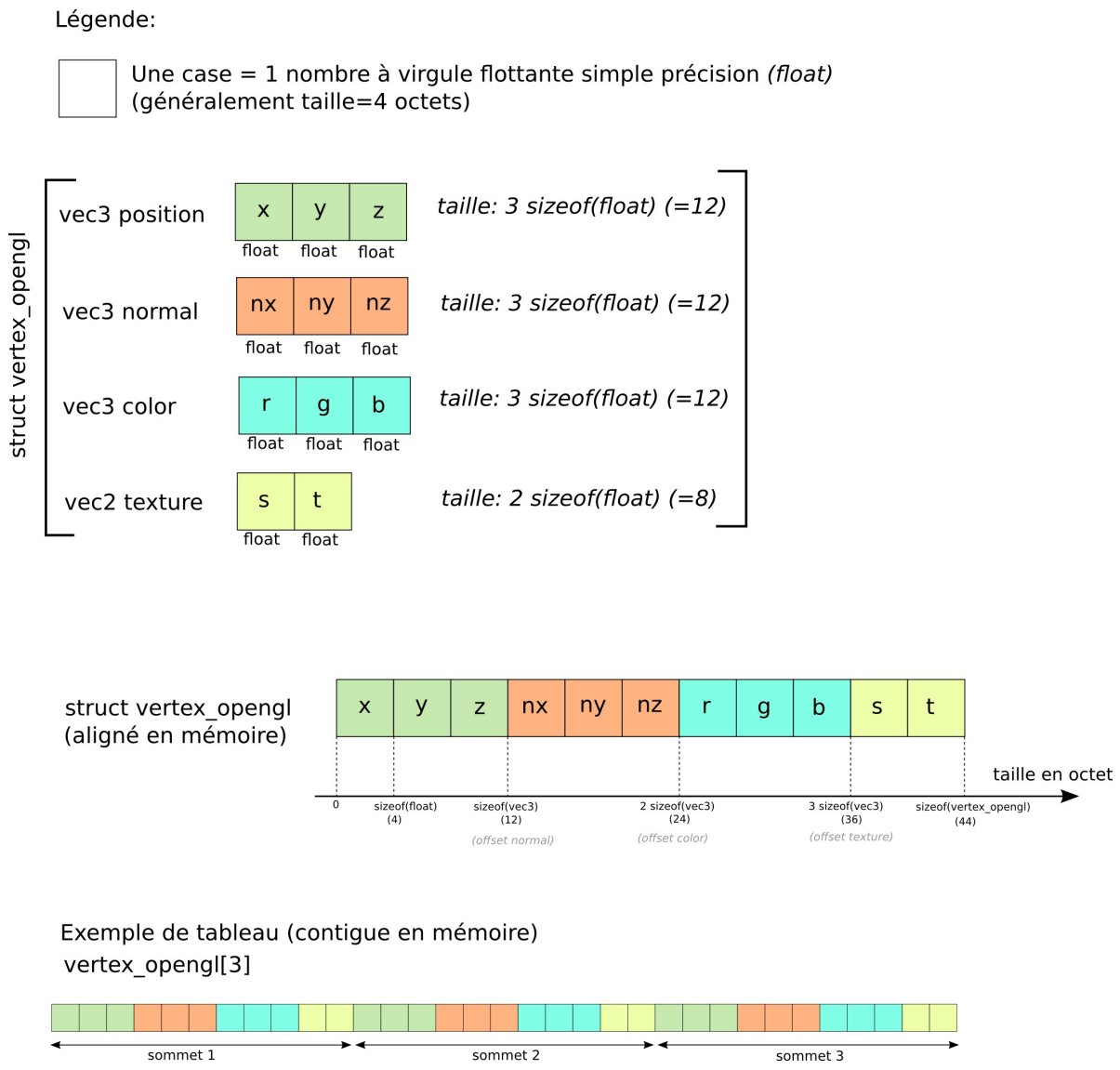


Fig. 6. Organisation de la mémoire pour un vertex_opengl[3].

➔ **Créez** la figure géométrique suivante (Fig. 7). Réfléchissez au placement des coordonnées de textures.

(Vous considérerez la texture de votre choix).

Fig. 7. Figure géométrique attendue
(Vous pouvez choisir la texture de votre choix)

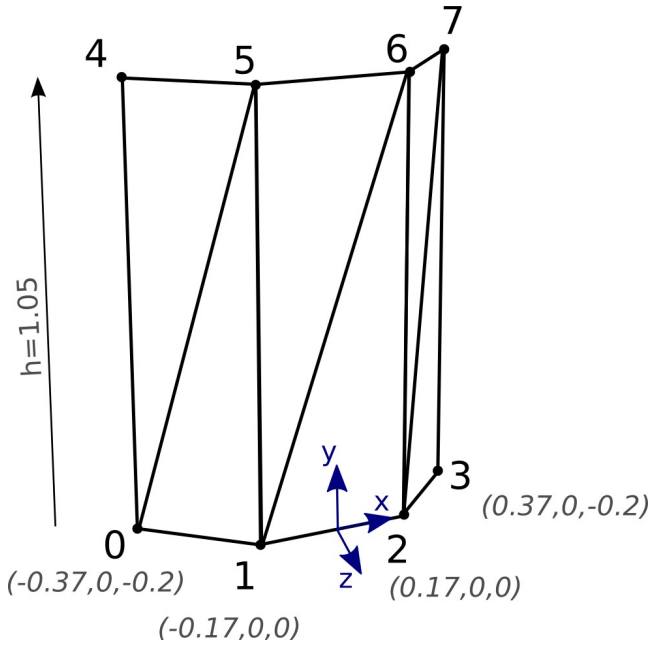


Schéma de coordonnées géométriques possibles

➔ Afin d'augmenter l'impression de profondeur, **modifiez** le maillage afin de créer un relief géométrique sur le nez du personnage comme illustré aux Fig. 8 et 9.

A vous de trouvez les coordonnées adéquates des nouveaux sommets.

Fig. 8. Le nez du personnage est cette fois réellement en relief.

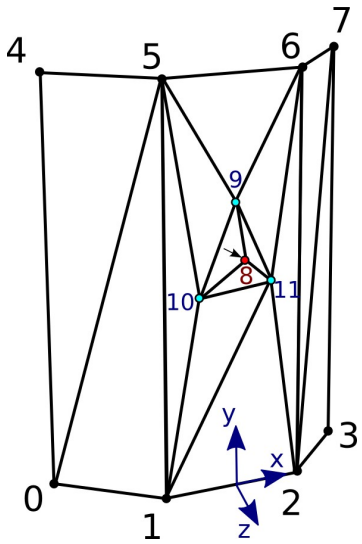


Fig. 9. Schéma de la structure maillée avec la géométrie du nez.

➔ Construisez un modèle 3D rassemblant à une voiture en suivant la démarche décrite ci-après.

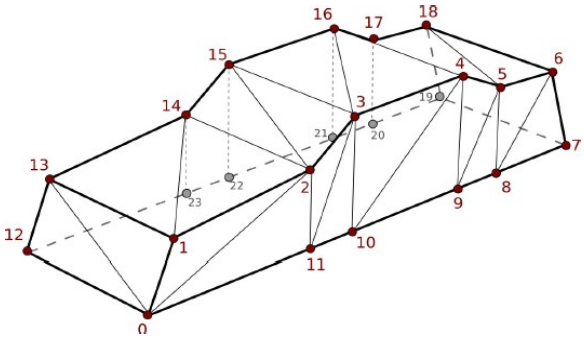


Fig. 10. Positions des sommets et connectivité

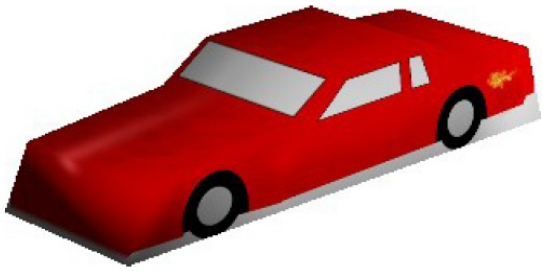


Fig. 11. Résultat visuel final possible

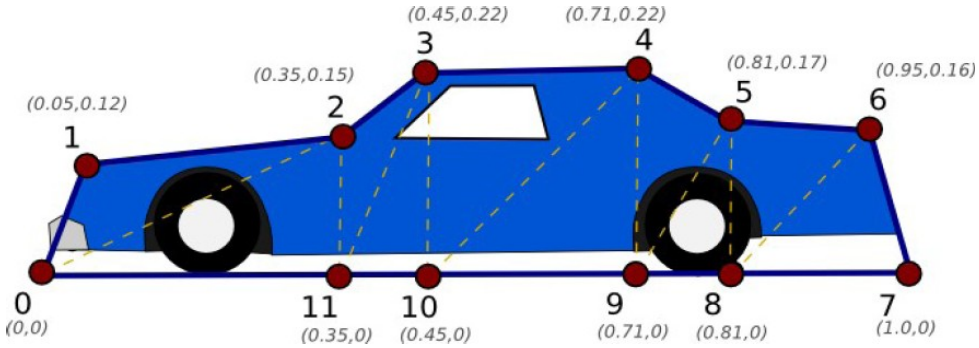


Fig. 12. Coordonnées (x,y) possibles des sommets du coté de la voiture.

Démarche

1. Dans un premier temps, construisez un seul coté de la voiture (voir Fig. 12. pour des coordonnées possibles) sans s'occuper des textures, cette face devrait être dans le plan d'équation z=0.

Fig. 13. Un seul coté de la voiture.

2. Dans un second temps, construisez la seconde moitié de la voiture (et complétez les triangles les reliants) pour obtenir la géométrie 3D de base. Réfléchissez à des normales adéquates.

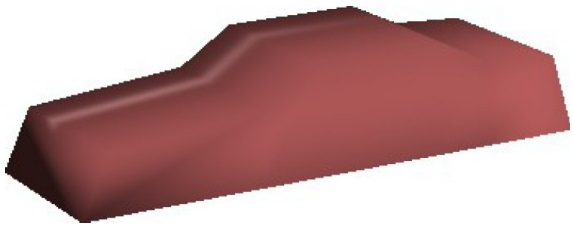


Fig. 14. Géométrie 3D de la voiture.

3. Dans un troisième temps, placez les coordonnées de textures sur la voiture. Pour vous aider, vous disposez de deux textures.

La première est une texture possible finale donnant le résultat visible sur la figure du dessus, l'autre étant un template visualisant la position des triangles, et annotée avec les coordonnées relatives (s,t) des positions particulières. Ces coordonnées peuvent servir de coordonnées de textures.

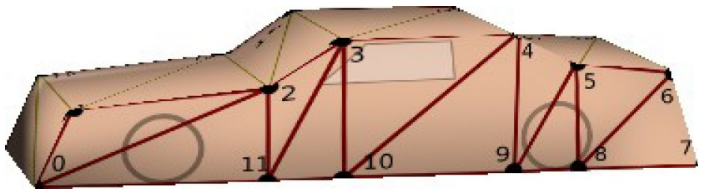


Fig. 15. Application du template de texture sur la géométrie.

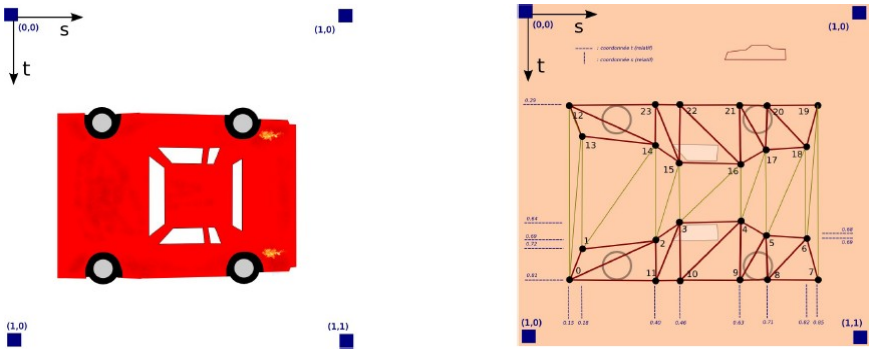


Fig. 16. Les deux textures fournies (gauche: texture de la voiture, droite: template avec mesures)

➔ **Réalisez** un jeu minimaliste à l'aide de votre voiture.

- L'utilisateur (la caméra) doit pouvoir se déplacer sur un terrain plat en avant ou en arrière, et pouvoir tourner sur lui même.
- La voiture se déplace elle aussi sur ce sol en suivant une trajectoire définie. On pourra tenter de modéliser un circuit automobile par exemple.
- L'utilisateur doit pouvoir lancer un projectile lors de l'appui sur la touche 'p'.
- Si le projectile atteint la voiture, la collision est détectée et la voiture est détruite.

Remarques

Vous êtes libres de choisir les différents détails de ce jeu (déplacement, vitesse, type de projectile et paramètres, type d'interaction avec la voiture, etc).

- Faites des choses **simples** dans un premier temps.
- Déplacement de la caméra par étapes.
- Voiture initialement immobile
- Projectile très simple, pas d'interaction avec l'environnement, etc.

Rendu

Rendez votre travail sous forme d'archive contenant:

- L'ensemble de vos fichiers sources (fichiers .cpp, .hpp, shaders).
- L'ensemble de vos textures.
- Une page avec des images de votre jeu illustrant chaque paramètre que vous avez mis en place.

Vous accompagnerez chaque image d'un commentaire expliquant la démarche que vous avez utilisés.
(Une page minimum. Essayez de vous restreindre à un maximum de 5 pages).