TP Synthèse d'images

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 évitez GEdit ou nano qui n'intentent pas automatiquement le code). Le fichier CMakeLists.txt est fourni dans le cas où vous souhaitez utiliser QtCreator (voir fiche annexe sur les IDE).

Notez que le code doit s'exécuter sans erreurs. Dans le cas contraire, il est possible que votre code soit appelé depuis un répertoire qui ne contient pas les fichiers de shaders (shader.frag et shader.vert). En particulier, il est nécessaire de paramétrer le répertoire d'execution avec QtCreator.

➔ **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[]={0.0f,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 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 (vous avez toujours une fenêtre vide).

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 rouge 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.

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

(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 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 600x600, les coordonnées x et y varient entre 0 et 600.
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/600.0;
  float g=gl_FragCoord.y/600.0;
  gl_FragColor = 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;

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

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

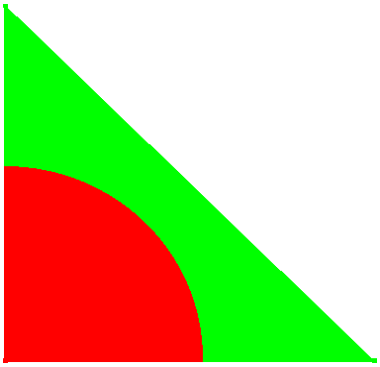


Fig. 1. *Exemple d'affichage obtenue*

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 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 valeur de `gl_Vertex` est affecté à `gl_Position`. `gl_Vertex` est une variable connue par défaut du vertex shader. Cette variable contient les coordonnées (dans l'espace 3D) de l'un des sommets du (ou des) triangle(s) de l'objet affiché (il s'agit de coordonnées à 4 dimensions, mais on ne traitera pas explicitement cette dimension dans notre cas).
Ici, cette variable contient donc les coordonnées de l'un des trois sommets du triangle. Notez bien que le vertex shader est executé en parallèle sur de nombreux sommets. Dans le cas présent, votre carte graphique exécute donc en parallèle 3 vertex shaders. L'un ayant la valeur (0,0,0) dans la variable `gl_Vertex`, 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 réalisé au niveau du fragment shader qui seront réalisés lorsque les 3 vertex shaders associés à chaque sommet du triangle auront terminés.

Il est possible 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;`

➔ **Expliquez** le résultat obtenu à l'écran.

➔ **Observez** également ce que réalise le code suivant.

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

gl_Position = p;
```

Passage de paramètres depuis le programme principal.

Il est possible de passer des variables depuis le programme principal (depuis la RAM du CPU) sur la carte graphique afin d'utiliser une valeur donnée dans le shader.

Considérez désormais le **programme_2**.
Notez que le vertex shader déclare désormais une variable vec4 qualifiée de "`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 executé sur le CPU.

Les valeurs de la variable translation sont envoyée sur la carte graphique par l'appel suivant dans la fonction display_callback:

```
glUniform4f(get_uni_loc(shader_program_id,"translation"),translati
on_x,translation_y,translation_z,0.0f);
```

Cette appel indique qu'une varible de type uniform du shader va recevoir un paramètre depuis ce programme. `get_uni_loc` permet de localiser la variable appelée textuellement "translation" dans le shader. Ensuite, les 4 valeurs flottantes sont envoyés dans le reste des paramètres.

➔ **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? Avez-vous une explication par rapport aux effets observés?

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.

➔ **Expliquez** les couleurs observées lorsque vous déplacer votre triangle dans la fenêtre.

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).
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 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.

- ➔ **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`.

Une dernière notion non pris en compte jusqu'à présent concerne la projection du triangle de l'espace 3D vers l'espace 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 apparaitra à la même taille qu'un objet proche.

Pour modéliser ce phénomène d'éloignement, il est nécessaire de considérer une troisième matrice: la matrice de projection qui va modéliser l'effet d'une caméra. La description de l'utilisation d'espace 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 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 cette distance la plus grande et la distance la plus faible.

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

- ➔ **Utilisez** les touches 'p' et 'm' pour déplacer votre triangle en profondeur. Observez l'effet de perspective (un triangle plus éloigné apparait plus petit qu'un triangle proche).
- ➔ **Notez** l'envoie d'une matrice de projection par le programme principal (ici uniquement envoyée dans la fonction d'init car les paramètres sont constants tout au long de l'affichage) ainsi que l'application de celle-ci dans le shader.

Tableau de sommets et affichage indexé.

Nous allons désromais 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,
                };
```

- ➔ **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`.

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'oublie 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).

➔ **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,
                };
```

➔ **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:

➔ **Créez** une variable globale `vboi` (signifiant "*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.

➔ Enfin, dans la fonction d'affichage, s**upprimez** la ligne du `glDrawArray`, et faites appel à: `glDrawElements(GL_TRIANGLES, 2*3, GL_UNSIGNED_INT, 0);`

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).

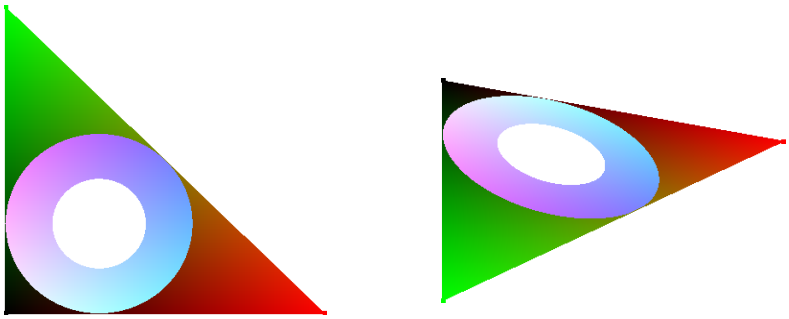
➔ **Executez** le programme et assurez vous que ayez le même résultat visuel que précédemment.

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 du triangle. Pour cela, la carte graphique va pouvoir donner une valeur de paramètre au fragment shader obtenue à partir de l'interpolation linéaire des valeurs données par le vertex shader.

Considérez le **programme_6**.
Notez cette fois la présence de la variable `coordonnee_3d` qualifiée de `varying`.
Dans le vertex shader cette variable est mise à jour avec la valeur du sommet considéré.
Ensuite, dans le fragment shader, `coordonnee_3d` contient la valeur des coordonnées interpolée linéairement en fonction de la position du fragment. Enfin, on récupère chaque coordonnée de cette valeur que l'on interprète comme une couleur *rouge*, *verte* ou *bleue*.
L'interpolation linéaire des coordonées aboutit donc 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.

➔ Tentez désormais de réaliser (ou d'approximer) cette figure sur l'un des triangles (les couleurs doivent cette fois être indépendamment de la position et 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.

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.

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

Observez l'envoie des coordonnées:
```
glVertexPointer(3, GL_FLOAT, 2*sizeof(vec3), 0);
```
Le second paramètre indique que l'écart 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)`.

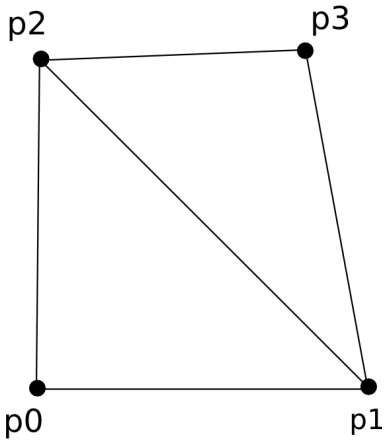
Observez également la mise en place d'un nouveau type de données, les normales:
```
    glEnableClientState(GL_NORMAL_ARRAY);
    glNormalPointer(GL_FLOAT, 2*sizeof(vec3),
                    buffer_offset(sizeof(vec3)));
```

Le second paramètre de glNormalPointer désigne également l'écart entre deux données de normales, et le 3ème paramètre 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)`). La fonction `buffer_offset` ne fait que changer le type d'une valeur entière vers un type pointeur attendue par la fonction.

Les informations de normales sont ensuites utilisées dans les shaders respectifs afin d'afficher une illumination de Phong.

➔ **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.

➔ **Ajoutez** un nouveau sommet p3 et un nouveau triangle (p1,p3,p2) afin d'obtenir le maillage suivant:



On considèrera `p3=(0.8,0.8,0.5)`. Et les différentes normales:
`n0=(0,0,1), n1=(-0.25,-0.25,0.85), n2=n1, n3=(-0.5,-0.5,0.707)`
associées respectivements à p0,p1,p2, et p3..

Notez que les normales pour p0 et p3 sont les normales approximatives de leur triangles respectifs. Notez également que les normales associées à p1 et p2 sont des moyennes des normales des deux triangles auquelles ils appartiennent.

Pour vous aider à réaliser cet ajout, notez que vous devez réaliser les actions suivantes sur le code:
- Créer un nouveau sommet p3 sous forme de `vec3`.
- Créer une nouvelle normale n3 sous forme de `vec3` et modifier les normales n1 et n2.
- Mettre à jour le tableau de geometrie en ajoutant p3 et n3.
- Creer un nouveau `triangle_index` (1,3,2) et l'ajouter au tableau `index`.
- Mettre à jour la demande d'affichage de 2 triangles (fonction `display_callback`).

➔ **Observez** que les deux triangles donnent l'impression de former une surface lisse (la séparation entre les deux triangles n'apparait 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:

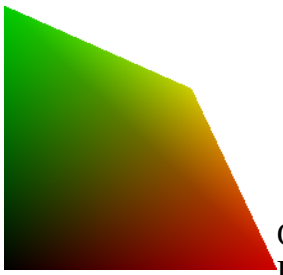
```
//coordonnees geometriques des sommets
vec3 p0=vec3(0.0f,0.0f,0.0f);
vec3 p1=vec3(1.0f,0.0f,0.0f);
vec3 p2=vec3(0.0f,1.0f,0.0f);
vec3 p3=vec3(0.8f,0.8f,0.5f);

//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 p0, p1, p2 et p3 devraient ainsi être respectivement: *noir*, *rouge*, *vert*, et *jaune*.

➔ **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 décallages pour les coordonnées et normales dans `glVertexPointer`.
- Ajout d'un nouveau type d'envoie de donnée: Activer l'utilisation des `GL_COLOR_ARRAY` (similairement aux `GL_VERTEX_ARRAY` et `GL_NORMAL_ARRAY`).
- Mise en place du pointeur de couleur à l'aide de la fonction fonction `glColorPointer()`. Réflechissez à l'écart et l'offset à appliquer.
- Dans le vertex shader, récupérez le contenu de la variable `gl_Color` contenant la couleur du sommet courant (sous forme de `vec4`) et passez le au fragment shader par le biais d'une variable varying.
- Dans le fragment shader, utilisez cette variable qualifiée de varying en tant que couleur (à la place de la couleur écrite en dur dans le shader actuel).

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

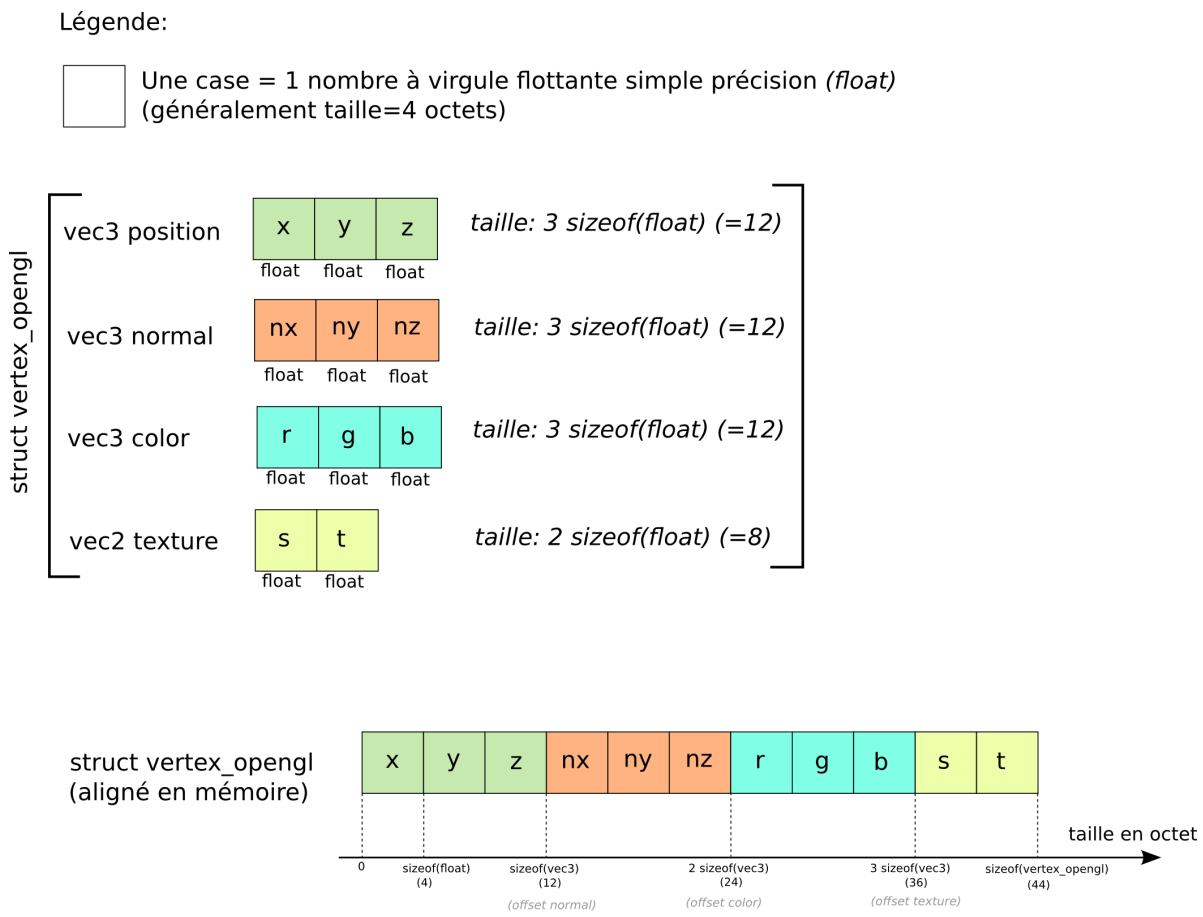


Considérez le **programme_8**.
En plus des couleurs, ce programme intègre également la gestion des textures.
De plus, cette fois, nous ajoutons une structure supplémentaire permettant d'organiser plus aisément les données: un `vertex_opengl` qui contient des coordonées 3D, une normale, une couleur, et une coordonnée de texture à 2 composantes. Une figure explicatrice est fournie ci après.

- ➔ **Observez** qu'à l'execution du programme, la couleur de la texture est *mixée* (multiplication) avec la couleur définie pour chaque sommet.
- ➔ **Observez** la mise en place du pointeur sur les coordonées de texture dans la fonction d'initialisation.
- ➔ **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 quelconque en tga à l'aide de Gimp ou d'ImageMagick). Notez que les images avec transparences ne sont pas acceptées.
- ➔ **Modifiez** les coordonnées de textures afin de comprendre leurs principes. Vous pouvez également modifier les couleurs et l'image utilisée.
- ➔ **Que ce passe t-il** lorsque les coordonnées de textures sont inférieurs à 0 ou supérieurs 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.



Organisation de la mémoire pour un vertex_opengl[3].

Scène complexe.

Considérez le **programme_9**.

Cette fois une scène plus complexe contenant plusieurs objets est affichée.
Pour faciliter la manipulation d'un objet spécifique, la structure `info_object_opengl` permet de stocker les informations des deux vbo, indice de texture, et nombre de triangle total de l'objet afin de permettre son affichage.
L'affichage d'un objet à partir d'un vbo prend de manière générique en paramètre une rotation et une translation qui permet de déplacer chaque objet de manière séparée.

Les fonctions contenues dans `opengl_aide_affichage` permettent de réaliser de manière concises différentes opérations à partir de la structure `info_object_opengl`.
On notera: la demande d'affichage d'un objet à partir de cette structure, le chargement d'une image de texture, et l'envoie de données d'un maillage sur la carte graphique.

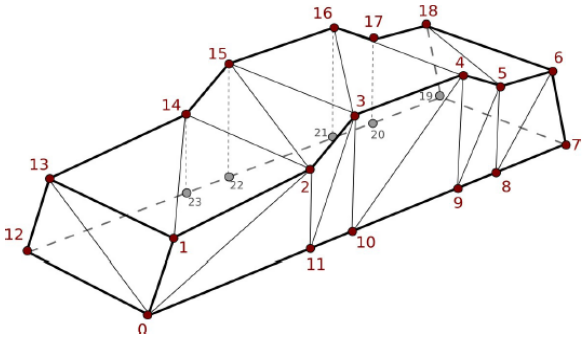
Une structure maillage est également fournie. Il est possible de charger un maillage à partir d'un fichier (format .off ou .obj) qui peut être réalisé par un logiciel de modélisation (ex. Blender).

Le chargement des différents objets ainsi que leurs paramètres spécifiques sont codés dans le fichier scene.cpp.

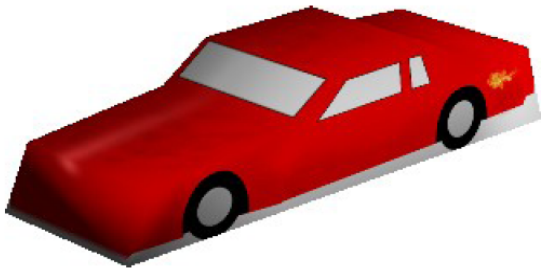
- ➔ Observez comment chaque objet est affiché avec des rotations et translations potentiellements différentes.

Notez que si l'ensemble des objets subissent une même transformation, alors on aura l'impression de se déplacer dans la scène.

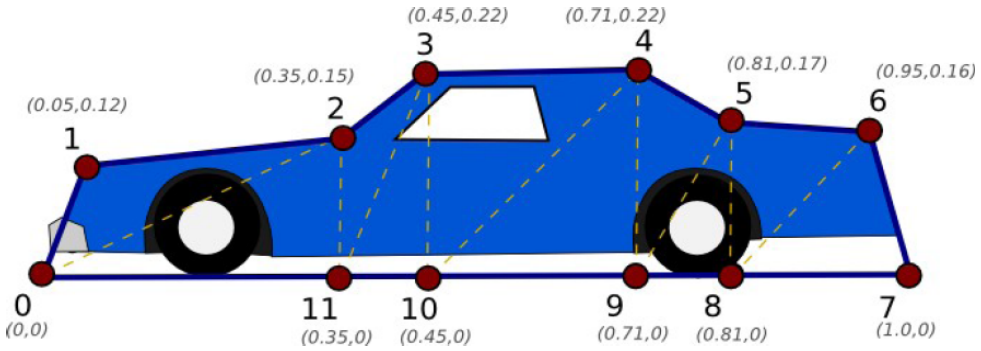
➔ Construisez un modèle 3D rassemblant à une voiture en suivant la démarche décrite ci-après.



Positions des sommets et connectivité



Résultat visuel final possible



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

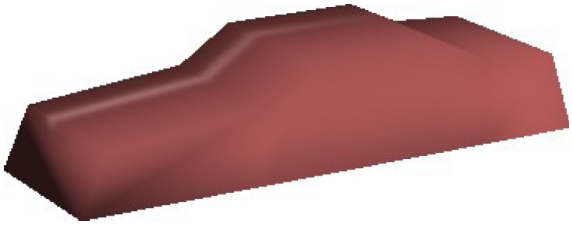
Démarche

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.

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

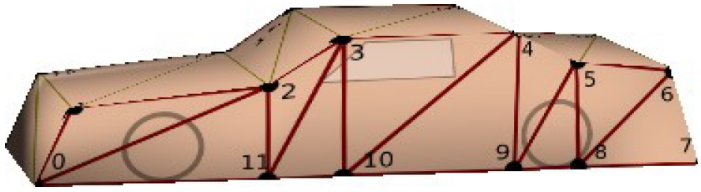
adéquates.



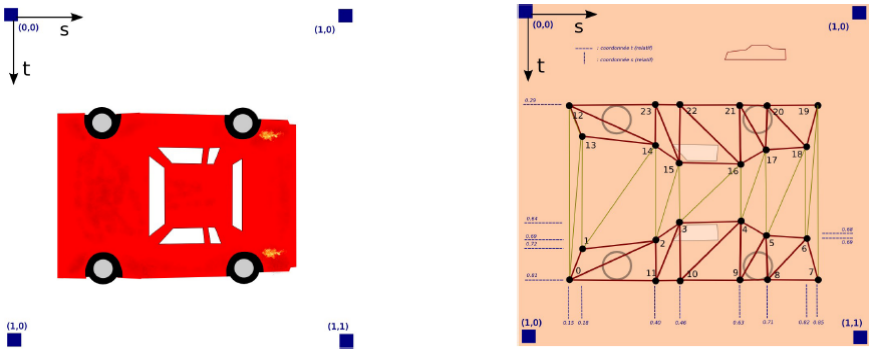
Géométrie 3D de la voiture.

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.



Application du template de texture sur la géométrie.



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).