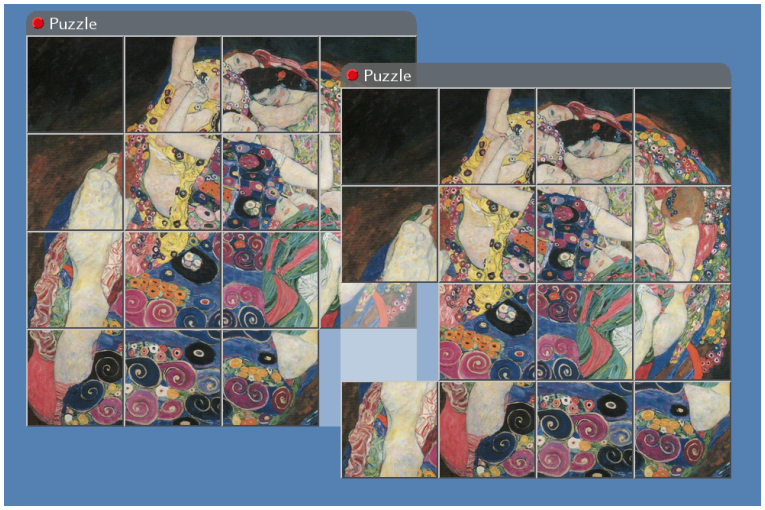
Ensimag — Printemps 2016

Projet Logiciel en C

Sujet : Interfaces Utilisateur Graphiques



Version : 2.2 (23 mai 2016)
Auteurs : F. Bérard, P. Reignier, JS. Franco, E. Frichot, N. Gesbert,
D. van Amstel, C. Ramisch, A. Shahwan.

Table des matières

1 Introduction **5**
- 1.1 Objectifs . . . 5
  - 1.1.1 Projet C . . . 5
  - 1.1.2 Bibliothèque de programmation des interfaces utilisateur graphiques . . . 5
- 1.2 Survol du projet . . . 6
  - 1.2.1 Bibliothèque . . . 6
  - 1.2.2 Applications . . . 6

2 Principes de la bibliothèque **7**
- 2.1 Programmation événementielle . . . 7
  - 2.1.1 Principe . . . 7
  - 2.1.2 Structure d'un programme événementiel . . . 7
- 2.2 Survol des modules et principales interactions . . . 8
  - 2.2.1 Interface système et matériel . . . 8
  - 2.2.2 Interacteurs . . . 9
  - 2.2.3 Gestionnaire d'événements . . . 11
  - 2.2.4 Gestionnaire de géométrie . . . 12
  - 2.2.5 Gestion de l'application . . . 13
  - 2.2.6 Programme principal . . . 13

3 Approfondissements **15**
- 3.1 Services de dessin . . . 15
  - 3.1.1 Niveau pixel . . . 15
  - 3.1.2 Niveau primitives graphiques . . . 17
- 3.2 Valeurs par défaut . . . 17
- 3.3 Polymorphisme . . . 17
  - 3.3.1 Polymorphisme des données . . . 18
  - 3.3.2 Polymorphisme des fonctions . . . 19
- 3.4 Classes et hiérarchie de widgets . . . 20
  - 3.4.1 Classes de widgets . . . 20
  - 3.4.2 Description des classes de widget demandées . . . 20
  - 3.4.3 Hiérarchie de widgets . . . 21
- 3.5 Gestion de la géométrie : le placeur . . . 22
  - 3.5.1 Mécanismes communs de gestion de géométrie . . . 22
  - 3.5.2 Algorithme du *placeur* . . . 22
- 3.6 Gestion des événements . . . 23
  - 3.6.1 Principes . . . 23
  - 3.6.2 Widget concerné par l'événement . . . 24
  - 3.6.3 Traitants externes et internes . . . 24
  - 3.6.4 Exemple du déplacement . . . 25
  - 3.6.5 Exemple du redimensionnement . . . 26
- 3.7 Gestion de l'affichage . . . 26
- 3.8 Programme principal et boucle principale . . . 27
  - 3.8.1 Initialisation de l'application . . . 27
  - 3.8.2 Boucle principale . . . 27

4 Travail à réaliser **29**
4.1 Code d'applications fournies . . . 29
4.1.1 Minimal . . . 29
4.1.2 Cadre (frame) . . . 29
4.1.3 Bouton simple (button) . . . 30
4.1.4 Fenêtre hello world . . . 30
4.1.5 Démineur (minesweeper) . . . 30
4.1.6 Puzzle . . . 31
4.2 Extensions . . . 31
4.2.1 Two048 (*) . . . 31
4.2.2 Widget bouton radio (**) . . . 31
4.2.3 Description de la hiérarchie dans un fichier externe (***) . . . 32
4.2.4 Gestion des tags des widgets (**) . . . 34
4.2.5 Widget champ de saisie (***) . . . 34
4.2.6 Gestionnaire de géométrie en grille (***) . . . 35
4.3 Évaluation . . . 35
4.3.1 Critères d'évaluation . . . 35
4.3.2 Rendu des fichiers de votre projet . . . 35
4.3.3 Soutenance . . . 36

5 Consignes et conseils **37**
5.1 Organisation du libre-service encadré . . . 37
5.2 Documentation "Doxygen" . . . 37
5.3 Cas de fraudes . . . 38
5.4 Styles de codage . . . 38
5.5 Outils . . . 39
5.6 Évaluation de performances . . . 39

A Étapes de progression **41**
A.1 Affichage de la fenêtre racine (root) . . . 41
A.2 Création de la classe de widget "frame" . . . 41
A.3 Mise en place d'un gestionnaire de géométrie . . . 42
A.4 Bilan . . . 42
A.5 Dessin de boutons en relief . . . 42
A.6 Mise en place d'un gestionnaire d'événements dans l'application button.c . . . 43
A.7 Généralité . . . 43

Index **43**

Chapitre 1

Introduction

Ce chapitre présente les objectifs du projet, puis un rapide survol des grandes familles d'algorithmes que vous aurez à programmer.

1.1 Objectifs

1.1.1 Projet C

Tout informaticien doit connaître le langage C. C'est une espèce d'espéranto de l'informatique. Les autres langages fournissent en effet souvent une interface avec le langage C (ce qui leur permet en particulier de s'interfacer plus facilement avec le système d'exploitation) ou sont eux-mêmes écrits en C. D'autre part c'est le langage de base pour programmer les couches basses des systèmes informatiques. Par exemple, on écrit rarement un pilote de périphérique en Ada ou Java. Le langage C est un excellent langage pour les programmes dont les performances sont critiques, en permettant des optimisations fines, à la main, des structures de données ou des algorithmes. Par exemple, les systèmes de gestions de base de données et d'une manière générale les logiciels serveurs sont majoritairement écrits en C. La programmation graphique interactive, c'est à dire nécessitant le calcul immédiat de nouvelles images en fonctions des actions de l'utilisateur, nécessite à la fois performance et accès au matériel (cartes graphiques et dispositifs d'interaction), c'est donc un domaine où la connaissance du C est indispensable.

En outre, le projet C a pour objectif de vous confronter au premier projet logiciel un peu conséquent, que vous devez développer dans les règles de l'art : mise en œuvre de tests, documentation, démonstration du logiciel, partage du travail, etc.

1.1.2 Bibliothèque de programmation des interfaces utilisateur graphiques

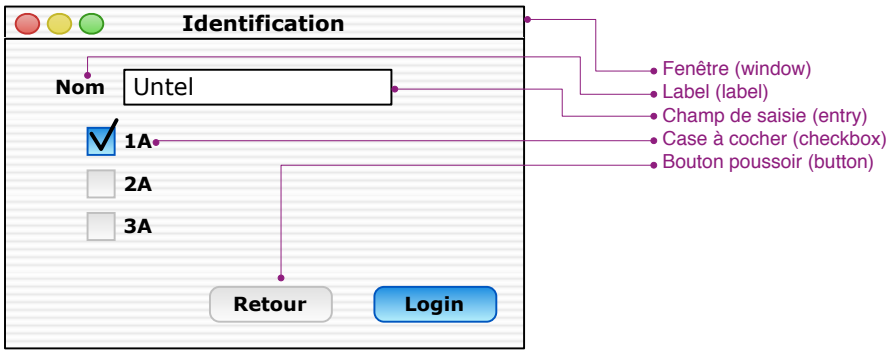


FIGURE 1.1 – Une fenêtre d'une interface utilisateur graphique.

L'objectif de ce sujet est de réaliser une bibliothèque logicielle qui facilite la programmation des Interfaces Utilisateur Graphiques (IUG). En utilisant cette bibliothèque, un programmeur pourra facilement créer

une interface graphique composée de fenêtres et d'interacteurs tels que boutons, champs de saisie, etc [1]. Un exemple d'interface graphique est donné sur la figure 1.1. Vous allez donc réaliser une *bibliothèque logicielle* (en gros, un ensemble de fonctions C) destinée à des programmeurs, et non une *application* destinée à des utilisateurs.

Il vous est donné des fonctions pour :
— l'accès aux pixels de l'écran,
— le dessin de texte,
— le dessin de primitives graphiques (dessin de lignes, de polygones),
— la réception des actions de l'utilisateur sur le clavier et la souris (événements d'appuis de touche, de déplacement de souris, etc.).

Vous devez réaliser les algorithmes :
— de configuration et de dessin des interacteurs (boutons, fenêtre, etc.),
— de gestion de la géométrie (position, taille) des interacteurs à l'écran, en particulier lors du changement de taille de la fenêtre,
— de gestion des événements des utilisateur (exécution des fonctions en réaction aux actions de l'utilisateur sur la souris et le clavier).

Le but du projet étant d'écrire du langage C, le principe des algorithmes ci-dessus vous est donné dans ce document. Votre rôle est de *programmer* ces algorithmes. Afin de vous simplifier le problème de *conception* de la bibliothèque, nous vous fournissons les fichiers d'en-têtes (.h) qui contiennent les spécifications des fonctions C correspondant à ces algorithmes, c'est à vous de programmer ces fonctions.

1.2 Survol du projet

1.2.1 Bibliothèque

La bibliothèque logicielle que vous devez réaliser offrira de nombreux services aux programmeurs qui l'utiliseront. Ces services sont les suivants :
— **Création et configuration des interacteurs**. Différentes fonctions permettent de configurer les différents types d'interacteurs (fenêtre, boutons, labels, etc.), c'est-à-dire définir les paramètres de l'interacteur créé. Par exemple, un bouton a pour paramètre, entre autres, le texte ou l'image qu'il contient, et s'il est affiché en relief ou non.
— **Gestion de la géométrie**. Le gestionnaire de géométrie définit la taille et la position d'un interacteur dans la fenêtre. Un bouton peut nécessiter une certaine largeur afin d'afficher son texte en entier.
— **Gestion des événements**. Votre bibliothèque doit traiter certains événements utilisateurs standard, comme par exemple l'appui du bouton de la souris lorsque le pointeur est sur la barre de titre d'une fenêtre, puis le déplacement de la souris pour déplacer la fenêtre. De plus, le programmeur d'interfaces graphiques doit pouvoir enregistrer des fonctions à appeler lors d'événements d'intérêt. Par exemple, le programmeur enregistre sa fonction "sauvegarde_document" auprès du bouton "Save". Votre bibliothèque se charge d'appeler cette fonction lorsque l'utilisateur clique sur le bouton "Save".

1.2.2 Applications

Afin de tester votre bibliothèque, nous vous fournissons le code source de différents programmes (affichage de bouton, de fenêtre, jeu de puzzle). Ces programmes sont présentés dans la section 4.1.

Au début du projet, ces programmes ne peuvent pas compiler : ils ont besoin de votre implémentation de la bibliothèque. Ces programmes vous permettent de tester si votre bibliothèque fonctionne selon les spécifications données. Bien sûr, il serait très imprudent d'attendre la fin du projet pour tester vos développements. Nous vous conseillons de développer vos propres petits programmes de test au fur et à mesure de vos développements. Nous vous donnons également en annexe A des indications pour mener les premières étapes de votre projet.

1. Le nom anglais "widget" (pour WIndow gaDGET) est souvent utilisé à la place d'interacteur.

Chapitre 2

Principes de la bibliothèque

Dans cette partie, nous donnons les éléments permettant de comprendre les principes essentiels d'une bibliothèque de programmation d'interfaces graphiques. Une vue synthétique de l'architecture de la bibliothèque à implémenter est fournie (figure 2.1) et les principaux modules décrits. Des explications plus détaillées pour chaque module seront présentées au chapitre suivant.

2.1 Programmation événementielle

2.1.1 Principe

Les applications interactives reposent sur le paradigme de *programmation événementielle*. Ce paradigme est différent de la programmation séquentielle que vous avez étudiée jusqu'ici, où l'exécution est linéaire avec des branchements prévus dans le programme. La programmation événementielle est, par opposition, fondée sur la réponse à des événements dont l'ordre d'apparition n'est pas connu au moment d'écrire le programme. Concernant les interacteurs, ces événements peuvent être l'appui d'une touche par l'utilisateur ou le déplacement de la souris. Afin de répondre à de tels événements, le programmeur enregistre, avant l'apparition des événements, des fonctions de traitement d'événements appelées *traitants* (*handler* ou *callback* en anglais).

Après une phase d'initialisation, le programme se met en attente jusqu'à recevoir un nouvel événement. Lorsque celui-ci intervient (par exemple l'appui sur une touche du clavier), la bibliothèque est chargée de communiquer cet événement au traitant à qui il est destiné.

Certains événements sont *situés*, d'autre non. Un évènement situé, par exemple l'appui sur le bouton de la souris, est en général destiné à l'interacteur situé à l'emplacement de la souris. La bibliothèque est chargée d'identifier l'interacteur placé sous le pointeur de la souris afin de communiquer les événements situés au traitant de cet interacteur. Pour un évènement non situé, par exemple l'appui d'une touche au clavier, la bibliothèque ne peut pas identifier à priori à quel interacteur est destiné l'événement. Elle doit donc fournir le moyen d'enregistrer des *traitant généraux* qui sont chargés de traiter les événements non situés.

La prise en compte d'un événement, qu'il soit situé ou non, nécessite en général une mise à jour de l'écran. La bibliothèque se charge de cette mise à jour à la suite du traitement des événements.

2.1.2 Structure d'un programme événementiel

Un programme de type événementiel est toujours articulé autour d'une *boucle principale* à quatre phases : attente d'un nouvel événements, recherche de l'interacteur concerné (s'il existe), appel du traitant enregistré, et mise à jour de l'affichage. Cette boucle est exécutée indéfiniment, jusqu'à ce qu'une requête de terminaison de l'application ait lieu.

Cette boucle principale est précédée par une phase d'initialisation, où l'utilisateur de votre bibliothèque (le programmeur de l'application) crée les interacteurs dont il a besoin, initialise leurs attributs (couleur, dimensions, traitants de l'application, etc.), et les place à l'écran grâce à un gestionnaire de géométrie (décrit plus loin), et enregistre les fonctions de traitement d'événement utiles à son application. Le déroulement d'un programme événementiel peut donc être résumé par le pseudo-code suivant :

créer les interacteurs en définissant leurs attributs
enregistrer les traitants d'événements utilisateur
placer à l'écran chaque interacteur via un gestionnaire de géométrie
tant que pas de demande d'arrêt du programme **faire**
    dessiner à l'écran les mises à jour nécessaires
    attendre un événement
    analyser l'événement pour trouver le traitant associé
    appeler le traitant associé
fin tant que
finaliser le programme : libérer toutes les ressources utilisées.

Nous détaillons ci-dessous les modules qui sont mis en œuvre dans un programme événementiel.

2.2 Survol des modules et principales interactions

La réalisation des tâches décrites ci-dessus est confiée à différents modules de la bibliothèque illustrés avec leurs principales interactions en figure 2.1. Ces modules font l'objet d'une description dans les paragraphes suivants.

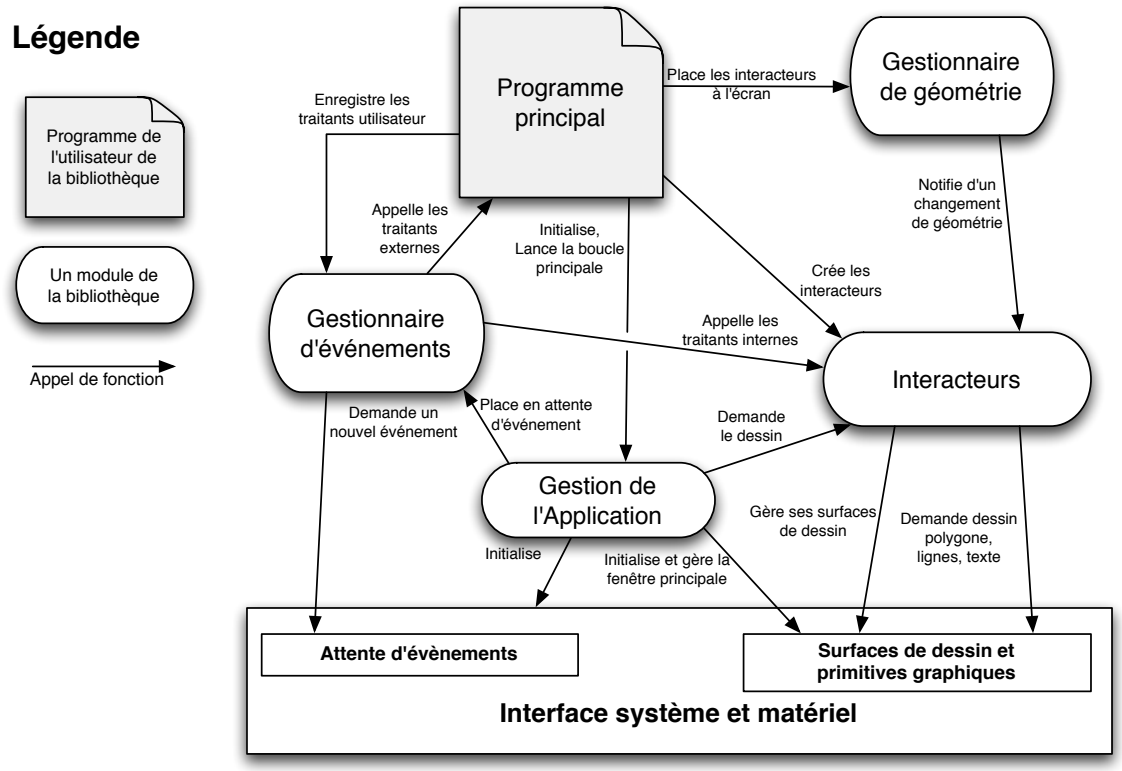


FIGURE 2.1 – Principaux modules et leurs interactions entre eux, avec l'interface matérielle, et avec le programme de l'utilisateur.

2.2.1 Interface système et matériel

Dans ce projet, vous accéderez à l'écran graphique et aux événements au travers d'un module d'interface matériel, représenté par le cadre du bas dans la figure 2.1. Pour les besoins du projet, l'ensemble de l'application et des interacteurs sera géré dans une unique fenêtre du système d'exploitation (UNIX, Mac OS, MS Windows). C'est à l'intérieur de cette fenêtre *système* que votre bibliothèque pourra dessiner ses propres fenêtres, elles-mêmes contenant les interacteurs, comme illustré sur la figure 4.3.

L'interface système / matériel fournit les fonctionnalités suivantes :

— initialisation de la fenêtre graphique système,
— gestion des surfaces de dessin, c'est-à-dire des zones mémoires où l'application "dessine" les interacteurs, pour les copier finalement sur la fenêtre système,
— dessins élémentaires (lignes, polygones, textes),
— attente des événements utilisateur (appui d'une touche, clic souris, etc.),
— mesure du temps.

L'ensemble de ces fonctionnalités vous est fourni dans le cadre de ce projet sous la forme d'une bibliothèque ("libeibase.a") et ne fait donc pas partie de ce que vous devez implémenter.

2.2.2 Interacteurs

Les interacteurs (ou "widgets") sont les objets graphiques interactifs au cœur de l'application. Ils permettent d'exposer graphiquement l'état du programme et les actions que l'utilisateur peut faire sur le programme. Chaque interacteur occupe un espace de l'écran, généralement rectangulaire. Les interacteurs sont de différentes natures : champs de texte, boutons, barres de défilement, conteneurs tels que les fenêtres qui définissent un espace dédié pour contenir d'autres interacteurs, etc.

Hiérarchie

Les interacteurs ont la propriété fondamentale d'être organisés hiérarchiquement en arbre. Un interacteur possède toujours un et un seul interacteur parent et peut avoir des interacteurs descendants. Par convention, chaque interacteur ne peut être dessiné qu'à l'intérieur des limites de son parent. C'est pour ça, par exemple, qu'en réduisant la taille d'une fenêtre, les interacteurs qu'elle contient (ses descendants) ne sont pas dessinés en dehors des limites de la fenêtre : ils sont *tronqués* ("clipped") sur ses limites. Au sommet de la hiérarchie, on considère un interacteur racine ("root") dont la fonction est d'inclure tous les autres, d'offrir un point d'accès unique à la hiérarchie, et de recevoir les événements par défaut lorsqu'ils ne sont associés à aucun autre interacteur. Cette racine est créée à l'initialisation de l'application. L'espace qu'elle occupe correspond à l'ensemble de la *fenêtre système* de l'application et sera en pratique dessiné avec un rectangle vide dont la couleur est paramétrable. Un exemple d'interface est illustré à gauche de la figure 2.3. La hiérarchie d'interacteurs correspondante est représentée sur la figure 2.2.

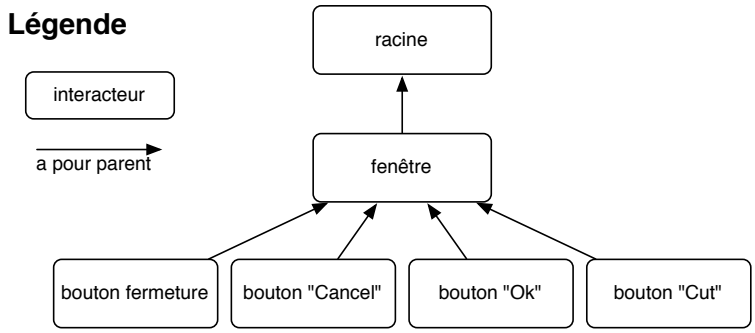


FIGURE 2.2 – Représentation de la hiérarchie de widgets de l'interface représentée à gauche sur la figure 2.3.

Les descendants d'un interacteur ont également la propriété d'être *ordonnés* et cet ordre va déterminer leur visibilité à l'écran : si deux descendants se chevauchent, celui qui sera dessiné en dernier "écrasera" l'autre sur la zone de chevauchement et apparaîtra donc *devant* lui sur l'écran. Cet ordre influence également la distribution des événements : si l'utilisateur clique dans la zone de chevauchement des deux interacteurs, l'événement doit être pris en compte par l'interacteur qui est "au dessus" de l'autre, donc celui qui est le plus proche de la fin dans la liste des descendants.

Description de la hiérarchie dans un fichier externe

Le code C nécessaire pour la mise en place de la hiérarchie de widgets d'une application est généralement un code fastidieux à écrire, peu lisible et ne faisant pas facilement ressortir cette hiérarchie à sa relecture. Les bibliothèques de widget offrent souvent la possibilité de décrire la hiérarchie de widget dans un fichier externe et dans un format simplifié. Ce fichier externe est lu lors du lancement de l'application ;

il est interprété afin de générer la hiérarchie de widgets correspondante. Externaliser la description de la hiérarchie de widgets a d'autres avantages, tel que pouvoir modifier facilement l'aspect de l'interface finale sans avoir besoin de recompiler l'application. Cette approche a aussi des limites, en particulier elle n'est pas adaptée à la génération "à la volée" d'une hiérarchie dynamique (telle qu'une liste de boutons correspondant à une liste de fichiers).

L'interprétation d'un fichier de description de hiérarchie de widget *n'est pas obligatoire* dans ce projet, mais c'est une des extensions proposées à la section 4.2.3.

Dessin des interacteurs, clipping

Une interface graphique doit pouvoir dessiner les interacteurs à l'écran et en modifier l'apparence en fonction des actions de l'utilisateur (l'appui sur un bouton doit lui donner l'aspect "enfoncé"). En informatique, l'unité de dessin est le *pixel* (contraction de "PICture ELement" en anglais). L'image affichée à l'écran est un tableau bidimensionnel de pixels carrés (un écran est par exemple constitué de 1400×900 pixels). Le programmeur définit la teinte et la luminosité de chaque pixel pour former l'image. Il serait bien sûr bien trop fastidieux de décrire l'ensemble des pixels à allumer et leur couleur à chaque fois qu'il est nécessaire de modifier l'aspect de l'interface graphique. Les bibliothèques fournissent donc des fonctions qui factorisent les tâches fréquentes dans le dessin : dessin de lignes, remplissage de polygones, dessin de lettres formant un texte, etc. À partir de ces trois primitives, il est relativement aisé de programmer le dessin d'une fenêtre, d'un bouton, ou de tout autre forme graphique.

Les primitives graphiques sont utilisées à chaque fois qu'il faut redessiner une partie de l'écran, c'est à dire très souvent. Par exemple, quand on déplace une fenêtre à l'écran, la bibliothèque doit effacer la fenêtre à son ancien emplacement (i.e. redessiner ce qui était "en dessous"), et la redessiner au nouvel emplacement, et ce à chaque micro-déplacement de la souris. Ce processus a lieu en général 60 fois par seconde pendant le déplacement de la souris. De plus, le dessin d'un polygone, même de taille modeste, peut nécessiter la modification de dizaines de milliers de pixels. Il en résulte que les algorithmes des primitives graphiques doivent être *extrêmement optimisées*.

On peut séparer les tâches de dessin en trois niveaux d'abstraction (du plus bas au plus haut) :

1. Niveau pixel : représentation d'un pixel en mémoire, allocation mémoire représentant une image, lecture et écriture de pixels ou de blocs de pixels.
2. Niveau primitive graphique : dessin de formes simples (lignes, rectangles, polygones) et de texte.
3. Niveau interacteur : dessin des interacteurs (boutons, fenêtres, barre de défilement, zone de saisie de texte, etc.).

Dans ce projet, les deux premiers niveaux (pixel et primitives graphiques) vous sont fournis. C'est à vous de réaliser le troisième niveau dans la fonction de dessin de vos interacteurs. Pour dessiner un interacteur, on dessine les *primitives graphiques* propres à l'interacteur (rectangles, polygones, lignes, textes, etc.), puis on dessine les descendants de l'interacteur. En dessinant les descendants *après* leur parent, on donne l'impression qu'ils sont devant lui (par exemple, les boutons de la figure 2.3, à gauche) puisque leur dessin vient écraser celui de l'ascendant.

Clipping

Par convention, chaque descendant ne peut être dessiné qu'à l'intérieur des limites de son parent (exemple du bouton "Cut" sur la figure 2.3, à gauche). Pour faciliter la programmation du dessin des interacteurs dans les limites de leur ascendant, toutes les fonctions de dessin des primitives graphiques (rectangles, etc.) acceptent en paramètre un *rectangle de clipping* : la primitive graphique est dessinée uniquement dans les limites de ce rectangle. Dans la figure 2.3 (à gauche), la fonction de dessin de la fenêtre `toplevel` définit un rectangle de clipping correspondant au rectangle blanc (le contenu de la fenêtre). Ce rectangle est passé en paramètre des fonctions de dessin de tous les descendants que contient la fenêtre (les 3 boutons), afin que ceux-ci ne se dessinent qu'à l'intérieur de cette zone blanche. Le bouton de fermeture de la fenêtre (carré rouge) est un cas particulier. Il est dessiné avec un rectangle de clipping qui englobe toute la fenêtre, y compris ses "décorations" (bordure et barre d'en-tête).

Classes d'interacteur

Différents types d'interacteurs sont nécessaires pour créer une interface. Les premiers interacteurs manipulés dans le cadre du projet sont des boutons, des labels (champs de texte non éditable) et des fenêtres

(ou “toplevel”). Ces différents types d’interacteurs partagent un certain nombre de caractéristiques et fonctionnalités communes à tout interacteur. Chaque interacteur a notamment besoin d’un espace mémoire pour mémoriser son état, de fonctions permettant d’allouer et de libérer cet espace mémoire, d’une fonction de configuration permettant de modifier l’état de l’interacteur, et d’une fonction de dessin qui se charge de dessiner l’interacteur à l’écran.

Cependant, la réalisation de ces différentes fonctions dépend du type d’interacteur. Par exemple, la fonction de dessin d’un interacteur de type bouton tracera une zone rectangulaire et un texte ou une image, tandis que la fonction de dessin d’un interacteur de type “toplevel” dessinera une barre d’en-tête, un cadre, et elle appellera la fonction de dessin des descendants de la toplevel. Les attributs, la taille mémoire nécessaire pour les stocker et la manière de les initialiser ou de les détruire dépendent également du type d’interacteur. Un bouton aura notamment des attributs d’état (bouton appuyé ou relâché, texte du bouton) qui lui sont propres.

C’est pourquoi les bibliothèques de programmation d’interfaces graphiques organisent généralement les interacteurs en *classes*. La classe d’un interacteur permet de définir les fonctionnalités communes à un type d’interacteur donné, et se présente sous la forme d’une structure de données stockant les attributs de tout interacteur de la classe. Cette structure contient en particulier une table de pointeurs vers les fonctions spécifiques de la classe. C’est cette table de fonctions qui permettra de spécifier la réalisation concrète des fonctions de dessin, d’initialisation, et de destruction. Le détail de l’implémentation des classes et des tables de fonction est donné au chapitre suivant en section 3.3.

2.2.3 Gestionnaire d’événements

En programmation événementielle, le comportement du programme est réalisé par les fonctions qui traitent les événements générés par les actions de l’utilisateur. On appelle ces fonctions les “traitants” (voir 2.1.1).

On distingue deux catégories de traitant :

— les “traitants internes” à la bibliothèque. Ils sont responsables du *comportement standard des interacteurs* et sont fournis par la bibliothèque.

— Les “traitants externes”. Ils sont responsables du *comportement de l’application* et sont fournis par le programmeur qui utilise la bibliothèque.

Par exemple, lorsque l’utilisateur clique sur un bouton graphique “Nouveau”, il serait fastidieux pour le programmeur d’application de changer lui-même l’affichage du bouton pour qu’il apparaisse “enfoncé”. C’est la bibliothèque qui réalise ce *comportement standard* dans un *traitant interne*. Par contre, lorsque le clic sur le bouton “Nouveau” est terminé, la bibliothèque n’a aucun moyen de savoir ce que doit faire l’application en réaction à cette action de l’utilisateur. C’est au programmeur de l’application de *définir* une fonction qui crée, par exemple, un nouveau contact. Et c’est au programmeur d’*enregistrer* cette fonction en tant que *traitant externe* lié au bouton. Finalement, ce sera à la bibliothèque d’*appeler* ce traitant externe lorsqu’elle détectera que le bouton a été cliqué.

Toutes les classes d’interacteurs n’ont pas forcément de traitant externe. Un interacteur de type “toplevel”, par exemple, n’offre pas au programmeur d’enregistrer de traitant externe car les programmes n’ont pas, en général, de traitement spécifique à exécuter lorsque l’utilisateur manipule une toplevel.

C’est le module *gestionnaire d’événement* qui gère les structures de données représentant les liens entre événement utilisateur et traitant. Pour plus de finesse dans le contrôle des événements, ce module donne la possibilité de conditionner l’appel d’un traitant à d’autres paramètres. Par exemple, le programmeur peut lier son traitant à l’événement “le bouton de la souris a été enfoncé” en demandant au gestionnaire d’événement d’appeler le traitant uniquement si la souris était au dessus d’un bouton particulier au moment de l’événement.

Aiguillage d’événement par picking

Dans le cas des *événement situés* (voir 2.1.1), le gestionnaire d’événement doit être capable d’identifier quel interacteur est positionné à n’importe quelle position (x, y) de l’écran. Il se peut également qu’il n’y ait pas d’interacteur à cet endroit là, hormis la “racine” définie à la section 2.2.2.

Lorsque l’utilisateur appuie sur le bouton de la souris, ou simplement lorsqu’il déplace la souris, le système d’exploitation envoie un événement à l’application qui précise la position du pointeur. Le gestionnaire d’événement doit choisir l’interacteur concerné. On appelle cette fonction le “picking”. Pour réaliser simplement le picking dans des conditions arbitraires de dessin des interacteurs, on utilise une technique

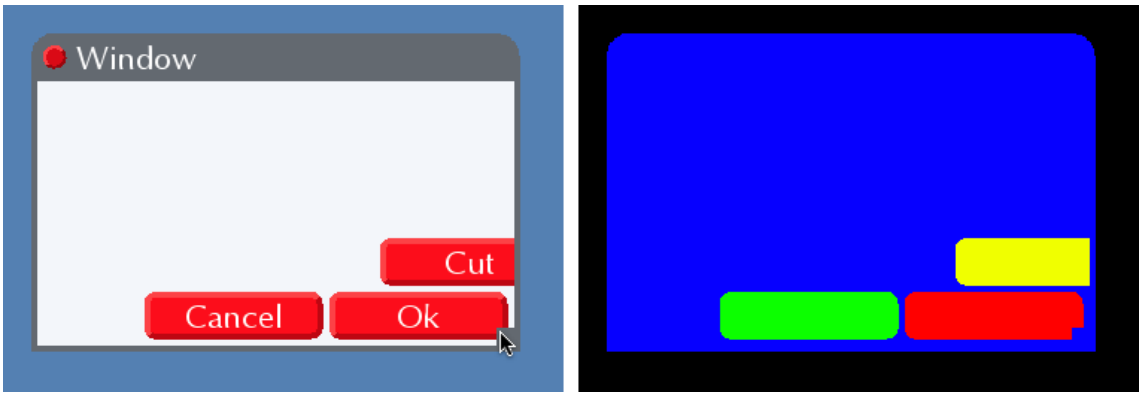


FIGURE 2.3 – Une interface affichée à l'écran (à gauche) et l'offscreen de picking correspondant (à droite). Dans l'offscreen, le fond (racine), la fenêtre `toplevel` et les boutons "Cancel", "Ok" et "Cut" sont représentés, respectivement, en noir, bleu, vert, rouge et jaune. Le pointeur de souris, représenté sur l'image de gauche, pointe sur le cadre de redimensionnement de la fenêtre, parce que celui-ci masque le bouton "Ok".

dite de dessin hors-écran ou "offscreen". Cette technique consiste à dessiner l'interacteur dans une surface dédiée, appelée l'offscreen de picking, qui ne sera jamais affichée à l'écran : elle ne sert qu'au picking. Au lieu d'utiliser différentes couleurs dans le dessin de l'interacteur (pour le fond, le texte, etc.), on utilise une seule "couleur" qui correspond en fait à un *numéro d'identifiant* propre à l'interacteur. Dans l'offscreen de picking illustré sur la figure 2.3, les identifiants sont représentés par des couleurs vives. Toute opération de dessin à l'écran sera donc dédoublée par une opération de dessin dans l'offscreen de picking. L'opération de picking devient triviale : par construction, l'identifiant de l'interacteur concerné par un clic en (x, y) est simplement la valeur du pixel en (x, y) dans l'offscreen de picking.

En conséquence, les fonctions de dessin des différentes classes d'interacteur reçoivent en paramètre non pas une, mais deux surfaces sur lesquelles dessiner l'interacteur. L'une des surfaces correspond à l'écran, l'autre à l'offscreen de picking.

Utilisations du gestionnaire d'événements

Le module de gestion d'événement offre des fonctions pour lier un couple événement-traitant ("bind") ou supprimer ce lien ("unbind"). Ces fonctions seront appelées à l'initialisation de la bibliothèque pour l'enregistrement des traitants internes, elles sont aussi appelées depuis le programme principal où le programmeur de l'application enregistre les traitants externes. Mais ces fonctions peuvent aussi être appelées depuis les traitants eux-mêmes pour modifier dynamiquement le comportement de l'interface. Au chapitre suivant, nous donnons au paragraphe 3.6.4 un exemple de modification dynamique des liens pour réaliser le déplacement des fenêtres.

2.2.4 Gestionnaire de géométrie

Définir la position et la taille des widgets dans leur parent peut être une tâche complexe. Dans le cas le plus simple, on peut spécifier position et taille de façon *absolue*, comme par exemple "place le bouton aux coordonnées $(10, 10)$ dans le repère de son parent et avec une taille de 80×30 pixels". Mais les programmeurs d'applications graphiques ont souvent besoin d'exprimer tailles et positions des widgets de façon plus abstraite. Par exemple, quand une fenêtre est redimensionnée, un bouton "valider" doit rester proche du coin en bas à droite de sa fenêtre. Il n'est pas souhaitable de laisser au programmeur d'application la tâche de recalculer les coordonnées du bouton à chaque modification de la taille de la fenêtre. Il faut donc proposer une fonction qui permet de spécifier "place ce bouton à l'angle en bas à droite de son parent, avec une marge de 10 pixels avec les bords du parent".

C'est le rôle d'un *gestionnaire de géométrie* d'enregistrer les contraintes de position et taille exprimées par le programmeur et d'être capable de les traduire en position et taille absolue dans le parent. C'est notamment nécessaire lorsque le parent est redimensionné ou lorsqu'un widget est détruit et que la place qu'il libère doit être redistribuée aux widgets restant dans le parent. Il existe différentes stratégies courantes de gestion de la géométrie. Le bouton "valider" évoqué ci-dessus pourra utiliser un gestionnaire de géomé-

trie qui accepte des contraintes simples par rapport au parent. Un gestionnaire de géométrie plus complexe prendra en compte également des contraintes liées à tous les widgets dont il a la charge. Un gestionnaire de type “grille” par exemple permet d’exprimer position et taille sous forme de lignes et colonnes. Par exemple, “place ce widget label dans la troisième ligne et deuxième colonne”. La taille de ce label dépendra notamment de la largeur de la deuxième colonne, qui elle-même pourra dépendre de la largeur requise par tous les widgets présents dans cette colonne et de la largeur du parent. Un exemple d’interface utilisant un gestionnaire de grille est illustré sur la figure 4.7.

Dans le projet, on impose la réalisation d’un seul gestionnaire de géométrie : le “placeur” (*placer* en anglais). C’est un gestionnaire simple qui permet de placer un interacteur en exprimant position et taille de façon *absolue* ou *relative au parent*. Davantage de détails sur ses spécifications seront donnés au paragraphe 3.5. Si vous avez du temps, vous pourrez proposer en extension au projet un autre gestionnaire de géométrie, comme indiqué en section 4.2.6.

2.2.5 Gestion de l’application

Le module de gestion de l’application est assez simple, il est en charge des principales étapes du cycle de vie de l’application. Le module offre des fonctions qui permettent d’initialiser l’application, de lancer la boucle principale, et finalement de libérer les ressources utilisées par l’application. En interne, le module alloue et initialise les structures de données que la bibliothèque gère pour la durée de vie de l’application. Il alloue en particulier la fenêtre graphique système qui joue le rôle d’interacteur racine dans la hiérarchie d’interacteurs (cf. 2.2.2). Le module initialise également les structures propres au gestionnaire d’événement, aux classes d’interacteurs et aux différents gestionnaires de géométrie.

Le module offre également :

— un point d’accès pour l’interacteur racine de l’application. Ceci permet de donner un parent aux interacteurs de plus haut niveau créés par le programmeur de l’application,

— une fonction permettant de demander la terminaison de l’application, c’est-à-dire de sortie de la boucle principale. Le programmeur appelle en général cette fonction en réaction à une action spécifique de l’utilisateur, comme l’appui sur un bouton “Quitter”.

2.2.6 Programme principal

Votre bibliothèque ne contient pas de programme principal : c’est le programmeur d’application graphique qui écrit un programme principal spécifiquement pour une application donnée. Dans le cadre du projet, nous vous donnons plusieurs exemples de programmes principaux qu’il vous faut faire fonctionner avec votre implémentation de la bibliothèque. Ces programmes sont présentés en section 4.1. Vous devrez en écrire d’autres pour tester tout ou une partie de vos modules.

Un programme principal a recours au module application pour gérer la vie du programme interactif. Il utilise aussi les modules interacteurs pour créer et configurer les éléments de son interface graphique. Il invoque les gestionnaires de géométrie pour placer les différents interacteurs créés et il enregistre, grâce au gestionnaire d’événements, les traitants à appeler lors des événements pertinents pour l’application.

Chapitre 3

Approfondissements

Dans ce chapitre, les concepts et algorithmes présentés au chapitre précédent sont détaillés. C'est aussi l'occasion de faire le lien avec les fichiers d'en-têtes C fournis. Ce chapitre doit être vu comme un complément des commentaires présents dans ces fichiers. Ces commentaires sont également regroupés sous forme HTML dans la documentation *doxygen* (cf. 5.2) ; consultez-la en parallèle de la lecture de ce chapitre.

3.1 Services de dessin

Les services de dessin permettent de définir la valeur des pixels de l'écran pour former l'image des interacteurs. Comme présenté en section 2.2.2, les deux plus bas niveaux d'abstraction vous sont fournis (niveau pixel et niveau primitives graphiques), c'est à vous de réaliser le dessin des interacteurs en utilisant les primitives graphiques.

3.1.1 Niveau pixel

Les fonctions de gestion des zones mémoire pour les surfaces de dessin sont déclarées dans le fichier `"hw_interface.h"`, leur nom commence par "hw_". Les autres fonctions des services de dessins sont déclarées dans le fichier `"ei_draw.h"`. La réalisation de ces fonctions vous est fournie dans la bibliothèque `"libeibase.a"` : vous n'avez pas à les réaliser.

Avant de pouvoir travailler sur des pixels, il faut d'abord initialiser l'accès au matériel, c'est à dire à la carte graphique de l'ordinateur (appel de `hw_init()`), puis allouer une zone mémoire qui représente l'image. On appelle ce type de zone mémoire une *surface de dessin*.

Surface de dessin

Il y a deux types de surfaces de dessin : celles qui apparaissent à l'écran et celles qui ne sont pas affichées. Ces dernières sont appelées "offscreen" (hors de l'écran). Elles servent par exemple à *l'offscreen de picking* (voir le paragraphe "Aiguillage d'événement par picking" en 2.2.3), ou bien à préparer un dessin qui sera ensuite copié sur une surface affichée à l'écran. Dans le projet, une surface de dessin est du type `ei_surface_t` qu'elle soit affichée ou offscreen. Toutes les fonctions de dessin ou d'accès aux pixels doivent recevoir un paramètre de type `ei_surface_t`. Vous ne créez qu'une seule surface qui sera affichée à l'écran, en appelant la fonction `hw_create_window(...)`. Cette fonction vous permet de choisir d'afficher l'application à l'intérieur d'une fenêtre système, ou bien sur tout l'écran ("fullscreen"). Vous travaillerez avec plusieurs surfaces de dessin offscreen que vous pouvez créer explicitement avec `hw_surface_create(...)`, ou bien qui sont renvoyées par les fonctions de dessin de texte (`hw_text_create_surface(...)`), ou bien de chargement d'image depuis un fichier (`hw_image_load(...)`). Quand le programme n'a plus besoin d'une surface de dessin, il faut penser à la libérer (`hw_surface_free(...)`).

Les surfaces de dessin sont une ressource partagée entre le programmeur et le système d'exploitation qui doit gérer leur transfert entre la mémoire principale et la mémoire de la carte graphique. Vous ne pouvez donc pas simplement modifier directement les pixels. Il faut au préalable bloquer la mémoire en appelant la fonction `hw_surface_lock(...)`. Une fois que vous avez modifié des pixels sur la surface affichée à l'écran, les modifications ne sont pas immédiatement visibles à l'écran : il faut d'abord débloquer la

surface (`hw_surface_unlock(...)`), puis signaler au système que la surface doit être mise à jour sur la carte graphique par appel de la fonction `hw_surface_update_rects(...)`. Cette fonction accepte une liste de rectangles en paramètre. Vous pouvez utiliser cette liste pour limiter la mise à jour de l'écran à ces rectangles, ce qui optimise la quantité de transfert mémoire : une image est une structure de donnée de grande taille. Transférer toute l'image pour une modification de quelques pixels serait un gaspillage important de la bande passante de la mémoire et aurait des effets négatifs sur la réactivité de l'application.

Pour obtenir l'adresse mémoire du premier pixel de l'image, vous appelez la fonction `hw_surface_get_buffer(...)`. Vous pouvez appeler cette fonction uniquement sur une surface qui a été préalablement bloquée (`hw_surface_lock(...)`). Par ailleurs, l'adresse mémoire renvoyée par cette fonction est valide uniquement pendant la durée où la surface est bloquée. Si la surface est débloquée, puis à nouveau bloquée, l'adresse du premier pixel peut avoir été changée par le système. En résumé, la surface de dessin affichée à l'écran s'utilise dans un cycle de ce type :

— La surface est bloquée (`hw_surface_lock(...)`).
— La surface est modifiée par des appels à des primitives graphiques ou par modification directe de ses pixels (`hw_surface_get_buffer(...)`).
— Avant la mise à jour, la surface est libérée (`hw_surface_unlock(...)`).
— Les zones de la surface à mettre à jour à l'écran sont signalées au système d'exploitation (`hw_surface_update_rects(...)`).

Représentation en mémoire

Dans ce projet, vous travaillez avec des pixels en couleur composite : chaque pixel est constitué de 3 composantes (rouge, vert, bleu), ou bien, en anglais (Red, Green, Blue). On parle de pixel *RGB*. Chaque composante représente l'intensité lumineuse du pixel dans sa bande de fréquence. En combinant les intensités lumineuses du rouge, vert et bleu d'un pixel, on peut afficher un grand nombre de couleurs différentes. C'est le mécanisme de synthèse additive [1]. On utilise, en général, trois octets (un pour chaque composante R, G, B). On peut alors représenter $2^{24} = 16777214$ couleurs différentes. Le rouge le plus lumineux est représenté par le triplet (255, 0, 0), un rouge plus sombre par (120, 0, 0). Le blanc est représenté par (255, 255, 255) et le noir par (0, 0, 0).

En pratique, il n'est pas commode de traiter des pixels de 3 octets car les ordinateurs ont des registres de 4 ou 8 octets (en fonction de leur architecture : 32 ou 64 bits). Il est préférable *d'aligner* les pixels sur les frontières de registre, donc de leur donner une taille en octets qui est un multiple de 4. On préfère donc ajouter un octet aux pixels en couleur pour leur donner une taille de 4 octets. Cet octet additionnel peut être inutilisé (on parle de pixel RGBX), mais il est souvent mis à profit pour représenter la *transparence* du pixel notée "Alpha" (voir la section "Transparence" plus bas). On parle alors de pixel RGBA.

Différents systèmes d'exploitation ordonnent les composantes de différentes façons (RGBA, ARGB, BGRA, ABGR). C'est le cas dans votre projet : si vous compilez votre projet sur MS Windows et Ubuntu, vous n'aurez pas forcément le même ordre des composantes. Pour connaître l'ordre des composantes d'une surface, on appelle la fonction `hw_surface_get_channel_indices(...)`. Utilisez cette fonction pour réaliser la fonction `ei_map_rgba(...)`. `ei_map_rgba(...)` permet de s'abstraire du problème de l'ordre des composantes : elle accepte un paramètre de type `ei_color_t` qui exprime explicitement les composantes rouge, verte, bleue et alpha de la couleur désirée et elle renvoie un entier sur 32 bits dans lequel chaque octet R, G, B, A a été correctement positionné. On peut alors copier cet entier directement en mémoire.

Transparence

L'apparence d'une interface graphique peut nécessiter des effets de *transparence*. Par exemple, la couleur de fond des fenêtres de l'application "Puzzle" (cf. section 4.1.6) est définie par la variable `toplevel_bg` dont la valeur est (255, 255, 255, 96). La valeur de transparence est donc 96. Par convention, 0 est la transparence maximale (complètement transparent) et 255 est la transparence minimale (complètement opaque). La valeur 96 correspond donc à une couleur transparente à 38%. On peut voir sur la figure 4.3 que la fenêtre "Puzzle" du premier plan révèle la fenêtre de second plan au travers de la case vide. L'apparence de la fenêtre de second plan est légèrement teintée en blanc.

Nous détaillons ici le principe de la programmation de l'effet de transparence. Vous n'aurez pas, à priori, à programmer vous-même l'effet de transparence puisqu'il est pris en charge par les primitives graphiques.

1. http://www.profil-couleur.com/lc/006-synthese-additive.php

Il est toutefois utile d'en connaître le principe pour bien utiliser ces primitives. Pour programmer cet effet de transparence, on dessine les objets dans l'ordre d'empilement : du plus *profond* au plus *proche* de l'écran. À chaque dessin d'un nouveau pixel, on calcule le pixel résultat comme la moyenne pondérée du pixel déjà présent dans la surface et du nouveau pixel, la valeur de transparence du nouveau pixel servant de coefficient de pondération. Soit P un pixel à afficher par transparence sur un pixel S de la surface. Soient P_R et P_A les composantes rouge et alpha du pixel à afficher et S_R la composante rouge du pixel de la surface, alors :

$$S_R = (P_A * P_R + (255 - P_A) * S_R)/255 \tag{3.1}$$

On calcule de façon similaire S_G et S_B, les composantes verte et bleue de la surface. Il est inutile de calculer S_A car la transparence résultante n'est jamais utilisée. Seule la transparence des objets à dessiner apparaît dans la formule 3.1.

3.1.2 Niveau primitives graphiques

Les fonctions de dessin du niveau *primitives graphiques* sont déclarées dans le fichier `"ei_draw.h"`. Les fonctions `ei_draw_polyline(...)` et `ei_draw_polygon(...)` prennent en paramètre une liste de points qui définissent la forme (ligne brisée ou polygone plein) à tracer. La fonction `ei_draw_text(...)` doit dessiner un texte dans une surface. Pour la réaliser, vous utilisez la fonction `hw_text_create_surface` qui crée une nouvelle surface de dessin contenant le texte à afficher, puis pour copiez les pixels de cette nouvelle surface grâce à la fonction `ei_copy_surface(...)` que vous devez aussi réaliser. Enfin, la fonction `ei_fill(...)` remplit une surface d'une couleur donnée, remplissage qui peut être limité à l'intérieur d'un rectangle : `ei_fill(...)` peut donc être utilisée pour dessiner un rectangle plein.

3.2 Valeurs par défaut

L'interface de programmation (API) de la bibliothèque contient des fonctions qui peuvent accepter de nombreux paramètres. Par exemple, la fonction de configuration d'un widget de type `button`, `ei_button_configure(...)` accepte 15 paramètres. Ces paramètres permettent de définir précisément chaque détail de l'aspect du bouton (texte du bouton, fonte utilisée, taille de la fonte, couleur du texte, etc.). Il serait fastidieux pour le programmeur d'applications de devoir spécifier ces 15 paramètres à chaque fois qu'il veut modifier un seul aspect du bouton, comme par exemple son texte. En général, le programmeur n'a pas d'intention particulière pour la plupart des paramètres. Il souhaite simplement que son bouton ait l'air *normal*.

L'API de la bibliothèque utilise donc un mécanisme de *valeurs par défaut*. Quand un paramètre n'a jamais été défini par le programmeur, sa valeur par défaut est celle qui donnera un aspect *normal* au widget. Quand le programmeur a déjà défini le paramètre lors d'un précédent appel à la fonction, alors la valeur par défaut est la valeur précédente. Au lieu de passer directement la valeur d'un paramètre dans un appel de fonction, on passe un *pointeur* vers une variable qui contient la valeur du paramètre. Par convention, on définit que lorsque ce pointeur est à `NULL`, c'est que le programmeur ne souhaite pas spécifier le paramètre, il souhaite conserver la valeur par défaut.

Par exemple, pour changer uniquement la couleur de fond d'un bouton, on passe en paramètre un pointeur vers cette couleur et on passe la valeur `NULL` pour tous les autres paramètres :

```
ei_color_t   red     = {0xff, 0x00, 0x00, 0xff};

ei_button_configure(my_button, NULL, &red, NULL, NULL, NULL, NULL, NULL,
                               NULL, NULL, NULL, NULL, NULL, NULL, NULL);
```

3.3 Polymorphisme

Pour implémenter différentes *classes de widgets*, de même que différents *gestionnaires de géométrie*, vous allez devoir utiliser du *polymorphisme* (= plusieurs formes). Par exemple, la bibliothèque doit pouvoir allouer l'espace mémoire nécessaire aux paramètres d'un nouvel interacteur sans même savoir de quel type d'interacteur il s'agit (un "bouton" ? une fenêtre "toplevel" ?). En d'autres termes, elle pourra appeler une

fonction d'interacteur sans connaître sa *forme* réelle. Autre exemple : à certains moments, la bibliothèque demande à un widget de se dessiner sur l'écran. Mais il y a autant de fonctions de dessin qu'il y a de *classes* de widgets (`draw_button(...)`, `draw_toplevel(...)`, etc.). On souhaite que la bibliothèque puisse appeler une fonction `draw`, mais que cet appel soit effectivement traduit par un appel différent en fonction de la classe du widget qui est dessiné.

Pour réaliser ce polymorphisme, il faut résoudre le problème des différentes formes de fonctions, mais également le problème des différentes formes des données.

3.3.1 Polymorphisme des données

Différentes classes de widget peuvent nécessiter différents attributs pour décrire leurs instances. Par exemple, un widget de la classe `toplevel` nécessitera le titre de la fenêtre, la couleur de fond, la présence ou l'absence d'un bouton de fermeture et d'un bouton de redimensionnement. Mais un widget de la classe `button` nécessitera uniquement le texte à afficher sur le bouton [2]. Par contre, quelle que soit sa classe, un widget appartient toujours à une classe, a toujours un parent, et a toujours une liste de descendants (possiblement vide). La classe du widget, le parent et les descendants sont donc trois attributs universels partagés entre toutes les classes.

Le polymorphisme des données est implémenté en utilisant une *structure commune* qui regroupe tous les attributs universels : `ei_widget_t` pour les widgets, et `ei_geometry_param_t` pour les gestionnaires de géométrie.

```
typedef struct ei_widget_t {
        ei_widgetclass_t*       wclass;
        (...)
        struct ei_widget_t*     parent;
        struct ei_widget_t*     children_head;
        (...)
} ei_widget_t;
```

Les attributs qui sont spécifiques à une classe de widgets particulière ou à un gestionnaire de géométrie particulier sont ajoutés en créant une nouvelle structure de données dont le premier champ est du type de la *structure commune*.

```
typedef struct ei_maclasse_t {
        ei_widget_t             widget;

        int                     specific_attribute1;
        char                    specific_attribute2[80];
} ei_maclasse_t;
```

Il est obligatoire d'avoir la structure commune en *premier* champ de la structure spécifique : c'est ce qui permet de réaliser certains traitements sur les données sans même savoir quelle est la forme spécifique de ces données. En effet, un *pointeur* vers une structure de type `ei_maclasse_t` *peut être utilisé comme* un pointeur vers une structure de type `ei_widget_t` parce que le premier champ de `ei_maclasse_t` est de type `ei_widget_t` et que le langage C ordonne les champs en mémoire dans l'ordre de déclaration.

En forçant le type de donnée ("typecast") vers la structure commune, on peut donc réaliser un traitement commun aux widgets de toute classe. Par exemple, la fonction `widget_set_parent` ci-dessous attribue un parent à un widget sans savoir à quelle classe ce widget appartient. Pour appeler cette fonction, le programmeur change le type du pointeur du widget.

```
void widget_set_parent (ei_widget_t* widget, ei_widget_t* parent)
{
  widget->parent = parent;
}

ei_maclasse_t   mon_widget;

widget_set_parent((ei_widget_t*)&mon_widget, root_window);
```

2. L'exemple est simplifié pour l'explication. Ces deux classes de widgets ont en réalité d'autres attributs.

Le polymorphisme des données permet donc de réaliser des *traitements communs* aux différentes formes. Nous allons voir comment le polymorphisme des fonctions permet de réaliser les *traitements spécifiques*.

3.3.2 Polymorphisme des fonctions

Pour qu'un traitement commun à toute forme puisse appeler des traitements spécifiques à la forme effectivement traitée, on utilise des pointeurs de fonctions. Le type `ei_widgetclass_t` regroupe les fonctions que doit implémenter toute classe de widgets :

```
typedef struct ei_widgetclass_t {
        ei_widgetclass_name_t                   name;
        ei_widgetclass_allocfunc_t              allocfunc;
        ei_widgetclass_releasefunc_t            releasefunc;
        ei_widgetclass_drawfunc_t               drawfunc;
        ei_widgetclass_setdefaultsfunc_t        setdefaultsfunc;
        ei_widgetclass_geomnotifyfunc_t         geomnotifyfunc;
        struct ei_widgetclass_t*                next;
} ei_widgetclass_t;
```

Le type `ei_widgetclass_drawfunc_t`, par exemple, spécifie la signature que doit respecter la fonction de dessin d'une classe de widgets :

```
typedef void (*ei_widgetclass_drawfunc_t) (struct ei_widget_t*    widget,
                                           ei_surface_t           surface,
                                           ei_surface_t           pick_surface,
                                           ei_rect_t*             clipper);
```

Pour ajouter une nouvelle classe de widgets dans la bibliothèque, on programme chacune de ces fonctions pour la nouvelle classe, puis on crée la structure de type `ei_widgetclass_t` dans laquelle on enregistre les pointeurs vers ces fonctions, puis on enregistre cette classe de widgets dans la bibliothèque par un appel à `ei_widgetclass_register(...)`.

```
void maclasse_drawfunc (struct ei_widget_t*    widget,
                        ei_surface_t           surface,
                        ei_surface_t           pick_surface,
                        ei_rect_t*             clipper)
{
  /* implémentation du dessin d'un widget de la classe "maclasse" */
}

ei_widgetclass_t       maclasse;
...
maclasse.drawfunc      = &maclasse_drawfunc;
...

ei_widgetclass_register(&maclasse);
```

La structure qui décrit une classe de widgets contient donc une *table des pointeurs* des fonctions spécifiques de la classe. On veille à ce qu'un pointeur vers cette structure soit présent dans la structure qui représente les widgets (c'est le premier champ, `wclass`, de la structure `ei_widget_t`). La bibliothèque peut alors appeler les fonctions spécifiques à une classe de widget sans la connaître, par simple déréférencement d'un pointeur de fonction. Par exemple, pour appeler le dessin d'un widget :

```
void traitement_commun(ei_widget_t* widget)
{
```

```
        ...
        widget->wclass->drawfunc(widget,
                                 draw_surface, pick_surface, clip_rect);
        ...
}
```

Les exemples que nous avons utilisés ici concernent les classes de widgets, mais les mêmes principes s'appliquent aux gestionnaires de géométrie :

- Le type `ei_geometry_param_t` définit les attributs communs à tout gestionnaire de géométrie.
- Tout gestionnaire de géométrie ajoute ses attributs spécifiques en créant une nouvelle structure dont le premier champ est de type `ei_geometry_param_t`.
- Le type `ei_geometrymanager_t` regroupe les pointeurs vers les fonctions spécifiques d'un gestionnaire.
- Pour ajouter un gestionnaire de géométrie à la bibliothèque, on crée une structure de type `ei_geometrymanager_t`, on y stocke les pointeurs vers les fonctions du gestionnaire et on enregistre le gestionnaire par un appel à `ei_geometrymanager_register(...)`.

3.4 Classes et hiérarchie de widgets

3.4.1 Classes de widgets

Tout widget appartient à une classe de widgets. Un widget est créé en appelant la fonction `ei_widget_create(...)` et en passant en paramètre le nom de la classe du widget, ainsi que le parent du widget. Cette fonction commence par vérifier que la classe dont le nom a été passé en paramètre est connue par la bibliothèque. Elle peut alors appeler la fonction d'allocation de widgets de la classe (`allocfunc`, du type `ei_widgetclass_allocfunc_t`) pour allouer un bloc mémoire assez grand pour stocker tous les attributs du nouveau widget de cette classe. La fonction `ei_widget_create(...)` se charge ensuite d'initialiser les attributs communs à tous les widgets (classe, parent, descendance, etc.), puis elle appelle la fonction d'initialisation des attributs spécifiques à la classe (`setdefaultsfunc`).

Pour que le programmeur puisse spécifier les attributs des widgets créés, toute classe fournit une fonction de configuration. Dans le projet vous devez au minimum gérer 3 classes de widgets et donc fournir l'implémentation des 3 fonctions de configuration déclarées dans le fichier `"ei_widget.h"` : `ei_frame_configure(...)`, `ei_button_configure(...)` et `ei_toplevel_configure(...)`. Ces fonctions de configuration sont appelées par le programmeur juste après avoir créé le widget. Mais elles sont aussi appelées à tout moment dans le programme pour modifier dynamiquement certains attributs du widget.

3.4.2 Description des classes de widget demandées

Au minimum, il vous est demandé d'implémenter trois classes de widgets : *Toplevel*, *Button* et *Frame*. Ces trois classes sont décrites plus précisément ci-dessous. Elles ont en commun leurs trois premiers attributs de présentation : la taille demandée pour le widget (que le gestionnaire de géométrie peu satisfaire ou non, en fonctions d'autres contraintes), la couleur de fond du widget et la taille en pixels des bords du widget.

Toplevel

Il s'agit d'une classe de widget ayant un rôle de "contenant" qui prend la forme d'une fenêtre. Les fenêtres sont configurées par un appel à `ei_toplevel_configure`. Un exemple de fenêtre est représenté en figure 2.3 (à gauche). Les fenêtres sont constituées d'une barre d'en-tête sur le haut avec un titre, d'un bouton en haut à gauche pour fermer la fenêtre, et d'une zone cliquable en bas à droite pour le redimensionnement. Les fenêtres doivent pouvoir être déplacées par l'utilisateur en maintenant le bouton de la souris appuyé après avoir cliqué sur la barre d'en-tête. En plus des attributs communs décrits ci-dessus, les attributs propres aux fenêtres sont :

- le titre qui sera affiché dans la barre d'en-tête,
- un booléen spécifiant si la fenêtre peut-être fermée ou non, c'est-à-dire si la fenêtre doit afficher ou non un bouton de fermeture à gauche dans la barre d'en-tête,

— un champ énuméré (`ei_axis_set_t`) qui spécifie si la fenêtre est redimensionnable ou non, et si oui, sur quels axes (horizontal et/ou vertical),
— la taille minimale de la fenêtre, contrainte dont devra tenir compte le gestionnaire de géométrie en cas de redimensionnement.

Frame

Un widget de la classe `frame` est un cadre rectangulaire qui peut être utilisé pour dessiner un simple cadre, ou un cadre contenant du texte ou une image. Les cadres sont configurés par appel de la fonction `ei_frame_configure`. En plus des attributs communs décrits en début de section, les attributs propres aux cadres sont :

— un énuméré (`ei_relief_t`) spécifiant le relief du widget. Le relief donne un aspect 3D (enfoncé, relevé, plat) au cadre. Le relief est dessiné sur la bordure du widget. Si la bordure est spécifiée de largeur 0, alors aucun relief n'est dessiné. Nous expliquons en section A.5 comment donner une impression de relief,
— un texte, ainsi que les attributs spécifiant son aspect : fonte (`ei_font_t`), couleur de texte, point d'ancrage dans le rectangle du widget. Pour dessiner ce widget, on fera donc appel aux fonctions de dessin de texte du module d'interface avec le système (`"hw_interface.h"`),
— une image à dessiner à la place du texte, ainsi que les attributs spécifiant son aspect : un rectangle permettant de n'utiliser qu'une sous-partie de l'image, et le positionnement de l'image dans le widget au cas où l'image serait plus petite que le widget.

Button

La classe de widget *bouton* permet de créer des boutons interactifs qui, lorsque l'utilisateur clique dessus, déclenchent l'appel à un traitant externe (fourni par le programmeur). Vous devez réaliser le comportement standard des boutons : apparence enfoncée quand l'utilisateur clique sur le bouton, puis retour à une apparence en relief quand le clic est terminé. Mais aussi : lorsque l'utilisateur maintient le bouton de la souris enfoncé, il peut déplacer le pointeur hors des limites du bouton graphique, ce qui a pour effet de remettre le bouton en relief. Il peut ensuite revenir sur le bouton graphique et le rendre enfoncé à nouveau, et ainsi de suite tant que le bouton de la souris n'est pas relâché (testez ça sur n'importe quel bouton, de l'application FireFox par exemple). L'appel au traitant externe du programmeur est effectué *uniquement* si le bouton de la souris est relâché alors que le pointeur est au dessus du bouton graphique.

Des boutons sont représentés sur la figure 2.3 (à gauche) où apparaissent les boutons "Ok", "Cancel", et "Cut". Configurés par appel de la fonction `ei_button_configure`, les boutons possèdent les mêmes attributs que les widgets de la classe frame : relief, texte ou image pouvant être dessinés dans le bouton. Seuls trois attributs sont spécifiques aux boutons :

— le *rayon* des arrondis aux angles du bouton,
— l'adresse d'une fonction *traitant*, de type `ei_callback_t`. Cette fonction doit être appelée par la bibliothèque lorsque l'utilisateur clique sur le bouton.
— une adresse mémoire permettant au programmeur de l'application de passer un paramètre spécifique à ce bouton particulier, lors de l'appel du traitant.

Les *cadres* et les *boutons* ont donc des fonctions de configuration très similaires, parce que leurs apparences sont très similaires. Il paraît donc judicieux de *mettre en facteur* leur implémentation.

3.4.3 Hiérarchie de widgets

Comme indiqué en section 2.2.2, les widgets sont organisés hiérarchiquement. Le champ `parent` de la structure `ei_widget_t` pointe vers l'unique parent du widget. Tout widget, sauf le widget racine, doit avoir un parent. Les champs `children_head` et `children_tail` pointent, respectivement, vers la tête et la queue de la liste chaînée des descendants. Si le widget n'a pas de descendant, alors cette liste est vide et ces deux champs sont à `NULL`. Le champ `next_sibling` pointe vers le descendant suivant ou est à `NULL` pour le dernier descendant.

L'ordre de la liste définit l'ordre de profondeur des descendants : le premier descendant peut être écrasé par les autres descendants s'ils le chevauchent et il apparaîtra donc *derrière* les autres. Il est parfois nécessaire de changer l'ordre des descendants pour modifier la présentation devant/derrière. Par exemple, quand l'utilisateur clique sur la barre d'en-tête d'une fenêtre `toplevel` pour la faire passer devant, il faut faire

passer le widget `toplevel` correspondant *en dernier* dans la liste des descendants de son parent (la fenêtre racine).

De nombreuses opérations sur un widget s'appliquent à sa descendance : la destruction d'un widget entraîne la destruction récursive de toute la descendance. Si un widget n'est pas détruit, mais simplement retiré de l'écran, alors sa descendance est également retirée de l'écran. Enfin, le déplacement d'un widget entraîne un déplacement similaire de toute sa descendance, puisque la position des descendants est exprimée dans le repère de leur parent (voir la section suivante "Gestion de la géométrie : le placeur").

3.5 Gestion de la géométrie : le placeur

Les types et fonctions utilisés pour la gestion de la géométrie sont principalement déclarés dans le fichier `"ei_geometrymanager.h"`. Comme expliqué en section 2.2.4, le programmeur a le choix entre plusieurs gestionnaires de géométrie pour gérer la taille et la position de ses widgets à l'écran. En pratique, il n'est *imposé* qu'un seul gestionnaire de géométrie dans le projet, le "placeur". Mais vous pouvez choisir de réaliser, en extension du projet, un gestionnaire en grille (voir 4.2.6). Dans tous les cas, la bibliothèque doit pouvoir appeler la fonction de calcul de la géométrie quel que soit le gestionnaire de géométrie utilisé. Vous utilisez pour cela l'approche de polymorphisme détaillée ci-dessus en section 3.3. Dans la suite de cette section, nous détaillons les mécanismes de gestion de géométrie communs à tous les gestionnaires, puis nous détaillons le principe du "placeur".

3.5.1 Mécanismes communs de gestion de géométrie

Après la création d'un widget par l'appel à `ei_widget_create(...)`, le widget n'est pas encore géré par un gestionnaire de géométrie. Il n'apparaît donc pas à l'écran. Cela se traduit par le champ `geom_params` à `NULL` dans la structure `ei_widget_t`.

Pour positionner un widget dans son parent, et donc le faire apparaître à l'écran, le programmeur appelle la fonction de configuration du gestionnaire de géométrie souhaité. Chaque gestionnaire propose sa propre fonction de configuration car les paramètres de configuration sont différents d'un gestionnaire à l'autre. La fonction de configuration du *placeur* est `ei_place(...)`. Après l'appel, le champ `geom_params` du widget n'est plus NULL : il pointe vers une structure qui décrit le gestionnaire utilisé, ainsi que tous les paramètres de géométrie définis pour ce widget.

Tout gestionnaire de géométrie fournit une fonction de type `ei_geometrymanager_runfunc_t` dont le rôle est de recalculer la position et la taille d'un widget dans son parent. Cette fonction est appelée à chaque fois qu'une opération peut avoir un effet sur la taille et la position d'un widget. Par exemple, en reconfigurant un widget de type `button`, on peut lui changer son label de "Ok" à "Cancel". Ce simple changement de label peut avoir un effet sur la géométrie puisque le nouveau label est plus long. Il peut nécessiter une largeur supérieure du bouton. La fonction de configuration du widget doit donc appeler la `runfunc` du gestionnaire du widget, si le widget est géré à ce moment-là. De même, si un widget a des descendants (par exemple si c'est une fenêtre `toplevel`), et que ce widget change de taille, alors la taille et la position de tous ses descendants peut être modifiée et doit être recalculée par leurs gestionnaires respectifs.

3.5.2 Algorithme du *placeur*

Le programmeur d'application demande au *placeur* de gérer un widget et fournit les paramètres de placement en appelant la fonction `ei_place(...)`.

```
void ei_place (ei_widget_t*  widget,
               ei_anchor_t*  anchor,
               int*          x,
               int*          y,
               int*          width,
               int*          height,
               float*        rel_x,
               float*        rel_y,
               float*        rel_width,
               float*        rel_height);
```

Cette fonction prend en paramètre les position et taille absolues désirées (`x, y, width, height`), ainsi que les position et taille relatives désirées (`rel_x, rel_y, rel_width, rel_height`). Tous ses paramètres peuvent être fixés à `NULL` pour conserver la valeur par défaut (voir 3.2 "Valeurs par défaut"). Tous ces paramètres sont exprimés dans le repère du parent du widget, dont l'origine est l'angle en haut à gauche du parent, l'axe des abscisses croît vers la droite et l'axe des ordonnées croît vers le bas [3].

Les paramètres relatifs sont représentés par des nombres flottants. Une ordonnée relative de 0.0 correspond au côté haut du parent, 1.0 à son côté bas, et 0.5 à son centre. Une hauteur relative de 0.5 correspond à la moitié de la hauteur du parent.

Les paramètres de position (`x, y, rel_x, rel_y`) définissent la position ponctuelle d'un seul *pixel* du parent, mais le widget a une surface rectangulaire de plusieurs pixels. Il faut donc aussi spécifier comment le widget s'attache, ou *s'ancre*, sur cette position. C'est le rôle du paramètre `anchor` : une ancre Nord-Ouest (`ei_anc_northwest`), par exemple, signifie que c'est l'angle supérieur droit du widget qui sera attaché au point défini par les paramètres de position.

Les paramètres *absolus* et *relatifs* peuvent être combinés. Par exemple les valeurs des paramètres `rel_x=1.0` et `x=-10` spécifient une abscisse relative au bord droit du parent, mais avec une marge de 10 pixels du bord droit. C'est par cette approche que l'on peut placer le bouton "Ok" de la figure 2.3 dans l'angle inférieur droit de sa `toplevel`, en laissant une petite marge par rapport aux bords droit et inférieur de la `toplevel`. Le *placeur* se charge alors de recalculer la position du bouton "Ok" lorsque la `toplevel` est redimensionnée :

```
ei_anchor_t         anchor      = ei_anc_southeast;
int                 x           = -4;
int                 y           = -4;
float               rel_x       = 1.0;
float               rel_y       = 1.0;

ei_place (ok_button, &anchor, &x, &y, NULL, NULL, &rel_x, &rel_y, NULL, NULL);
```

Un gestionnaire de géométrie peut avoir à résoudre des contraintes incompatibles. Par exemple, un bouton est configuré, par appel de la fonction `ei_button_configure(...)`, avec le label "Validation". Ce label nécessite une certaine largeur pour pouvoir être affiché sans être tronqué, il définit donc une certaine *largeur par défaut* du bouton. Mais le programmeur peut aussi demander explicitement une largeur en fournissant un paramètre `requested_size` lors de l'appel à `ei_button_configure(...)`. Enfin, le programmeur peut encore demander une autre taille avec les paramètres de l'appel à `ei_place(...)`. Le *placeur* doit donc gérer une priorité : les paramètres de `ei_place(...)` sont prioritairement respectés. S'ils ne sont pas fournis (i.e. `width=NULL, rel_width=NULL`), alors c'est la taille demandée qui est respectée (paramètre `requested_size` de l'appel à `ei_button_configure(...)`). Si ce paramètre n'est pas fourni non plus, alors c'est la *taille par défaut* qui est respectée.

3.6 Gestion des événements

3.6.1 Principes

Les fonctions utilisées pour la gestion des événements sont définies dans le fichier `"ei_event.h"`. Le programmeur utilise les fonctions `ei_bind(...)` et `ei_unbind(...)` pour créer et détruire un lien entre :

— un *widget* ou une étiquette (*tag*),
— un *type* d'événement,
— une fonction "traitant" d'événement ou *callback*.

La bibliothèque doit se charger d'appeler le *traitant* lorsqu'un événement du *type* concerné a lieu sur le *widget* ou sur tout widget qui possède le *tag*.

Les types d'événements sont définis dans l'énumération `ei_eventtype_t`. Ils désignent l'appui et le relâchement d'une touche du clavier (`ei_ev_keydown`, `ei_ev_keyup`), l'appui et le relâchement d'un des boutons de la souris (`ei_ev_mouse_buttondown`, `ei_ev_mouse_buttonup`), et le déplacement du pointeur de la souris (`ei_ev_mouse_move`).

3. L'axe des ordonnées qui croît vers le bas est une convention informatique liée à l'organisation des images en mémoire : le début de la mémoire qui représente une surface graphique correspond à la ligne du haut de la surface.

Dans le projet, nous supposons que les deux seuls tags que possède un widget sont le *nom de sa classe* et le tag “all”. En particulier, nous ne proposons pas d’interface de programmation pour mettre et enlever des tags à des widgets, mais c’est une des extensions proposées (voir 4.2.4). Le tag du nom de la classe permet de définir des comportements au niveau d’une classe plutôt qu’un niveau d’un widget particulier. Par exemple, quand on clique sur tout widget de type “button”, on souhaite que le widget apparaisse avec un relief *enfoncé*. On pourrait pour cela créer un binding sur l’événement “appui sur le bouton de la souris” à la création de tout nouveau widget de type “button”, et lier un traitant qui s’occupe de changer l’apparence du bouton. Mais grâce aux tags, il suffit à l’initialisation de la bibliothèque de faire un seul binding sur le tag “button” pour que le traitant soit appelé quand l’événement a lieu sur n’importe quel bouton. Le tag “all” permet de réaliser des liens qui se déclenchent à chaque fois que l’événement associé intervient, qu’il y ait ou non un widget concerné par celui-ci. Par exemple, pour quitter l’application lorsque l’utilisateur appuie sur la touche “Escape”, on peut faire un binding sur le tag “all” et l’événement de type `ei_ev_keydown`.

Lorsqu’un événement a lieu, la bibliothèque appelle le ou les traitant(s) liés, en leur passant un paramètre de type `ei_event_t`. Ce paramètre permet au traitant de savoir quel est le type de l’événement, mais aussi de recevoir des *paramètres d’événements*. Par exemple, pour un événement de type `ei_ev_keydown` ou `ei_ev_keyup`, la structure `ei_event_t` contient le code de la touche qui a été enfoncée [4] et un champ de bits qui décrit l’état enfoncé ou relâché de toutes les touches spéciales (“majuscule”, “control”, “alt”, etc.). Pour les événements qui concernent la souris, les paramètres d’événements sont la position du pointeur de la souris et, le cas échéant, le numéro de bouton de souris qui a été enfoncé ou relâché.

3.6.2 Widget concerné par l’événement

Pour les événements qui concernent la souris, le programmeur peut limiter l’appel du traitant à un widget particulier. En d’autre termes, l’appel suivant :

```
ei_bind(ei_ev_mouse_move, my_window, NULL, my_callback, NULL)
```

demande à la bibliothèque d’appeler la fonction traitant “my_callback” lorsque la souris se déplace (événement de type `ei_ev_mouse_move`), mais uniquement lorsqu’elle se déplace sur le widget “my_window”. La bibliothèque doit donc être capable de déterminer quel est le widget concerné par les événement souris, c’est à dire celui que se trouve “sous” le pointeur de la souris. Pour cela, vous devez implémenter un *off-screen de picking*, tel que présenté dans le paragraphe “Aiguillage d’événement par picking” de la section 2.2.3.

Pour les événements qui concernent les touches du clavier il n’y a pas, dans l’événement, d’information de position sur l’écran. Ces événements ne sont pas liés à un widget spécifique et doivent être associés à leurs traitants par le tag “all”. Une des extensions proposées pour ce projet concerne la gestion d’un *focus clavier*, c’est à dire d’un widget désigné pour recevoir les événements clavier, et des interactions pour permettre à l’utilisateur de contrôler le focus clavier (voir 4.2.5).

3.6.3 Traitants externes et internes

Comme présenté en section 2.2.3, développer une application interactive consiste, entre autres, à programmer un ensemble de *traitants externes*. En plus de ces traitants créés par le programmeur de l’application, la bibliothèque fournit également un ensemble de *traitants internes* qui permettent de gérer les comportements standard des différents widgets, par exemple :

— un bouton s’enfonce lorsqu’on clique dessus,
— une toplevel peut être déplacée en cliquant sur son bandeau de titre et en maintenant le bouton appuyé pendant que la souris est déplacée,
— une toplevel peut être redimensionnée avec la même interaction, mais en cliquant sur le bouton de redimensionnement en bas à droite de la fenêtre.

Les deux derniers points sont détaillés dans la section suivante.

Trois boutons différents, par exemple les trois boutons “Ok”, “Cancel” et “Cut” de la figure 2.3, auront trois *traitants externes* différents, car l’appui sur chacun de ces boutons doit engendrer un traitement différent de la part de l’application. Par contre, il n’y a qu’un seul *traitant interne* qui gère tous les boutons,

4. Les identificateurs des code de touche clavier sont déclarés dans le fichier `"SDL_keysym.h"` de la bibliothèque SDL.

puisque le comportement standard ne varie pas en fonction des boutons (i.e. tous les boutons doivent apparaître enfoncés quand on clique dessus). La bibliothèque doit se charger de "lier" (`ei_bind(...)`) l'unique traitant interne qui gère l'enfoncement des boutons au tag "button" de cette classe de widget.

Afin de pouvoir gérer à la fois l'exécution de *traitants internes* et de *traitants externes*, la bibliothèque devra être capable d'appeler plusieurs *traitants* par événements. Lorsqu'un traitant a fini son traitement, il peut retourner la valeur `EI_FALSE` pour spécifier à la bibliothèque qu'elle pourra appeler un autre traitant par la suite. Par contre, s'il retourne `EI_TRUE`, il sera le dernier *traitant* appelé pour cet événement.

3.6.4 Exemple du déplacement

Une action de type *déplacement* sur un widget est le résultat d'une succession d'événements :
— un événement initiateur de type `ei_ev_mouse_buttondown` : à partir de ce moment, le widget doit être tenu au courant des événements de type `ei_ev_mouse_move` et `ei_ev_mouse_buttonup`,
— une succession de mouvements `ei_ev_mouse_move` : à chaque événement `ei_ev_mouse_move`, la position absolue du widget est alignée sur celle de la souris, permettant d'obtenir le mouvement interactif de la fenêtre correspondant au glissé de l'utilisateur,
— un événement de fin d'action `ei_ev_mouse_buttonup` : le widget n'est plus sensible aux événements `ei_ev_mouse_move` et `ei_ev_mouse_buttonup`.

Cette action est disponible de façon standard pour la classe de widget `toplevel`. C'est donc à la bibliothèque, et non au programmeur, de réaliser ce comportement. Ces bindings seront donc réalisés dans le code de la librairie au moment où la classe de widget `toplevel` est déclarée, et *non pas* dans le code de l'application par le programmeur.

Initialisation

Le *binding* (cf. 3.6) se fait au niveau de l'enregistrement de la classe : la fonction `ei_bind(...)` reçoit la valeur "`toplevel`" pour son paramètre `tag`, le paramètre `widget` étant laissé à `NULL`. L'objectif de ce premier binding est de détecter le démarrage du *déplacement*. Il est donc fait sur les événements de type `ei_ev_mouse_buttondown`.

Début d'action

L'utilisateur vient de cliquer avec le pointeur de souris sur la barre d'en-tête de la `toplevel`. Le traitant associé à `ei_ev_mouse_buttondown` pour la classe `toplevel` est appelée. L'action de *déplacement* démarre. La position du widget sera maintenant contrôlée par la souris. Il est donc nécessaire de remplacer sa gestion de géométrie actuelle, quelle qu'elle soit, par une gestion en positionnement absolu. Par ailleurs, il faut notifier la bibliothèque que nous sommes maintenant intéressés par les événements `ei_ev_mouse_move` et `ei_ev_mouse_buttonup`. Ceci est fait par deux appels à `ei_bind` effectués *dans le traitant* de l'événement `ei_ev_mouse_buttondown`. On pourra utiliser le tag "all", puisqu'aucune autre interaction ne peut avoir lieu pendant le déplacement de la fenêtre : nous sommes donc intéressés par ces événements quel que soit le widget concerné.

Micro-déplacements de la fenêtre

Chaque mouvement élémentaire de la souris déclenche maintenant l'appel du traitant liée à l'événement de type `ei_ev_mouse_move`. La position du curseur est utilisée pour calculer la nouvelle position du widget. On notera que la position du pointeur de la souris est donnée, dans les paramètres de l'événement, dans le repère de la fenêtre racine. Par contre, le positionnement absolu d'un widget, grâce au gestionnaire de géométrie, s'exprime dans le repère du parent de ce widget.

Fin d'action

L'utilisateur relâche le bouton de la souris. Le traitant associé à l'événement `ei_ev_mouse_buttonup` est appelée. Ce traitant doit simplement *désabonner* les traitants liés aux événements `ei_ev_mouse_move` et `ei_ev_mouse_buttonup`.

3.6.5 Exemple du redimensionnement

Le redimensionnement suit le même principe que l'action de *déplacement* :

— Initialisation par appel d'un traitant liée à `ei_ev_mouse_buttondown`. Ce traitant est enregistrée sur la classe de widget `toplevel`. Enregistrement de deux nouvelles traitants sur les événements de type `ei_ev_mouse_move` et `ei_ev_mouse_buttonup`,
— Déplacement de la souris alors que le bouton est toujours enfoncé : mise à jour de la taille du widget,
— Fin de l'action par détection de l'événement de type `ei_ev_mouse_buttonup`. Dé-enregistrement des traitants associés à `ei_ev_mouse_move` et `ei_ev_mouse_buttonup`.

3.7 Gestion de l'affichage

Dans la durée de vie d'une application graphique, l'écran doit être mis à jour de nombreuses fois pour de multiples raisons, par exemple :

— le programme modifie une option de configuration d'un widget (par exemple le label d'un bouton passe de "Démarrer" à "Arrêter"),
— une action de l'utilisateur provoque la création d'une nouvelle fenêtre,
— l'utilisateur déplace une fenêtre,
— l'utilisateur fait défiler le contenu d'une fenêtre.

Par souci d'optimisation et de simplification, les mises à jour à l'écran ne sont pas réalisées immédiatement. Elles sont *programmées* pour être réalisées plus tard lors de la phase de re-dessin dans la boucle principale. Ainsi, lorsqu'une mise à jour est nécessaire, la bibliothèque se contente d'appeler la fonction `ei_app_invalidate_rect(...)` et lui passe en paramètre le rectangle de l'écran qui doit être mis à jour. La bibliothèque mémorise ainsi l'ensemble des rectangles qui doivent être mis à jour.

Dans la boucle principale, après avoir traité un événement, la bibliothèque se charge de mettre à jour tous les rectangles qui ont été enregistrés par `ei_app_invalidate_rect(...)`. La surface de l'écran et la surface de picking sont bloquées (3.1.1 "Surface de dessin", 2.2.3 "Aiguillage d'événement par picking"). Puis, pour chaque rectangle à redessiner, il faut faire en sorte que la `drawfunc` de tous les widgets de la hiérarchie soit appelée en utilisant le rectangle comme *rectangle de clipping* (2.2.2 "Dessin des interacteurs, clipping"). L'ordre d'appel des `drawfunc` est important pour créer l'effet de profondeur (relation devant/derrière entre les widgets). Quand tous les rectangles à mettre à jour ont été traités, la bibliothèque débloque la surface de l'écran et demande au système d'exploitation de copier ces pixels sur l'écran par un appel à `hw_surface_update_rects`.

Pour *effacer* un widget de l'écran, parce qu'il a été détruit ou simplement retiré de l'écran, on se contente d'appeler `ei_app_invalidate_rect(...)` sur le rectangle qu'il occupait avant d'être effacé. Suivant le principe décrit ci-dessus, la bibliothèque appellera la `drawfunc` des widgets qui étaient sous le widget à effacer (il y a toujours au minimum le widget racine), ce qui aura pour conséquence d'écraser ses pixels, et donc de l'effacer de l'écran. Le même principe est utilisé pour effacer un widget qui a été déplacé : on appelle `ei_app_invalidate_rect(...)` sur le rectangle qu'occupait le widget *avant* le déplacement, afin de l'en effacer, puis on fait un deuxième appel à `ei_app_invalidate_rect(...)`, cette fois sur le rectangle qu'occupe le widget *après* déplacement, afin de l'y dessiner.

Le principe de re-dessin présenté ci-dessus suppose donc que la `drawfunc` de *tous* les widgets sera appelée *plusieurs fois* à chaque mise à jour de l'écran : une fois par rectangle à redessiner. Or, la mise à jour peut ne concerner qu'une toute petite surface de l'écran. Par exemple, le déplacement d'un widget de petite taille provoque le re-dessin de deux petits rectangles : ceux qu'occupe le widget avant et après son déplacement. Il serait inutile et coûteux en temps de calcul de redessiner à l'écran tous les widgets, même ceux qui n'ont aucune intersection avec ces deux petits rectangles. Le bon usage du *rectangle de clipping* est essentiel pour éviter ces traitements inutiles. Il faut veiller à tester si le rectangle englobant d'un widget intersecte le rectangle de clipping : dans le cas contraire (fréquent), aucun dessin n'est nécessaire.

Lorsque la bibliothèque parcourt tous les rectangles à redessiner, il se peut que de grandes zones de l'écran soient redessinées plusieurs fois. C'est le cas lors d'un petit déplacement d'une fenêtre, comme illustré sur la figure 3.1 : le rectangle d'intersection de l'ancienne et de la nouvelle position de la fenêtre sera redessiné deux fois, inutilement. En conséquence, avant de parcourir tous les rectangles de re-dessin, la bibliothèque pourra appliquer des stratégies d'optimisation, comme par exemple fusionner les rectangles ayant une intersection importante.

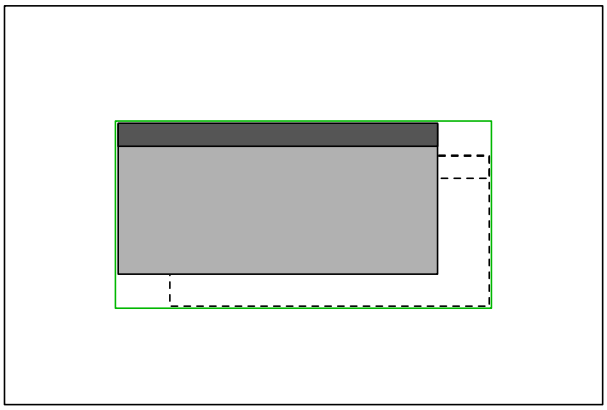


FIGURE 3.1 – Rectangles de re-dessin lors d'un déplacement de fenêtre. La fenêtre a été légèrement déplacée entre l'ancienne position (pointillés) et la nouvelle (fenêtre pleine). Deux rectangles de re-dessin sont générés : à l'ancienne et à la nouvelle position de la fenêtre. Il est préférable de ne redessiner qu'un seul rectangle (en vert) qui englobe les deux rectangles.

3.8 Programme principal et boucle principale

Comme cela a été introduit dans la section 2.1.2 "Structure d'un programme événementiel", un programme de type événementiel se décompose en deux grandes parties : l'initialisation et le lancement de la boucle principale. Ces deux parties sont détaillées dans les sous-sections suivantes.

3.8.1 Initialisation de l'application

Le programmeur initialise la bibliothèque par un appel à la fonction `ei_app_create(...)`. Cette fonction réalise les actions suivantes :

— initialisation de la couche graphique : `hw_init()`,
— enregistrement des classes de widgets disponibles : `ei_widgetclass_register(...)`,
— enregistrement des geometry manager disponibles (au minimum, le `placer`) : `ei_geometrymanager_register(...)`,
— création du widget racine de la classe `frame`,
— création d'une surface *offscreen* pour la gestion du picking : `hw_surface_create(...)`.

La bibliothèque étant initialisée, l'étape suivante pour le programmeur consiste à construire sa hiérarchie de widgets par appels successifs à la fonction `ei_widget_create(...)`. Le programmeur configure ses widgets (labels, couleurs, etc.) par appels aux fonctions correspondantes (`ei_button_configure(...)`, `ei_frame_configure(...)`, etc.). Il utilise `ei_bind(...)` pour spécifier les traitants externes des différents interacteurs. La "structure hiérarchique" de l'interface est en place. Les widgets sont également placés à l'écran par appel au gestionnaire de géométrie (`ei_place(...)`).

L'état initial de l'application étant créé, il faut maintenant lui "donner vie" en écoutant les événements, en les redirigeant vers les widgets correspondants et en gérant le rafraîchissement de l'écran. C'est la responsabilité de la boucle principale qui est implémentée par la bibliothèque et invoquée par appel de la fonction `ei_app_run()`.

3.8.2 Boucle principale

L'appel à la fonction `ei_app_run()` déclenche une boucle dont la sortie est contrôlée par un booléen. Le programmeur positionne ce booléen à vrai, lorsqu'il souhaite terminer l'application, par un appel à la fonction `ei_app_quit_request()`. Cet appel est généralement effectué depuis un traitant.

La boucle principale réalise les étapes suivantes :

— re-dessin des différentes zones nécessitant un rafraîchissement. Le principe est expliqué dans la section 3.7 "Gestion de l'affichage",
— attente du prochain événement : `hw_event_wait_next(...)`. Cette fonction endort le processus jusqu'à ce qu'il soit réveillé par le système d'exploitation lorsqu'un événement utilisateur a eu lieu.

— traitement de l'événement. La bibliothèque doit analyser l'événement pour identifier le widget concerné, s'il y en a un. Elle doit déterminer si un *traitant* est liée à cet événement et appeler ce traitant. Le principe de gestion des événements est détaillé dans la section 3.6 "Gestion des événements"

Chapitre 4

Travail à réaliser

Vous devez programmer une implémentation de la bibliothèque `libei.a` spécifiée aux chapitres précédents. Les structures de données et les signatures des fonctions de la bibliothèque sont imposées à travers les fichiers d'en-tête fournis. On appelle cet ensemble de structures et signatures "l'interface de programmation" de la bibliothèque, ou "Application Programming Interface" (**API**). Ces fichiers d'en-tête **ne doivent absolument pas être modifiés**. Vous êtes en revanche libre d'ajouter dans des fichiers d'en-tête séparés tous types ou fonctions qui vous seront nécessaires pour implémenter cette API. Les seules fonctions que vous pouvez utiliser sont celles que vous aurez écrites et celles qui vous sont fournies dans les headers. Vous pouvez, toutefois, utiliser les librairies standard du C (`"math.h"`, `"stdio.h"`, etc.)

La spécification de la bibliothèque, décrite dans ce document et dans les fichiers d'en-tête, est incomplète. Vous aurez parfois à faire des choix quant au fonctionnement de votre bibliothèque. Quand vous identifiez un point non spécifié, réfléchissez au choix le plus pertinent et *parlez-en aux encadrants*. Les encadrants discuteront avec vous de la pertinence de votre choix, et dans certains cas pourront vous corriger en vous rappelant les points de spécification que vous avez manqués.

Le projet s'appuie sur une bibliothèque d'interface avec le matériel, `SDL`, pour accéder aux pixels de l'écran et aux événements utilisateur. `SDL` est multiplateforme, vous pouvez donc développer le projet sur votre environnement préféré (Linux, Windows, OSX). Mais votre **projet sera évalué sous Linux sur les machines de l'Ensimag**. Il est donc de votre **responsabilité** de valider **régulièrement** que votre solution fonctionne correctement à l'Ensimag. D'autre part, le développement sur machine personnelle est de votre entière responsabilité. Les encadrants ne vous fourniront pas de support en séance pour faire fonctionner votre projet sur votre machine personnelle.

4.1 Code d'applications fournies

Afin de vous guider dans vos développements, nous vous fournissons dans le répertoire `tests` le code source de plusieurs applications de complexité croissante : une application minimale, un simple cadre, un bouton, une fenêtre contenant un bouton. Le répertoire `tests` contient également des applications réellement utilisables : un jeu de démineur et un jeu de puzzle. Ces applications utilisent une très grande partie des services de la bibliothèque et nécessitent donc une implémentation presque complète pour fonctionner. Tous ces fichiers de code source devront compiler et fonctionner avec votre bibliothèque.

4.1.1 Minimal

Cette application est la seule que vous pouvez compiler et exécuter dès le début du projet (`make minimal ; ./minimal`) : elle utilise uniquement les fonctions qui vous sont fournies dans `libeibase`. L'application se contente de remplir la fenêtre système en rouge et de dessiner un polygone à l'intérieur. Puis elle se met en attente d'un événement clavier (appui d'une touche) pour terminer.

4.1.2 Cadre (frame)

Cette application affiche un simple cadre ("frame" en anglais) tel qu'illustré sur la figure 4.1, à gauche. Elle est constituée d'un widget racine ayant pour descendant un widget de type `frame`. Le `frame` a une

taille fixée. Son positionnement est définit de façon absolue à l'intérieur du widget racine. Cette simple application ne gère pas les événements, il n'y a donc aucun moyen de la quitter, si ce n'est en tuant le processus (commande shell `kill` ou `xkill`).

Bien que simple, cette application nécessite le développement d'une partie importante de la bibliothèque. Pour vous aider dans ce développement, nous vous proposons un ensemble d'étapes dans l'annexe A.

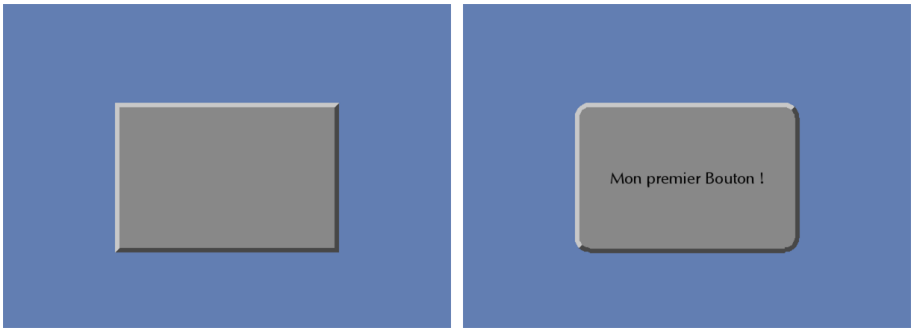


FIGURE 4.1 – À gauche : un simple cadre (frame.c), à droite : un simple bouton (button.c).

4.1.3 Bouton simple (button)

Cette application est similaire à la précédente (Cadre), si ce n'est que le widget `frame` est remplacé par un widget `button` (voir figure 4.1, à droite). Cette application introduit la gestion d'événements en prenant en compte les deux événements suivants :

— Sortie de l'application par appui sur la touche escape.
— Exécution d'une callback lors d'un clic souris sur le bouton. Cette callback affiche un message à l'écran.

Vous trouverez en annexe A.6 des indications pour la réalisation de la gestion des événements.

4.1.4 Fenêtre hello world

Cette application introduit un widget de la classe `toplevel` : une fenêtre avec une barre d'en-tête qui permet de la déplacer sur l'écran. Cette fenêtre a pour titre "hello world". Elle possède un bouton en bas à droite (voir figure 4.2). La fenêtre peut être déplacée et redimensionnée. Le bouton est placé "relativement" par rapport à la fenêtre, ce qui lui permet de rester dans le coin inférieur droit. Par ailleurs, le bouton conserve une largeur relative de la moitié de la largeur de la fenêtre.

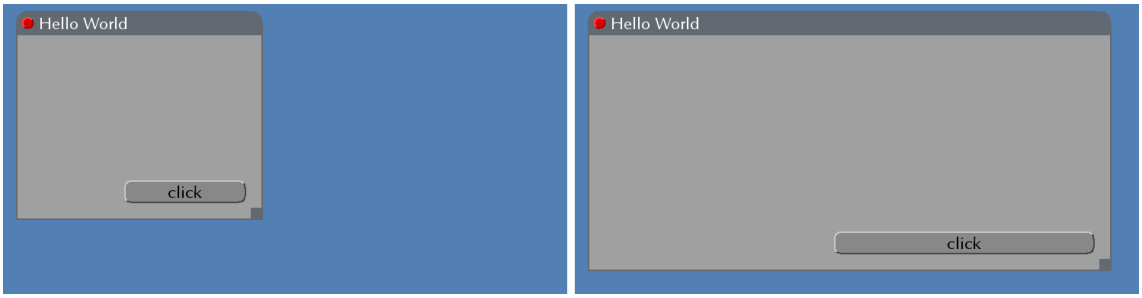


FIGURE 4.2 – Fenêtre redimensionnable et bouton de taille et de position relatives.

4.1.5 Démineur (minesweeper)

Cette application est un exemple d'utilisation de la bibliothèque pour un jeu de démineur. Chaque case du jeu est un bouton qui doit afficher une image (drapeau ou explosion) ou un texte (nombre de mines sur les cases voisines). La taille du champ de mines et le nombre de mines peuvent être passés en arguments

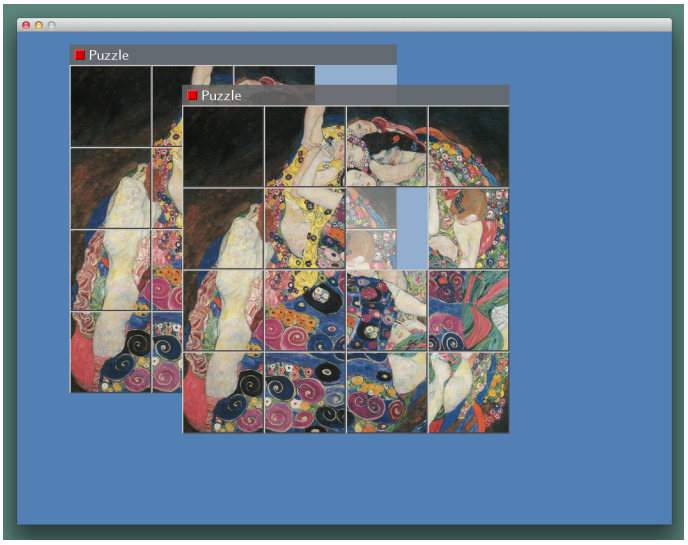


FIGURE 4.3 – Le jeu de taquin. Deux `toplevel` contenant chacune un jeu de taquin ont été ouvertes dans l'application. La fenêtre de plus haut niveau (fond bleu uni) est la fenêtre système de l'application, elle est gérée par l'OS (Linux, OSX, Windows) et non pas par votre bibliothèque.

dans la ligne de commande du programme. Les boutons du champ de mines utilisent volontairement le placeur (et pas le gestionnaire en grille, cf. section 2.2.4) pour que le programme puisse compiler avec une version minimale de votre bibliothèque. Une partie du haut de la fenêtre de jeu est utilisée pour le bouton "recommencer" ainsi que pour l'affichage du statut de la partie.

4.1.6 Puzzle

Cette application a elle aussi une utilité réelle : il s'agit d'un jeu de taquin de 15 pièces carrées (voir figure 4.3). Le contenu de chaque pièce provient du découpage d'une image initiale, dont le fichier peut être passé en paramètre du programme (sur la ligne de commande). Une pièce peut être déplacée si une des cases voisines est libre. Le déplacement est déclenché par un clic sur la pièce que l'on souhaite bouger.

4.2 Extensions

Le contrat minimum consiste à fournir une implémentation complète de l'API. Cette implémentation doit permettre la compilation et l'exécution correcte des applications décrites dans la section précédente. **Si ce contrat est rempli**, vous pouvez proposer des extensions. Nous proposons ci-dessous quelques idées d'extensions, mais cette liste n'est pas exhaustive. Le nombre d'étoiles entre parenthèses donne une indication sur la difficulté de l'extension, d'assez simple (*) à très compliquée (***).

4.2.1 Two048 (*)

Cette application présente dans le répertoire `test` est une implémentation du jeu 2048 (https://gabrielecirulli.github.io/2048/). Cependant, pour compiler, elle nécessite l'ajout d'un nouveau service dans la bibliothèque qui n'est pas spécifié dans les fichier d'en-tête : il s'agit de l'enregistrement d'un traitant appelé à la destruction d'un widget. L'interface de programmation est définie au début du fichier `"two048.c"` : c'est la fonction `ei_widget_set_destroy_cb(...)`. À vous de voir comment ajouter ce service *sans modifier* le reste de l'API.

4.2.2 Widget bouton radio (**)

Cette extension consiste à ajouter une classe de widget "bouton radio" (ou "radiobutton" en anglais) à votre bibliothèque. Un widget bouton radio fonctionne toujours avec d'autres widgets bouton radio. À un

FIGURE 4.4 – Exemple de boutons radio.

instant donné, un seul bouton peut être actif. Les boutons radios sont utilisés pour permettre 1 choix parmi n, comme par exemple donner un note, tel qu'illustré sur la figure 4.4.

À la différence des classes de widget `frame`, `button`, et `toplevel`, nous ne fournissons pas l'interface de programmation des `radiobutton`. À vous de spécifier une fonction `radiobutton_configure(...)`. Vous pourrez vous inspirer de l'interface de programmation des boutons classiques. Il faudra prévoir un moyen pour que tous les boutons radios d'un groupe se *connaissent*, de façon à se désactiver lorsqu'un autre bouton du groupe est activé. Par exemple, la fonction de configuration d'un bouton radio peut accepter en paramètre une liste chaînée de widgets qui contient tous les widgets radio bouton du groupe.

4.2.3 Description de la hiérarchie dans un fichier externe (***)

Le but de cette extension est de permettre à la librairie de charger un fichier externe contenant la description des widgets associés à l'application. L'interface de programmation est définie dans le fichier `"ei_parse.h"`. La fonction `ei_parse(..)` interprète un fichier dont le nom est passé en paramètre et construit la hiérarchie de widgets correspondante. Chaque widget est associé à un nom unique représenté par une chaîne de caractère. Il est possible dans le programme d'accéder à l'adresse du widget à partir de son nom grâce à la fonction `ei_parse_widget_from_name`. C'est nécessaire par exemple pour associer des traitants aux widgets, car les traitants ne peuvent pas être définis dans le fichier externe. La fonction `free_name_to_widget_list` est appelée par le programme pour libérer la mémoire utilisée pour la correspondance entre noms de widget et adresse. Le programme de test `"parsing.c"` donne un exemple d'utilisation d'un fichier externe.

Le format du fichier de description est spécifié à l'aide d'une grammaire.

spécification lexicale

```
OB : '{'
CB : '}'

EQUAL : '='

END_LINE : '\n'

IGNORE_END_LINE : '\\n'

PLACE : 'place'
COMMENT : '#' ( ~('\n') )* {skip();}
NAME : ('a'..'z' | 'A'..'Z' | '_')('a'..'z' | 'A'..'Z' |'0'..'9' | '_')*
INTEGER : ('0'..'9')+
REAL : ( ('0' .. '9')+ '.' ('0' .. '9')* ) | ('.' ('0' .. '9')+ )

// Ignore spaces, tabs
WS  :    ' '  | '\t'
```

spécification syntaxique

```
// Top Level rule
ig  -> list_commands EOF

list_commands -> (command)*

command -> widget_command END_LINE
                | place_command END_LINE
                | END_LINE

widget_command -> widget_type widget_name parent_name list_option

widget_name -> NAME

widget_type -> NAME

parent_name : NAME

list_option -> (option)*

option -> NAME EQUAL option_value

option_value -> number
                      | NAME
                      | OB list_number CB
                      | OB list_name CB

number -> INTEGER
             | REAL

list_number -> (number)*

list_name -> (NAME)*

place_command -> PLACE widget_name list_option
```

Remarques :
— Attention : la grammaire n'est pas complètement LL(1) (cf. `option_value`). Vous devez donc d'abord la transformer en une grammaire LL(1) avant d'écrire le parser.
— La grammaire fournie n'est pas attribuée. Vous devez déterminer les traitements à effectuer pour chacune des règles.

Exemple

Le fichier suivant :

```
toplevel the_top root \
title = Test_External_Definition \
requested_size = {320 240} \
color = {200 200 200 255} \
border_width = 4 \
closable = 1 \
resizable = both \
min_size = {320 240}

place the_top rel_x=.5 rel_y=.5 anchor=center
```

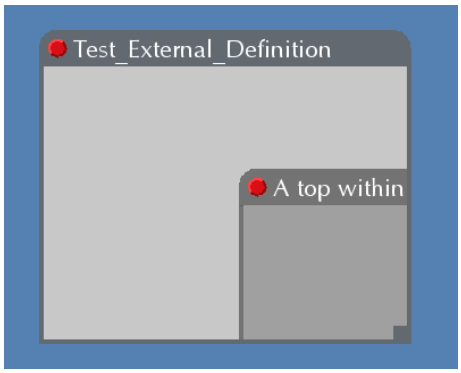


FIGURE 4.5 – Exemple de fichier de description externe d'une hiérarchie de widget : résultat de l'interprétation.

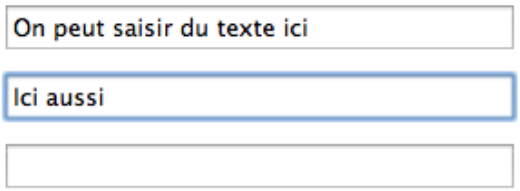


FIGURE 4.6 – Exemple de widgets de champ de saisie (`entry`). C'est le widget au centre qui a le *focus* clavier.

```
toplevel within the_top title = {A top within} requested_size = {160 120}

place within rel_x=.8 rel_y=.7 anchor=center
```

doit produire l'interface illustrée sur la Figure 4.5.

4.2.4 Gestion des tags des widgets (**)

La gestion des événements utilise le concept de *tag* pour limiter l'appel des traitants aux widgets qui possèdent un tag particulier (voir 3.6 "Gestion des événements"). Cette extension consiste à ajouter une fonction à la bibliothèque pour permettre au programmeur de définir des nouveaux tags et d'affecter lui-même les tags qu'il désire à un widget particulier.

Cette fonction permet de donner des comportements à des widgets de façon très simple : en leur donnant simplement un tag. Par exemple, le programmeur peut vouloir implémenter des infos bulle ("tooltips"). Une tooltip est une petite fenêtre d'aide qui s'affiche sur un widget (par exemple un bouton) lorsque l'utilisateur laisse le pointeur de la souris à l'arrêt sur le widget pendant 1 seconde. La fenêtre affiche un message d'aide qui explique le rôle du bouton dans l'application. Pour implémenter les tooltips, le programmeur crée les *bindings* sur le tag "tooltip" plutôt que sur un widget particulier. Ensuite, il n'a plus qu'à utiliser votre fonction de gestion des tags pour donner le tag "tooltip" a tout widget qui doit pouvoir afficher une tooltip. De la même manière, le programmeur pourra donner le tag "movable", par exemple, à tout widget qui peut être déplacer par un glisser-déposer.

Votre fonction peut aussi servir à *enlever* un tag à un widget. Par exemple, pour donner un comportement particulier à un bouton, on commence par lui enlever le tag qui correspond à sa classe (`button`). Le bouton perdra donc le comportement par défaut des boutons (comme s'enfoncer quand on clique dessus) puisque ce comportement est implémenté par des bindings sur le tag `button`.

4.2.5 Widget champ de saisie (***)

Cette extension consiste à ajouter une classe de widget "champ de saisie" (ou "entry", en anglais) permettant à l'utilisateur de saisir une ligne de texte (par exemple son nom, sa date de naissance, etc.) dans

FIGURE 4.7 – Exemple de gestion de géométrie en grille. Le programmeur spécifie les numéros de ligne et de colonne où positionner le widget, sur combien de colonnes il s'étale, s'il remplit toute sa cellule ou non, et dans ce dernier cas, comment il s'aligne dans sa cellule.

l'application. De même que pour les `radiobutton`, c'est à vous de spécifier l'interface de programmation de la classe `entry`. Trois widgets `entry` son illustrés sur la figure 4.6. L'ajout de la saisie de texte dans l'application nécessite la gestion du *focus clavier* : il peut y avoir à l'écran plusieurs fenêtres, chacune contenant plusieurs widgets `entry`. Mais quand l'utilisateur enfonce une touche de caractère sur le clavier, le caractère correspondant doit apparaître *uniquement dans une seule entry* : c'est l'`entry` qui a le *focus clavier*. L'utilisateur décide quel est l'`entry` qui a le focus clavier en cliquant dedans, ou en naviguant entre les différentes `entry` d'une fenêtre avec la touche "Tab".

4.2.6 Gestionnaire de géométrie en grille (***)

Cette extension consiste à ajouter un nouveau gestionnaire de géométrie qui place et dimensionne les widgets dans une grille, tel qu'illustré sur la figure 4.7. Là aussi, c'est à vous de définir l'interface de programmation du *griddeur*. Vous pourrez vous inspirer de l'interface de programme du *placeur*, c'est à dire de la fonction `ei_place`.

4.3 Évaluation

L'évaluation de votre projet se fera sur les fichiers du projet et lors d'une soutenance. La note du projet est la note de soutenance, elle intègre une évaluation des fichiers de votre projet. La chronologie est la suivante :

— 2 jours avant la fin du projet, dans la soirée, vous rendez les fichiers du projet sur TEIDE (voir ci-dessous),
— 1 jour avant la fin du projet, vous préparez vos soutenances (voir ci-dessous),
— le jour de la fin du projet, vous présentez votre projet en soutenance.

4.3.1 Critères d'évaluation

Par ordre d'importance :

— Exactitude : le projet fait ce qui est demandé.
— Qualité de la structure de votre code (modules, fonctions).
— Qualité de la forme du code (identificateurs, indentation, commentaires).
— Performance : les applications sont réactives, le processeur n'est pas chargé inutilement (voir 5.6 "Évaluation de performances").
— Extensions réalisées.

4.3.2 Rendu des fichiers de votre projet

L'archive de votre projet devra être déposée sur Teide. Elle contiendra :

— les sources et fichiers en-têtes commentés (y compris les en-têtes qui vous sont fournis mais que vous n'aurez *absolument pas modifiés*),
— les sources de vos applications de tests (y compris les applications fournies),
— un fichier `README.txt` décrivant en une ligne, pour chaque application, le ou les éléments testés,
— un fichier Makefile. Ce fichier Makefile doit permettre au minimum de nettoyer votre archive (`make clean`) et de régénérer la bibliothèque et l'ensemble des exécutables (`make all`).

4.3.3 Soutenance

Les soutenances durent 1/2h. Les notes des membres d'un même trinôme peuvent être différentes, si l'enseignant l'estime juste.

— Pendant les *dix* premières minutes, le trinôme expose brièvement un *bilan* du projet et présente une *démonstration* du fonctionnement du programme. Le bilan doit préciser :
  — l'état du programme vis-à-vis du cahier des charges : ce qui a été réalisé, ce qui n'a pas été réalisé, les bugs non corrigées, etc.
  — l'organisation du projet dans le trinôme (répartition des tâches, synchronisation, etc.),
  — les principaux choix de conception *intéressants* du programme : structures de données choisies, architecture du programme, etc.
  — les facilités/difficultés rencontrées, les bonnes et mauvaises idées.

  La démonstration illustre le fonctionnement du programme sur quelques exemples, afin de montrer son adéquation vis-à-vis des spécifications. Il est conseillé d'utiliser plusieurs exemples courts et pertinents pour illustrer les différents points de la spécification. La démonstration pourra contenir 1 ou 2 exemples plus longs pour montrer "le passage à l'échelle".
— Pendant les 20 minutes suivantes, l'enseignant teste vos programmes et vous interroge sur le projet. Les questions peuvent porter sur tous les aspects du projet, mais plus particulièrement sur des *détails* de votre implémentation, comment vous procéderiez pour terminer les *fonctionnalités manquantes*, et comment vous procéderiez pour ajouter une *nouvelle fonctionnalité*.

Vous aurez au minimum une demi-journée pour préparer votre soutenance entre la date de rendu des fichiers et votre créneau de soutenance. Préparez la soutenance sérieusement ! Il serait dommage d'avoir fait du bon travail sur le projet, mais perdre des points à cause d'une soutenance mal préparée. Il n'est pas demandé des *transparents*, mais répétez plusieurs fois la soutenance pour vous assurer :

— de faire une présentation de *10 minutes* (plus ou moins 1 minute),
— d'aborder tous les sujets demandés (voir ci-dessus),
— de répartir le temps de parole dans le trinôme,
— de vous exercer aux démonstrations.

Chapitre 5

Consignes et conseils

5.1 Organisation du libre-service encadré

Pendant tout le libre-service encadré, il faut consulter régulièrement la page d'EnsiWiki du projet [1]. En effet cette page contient les informations de dernière minute sur le déroulement et l'organisation du projet. En particulier, il est impératif de consulter régulièrement (au moins une fois par jour) la page des nouvelles du projet C – Interaction Graphique [2].

Pour réaliser le projet, les étudiants bénéficient d'un encadrement du travail en libre-service (voir la page EnsiWiki pour les détails des horaires). Pendant ces heures de libre-service encadré, des salles machines de l'Ensimag ont été réservées pour les étudiants participant au projet. De plus, les enseignants assurent une permanence pour aider les étudiants sur :

— la programmation en langage C.
— l'environnement de développement (make, autres outils gnu, etc.) et les programmes fournis.
— la conception du programme.
— l'organisation du projet.
— la compréhension générale des sujets donnés aux étudiants.

Les enseignants ne sont pas là pour corriger les bugs, pour programmer ou concevoir le programme à la place des étudiants. Si les enseignants l'estiment nécessaire, ils peuvent débloquer les groupes en difficulté. Les questions posées par les étudiants doivent être précises et réfléchies.

5.2 Documentation "Doxygen"

Les fichiers d'en-tête qui vous sont fournis contiennent des commentaires. Ces commentaires documentent, notamment, tous les types, fonctions et paramètres de la bibliothèque. Pour vous éviter d'avoir à lire cette documentation directement dans les fichiers d'en-tête, nous utilisons le système de documentation automatique *Doxygen* [3]. Ce système parcours les fichiers de code, récupère les commentaires, et en fait une documentation structurée sous forme HTML ou Latex (compilable en .pdf). Pour générer la documentation dans ces deux formats, exécutez la commande `make doc` à la racine du projet. Les documentations générées par doxygen sont placées dans les répertoire `docs/html` et `docs/latex`.

Pour consulter la documentation sous forme HTML, ouvrez le fichier `"docs/html/index.html"` dans un navigateur (tel que Firefox). La liste des fonctions de la bibliothèque est disponible dans le fichier `"docs/html/globals_func.html"`.

Pour consulter la documentation sous forme pdf, placez-vous dans le répertoire `docs/latex` et exécutez la commande `make`. La documentation est générée dans le fichier `"refman.pdf"`. **N'imprimez pas ce fichier !** Ce serait du gaspillage pour les quelques lignes que vous allez consulter, la consultation interactive par index et liens, sur la version HTML, est beaucoup plus pratique.

1. http://ensiwiki.ensimag.fr/index.php/Projet_C
2. https://ensiwiki.ensimag.fr/index.php/Nouvelles_du_projet_C_-_Interaction_Graphique
3. http://www.doxygen.org/

5.3 Cas de fraudes

Il est interdit de copier ou de s'inspirer, même partiellement, de fichiers concernant le projet C, en dehors des fichiers donnés explicitement par les enseignants et des fichiers écrits par des membres de son trinôme. Il est aussi interdit de communiquer des fichiers du projet C à d'autres étudiants que des membres de son trinôme. Les sanctions encourues par les étudiants pris en flagrant délit de fraude sont le zéro au projet (sans possibilité de rattrapage en deuxième session), plus les sanctions prévues dans le règlement de la scolarité en cas de fraude aux examens. Dans ce cadre, il est en particulier interdit :

- d'échanger (par mail, internet, etc.) des fichiers avec d'autres étudiants que les membres de son trinôme.
- de lire ou copier des fichiers du projet C dans des répertoires n'appartenant pas à un membre de son trinôme.
- de posséder dans son répertoire des fichiers de projets des années précédentes ou appartenant à d'autres trinômes.
- de laisser ses fichiers du projet C accessibles à d'autres étudiants que les membres du trinôme. Cela signifie en particulier que les répertoires contenant des fichiers du projet C doivent être des répertoires privés, avec autorisation en lecture, écriture ou exécution uniquement pour le propriétaire, et que les autres membres du trinôme ne peuvent y accéder que par ssh (échange de clef publique) ou un contrôle de droit avec les ACL (dans les deux cas, des documentations se trouvent sur la page d'EnsiWiki pour vous aider). Pendant la durée du projet, seuls les membres du trinôme doivent pouvoir accéder au compte.
- de récupérer du code sur Internet ou toute autre source (sur ce dernier point, contactez les responsables du projet si vous avez de bonnes raisons de vouloir une exception).

Les fichiers concernés par ces interdictions sont tous les fichiers écrits dans le cadre du projet : fichiers C, images, scripts de tests, etc.

Dans le cadre du projet C, la fraude est donc un gros risque pour une faible espérance de gain, car étant donné le mode d'évaluation du projet (voir section 4.3.3), la note que vous aurez dépend davantage de la compréhension du sujet et de la connaissance de l'implantation que vous manifestez plutôt que de la qualité "brute" de cette implantation. Notez également que des outils automatisés de détection de fraude seront utilisés dans le cadre de ce projet.

5.4 Styles de codage

Indépendamment de la correction des algorithmes, un code de bonne qualité est aussi un code facile et agréable à lire. Dans un texte en langue naturelle, le choix des mots justes, la longueur des phrases, l'organisation en chapitres et en paragraphes peuvent rendre la lecture fluide, ou bien au contraire très laborieuse.

Pour du code source, c'est la même chose : le choix des noms de variables, l'organisation du code en fonctions, et la disposition (indentation, longueur des lignes...) sont très importants pour rendre un code clair. La plupart des projets logiciels se fixent un certain nombre de règles à suivre pour écrire et présenter le code, et s'y tiennent rigoureusement. Ces règles (*Coding Style* en anglais) permettent non seulement de se forcer à écrire du code de bonne qualité, mais aussi d'écrire du code *homogène*. Par exemple, si on décide d'indenter le code avec des tabulations, on le décide une bonne fois pour toutes et on s'y tient, pour éviter d'écrire du code dans un style incohérent comme :

```
if (a == b) {
  printf("a == b\n");
} else
{
        printf ( "a et b sont différents\n");
}
```

Pour le projet C, les règles que nous vous imposons sont celles utilisées par le noyau Linux. Pour vous donner une idée du résultat, vous pouvez regarder un fichier source de noyau au hasard (a priori, sans comprendre le fond). Vous trouverez un lien vers le document complet sur EnsiWiki, lisez-le. Certains chapitres sont plus ou moins spécifiques au noyau Linux, vous pouvez donc vous contenter des Chapitres 1

à 9. Le chapitre 5 sur les `typedef` et le chapitre 7 sur les `goto` sont un peu complexes à assimiler *vraiment* et sujets à discussion. Vous pouvez ignorer ces deux chapitres pour le projet C.

Nous rappelons ici le document dans les grandes lignes :

— Règles de présentation du code (indentation à 8 caractères, pas de lignes de plus de 80 caractères, placements des espaces et des accolades, etc.)
— Règles et conseils pour le nommage des fonctions (trouver des noms courts et expressifs à la fois).
— Règles de découpage du code en fonctions : faire des fonctions courtes, qui font une chose et qui le font bien.
— Règles d'utilisation des commentaires : en bref, expliquez *pourquoi* votre code est comme il est, et non *comment*. Si le code a besoin de beaucoup de commentaires pour expliquer comment il fonctionne, c'est qu'il est trop complexe et qu'il devrait être simplifié.

Certaines de ces règles (en particulier l'indentation) peuvent être appliquées plus ou moins automatiquement. Le chapitre 9 du *Coding Style* du noyau Linux vous présente quelques outils pour vous épargner les tâches les plus ingrates : GNU Emacs et la commande `indent` (qui fait en fait un peu plus que ce que son nom semble suggérer). Pour le projet C, nous vous laissons le choix des outils, mais nous exigeons un code conforme à toutes ces directives.

5.5 Outils

Les outils pour développer et bien développer en langage C sont nombreux. Nous en présentons ici quelques-uns, mais vous en trouverez plus sur EnsiWiki (les liens appropriés sont sur la page du projet) et, bien sûr, un peu partout sur Internet !

— `Emacs`, `Vim`, `gedit` sont d'excellents éditeurs de texte, qui facilitent la vie du développeur en C (et pas seulement) : indentation automatique, coloration syntaxique, navigation de code...
— `gdb` le debugger ou son interface graphique `ddd` permettent de tracer l'exécution. Son utilisation est très intéressante, mais il ne faut pas espérer réussir à converger vers un programme correct par approximations successives à l'aide de cet outil, etc.
— `valgrind` sera votre compagnon tout au long de ce projet. Il vous permet de vérifier à l'exécution les accès mémoire faits par vos programmes. Ceci permet de détecter des erreurs qui seraient passées inaperçues autrement, ou bien d'avoir un diagnostic pour comprendre pourquoi un programme ne marche pas. Il peut également servir à identifier les fuites de mémoire (c'est-à-dire vérifier que les zones mémoires allouées sont bien désallouées). Pour l'utiliser : `valgrind [options] <executable> <paramètres de l'exécutable>`
Pour les fuites mémoires, vous constaterez que `libeibase` en contient 3 situées dans `SDL_Init`, `SDL_VideoInit`, et `XGetDefaults`. Nous n'avons pas le contrôle sur ces fonctions et ne pouvons donc pas supprimer ces erreurs. Vous pouvez utiliser l'option `--suppressions` de `valgrind` pour cacher ces erreurs de la ligne de commande.
— Pour travailler à plusieurs en même temps, des outils peuvent vous aider. Les *gestionnaires de versions* permettent d'avoir plusieurs personnes travaillant sur le même code en parallèle, et de fusionner automatiquement les changements. Vous pouvez au choix utiliser Subversion (alias SVN), relativement simple d'utilisation, ou des outils plus évolués comme Git. EnsiWiki vous en dira plus sur l'utilisation de ces outils à l'Ensimag.
— Finalement, pour tout problème avec les outils logiciels utilisés, ou avec certaines fonctions classiques du C, les outils indispensables restent l'option `--help` des programmes, le manuel (`man <commande>`), et en dernier recours, Google ![4]

5.6 Évaluation de performances

`gprof` est un outil de "profiling" du code, qui permet d'étudier les performances de chaque morceau de votre code. `gcov` permet de tester la couverture de votre code lors de vos tests. L'utilisation des deux programmes en parallèle permet d'optimiser de manière efficace votre code, en ne vous concentrant que sur les points qui apporteront une réelle amélioration à l'ensemble. Pour savoir comment les utiliser, lisez le manuel.

4. 7g de CO_2 par requête

Vous pouvez également utiliser la fonction `hw_now()` déclaré dans `"hw_interface.h"` pour mesurer un temps d'exécution avec précision. Par exemple, pour mesurer le temps d'exécution d'une fonction `ma_fonction()`, et étudier si vos optimisations sont efficaces, exécutez le code suivant :

```
int     i, nb_loop;
double  end, start;

nb_loop    = 1000;     /* precision de mesure améliorée par repetition */
start      = hw_now();
for (i=0; i < nb_loop; i++)
  ma_fonction();
end        = hw_now();
printf("Execution time for ma_fonction: %f s.", (end - start) / (double)nb_loop);
```

Pour mesurer une *fréquence* de fonctionnement, vous pouvez utilisez l'estimateur de fréquence `frequency_counter_t` déclaré dans `"freq_counter.h"`. Vous devez :

— déclarer une variable de type `frequency_counter_t` dont la durée de vie est suffisante (par exemple en variable globale),

— initialiser cette variable une seule fois en donnant son pointeur à `frequency_init(...)`,

— appeler `frequency_tick(...)` dans le code dont vous souhaitez estimer la fréquence d'appel. Cette fonction se charge d'afficher régulièrement un message sur la sortie standard qui donne l'estimation de fréquence.

Annexe A

Étapes de progression

Même si la première application `"frame.c"` est rudimentaire, elle repose sur de nombreuses fonctions de la bibliothèque : `ei_app_create`, `ei_frame_configure`, `ei_widget_create`, `ei_place`, `ei_app_run`, `ei_app_free`. Mais vous n'allez pas attendre d'avoir entièrement développé chacune de ces fonctions avant de compiler l'application `"frame.c"` et tester si votre code fonctionne. Afin de vous faciliter le travail et vous permettre d'obtenir plus rapidement une application produisant déjà de premiers résultats, nous vous proposons de suivre les étapes suivantes pour le développement des différentes fonctions. Attention ! Réaliser toutes ces étapes ne signifie pas que vous avez un projet complet. Veillez à la *généralité* de vos développements (i.e. l'étape A.7).

A.1 Affichage de la fenêtre racine (root)

Cette étape a pour but d'initialiser la bibliothèque et de créer la fenêtre système qui joue le rôle de widget racine ("root window").

— Écrivez une implémentation de la fonction `ei_app_create(...)` qui crée la fenêtre racine de votre application. Il faudra en particulier prendre en compte la possibilité de créer une fenêtre en mode plein écran.

— Écrivez toutes les autres fonctions de la bibliothèque nécessaires à la compilation de `"frame.c"` en leur donnant un corps vide. Seule `ei_app_run()` n'aura pas un corps vide, mais un corps contenant un simple appel à `getchar()`. Ainsi, le programme ne se terminera pas immédiatement mais attendra l'appui sur la touche "entrée" de la part de l'utilisateur.

À ce stade, l'application doit ouvrir une fenêtre vraisemblablement noire de taille `600x600`.

A.2 Création de la classe de widget "frame"

Tout widget appartient à une classe. La fenêtre racine est un widget particulier : ce n'est pas le programmeur qui crée ce widget, c'est la bibliothèque elle-même à l'initialisation de l'application. Par ailleurs, ce widget n'a pas besoin d'être géré par un gestionnaire de géométrie puisqu'il est en permanence affiché sur toute la surface disponible de l'application. Malgré ces différences, le widget racine se comporte comme un cadre dont on peut notamment changer la couleur de fond. Vous ferez en sorte qu'il soit de la classe de widget `frame`. Outre la fenêtre racine, le programme crée également un widget `frame` ayant un effet de relief (voir la figure 4.1, à gauche).

Il s'agit maintenant d'ajouter la classe de widget `frame` dans la bibliothèque. La création d'une classe de widget est détaillée en section 3.3. Par ailleurs, les applications interactives sont constituées d'une hiérarchie de widgets telle que présentée en section 2.2.2. Vous allez gérer cette hiérarchie pour que le cadre en relief soit descendant de la fenêtre racine, et vous parcourrez cette simple hiérarchie pour dessiner l'interface à l'écran.

— Définissez la classe de widget `frame`. Cette définition nécessite en particulier la création d'une structure de type `ei_widgetclass_t` dans laquelle les champs `allocfunc`, `releasefunc`, `drawfunc`, et `setdefaultsfunc` devront pointer sur des fonctions que vous implémenterez.

— Complétez la partie initialisation de la fonction `ei_app_create(...)` afin d'enregistrer votre nouvelle classe `frame` dans votre bibliothèque, c'est à dire d'appeler la fonction `ei_widgetclass_register(...)`.
— Modifiez la fonction `ei_app_run()`. Cette fonction doit maintenant parcourir la hiérarchie de widgets et appeler leur `drawfunc` pour dessiner toute l'interface de l'application.
— Afin de pouvoir contrôler l'apparence de la fenêtre racine, proposez une implémentation de `ei_frame_configure(...)` permettant au minimum pour cette étape de gérer la couleur de fond.

A ce stade, l'application doit ouvrir une fenêtre bleue de taille `600x600`. En théorie, le cadre en relief n'est pas encore affiché car l'appel au gestionnaire de géométrie `ei_place` est vide. Cependant, votre programme pourra ignorer le problème de gestion de géométrie et choisir de placer le cadre en relief à l'écran.

A.3 Mise en place d'un gestionnaire de géométrie

Il s'agit maintenant de faire une implémentation de la fonction `ei_place`. Le "placeur" est le seul gestionnaire de géométrie requis pour le projet, mais vous pourrez en proposer d'autres en extension (cf. section 4.2.6). Le "placeur" permet de spécifier la position et la taille d'un widget de façon absolue ou relative (par rapport à son parent). Dans l'application `"frame.c"`, nous n'utilisons que du placement absolu.

— De la même façon que pour la création de la classe de widget `frame`, créez et enregistrez le gestionnaire de géométrie "placeur" dans la bibliothèque. Il faut donc créer et initialiser une structure `ei_geometrymanager_t`, et l'enregistrer dans la bibliothèque avec `ei_geometrymanager_register(...)`.
— Implémenter la fonction `ei_place()`. Pour le moment, vous pouvez prendre en compte uniquement le positionnement absolu.

Le cadre en relief doit maintenant être géré par le gestionnaire de géométrie "placeur", il doit donc être dessiné lors du passage dans la fonction `ei_app_run(...)`.

A.4 Bilan

A ce stade, la première application `"frame.c"` doit maintenant fonctionner. Une première implémentation de l'ensemble des fonctions listées en début de cette annexe a été réalisée. Cette implémentation reste bien entendu *partielle* et doit être complétée pour fournir l'ensemble des services demandés.

A.5 Dessin de boutons en relief

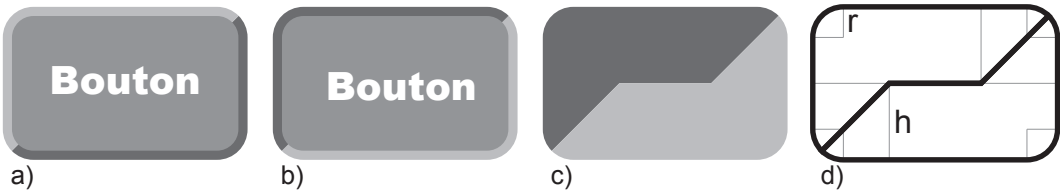


FIGURE A.1 – Dessin des boutons. a) effet "relevé", b) effet "enfoncé", c) les deux formes du fond, d) schéma : r est le rayon des arrondis, h est la moitié de la hauteur ou de la largeur du bouton (le plus petit des deux).

Un interacteur de type `button` doit apparaître comme un cadre ayant les angles arrondis, et en relief "relevé" ou "enfoncé" (figure A.1a,b). Pour créer l'effet de relief, on dessine d'abord deux moitiés de forme, l'une plus sombre et l'une plus claire que la couleur de bouton demandée (figure A.1c). Puis on dessine par dessus le fond lui-même, puis le contenu (texte ou image) du bouton. Pour un effet "enfoncé", les couleurs des 2 moitiés sont échangées et le contenu du bouton est décalé un petit peu en bas à droite. Le dessin des formes se fait par appel de `ei_draw_polygon(...)` en passant la liste des points qui définissent la forme.

— Écrivez une fonction appelée par exemple "arc" qui génère une liste de points définissant un arc, paramétrée par le centre, le rayon, et les angles de début et fin de l'arc. Utilisez `ei_draw_polygon(...)` pour voir le résultat.
— Écrivez une fonction appelée par exemple "rounded_frame" qui renvoie une liste de points définissant un cadre aux bords arrondis. Cette fonction prendra en paramètre un rectangle (`ei_rect_t`) et le rayon des arrondis. Aidez-vous du schéma de la figure A.1d sans prendre en compte le paramètre *h* et les traits intérieurs.
— Paramétrez votre fonction "rounded_frame" pour qu'elle génère uniquement la partie haute, ou basse, ou bien la totalité de la forme.
— Écrivez une fonction appelée par exemple "draw_button" qui dessine un bouton en relief.

A.6 Mise en place d'un gestionnaire d'événements dans l'application button.c

L'application `"button.c"` est très similaire à `"frame.c"`. Les principales différences sont :
— l'utilisation d'une nouvelle classe de widgets `button`,
— la gestion des événements.

La prise en compte de la nouvelle classe de widgets se fera de la même façon que pour la classe `frame`. Pour ce qui est de la gestion des événements, il y a deux aspects principaux : l'association d'une fonction traitant (callback) à un type d'événements via la fonction `ei_bind(...)` ; et la mise en place de la boucle de traitement des événements, dans `ei_app_run()`, qui permet de récupérer les événements système et d'appeler les traitants concernées.

— Implémentez la fonction `ei_bind(...)` permettant de mettre en place l'association événement ↔ callback, telle que décrite en section 3.6.
— Modifiez la fonction `ei_app_run()`. La boucle doit se terminer s'il y a eu un appel `ei_app_quit_request()`. Le corps de la boucle doit inclure une mise en attente d'un événement utilisateur (`hw_event_wait_next()`), et l'analyse de cet événement pour rechercher le widget concerné si c'est un événement situé (voir 2.1.1). L'analyse des événements aboutie à l'appel éventuel des traitants concernés. Le principe de la boucle principale est donné en section 3.8.2.
— Complétez la définition de la classe `button` afin d'ajouter le traitement des événements *mouse button down* et *mouse button up*. Le bouton doit avoir une apparence "enfoncée" après un événement *mouse button down* et tant que le pointeur de la souris et au dessus du bouton. Il doit y avoir un retour à l'apparence relâchée si le pointeur sort des limites du bouton et après *mouse button up*.

A.7 Généralité

Gardez en tête que votre bibliothèque doit avoir le comportement attendu, même dans certains cas d'utilisation qui ne sont pas décrits dans ce documents, et qui ne sont pas présent dans le "Code d'applications fournies" (section 4.1). Par exemple, est-ce que votre bibliothèque est capable de gérer correctement le cas illustré sur la figure 4.5 (même s'il n'est pas généré par un fichier de description externe, mais par des appels de fonctions) ? Autre exemple : l'utilisateur déplace une fenêtre toplevel avec la souris. Il déclenche alors le raccourci clavier <ctrl>-W qui détruit la fenêtre alors qu'il a toujours le bouton de la souris enfoncé pendant le déplacement. Que se passe-t-il avec votre bibliothèque ?

Les deux exemples ci-dessus peuvent paraître difficiles, mais ils ne devraient nécessiter aucun développement supplémentaires de votre part si vous avez programmé votre bibliothèque dans un esprit de "généralité" : ne réalisez pas un module simplement pour atteindre un objectif particulier. Demandez-vous plutôt quel est le rôle de ce module, et imaginez et *testez* toutes ses utilisations possibles.

Lors de la soutenance de votre projet, nous testerons votre bibliothèque sur des cas qui ne sont pas décrits dans ce document, justement pour évaluer sa généralité.

Index

"ei_event.h", 23
"ei_geometrymanager.h", 22

all, *voir* gestionnaire d'événements (étiquette)
API, 17, 29
application, 6, 13
attributs, 18

bibliothèque, *voir* bibliothèque logicielle, 6
bibliothèque logicielle, 6
boucle principale, 7, 13, 27
bouton, 10, 20, 21, 43
    dessin, 42
bouton radio, 31
button, *voir* bouton
"button.c", 30

cadre, 20, 21, 41
callback, *voir* traitant
champ de saisie, 34
classe, *voir* interacteur (classe d'interacteur)
clipping, 9, 10
code de touche clavier, 24
critères d'évaluation, 35

déplacement, 25
descendant, *voir* interacteur (hiérarchie)
doxygen, 15, 37
drawfunc, *voir* ei_widget_class_t (drawfunc)

ei_app_create, 27, 41
ei_app_invalidate_rect, 26
ei_app_quit_request, 27, 43
ei_app_redraw, 27
ei_app_run, 27, 41, 43
ei_axis_set_t, 20
ei_bind, 12, 23, 25, 27, 43
ei_button_configure, 17, 20, 21, 23
ei_color_t, 16
ei_copy_surface, 17
"ei_draw.h", 15, 17, 21
ei_draw_polygone, 17
ei_draw_polyline, 17
ei_draw_text, 17
ei_ev_mouse_buttondown, 25
ei_ev_mouse_buttonup, 25
ei_ev_mouse_move, 25
ei_event_t, 24
ei_eventtype_t, 23
    ei_ev_keydown, 23
    ei_ev_keyup, 23
    ei_ev_mouse_buttondown, 23
    ei_ev_mouse_buttonup, 23
    ei_ev_mouse_move, 23
ei_fill, 17
ei_font_t, 21
ei_frame_configure, 20, 21, 42
ei_geometry_param_t, 18, 20
ei_geometrymanager_register, 20, 27, 42
ei_geometrymanager_runfunc_t, 22
ei_geometrymanager_t, 20, 42
    runfunc, 22
ei_map_rgba, 16
ei_place, 22, 42
    anchor, 23
ei_relief_t, 21
ei_surface_t, 15
ei_toplevel_configure, 20
ei_unbind, 12, 23
"ei_widget.h", 20
ei_widget_create, 20, 27
ei_widget_t, 18
    children_head, 21
    children_tail, 21
    geom_params, 22
    next_sibling, 21
    parent, 21
    wclass, 19
ei_widgetclass_allocfunc_t, 20
ei_widgetclass_drawfunc_t, 19
ei_widgetclass_register, 19, 27, 41
ei_widgetclass_t, 19, 41
    allocfunc, 20, 41
    drawfunc, 26, 41
    releasefunc, 41
    setdefaultsfunc, 20, 41
ensiwiki, 37
évaluation de performances, 39
évaluation du projet, 35
événement, *voir* gestionnaire d'événements

fenêtre système, 8, 13, 41
fenêtre système, 15
fichiers, 35
focus clavier, 34
fonte, 21
frame, *voir* cadre

"frame.c", 29, 41
"freq_counter.h", 40
fréquence d'appel, 40
frequency_counter_t, 40
frequency_init, 40
frequency_tick, 40

gestion de l'application, 8, 13
gestionnaire d'événements, 6, 8, 11, 12, 23, 43
    étiquette, 23, 25, 34
    clavier, 24
    événement non situé, 7
    événement situé, 7
    lien, 12, 25, 43
    picking, 11, 24
    type d'événements, 23
gestionnaire de géométrie, 6, 8, 12, 17, 22, 30, 42
    absolu, 13, 23
    en grille, 12, 35
    placeur, 13, *voir* placeur, 22
    relatif, 13, 23

handler, *voir* traitant
"hello_world.c", 30
hw_create_window, 15
hw_event_wait_next, 27, 43
hw_image_load, 15
hw_init, 15, 27
"hw_interface.h", 15, 21
hw_interface_update_rects, 15
hw_now, 39
hw_surface_create, 15, 27
hw_surface_free, 15
hw_surface_get_buffer, 16
hw_surface_get_channel_indices, 16
hw_surface_lock, 15, 16
hw_surface_unlock, 15
hw_surface_update_rects, 26
hw_text_create_surface, 15

image, 15, 21
interacteur, 6, 8, 9
    classe d'interacteur, 10, 17, 20, 41
    clipping, 10
    hiérarchie, 9, 21
    identifiant, 11
    ordre, 9, 10, 21
    racine, 9, 13, 27, 41
interface système et matériel, 8
IUG, 5

"libei.a", 29
"libeibase.a", 9, 15

"minesweeper.c", 30
"minimal.c", 29
modules, 8

offscreen de picking, 11, 15, 24, 27
ordre, *voir* interacteur (ordre)

paramètres d'événement, 24
parent, *voir* interacteur (hiérarchie)
"parsing.c", 32
picking, *voir* gestionnaire d'événement (picking)
pixel, 10, 16
    alpha, 16
    représentation en mémoire, 16
    RGB, 16
    RGBA, 16
    RGBX, 16
    transparence, 16
placeur, 22, 27
    algorithme, 22
polymorphisme, 17
primitives graphiques, 10, 17
    lignes brisées, 17
    polygones, 17
programmation événementielle, 7
programme principal, 8, 13, 27
"puzzle.c", 31

racine, *voir* interacteur (racine)
redimensionnement, 26
relief, 21
remplissage, 17
root, *voir* interacteur (racine)
runfunc, *voir* ei_geometrymanager_t runfunc

SDL, 29
soutenance, 36
surface de dessin, 9, 11, 15
    adresse en mémoire, 16
    allocation, 15
    blocage, 15, 26
    copie, 17
    cycle de mise à jour, 16
    déblocage, 15, 26
    fenêtre système, 15
    libération, 15
    mise à jour, 15, 26

table des pointeurs, 19
tag, *voir* gestionnaire d'événements (étiquette)
tests, 6, 29
texte, 15, 21
toplevel, 10, 20, 21, 25, 30
touches spéciales, 24
traitant, 7, 11, 21, 23–25, 30
    externe, 11
    interne, 11
traitant par défaut, 7
transparence, 16
"two048.c", 31

valeurs par défaut, 17

widget, *voir* interacteur, *voir* interacteur