
L3IUP Informatique— UBO

Projet “Jeux Interactifs” en langage C

Réalisation d’une application graphique

1 Objectifs

Ce projet a pour objet de faire développer une application complète, plus complexe que celles habituellement proposées lors de sujets de TP en temps limité.

En laissant une grande part à l’initiative des étudiants, ceux-ci pourront approfondir leur maîtrise du langage C, mettre en évidence leurs capacités à trouver des solutions originales mais également être confrontés aux problèmes de complexité de développement d’applications de moyenne envergure.

Ce projet pourra commencer par une familiarisation avec la bibliothèque graphique fournie à travers l’étude et la modification d’un exemple fourni.

Ensuite les étudiants pourront choisir une application de leur choix (casse-brique, circuit automobile, tetris, tennis ...) et ils proposeront alors à l’enseignant une étude de la solution envisagée (sur papier, technique utilisée, analyse descendante ...). Une fois les étudiants et l’enseignant d’accord sur la réalisation à entreprendre, celle-ci pourra avoir lieu.

Un rapport présentant l’analyse du problème, les détails de réalisation et les possibilités d’évolution du logiciel fera l’objet d’une évaluation.

2 La bibliothèque graphique

Dans le répertoire `JeuxInteractifs` se trouvent des fichiers et répertoires utiles au développement de votre application. Le fichier d’interface (`JeuxInteractifs/include/tpC_graph.h`) donne accès à une bibliothèque vous permettant de manipuler des `Object2Ds`. Il s’agit d’un type opaque décrivant un objet graphique situé dans un plan (x, y, θ) structuré en couches (avant/arrière-plan).

Voici les fonctions qui sont associées à ce type de donnée :

- ▷ `Object2D *`
`Object2D_new(void);`
 - ◊ Instancier un `Object2D`.
 - ◊ Sa représentation est un point blanc.
 - ◊ Il se trouve sur la couche de visualisation 0.
 - ◊ Sa situation absolue sur le plan est $(x = 0, y = 0, \theta = 0)$
- ▷ `void`
`Object2D_delete(Object2D * obj2d);`
 - ◊ Détruire `obj2d`.

- ▷ void
Object2D_setLocation(Object2D * obj2d,
double x,
double y,
double theta);
◊ Situer obj2d en absolu dans le plan.
- ▷ void
Object2D_getLocation(const Object2D * obj2d,
double * xOut,
double * yOut,
double * thetaOut);
◊ Obtenir la situation absolue d'obj2d dans le plan.
- ▷ void
Object2D_setX(Object2D * obj2d,
double x);
◊ Fixer l'abscisse absolue d'obj2d dans le plan.
- ▷ double
Object2D_getX(const Object2D * obj2d);
◊ Obtenir l'abscisse absolue d'obj2d dans le plan.
- ▷ void
Object2D_setY(Object2D * obj2d,
double y);
◊ Fixer l'ordonnée absolue d'obj2d dans le plan.
- ▷ double
Object2D_getY(const Object2D * obj2d);
◊ Obtenir l'ordonnée absolue d'obj2d dans le plan.
- ▷ void
Object2D_setTheta(Object2D * obj2d,
double theta);
◊ Fixer l'orientation absolue d'obj2d dans le plan.
- ▷ double
Object2D_getTheta(const Object2D * obj2d);
◊ Obtenir l'orientation absolue d'obj2d dans le plan.
- ▷ void
Object2D_translate(Object2D * obj2d,
double dx,
double dy)
◊ Translation relative d'obj2d.
◊ dx et dy sont exprimés dans le repère local d'obj2d.
- ▷ void
Object2D_rotate(Object2D * obj2d,
double dTheta)
◊ Rotation relative d'obj2d.
◊ dTheta est exprimé dans le repère local d'obj2d.

- ▷ `int /* 0: outside !=0: inside */`
`Object2D_isInside(const Object2D * obj2d,`
`double x,`
`double y);`
 - ◇ Non nul si le point (x, y) est situé à l'intérieur de la représentation d'obj2d.
 - ◇ x et y sont exprimés dans le repère global.
- ▷ `int /* 0: no intersection found !=0: intersection found */`
`Object2D_intersectRay(const Object2D * obj2d,`
`double xRay,`
`double yRay,`
`double thetaRay,`
`double * xOut,`
`double * yOut);`
 - ◇ Non nul si la représentation d'obj2d est intersectée par la demi-droite issue de $(xRay, yRay)$ et orientée selon $thetaRay$.
 - ◇ $xOut$ et $yOut$ reçoivent alors le point d'intersection.
 - ◇ $xRay, yRay, thetaRay, xOut$ et $yOut$ sont exprimés dans le repère global.
- ▷ `void`
`Object2D_globalToLocalPosition(const Object2D * obj2d,`
`double * xInOut,`
`double * yInOut);`
 - ◇ Conversion d'une position globale dans un repère local.
 - ◇ $xInOut$ et $yInOut$ sont exprimés dans le repère global en entrée et dans le repère local d'obj2d en sortie.
- ▷ `void`
`Object2D_localToGlobalPosition(const Object2D * obj2d,`
`double * xInOut,`
`double * yInOut);`
 - ◇ Conversion d'une position locale dans le repère global.
 - ◇ $xInOut$ et $yInOut$ sont exprimés dans le repère local d'obj2d en entrée et dans le repère global en sortie.
- ▷ `double`
`Object2D_globalToLocalOrientation(const Object2D * obj2d,`
`double orientation);`
 - ◇ Conversion d'orientation dans le repère local d'obj2d.
 - ◇ `orientation` est exprimé dans le repère global.
- ▷ `double`
`Object2D_localToGlobalOrientation(const Object2D * obj2d,`
`double orientation);`
 - ◇ Conversion d'orientation dans le repère global.
 - ◇ `orientation` est exprimé dans le repère local d'obj2d.

- ▷ void

```
Object2D_setColor(Object2D * obj2d,
                  const char * colorName);
```

 - ◇ Modifier la couleur d'obj2d.
 - ◇ `colorName` peut désigner le nom d'une couleur ("blue", "grey" ...) ou bien le motif "rgb:RR/GG/BB" où RR, GG et BB sont les composantes de la couleur choisie exprimées sur deux chiffres hexadécimaux (00 à FF).
- ▷ const char *

```
Object2D_getColor(Object2D * obj2d);
```

 - ◇ Obtenir la couleur d'obj2d.
- ▷ void

```
Object2D_setLayer(Object2D * obj2d,
                  int layer);
```

 - ◇ Placer obj2d sur une couche de visualisation.
 - ◇ Les valeurs croissantes de `layer` progressent vers l'avant-plan.
- ▷ int

```
Object2D_getLayer(const Object2D * obj2d);
```

 - ◇ Obtenir la couche de visualisation sur laquelle évolue obj2d.
- ▷ void

```
Object2D_point(Object2D * obj2d);
```

 - ◇ obj2d prend la forme d'un point.
- ▷ void

```
Object2D_text(Object2D* obj2d,
              const char * text);
```

 - ◇ obj2d prend la forme d'une chaîne de caractères.
- ▷ void

```
Object2D_line(Object2D * obj2d,
              double length);
```

 - ◇ obj2d prend la forme d'une ligne.
 - ◇ La ligne va de (0,0) à (*length*,0) dans le repère local d'obj2d.
- ▷ void

```
Object2D_square(Object2D * obj2d,
                double side,
                int filled);
```

 - ◇ obj2d prend la forme d'un carré.
 - ◇ Le carré est centré en (0,0) sur le repère local d'obj2d, les cotés ont une longueur `side` et sont orientés selon les axes du repère local d'obj2d.
 - ◇ Si `filled` est non nul, le carré est plein.
- ▷ void

```
Object2D_rectangle(Object2D * obj2d,
                  double length,
                  double width,
                  int filled);
```

 - ◇ Même principe qu'avec `Object2D_square()` mais pour un rectangle.
 - ◇ `length` et `width` donnent respectivement la longueur selon l'axe \vec{x} local et la largeur selon l'axe \vec{y} local.

- ▷ void

```
Object2D_polyline(Object2D * obj2d,
                  unsigned int nbPoints,
                  const double * xPoints,
                  const double * yPoints);
```

 - ◇ Même principe qu'avec `Object2D_line()` mais pour une ligne brisée.
 - ◇ Les coordonnées (x, y) sont exprimées dans le repère local d'`obj2d`.
- ▷ void

```
Object2D_polygon(Object2D * obj2d,
                  unsigned int nbPoints,
                  const double * xPoints,
                  const double * yPoints,
                  int filled);
```

 - ◇ Même principe qu'avec `Object2D_square()` mais pour un polygone.
 - ◇ Les coordonnées (x, y) sont exprimées dans le repère local d'`obj2d`.
 - ◇ Le dernier point et le premier point sont reliés.
- ▷ void

```
Object2D_circle(Object2D * obj2d,
                 double radius,
                 int filled);
```

 - ◇ Même principe qu'avec `Object2D_square()` mais pour un cercle.
- ▷ int /* 0: failure !=0: success */

```
Object2D_image(Object2D * obj2d,
               const char * fileName,
               double pixelScale);
```

 - ◇ Même principe qu'avec `Object2D_square()` mais pour une image.
 - ◇ Le fichier `fileName` doit être au format `bmp` ou `ras` et utiliser une palette (pas d'image 24 bits).
 - ◇ `pixelScale` indique la taille d'un pixel de l'image dans le plan.
 - ◇ Retourne un résultat nul si le fichier n'est pas au format attendu.
- ▷ int /* 0: failure !=0: success */

```
Object2D_getImagePixelAt(Object2D * obj2d,
                         double x,
                         double y,
                         int * redOut,
                         int * greenOut,
                         int * blueOut);
```

 - ◇ Donne la couleur du pixel qui se trouve en (x, y) dans le plan (repère global).
 - ◇ Retourne un résultat nul si `obj2d` n'a pas la forme d'une image ou si (x, y) n'est pas à l'intérieur d'`obj2d`.

Le fichier d'interface donne également accès à des fonctions permettant d'initialiser et d'activer l'application graphique :

- ▷ void
 `graphic_init(const char * windowName,
 const char * fontName);`
 - ◇ Créer la fenêtre graphique en lui affectant le titre `windowName`.
 - ◇ Les `Object2Ds` ayant une représentation sous forme de texte utiliseront la police `fontName`.
 - ◇ Cette fonction doit être appelée avant toutes les autres fonctions de la bibliothèque.
- ▷ void
 `graphic_setWidth(int width);`
 - ◇ Fixer la largeur en pixels de la fenêtre graphique.
- ▷ void
 `graphic_setHeight(int height);`
 - ◇ Fixer la hauteur en pixels de la fenêtre graphique.
- ▷ void
 `graphic_setBackground(const char * colorName);`
 - ◇ Changer la couleur du fond de la fenêtre graphique.
- ▷ void
 `graphic_setViewPoint(double x,
 double y,
 double scale);`
 - ◇ Modifier le point de vue de la fenêtre
 - ◇ Le point (x, y) représente le point du plan qui sera situé au centre de la fenêtre.
 - ◇ `scale` représente le facteur qui permet de passer des grandeurs sans dimension du plan aux *pixels* de l'écran.
- ▷ void
 `graphic_getViewPoint(double * xOut,
 double * yOut,
 double * scaleOut);`
 - ◇ Obtenir le point de vue courant de la fenêtre.
 - ◇ Voir `graphic_setViewPoint()` .
- ▷ void
 `graphic_autoscale(void);`
 - ◇ Recadrer la fenêtre pour qu'elle montre l'ensemble des `Object2Ds`.
- ▷ void
 `graphic_run(void * userData);`
 - ◇ Lancer la partie active (événementielle) du programme.
 - ◇ `userData` désigne généralement une structure créée par vos soins, permettant d'accéder à l'ensemble des données de l'application.

- ▷ void
`graphic_mainLoop(void * userData);`
 - ◇ Cette fonction est appelée perpétuellement à partir de `graphic_run()` ; c'est la cinématique de l'application.
 - ◇ **Vous devez définir cette fonction !**
 - ◇ Le paramètre `userData` est celui qui a été transmis à `graphic_run()`.
- ▷ void
`graphic_keyPressCallback(Object2D * obj2d,
 const char * key,
 void * userData);`
 - ◇ Cette fonction est appelée lorsqu'une touche du clavier est enfoncée.
 - ◇ **Vous devez définir cette fonction !**
 - ◇ Si `obj2d` est non nul, il s'agit d'un objet graphique qui était sélectionné lors de l'appui sur la touche.
 - ◇ Si `obj2d` est nul, aucun objet graphique n'était sélectionné au moment de l'appui sur la touche.
 - ◇ La touche enfoncée est décrite de manière lisible par `key`.
 - ◇ Le paramètre `userData` est celui qui a été transmis à `graphic_run()`.
- ▷ void
`graphic_mouseDragCallback(Object2D * obj2d,
 double dx,
 double dy,
 void * userData);`
 - ◇ Cette fonction est appelée lorsqu'un mouvement de souris est appliqué à un `Object2D`.
 - ◇ **Vous devez définir cette fonction !**
 - ◇ `obj2d` désigne l'objet sélectionné lors du mouvement de souris.
 - ◇ Le mouvement est décrit par le vecteur (dx, dy) dans le repère global.
 - ◇ Le paramètre `userData` est celui qui a été transmis à `graphic_run()`.

Tous les mouvements et les dimensions des `Object2D` sont exprimés dans une grandeur sans dimension ; ce ne sont pas des pixels. L'axe \vec{x} croît de gauche à droite et l'axe \vec{y} de bas en haut. Le point de vue de la fenêtre peut être modifié selon des translations (`Ctrl+Click Gauche`) et selon un facteur de grossissement (`Ctrl+Click Droit`).

3 L'exemple fourni

Le fichier `graphTest.c` est un programme qui utilise de nombreuses fonctionnalités de la bibliothèque graphique. On y trouve traité un certain nombre de points qui pourront vous guider dans votre réalisation :

- ▷ Squelette général de l'application.
- ▷ Fonctions utilitaires (temps, aléatoire ...).
- ▷ Utilisation des formes, des couleurs ...
- ▷ Changement de repère, suivi de cible.
- ▷ Cadencement de l'application (mesure du temps).
- ▷ Lancers de rayons.
- ▷ Interactions clavier/souris.
- ▷ Lecture des couleurs d'une image.
- ▷ ...

Le `makefile` qui l'accompagne vous sera également utile pour vos développements. Vous pourrez constater sur certaines machines peu puissantes que l'utilisation d'images est beaucoup plus pénalisante que l'utilisation d'autres représentations. Évitez donc autant que possible d'y avoir recours et limitez vous à des images de petite taille (peu de pixels).

4 Travail demandé

Après avoir pris en main et manipulé l'exemple fourni, choisissez le programme graphique que vous souhaitez réaliser. Procédez à une analyse descendante des fonctionnalités en identifiant pour chaque traitement les entrées/sorties et les effets de bord. Après concertation avec l'enseignant, la réalisation pourra débuter. Elle donnera lieu à plusieurs versions successives dans des répertoires distincts (`V1`, `V2` ...). Vous devrez rendre un rapport relatant l'analyse et les détails de réalisation du projet.