






# TP13 – Prise en main de LibGraph2

 LibGraph2 est une bibliothèque C++ qui permet à l'aide d'une interface de développement simplifiée de développer des applications munies d'une interface graphique.

## Exercice 0 Récupération de la bibliothèque

-  Récupérez le fichier **LibGraph2.zip** sur la plateforme pédagogique de l'ENT.
-  Décompressez cette archive dans votre répertoire de travail.
-  Cela crée 6 répertoires de documentation de la bibliothèque (**doc**, **doc-0**, **doc-1**, **doc-2**, **doc-3**, **doc-4**), un répertoire **Demonstration** contenant des programmes de démonstration de la bibliothèque, et deux répertoires **inc** et **lib** contenant les fichiers de développement de la bibliothèque.

 La bibliothèque s'adapte au niveau d'expérience de l'utilisateur selon 5 niveaux :

**NIVEAU 0** : *Novice* : Aucun pointeur, aucun objet, pas de passage par référence, seulement des appels de fonctions → aide en ligne dans le répertoire **doc-0**


**NIVEAU 1** : *Débutant* : Des passages par références, pas de pointeurs, pas d'objet → aide en ligne dans le répertoire **doc-1**

**NIVEAU 2** : *Intermédiaire* : Des passages par références, des pointeurs, pas d'objet → aide en ligne dans le répertoire **doc-2**

**NIVEAU 3** : *Avancé* : Quelques objets → aide en ligne dans le répertoire **doc-3**

**NIVEAU 4** : *Expert* : (niveau par défaut) Tout objet → aide en ligne dans le répertoire **doc-4**

## Exercice 1 Installation et création d'un projet Visual Studio

-  Le point d'entrée principal de l'aide en ligne est le fichier **doc\html\index.html**. Ouvrez ce fichier dans un navigateur web et suivez la section « **Installation** » pour créer un nouveau projet Visual Studio capable d'utiliser la bibliothèque LibGraph2.

## Exercice 2 Philosophie de LibGraph2

### 2.1 Structure d'une application LibGraph2

LibGraph2 permet de développer des applications interactives. Dans le code, l'interaction se construit autour d'une boucle principale qui « attend » les événements provoqués par l'utilisateur. La fonction **waitForEvent(...)** permet d'attendre qu'un événement se produise, puis en fonction de son type (variable de type **evt\_type**), le développeur doit programmer le comportement adéquat pour son application. Une fois l'événement géré, la fonction **waitForEvent(...)** doit être appelée à nouveau pour gérer le prochain événement.

Le tableau suivant récapitule les différents types d'événements. Le seul événement qui doit absolument être géré par le développeur est l'événement **evt\_type::evtRefresh**, car c'est le seul endroit où il est judicieux de faire l'affichage.

Type d'événement (valeur de type <code>evt_type</code> )	Signification	Remarque
<code>evt_type::evtMouseDown</code>	Enfoncement du bouton de la souris.	Pas de différenciation entre les boutons gauche, milieu et droit. Possibilité de récupérer la position de la souris (voir l'aide).
<code>evt_type::evtMouseUp</code>	Relâchement du bouton de la souris.	Pas de différenciation entre les boutons gauche, milieu et droit. Possibilité de récupérer la position de la souris (voir l'aide).
<code>evt_type::evtMouseMove</code>	Déplacement de la souris.	Possibilité de récupérer la position de la souris (voir l'aide).
<code>evt_type::evtKeyDown</code>	Enfoncement d'une touche du clavier.	Possibilité de récupérer la touche enfoncée (voir l'aide).
<code>evt_type::evtKeyUp</code>	Relâchement d'une touche du clavier.	Possibilité de récupérer la touche enfoncée (voir l'aide).
<code>evt_type::evtRefresh</code>	Rafraîchissement de toute la fenêtre.	Toutes les opérations de dessin doivent être effectuées en réponse à cet événement.
<code>evt_type::evtSize</code>	Modification de la taille de la fenêtre.	Possibilité de récupérer la taille de la fenêtre
<code>evt_type::evtClose</code>	Fermeture de la fenêtre	Utile pour éventuellement libérer des ressources.

## 2.2 Principe de l'affichage

L'utilisation de la bibliothèque LibGraph2 pour dessiner est une métaphore du peintre. Au même titre qu'un peintre choisi son pinceau, y dépose de la peinture puis dessine des formes, le dessin sous LibGraph2 nécessite de choisir le type et la couleur de trait (le stylo – *pen* en anglais) ainsi que le type et la couleur de remplissage (le pinceau – *brush* en anglais) avant de dessiner des formes.

Les couleurs (de type **ARGB**) sont interprétées comme étant une combinaison de 4 composantes numériques comprises entre **0** et **255** ; **Alpha**, **Rouge**, **Vert** et **Bleu**. La composante **Alpha** permet de choisir le niveau d'opacité de la couleur : **0** pour totalement transparente et **255** pour totalement opaque. Pour générer les codes numériques des couleurs, le développeur peut utiliser la fonction utilitaire **MakeARGB(...)**. Le code suivant montre le principe du dessin sous LibGraph2.

```
//Choisit un stylo rouge d'épaisseur 5 pixels traçant des pointillés
setPen(MakeARGB(255,255,0,0), 5.0f, LibGraph2::pen_DashStyles::Dash);
//Choisit un pinceau jaune
setSolidBrush(MakeARGB(255,255,255,0));
//Dessine un rectangle avec le stylo et le pinceau sélectionné
drawRectangle(20,40,200,100);
//Choisit un pinceau cyan translucide
setSolidBrush(MakeARGB(128,0,255,255));
//Dessine une ellipse
drawEllipse(100,100,200,200);
```

Code 1 : Exemple de code pour dessiner

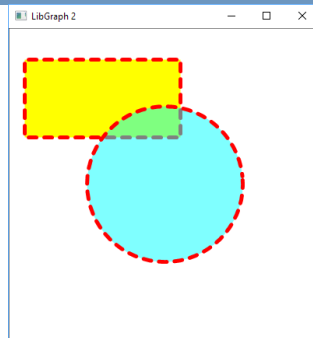


Figure 1 : Résultat de l'exécution du code 1.

### 2.2.1 Optimisation de l'affichage

Il y a deux modes d'affichages pris en charge par LibGraph2 : un affichage direct (les formes s'affichent à l'écran dès l'exécution d'une fonction de dessin) et un affichage optimisé (les formes sont toutes affichées en même temps lorsque l'affichage est prêt). Pour activer l'affichage optimisé, il suffit de faire précéder le bloc des fonctions d'affichage par la fonction **beginPaint()** et de le terminer par la fonction **endPaint()**, comme dans l'exemple de code suivant :

```
//Prépare l'affichage. A partir de ce point, les fonctions d'affichage
//sont exécutées uniquement en mémoire, et non à l'écran
beginPaint();
//Choisit un stylo rouge d'épaisseur 5 pixels traçant des pointillés
setPen(MakeARGB(255,255,0,0), 5.0f, LibGraph2::pen_DashStyles::Dash);
//Choisit un pinceau jaune
setSolidBrush(MakeARGB(255,255,255,0));
//Dessine un rectangle avec le stylo et le pinceau sélectionné
drawRectangle(20,40,200,100);
//Choisit un pinceau cyan translucide
setSolidBrush(MakeARGB(128,0,255,255));
//Dessine une ellipse
drawEllipse(100,100,200,200);
//Affiche réellement à l'écran l'ensemble du dessin
endPaint();
```

Code 2 : Exemple de code optimisé pour dessiner

Il est conseillé de toujours utiliser l'optimisation à l'aide des fonctions **beginPaint()** et **endPaint()**, car cela est plus rapide et évite les effets de scintillements lors de l'affichage d'animations. La désactivation de cette fonctionnalité peut toutefois être utile à des fins de débogage.

### 2.2.2 Où appeler les fonctions de dessin ?

Les fonctions de dessin doivent être appelées uniquement en réponse à l'événement **evt\_type::evtRefresh**. Si vous les appelez ailleurs, elles n'auront aucun effet. Si votre affichage doit évoluer au cours de l'exécution de votre programme en conséquence d'une action de l'utilisateur, alors vous devez utiliser des variables d'état qui seront modifiées dans les fonctions d'écoute de l'utilisateur (à la suite d'un **evt\_type::evtKeyUp** par exemple) et lues lors de la réponse à l'événement **evt\_type::evtRefresh**.

Lorsqu'une variable d'état est modifiée, il faut alors demander à LibGraph2 de rafraîchir l'affichage (c'est-à-dire de générer un événement **evt\_type::evtRefresh**). Cela se fait en appelant la fonction **askForRefresh()**.

## 2.3 Gestion de la fenêtre graphique

🔗 Référez-vous à la section « Modules / Gestion des fenêtres / Gestion des fenêtres graphiques » de l’aide en ligne pour le détail des fonctionnalités

La fonction **show(...)** permet d’afficher la fenêtre alors que la fonction **hide()** permet de la masquer. En début de programme, vous devez évidemment appeler la fonction **show(...)**.

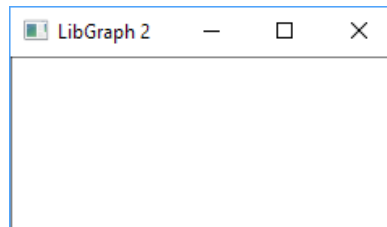


Figure 2 : Fenêtre graphique de LibGraph2

## 2.4 Gestion de la console

🔗 Référez-vous à la section « Modules / Gestion des fenêtres / Gestion de la console » de l’aide en ligne pour le détail des fonctionnalités

Dès que votre programme réalise une écriture vers le flux de sortie standard (typiquement un **cout**), la console s’affiche en bas de la fenêtre graphique. Il est également possible de contrôler l’affichage de la console par le code, comme spécifié dans l’aide en ligne.

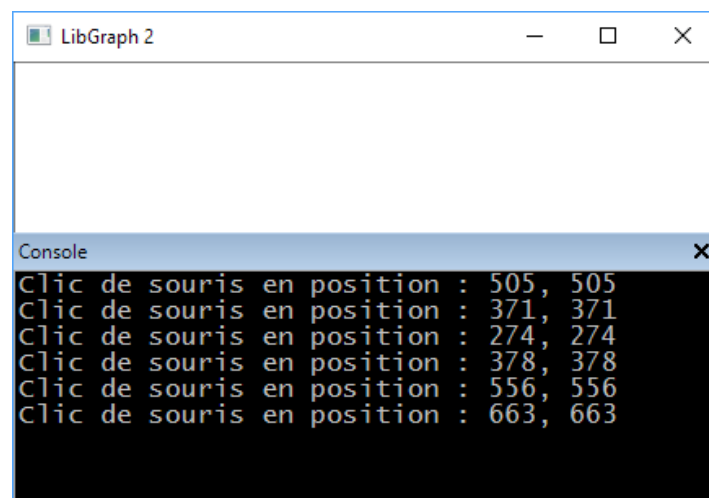


Figure 3 : Console affichée en bas de la fenêtre graphique

## 2.5 Boîte de dialogues

Enfin, il est possible d’interagir avec l’utilisateur à l’aide de boîtes de dialogues. Référez-vous à la section « Modules / Gestion des fenêtres / Boîtes de dialogues » de l’aide en ligne.

## Exercice 3 Pour s'exercer

### 3.1 LibGraph niveau 2 (passages par références, pas de pointeur, pas d'objet)

Créez une application utilisant LibGraph2 au niveau 2 affichant l'image suivante :

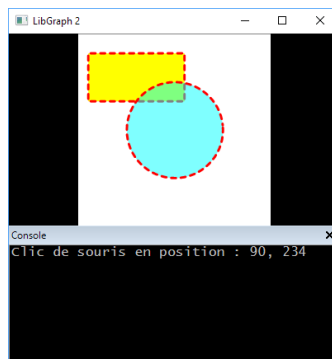


Figure 4 : Capture d'écran de l'application à obtenir

La fenêtre est de taille **400x400**. Sur l'appui de la touche '**C**', la console doit apparaître et disparaître sur l'appui de la touche '**A**'. Sur le clic de souris, la console doit afficher le message « **Clic de souris en position : X, Y** » et le disque doit soit s'afficher soit se masquer en fonction de son état actuel. Lorsqu'il est visible, le disque est affiché translucide sur le rectangle.

### 3.2 LibGraph niveau 4 (tout objet)

Passez LibGraph2 au niveau 4 et recommencez l'exercice précédent.

### 3.3 Animation (facultatif)

Il est possible de réaliser simplement des animations avec LibGraph2. Pour cela, il suffit d'appeler la fonction `askForRefresh()` en réponse à l'événement `evt_type::evtRefresh`. Cela signifie qu'à chaque affichage de la fenêtre, un prochain rafraîchissement est demandé. En mesurant le temps écoulé entre deux rafraîchissements, il est possible de contrôler la vitesse de l'animation.

#### 3.3.1 Mesure du temps

En C++, le temps est géré par la bibliothèque `<chrono>`. La fonction `chrono::high_resolution_clock::now()` permet d'obtenir un point temporel (**timepoint**). La différence entre deux points temporels devient une durée (**duration**). Il est possible de convertir simplement une durée dans l'unité temporelle voulue (nanoseconde, microseconde, milliseconde, seconde, ...). Pour savoir comment utiliser ces objets, référez-vous à l'exemple suivant :

```
//Point temporel de début de mesure
auto tpStart = chrono::high_resolution_clock::now();
//Point temporel de fin de mesure
auto tpStop = chrono::high_resolution_clock::now();
//Durée entre les deux points temporels exprimée en secondes avec une
// précision de float.
chrono::duration<float, chrono::seconds::period> dur = tpStart- tpStop;
//Récupération de la valeur numérique de la durée
float fDuree = dur.count();
```

Code 3 : Exemple d'utilisation de la bibliothèque chrono pour mesurer une durée.

Notre objectif est de faire rebondir le disque comme s'il s'agissait d'une balle. Sa position dans le code 2 est exprimée par le doublet 100, 100. Il s'agit donc de faire varier son ordonnée entre 100 et 200 en fonction du temps écoulé. L'équation de mouvement suit une parabole d'expression  $y = 100(t - 1)^2 + 100$  où  $t$  est le temps exprimé en secondes et compris entre 0s et 2s. La courbe ci-dessous représente cette expression :

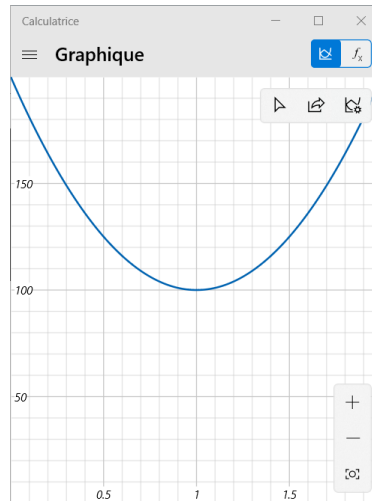



Figure 5 : Représentation graphique de l'équation du mouvement de la balle

Dès que la durée mesurée atteint 2s, le point temporel initial doit alors être réinitialisé au temps actuel.

 Fort-e de vos nouvelles connaissances, implémentez ce comportement.