# Initiation à OpenGL et SDL

L'objectif de cette matière est de vous initier à OpenGL (en 2D, cette année).
Ce premier TP va également vous permettre de prendre en main la SDL afin de créer une fenêtre dans laquelle un rendu OpenGL pourra être effectué. Nous verrons également comment intégrer la gestion des événements de type clavier, souris, etc…

## Exercice 01 – Késako

Avant de commencer à travailler, il serait bon de savoir quels outils nous utilisons.

**Questions :**
À l'aide d'internet, définissez rapidement :
**01.** Qu'est-ce qu'OpenGL ?
**02.** Qu'est-ce que la SDL ?

## Exercice 02 – Votre première fenêtre

Maintenant que vous savez ce que sont OpenGL et SDL, il est temps de s'en servir.
Cet exercice ne vous demandera pas d'écrire de code, il vise à vous montrer la manière dont nous allons travailler, notamment via l'outil de compilation **make**.

L'archive TD01.zip est disponible sur le site du chargé de TD. (A définir)

Le fichier *minimal.c* , situé dans le répertoire *TD01/doc/*, constitue la base de code sur laquelle nous travaillerons avec OpenGL et SDL.
Vous trouverez également le fichier *makefile*, à la racine du répertoire *TD01/* .

**Questions :**
**01.** Que signifient les directives `#include` au début de *minimal.c* ?
**02.** Quelles sont les bibliothèques utilisées pour OpenGL et SDL ?
**03.** Comment sont elles appelées à la compilation dans le *makefile* ?
**04.** Quelle est l'utilité de la fonction `SDL_SetVideoMode(…)` ?

Le fichier source pour ce programme est donné dans *TD01/src/ex02/* .
(Il s'agit d'une simple copie de *minimal.c*)

Pour compiler à partir du makefile, ouvrez un terminal dans *TD01/*, et entrez la commande :

```
make ex02
```

Le programme *td01_ex02.out* est crée dans le répertoire *TD01/bin/* .
Pour l'exécuter , entrez la commande (toujours à partir de *TD01/*) :

```
bin/td01_ex02.out
```

## Notice – Organisation d'un répertoire de TD

En faisant l'exercice 02, vous avez pu prendre en main la compilation et l'exécution d'un programme utilisant OpenGL et SDL.
Vous aurez sans doute remarqué que les fichiers de travail se trouvent à différents endroits. (Et peut être même remarqué l'apparition d'un répertoire *TD01/obj/*)

Un répertoire de TD est pensé comme un objet indépendant, vous pourrez l'extraire du .zip sur votre machine la où vous le souhaitez, et travailler immédiatement dedans (ou presque).

Le répertoire pour chaque TD sera divisé en trois sous-répertoires :
- *bin/* : les exécutables crées via **make** seront placés ici
- *doc/* : contient divers fichiers de référence, notamment *minimal.c* et l'énoncé du TD
- *src/ :* contient les fichiers sources sur lesquels vous travaillez pour chaque exercice
- *obj/* : contient les fichiers **.o* générés lors de la compilation

Avant de démarrer un nouvel exercice, vous aurez besoin de faire ceci :
- créer un répertoire pour votre exercice dans */src*, comprenant un nouveau fichier source
- mettre a jour le *makefile* du TD, pour qu'il puisse trouver ce fichier et le compiler

**A faire :**
**01.** Créez le répertoire *TD1/src/ex03/*

```
mkdir -p src/ex03
```

**02.** Copiez le fichier *minimal.c*, et renommez le en *td01_ex03.c*
(Une autre option est de repartir du fichier source de l'exercice précédent)

```
cp doc/minimal.c src/ex03/td01_ex03.c
cp src/ex02/td1_ex02.c src/ex03/td01_ex03.c
```

**03a.** Ouvrez *TD1/makefile*

```
gedit makefile
```

**03b.** Ajoutez les lignes suivantes, dans la partie `# Fichiers TD 01` :

```
# Fichiers exercice 03
OBJ_TD01_EX03= ex03/td01_ex03.o
EXEC_TD01_EX03= td01_ex03.out
```

Ajoutez les lignes suivantes, dans la partie `# Regles compilation TD 01` :

```
ex03 : $(OBJDIR)$(OBJ_TD01_EX03)
    $(CC) $(CFLAGS) $(OBJDIR)$(OBJ_TD01_EX03) -o $(BINDIR)$
(EXEC_TD01_EX03) $(LDFLAGS)
```

Vous êtes désormais en mesure de compiler les fichiers pour l'exercice 03 et de générer l'exécutable *td01_ex03.out* dans le répertoire *TD01/bin*, en entrant simplement :

```
make ex03
```

## Exercice 03 – Domptage de la fenêtre

La fonction `SDL_SetVideoMode` permet d'ouvrir une fenêtre et de fixer ses paramètres. Elle à la signature suivante :

```
surface = SDL_SetVideoMode(
    WINDOW_WIDTH, WINDOW_HEIGHT, BIT_PER_PIXEL,
    SDL_OPENGL | SDL_GL_DOUBLEBUFFER | SDL_RESIZABLE);
```

Le pointeur qu'elle renvoie ne nous servira pas dans le cas d'une application OpenGL.

Les paramètres `width` et `height` permettent de fixer les dimensions de la fenêtre.

Le paramètre `bitsperpixel` spécifie le nombre de bits que la SDL doit utiliser pour encoder un pixel. Les valeurs qu'on utilise le plus souvent sont :
- 8 pour une fenêtre affichant des images en niveau de gris,
- 24 pour un affichage en rouge, vert, bleu
- 32 pour un affichage en rouge, vert, bleu + transparence.

Le paramètre `flags` est un champ de bits permettant d'énumérer divers paramètres à l'aide de l'opérateur `|` de la manière suivante : `param1 | param2 | … | param_n`.

Voici un exemple simple d'appel permettant d'ouvrir une fenêtre de dimensions 10x10 en niveaux de gris, pour OpenGL et en plein écran :

```
SDL_SetVideoMode(
    10, 10, 8, SDL_OPENGL | SDL_GL_DOUBLEBUFFER |SDL_FULLSCREEN)
```

Attention :
Si vous exécutez ce code vous ne pourrez plus quitter l'application proprement
car la croix permettant de fermer la fenêtre disparaît.

Vous pourrez trouver sur cette page la liste des paramètres pouvant être placés dans `flags` :
http://sdl.beuc.net/sdl.wiki/SDL_SetVideoMode

La fonction `SDL_WM_SetCaption` permet quand à elle de modifier le titre de la fenêtre et son icône. La page suivante décrit son fonctionnement :
http://sdl.beuc.net/sdl.wiki/SDL_WM_SetCaption

**A faire :**
À partir du fichier *minimal.c* et des fonctions ci-dessus :

**01.** Créez une fenêtre redimensionnable ( `SDL_RESIZABLE` ) pour OpenGL de taille 400x400.
**02.** Changez le titre.

Finalement, nous voulons effacer le contenu de la fenêtre ( i.e. appliquer une couleur unique).
**03.** Insérez au bon endroit du `main` l'instruction `glClear(GL_COLOR_BUFFER_BIT);`

## Exercice 04 – Des événements …

La SDL permet une gestion poussée et très libre des événements utilisateur.
Lorsqu'elle détecte un événement, elle place dans une file interne une structure de type `SDL_Event` décrivant l'événement.
Il est nécessaire de traiter tous les événements reçus à chaque tour de la boucle d'affichage.

La fonction `int SDL_PollEvent(SDL_Event *event)` permet de défiler un événement stocké dans la file interne. Elle renvoie 0 si il n'y a aucun événement a défiler.
Sinon, elle renvoie 1, et remplit `*event` avec les informations concernant l'événement.

On peut défiler et traiter tous les événements de la file avec une boucle de la forme suivante :

```
while(SDL_PollEvent(&e)) {
    /* Traiter evenement */
}
```

La structure `SDL_Event` est détaillée sur la page suivante :
http://sdl.beuc.net/sdl.wiki/SDL_Event

Comme on peut le voir, ce n'est pas vraiment une structure mais une union.
Le champs `type` indique le type de l'événement reçu. Il indique quel champs doit être manipulé par la suite dans l'union.

Par exemple, si `e.type` vaut `SDL_KEYDOWN` ou `SDL_KEYUP`, on doit accéder au champ `e.key` pour connaître les propriétés de l'événement.
Par contre si `e.type` vaut `SDL_SDL_MOUSEBUTTONUP` il faut accéder à `e.button`.

Chaque type d'événement est décrit sur la page :
http://sdl.beuc.net/sdl.wiki/SDL_Event_Structures

**A faire :**
**01.** Modifiez le programme pour qu'il s'arrête lorsque l'utilisateur appuie sur la touche `Q`.
Vous pourrez ainsi quitter le programme proprement, même en plein écran. (c.f. exercice 03)

**02.** Faites en sorte que lorsque l'utilisateur clique en position (x, y), la fenêtre soit redessinée avec la couleur (rouge = x mod 255, vert = y mod 255, bleu = 0) au prochain tour de la boucle d'affichage, plus précisément au prochain appel de `glClear(GL_COLOR_BUFFER_BIT)`.

Il faut utiliser la fonction `glClearColor(r, g, b, a)` pour modifier la couleur de remplissage / nettoyage de la fenêtre.
`r`, `g`, `b` et `a` doivent être compris entre 0 et 1, il faut donc diviser les valeurs `r`, `g`, `b` comprises entre 0 et 255 par 255, avant de les passer à OpenGL. Pour `a` il suffit de passer 1.

**03.** Faites de même mais lorsque la souris bouge (événement `SDL_MOUSEMOTION`).
Pour distinguer du précédent événement, faites en sorte que la fenêtre soit redessinée avec la couleur (rouge = (x / largeur_fenetre, vert = 0, bleu = y / hauteur_fenetre).
(Vous pouvez récupérer la largeur et hauteur de la fenêtre via la structure `SDL_Surface`)

## Exercice 05 – Dessinons !

Nous avons précédemment assigné une taille en pixels à notre fenêtre de rendu, grâce à la fonction `SDL_SetVideoMode(…)` .

Or, pour pouvoir y dessiner une scène, nous devons également indiquer à OpenGL la taille virtuelle qu'aura notre fenêtre à l'intérieur de cette scène.
Ainsi, à chaque fois qu'un point sera dessiné, les coordonnées de ce point seront fournies à OpenGL dans l'espace virtuel de la scène, et OpenGL se chargera de convertir celles-ci en coordonnées pixels (aussi appelées coordonnées écran) dans le repère 2D de la fenêtre.

Cette étape de conversion entre les coordonnées virtuelles et les coordonnées pixels / écran s'appelle la **projection**.

**A faire :**
**01a.** Ajoutez la fonction `reshape()` dans votre code :

```
void reshape(SDL_Surface** surface)
{
    SDL_Surface* surface_temp = SDL_SetVideoMode(
        WINDOW_WIDTH, WINDOW_HEIGHT, BIT_PER_PIXEL,
        SDL_OPENGL | SDL_RESIZABLE);
    if(NULL == surface_temp) {
        fprintf(
            stderr,
            "Erreur lors du redimensionnement de la fenetre.\n");
        return EXIT_FAILURE;
    }
    *surface = surface_temp;

    glViewport(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-1., 1., -1., 1.);
}
```

Le viewport définit la taille en pixels de la fenêtre réelle.
Puis on passe en mode projection, et après avoir chargé la matrice identité, on crée une projection orthogonale qui, en fait, identifie coordonnées virtuelles et coordonnées pixels.

**01b.** Faites le nécessaire pour que la fonction `reshape()` soit appelée à chaque redimensionnement de la fenêtre. (événement `SDL_VIDEORESIZE` )

Attention :
Il faut passer l'adresse du pointeur `surface` en paramètre de `reshape` .

Nous pouvons maintenant dessiner !

Pour spécifier une couleur, on utilise l'instruction `glColor3ub(r, v, b);`
où `r`, `v`, `b` sont trois entiers compris entre 0 et 255 qui définissent les trois composantes rouge, verte et bleue de la couleur désirée.
Une fois cette fonction appelée, tout se qui sera dessiné par la suite aura cette couleur.

Écrire un ou plusieurs points se fait grâce aux instructions :

```
glBegin(GL_POINTS);
    glVertex2f(
        -1 + 2. * x / WINDOW_WIDTH,
        -(-1 + 2. * y / WINDOW_HEIGHT));
    // eventuellement d'autres points …
glEnd();
```

où `x` et `y` sont les coordonnées du pixel cliqué, avec (0,0) en haut à gauche de la fenêtre.

**Questions :**

**02.** Pourquoi ne peut on pas faire directement : `glVertex2i(x, y);` ?
**03.** Pourquoi utilise t-on `-1 + 2. * x / WINDOW_WIDTH` et pas simplement `x` ?
(cette question s'applique également pour `y`)

**A faire :**
A l'aide de ces fonctions, faire une application de dessin :

**04.** Lorsqu'on clique dans la fenêtre, affichez un point rouge à l'endroit cliqué.

Attention :
Il faut désactiver le double buffering avant d'ouvrir la fenêtre
avec `SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 0)` et nettoyer la fenêtre qu'une seule fois avec `glClear(GL_COLOR_BUFFER_BIT)` avant la boucle de rendu.
Autrement, les points ne resteront pas affichés !

## Exercice 06 – Vers un outil de dessin ? (A faire à la maison)

Vous allez maintenant améliorer votre programme de dessin.

L'utilisateur devrait notamment 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` , le viewport 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.
(Une solution plus aboutie en C serait de créer une énumération recensant nos deux états.)

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 ne dessinera plus en direct dans le viewport, mais écrira les points dans une structure de données prévue à cet effet.
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 points :

Un point stockera sa position en coordonnées de clic, et sa couleur.

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

```
typedef struct point_link {
     int x, y;               // Position 2D du point
     unsigned char r, g, b;  // Couleur du point
     struct point_link *next;    // Suivant
} Point, *PointList;
```

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.** `Point* allocPoint(int x, int y,`
`unsigned char r, unsigned char g, unsigned char b);`

Alloue la mémoire pour stocker un point et initialise le point

**03.** `void deletePoints(PointList* list);`

Libère la mémoire allouée pour une liste de points.

**04.** `void addPoint(Point* point, PointList* list);`

Ajoute en fin de liste le point passé en paramètre.
(Il est important d'ajouter en fin de liste pour la suite de l'exercice)

**05.** `void drawPoints(PointList list);`

Dessine tous les points de la liste. (A l'aide de `glVertex2f` et `glColor3ub` )

**Note :**
On stocke les cordonées pixels des points cliqués, il faudra les convertir pour les dessiner.

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

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

```
typedef struct primitive_link {
    GLenum primitiveType;   // Type de primitive
    PointList points;       // 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.** `Primitive* allocPrimitive(GLenum primitiveType);`

Alloue la mémoire pour stocker les points de plusieurs primitives d'un même type. Le champs points doit être fixé à `NULL`.

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

Libère la mémoire allouée à une liste de primitives, et aux listes de points sous-jacentes.

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

Ajoute en tête de liste la primitive passée en paramètre.

**10.** `void drawPrimitives(PrimitiveList list);`

Dessine l'ensemble des primitives. (Utilise la fonction `drawPoints` pour chaque primitive)

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.
**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 ne vous reste plus désormais qu'à implémenter le changement de couleur.
Les exercices 04 et 05 vous ont donnés les éléments nécessaires pour faire cette partie.

**14.** 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.

**15a.** Lorsque le programme est en mode palette, affichez des carrés de couleur.
**15b.** 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 SDL).

Optionnel :
16. 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. (Ex : `void handleEvents_paintState()`, `void draw_paletteState()`, …)

En arrivant au bout de cet exercice, vous avez obtenu un mini-programme de dessin, et ce faisant avez assimilé les premières notions pour écrire un programme graphique pouvant interagir avec l'utilisateur en temps réel.

Félicitations ! Un océan de possibilités s'ouvre devant vous !