OpenGL - TD 02

Dessin, Objets canoniques et Transformations

Lors de cette séance, nous allons dessiner des formes basiques à la souris. Nous aborderons aussi un des concepts les plus simples de la modélisation en 2D (et en 3D) : La construction d'objets canoniques.

Exercice 01 – Premier points

Au précédent TD, nous avons mis en place la fenêtre et vu les interactions hommes machines, et nous allons pouvoir maintenant dessiner ! Avant toute chose, il faut savoir que la bibliothèque OpenGL exploite intensemment la notion d'état. A savoir qu'il existe un grand nombre d'état permettant le dessin, que chaque état a une valeur par défaut, et qu'il est possible de modifier les états. Une fois un état modifié, il conserve cette valeur.

Ainsi, un état « classique » dans OpenGL (ancien) est la couleur des objets (primitives graphiques) que l'on dessine. Par défaut, c'est blanc. Mais cet état peut être modifié en appelant l'instruction l'instruction `glColor3f(r, v, b)` où `r`, `v`, `b` sont trois flotants compris entre 0 et 1, et définissent les trois composantes rouge, verte et bleue de la couleur désirée. Une fois cette fonction appelée, l'état « couleur de dessin » change et alors tous les dessins effectués par la suite auront cette couleur.

Nous allons maintenant commencer par dessiner de simples points. Pour ce faire, nous allons utiliser les instructions suivantes :

```
glBegin(GL_POINTS);
    glVertex2f(x, y);
    // eventuellement d'autres points …
glEnd();
```

où `glBegin()` / `glEnd()` définissent la primitive graphique à utiliser (ici des points : `GL_POINTS`) et où à l'intérieur nous spécifions les coordonnées des points à tracer. On définit ces points grâce à l'instruction `glVertex2f(x,y)` où `x` et `y` sont les coordonnées du point à afficher dans l'espace virtuel (voir TD précédent, pour rappel les coordonnées (0,0) sont en haut à gauche de la fenêtre et l'espace virtuel pour ce TD sera [-1,1]x[-1,1]).

Notez qu'entre `glBegin()` et `glEnd()`, il est possible d'utiliser les commandes `glColor3f` (ou autres variantes de `glColor`), `glVertex2f` (ou autres variantes de `glVertex`) et tout ce que vous avez appris en C (boucle, tests, etc). Les autres fonctions OpenGL ou GLFW ne sont pas acceptées.

L'ensemble des **vertices** (sommets en français) que vous créez entre ces deux instructions constitue la liste des sommets qui formeront la primitive à afficher. Par exemple, si vous utilisez `GL_POINTS` en paramètre de `glBegin`, alors chaque appel à `glVertex`, OpenGL dessinera un unique point.

A faire :

01. Pour vous échauffer, modifiez le code de dessin pour afficher :
- un point blanc en (0. , 0.)
- un point rouge en (0.5 , 0.)
- un point vert en (0. , 0.5)
- un point violet en (-0.5 , -0.5)

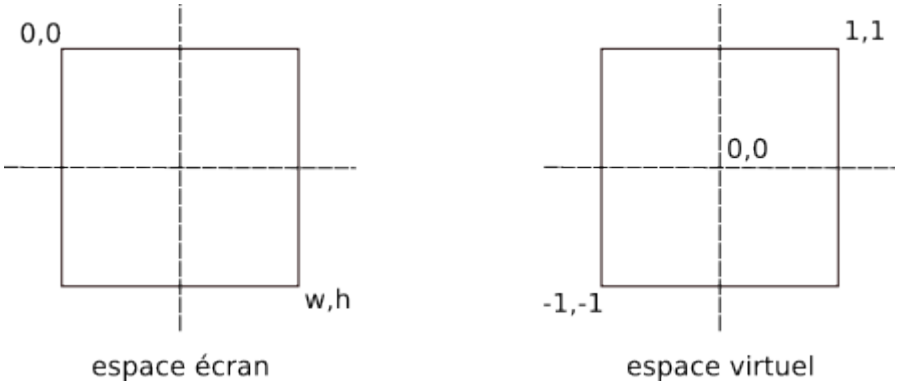
Exercice 02 – Dessin à la souris

Nous voudrions maintenant placer un sommet à chaque clic dans la fenêtre. L'objectif de cet exercice est donc de redessiner tous les points cliqués à chaque tour de la boucle principale. Pour cela, vous devrez enregistrer ces derniers en mémoire au moment d'un clic.

A faire :

01a. Faites en sorte que lorsque l'utilisateur clique, les coordonnées du vertex à dessiner soient stockées dans des tableaux. (Ou mieux : dans un tableau de structures `Vertex` ?)

Ayez en tête que l'origine du repère de la fenetre GLFW est en haut à gauche de cette dernière, là ou l'origine du repère de la scène est au centre de la fenêtre. Pour cette raison, vous devrez convertir les coordonnées de l'espace écran à l'espace virtuel que nous avons défini (ici il va de -1 à 1 sur chacun des axes, car `GL_VIEW_SIZE` est à 2.0 dans notre fichier de base).



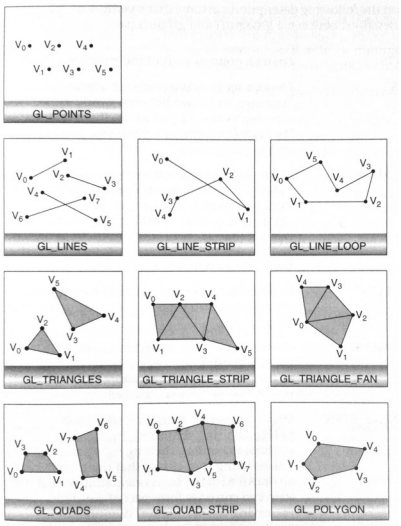
Il vous faudra donc trouver la bonne conversion. Dans un premier temps en considérant une fenêtre carrée (comme ci-dessus). Puis vous devrez exploiter `aspectRatio`, selon l'orientation de la fenêtre (portrait ou paysage), pour obtenir le bon « mapping » dans le cas d'une fenêtre rectangulaire.

01b. Assurez vous ensuite qu'à chaque tour de boucle le contenu de la fenêtre soit effacé et que tous les sommets enregistrés dans le(s) tableau(x) soient redessinés à l'écran.

Note : Lorsque nous avons dessiné des primitives graphiques à l'aide de `glBegin` / `glEnd`, nous avons précisé, dans l'argument de `glBegin`, la primitive graphique à dessiner, dans ce cas `GL_POINTS`. Il est évidemment possible de dessiner d'autres primitives graphiques.

Voici la liste des primitives (OpenGL ancien) :

- `GL_POINTS`
- `GL_LINES`
- `GL_LINE_STRIP`
- `GL_LINE_LOOP`
- `GL_TRIANGLES`
- `GL_TRIANGLE_STRIP`
- `GL_TRIANGLE_FAN`
- `GL_QUADS`
- `GL_QUAD_STRIP`
- `GL_POLYGON`



A l'intérieur du `glBegin` / `glEnd`, OpenGL attend le bon « nombre » de vertex avant de dessiner la primitive. Dans le cas où il y aurait un nombre insuffisant de vertex, la primitive demandée ne sera pas dessinée.

02. Faites en sorte qu'appuyant sur une touche du clavier, une ligne bouclant sur elle même remplace les points dessinés jusqu'ici.

Indice : Vous devriez stocker le type de primitive à dessiner dans une variable

Exercice 03 – Des objets canoniques…

Votre maîtrise des primitives fraîchement acquise, vous allez maintenant pouvoir créer des objets un peu plus complexes. Ces objets sont de taille unitaire, et centrés sur l'origine. On les considère comme des objets « canoniques » car en les composant on peut obtenir des objets encore plus complexes.

A faire :

Implémentez les fonctions suivantes permettant de dessiner des objets canoniques :

01. `drawOrigin()`

Dessine les axes à l'origine, c'est à a dire :
- Un segment de taille 1 et de couleur rouge sur l'axe horizontal (axe des *x*)
- Un segment de taille 1 et de couleur verte sur l'axe vertical (axe des *y*)

02. `drawSquare()`

Dessine un carré de coté 1, centré sur l'origine.

03. `drawCircle()`

Dessine un cercle de diamètre 1, centré sur l'origine. Vous représenterez ce cercle en l'approximant par une série de segment de droite car OpenGL ne sait dessiner que des points, des segments et des triangles. Ce nombre de segment est à définir en tant que constante dans votre code. Vous devez ajouter en début de code la bibliothèque `math.h` pour l'accès aux fonctions trigonométrique.

04. Pour finir, réalisez l'affichage de ces objets canoniques dans une fenêtre GLFW. A l'aide d'une touche on doit 'boucler' sur l'affichage des trois objets canoniques.

Rappel : Tout dessin doit se faire entre `glClear` et `glfwSwapBuffers`.

Optionnel : 05. Ajouter un paramètre `full` aux fonctions précédentes. Ce paramètre permettra choisir si l'objet est à dessiner plein ou seulement en contour : si `full` vaut 0, afficher uniquement le contour, si `full` vaut 1, remplir le polygone (pour le carré et le cercle bien sûr).

Exercice 04 – Transformations !

Les deux objets canoniques (cercle et carré) définis dans l'exercice 03 sont peu intéressants en l'état… Cependant OpenGL à la particularité de pouvoir déplacer et/ou déformer les objets qu'il dessinera par la suite. Cela se fait en modifiant la matrice de transformation d'OpenGL.

Avant de modifier cette matrice, il faut utiliser l'instruction suivante :

```
glMatrixMode(GL_MODELVIEW);
```

Cette ligne permet de dire que la matrice courante (celle que l'on va modifier) est la matrice de transformation (il y en a d'autres comme la matrice de projection `GL_PROJECTION`).

Pour demander le chargement de la matrice identité, on utilise `glLoadIdentity()`. Cette fonction fixe la matrice courante à la matrice identité.

On peut effectuer une translation à l'aide de `glTranslatef(x, y, z)` où `x`, `y` et `z` doivent être des flottants.

On peut effectuer une rotation dans le plan (*x*0*y*), à savoir autour de l'axe *z*, à l'aide de `glRotatef(alpha, 0., 0., 1.)` où `alpha` est l'angle de rotation en degrés.

Enfin on peut effectuer une mise à l'échelle à l'aide de `glScalef(x, y, z)` où `x`, `y` et `z` doivent être des flottants.

Note : La matrice de transformation `GL_MODELVIEW` s'applique à tous les points que l'on transmet à la carte graphique par la suite (i.e. jusqu'à nouvel ordre).

A faire :

01. Modifiez la taille de l'epace virtuel (`GL_VIEW_SIZE`) pour qu'il fasse 6 de coté. La fonction `onWindowResized` se chargera d'ajuster les dimensions de cet espace aux proportions de la fenêtre, pour éviter d'étirer les formes dessinées à l'écran.

02. Dessinez le repère dans votre fenêtre.

03. Modifiez le code d'affichage pour afficher un cercle orange centré en (1, 2). Bien entendu, vous ne devez pas modifier `drawCircle`

04. Changez le code d'affichage pour ajouter un carré rouge mais après qu'il ait subi une rotation de 45°, puis une translation de 1.0 sur l'axe des *x*.

05. Permutez la rotation et la translation pour dessiner un carré violet supplémentaire. Relevez la différence de positions entre ce carré violet et le carré rouge dessiné en 04.

Question :

06. Expliquez la différence de positions entre les carrés rouge et violet, dessinés en 04 et 05. Laquelle des deux approches vous semble la plus adaptée pour dessiner un objet en OpenGL ?

A faire :

07. Faites en sorte qu'en cliquant dans la fenêtre, on déplace le centre d'un carré vert, situé initialement en (0, 0), à l'emplacement du clic.

Optionnel 08. Faites en sorte qu'en maintenant le bouton droit de la souris, on puisse effectuer une rotation du carré jaune (par rapport a son centre) en déplaçant le curseur. (Plus simplement, vous pouvez aussi faire tourner le carré jaune à une vitesse constante en maintenant appuyé un bouton ou une touche du clavier)

Optionnel 09. Faites en sorte qu'un cercle bleu se déplace aléatoirement dans la fenêtre, en permanence.

Exercice 05 – Vers un outil de dessin ? (Optionnel)

Attention : Ignorez cet exercice tant que vous n'avez pas vu les structures de données en algorithmique.

Cet exercice vous propose d'améliorer le code fourni afin de réaliser un petit programme de dessin. Un utilisateur voudrait pouvoir :

- changer le type de primitive (point, ligne, triangle, …) qu'il souhaite dessiner
- choisir la couleur dans laquelle il dessine ces primitives

Choix des primitives :

L'utilisateur devrait pouvoir basculer d'un type de primitive à l'autre en appuyant sur une touche du clavier, par exemple :

- `P` pour dessiner des points
- `L` pour dessiner des lignes
- `T` pour dessiner des triangles

Pour cela, vous devrez faire varier le mode de primitive que vous passez à `glBegin()` lorsque un événement associé à l'une de ces touches est détecté. Attention : Chaque primitive attend un certain nombre de points pour pouvoir être dessinée : 1 pt pour `GL_POINTS`, 2 pts pour `GL_LINES` , 3 pts pour `GL_TRIANGLES`

Choix de la couleur :

Idéalement, l'utilisateur pourrait aussi changer la couleur de dessin, en la sélectionnant sur une palette qu'il afficherait au moyen de la touche `Espace`.

- Lorsque l'utilisateur appuie sur `Espace`, la fenêtre n'affiche plus le dessin en cours, mais des carrés de différentes couleurs.
- Si l'utilisateur clique sur l'un de ces carrés, le programme doit enregistrer la couleur de ce carré comme étant la couleur avec laquelle seront dessinées les prochaines primitives.
- Lorsque l'utilisateur relâche `Espace`, le programme revient en mode dessin.

Il faut donc désormais gérer deux états : le mode dessin et la palette, qui ont chacun leur comportements distincts. Pour garder trace de l'état un simple entier à 0 où à 1 peut suffire.

Le point central de cet exercice est de faire en sorte que l'état du mode dessin garde en mémoire les primitives, qui sont effacées lorsque l'utilisateur bascule sur la palette. Il faut donc stocker ces primitives et leurs points dans des structures appropriées, afin de garder trace de l'ordre dans lequel les points ont été dessinés, et de pouvoir ré-afficher les primitives aussitôt que le programme revient en mode dessin.

Désormais, votre programme écrira les sommets cliqués dans une structure de données. Il lira cette structure lors de l'affichage pour savoir où et comment dessiner les points.

A faire : Structurez le programme, pour stocker les primitives et leurs sommets. Un vertex stockera sa position en coordonnées OpenGL, et sa couleur.

01. Ajoutez la structure `point_link` dans votre code :

```
typedef struct vertex_link {
 float x, y;                 // Position 2D du vertex
 float r, g, b;              // Couleur du vertex
 struct point_link *next;    // Suivant
} Vertex, *VertexList;
```

Une liste chaînée vous permet de stocker un nombre indéfini de points, par ordre de création.

Pour utiliser `point_link` , vous aurez besoin d'implémenter les fonctions suivantes :

02. Allocation de mémoire pour stocker un vertex et l'initialiser :

```
Vertex* allocVertex(int x, int y,float r, float g, float b);
```

03. Libération de mémoire allouée pour une liste de vertex :

```
void deleteVertices(VertexList* list);
```

04. Ajout en fin de liste d'un vertex passé en paramêtre

```
void addVertex(Vertex* vertex, VertexList* list);
```

05. Dessin de tous les vertex d'une liste (avec `glVertex2f` et `glColor3f`)

```
void drawVertices(VertexList list);
```

A faire : Vous aurez également besoin de stocker des informations sur les primitives à utiliser. Une primitive encapsulera une liste de vertex, à afficher selon le type de primitive donné.

06. Ajoutez la structure `primitive_link` dans votre code:

```
typedef struct primitive_link {
    GLenum primitiveType;           // Type de primitive
    VertexList vertices;            // Liste des vertex
    struct primitive_link *next;    // Suivant
} Primitive, *PrimitiveList;
```

Les primitives seront également stockées sous la forme d'une liste chaînée.

Pour utiliser `primitive_link` , vous aurez besoin d'implémenter les fonctions suivantes :

07. Allocation de la mémoire pour stocker une primitive. Le champs `vertices` doit être fixé à `NULL`.

```
Primitive* allocPrimitive(GLenum primitiveType);
```

08. Libération de la mémoire allouée à une liste de primitive (et aux listes de sommets sous jacentes) :

```
void deletePrimitives(PrimitiveList* list);
```

09. Ajout en tête de liste la primitve passée en paramètre

```
void addPrimitive(Primitive* primitive, PrimitiveList* list);
```

10. Dessine l'ensemble des primitives d'une liste

```
void drawPrimitives(PrimitiveList list);
```

Vous avez désormais les éléments nécessaires pour gérer le changement et le dessin de primitives dans votre programme.

11a. A l'initialisation, créez une liste de primitives, contenant par défaut première primitive de type `GL_POINTS`.

11b. En fin de programme, n'oubliez pas de libérer l'espace alloué aux primitives.

12a. Lorsque l'utilisateur appuie sur une touche pour changer de primitive (`P` , `L` , `T`), ajoutez une nouvelle primitive à la liste et tant pis si la précédente n'est pas finie.

12b. Lorsque l'utilisateur clique dans le viewport, ajoutez un point de la couleur courante à la primitive courante.

12c. A chaque tour de boucle, dessinez l'ensemble des primitives.

A faire :

Il vous reste désormais à implémenter le changement de couleur.

13. Faites en sorte que le programme passe en mode palette quand l'utilisateur maintient le bouton `Espace` appuyé, et revienne en mode dessin lorsqu'il le relâche.

14a. Lorsque le programme est en mode palette, affichez des carrés de couleur.

14b. Faites en sorte que lorsque l'utilisateur clique sur l'un de ces carrés, la couleur courante devienne la couleur de ce quad.

Notez que vous pouvez faire fonctionner cette partie avec autre chose que des carrés. Utiliser des bandes de couleurs peut constituer une approche plus facile à mettre en œuvre, et travailler avec des cercles pourrait constituer une solution très élégante.

L'essentiel est que vous ayez des zones de couleur, via lesquelles il est possible de changer la couleur de dessin en cliquant dessus (via les coordonnées stockées dans l'événement GLFW).

Optionnel 15. A ce stade, il serait intéressant de séparer le code pour le dessin et la gestion des événements de chaque état dans des fonctions dédiées, pour obtenir un code plus lisible. Par exemple avec des fonction du type :

`void handleEvents_paintState(), void draw_paletteState()...`