

Ensimag — Printemps 2021

—
Projet Logiciel en C
—

Sujet : Interfaces Utilisateur Graphiques

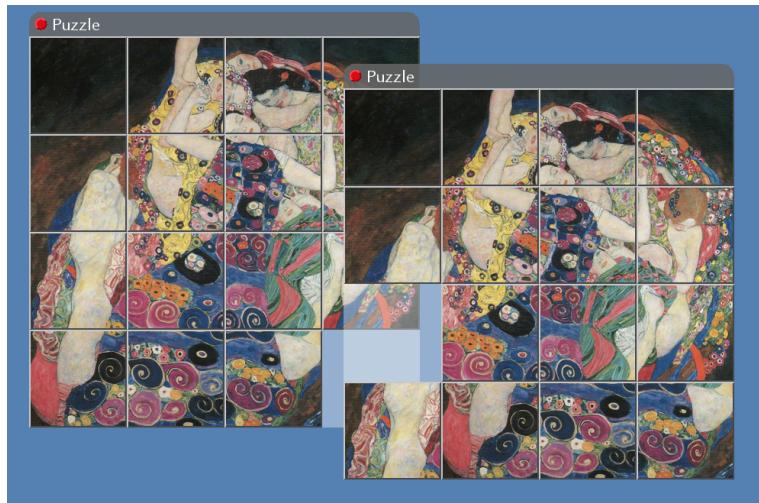


Table des matières

1	Introduction	5
1.1	Objectifs	5
1.1.1	Langage C, Projet	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	Primitives Graphiques	9
2.2.3	Interacteurs	10
2.2.4	Gestionnaire d'événements	11
2.2.5	Gestionnaire de géométrie	12
2.2.6	Gestion de l'application	12
2.2.7	Programme principal	12
3	Approfondissements	13
3.1	Services de dessin	13
3.1.1	Niveau pixel	13
3.1.2	Niveau primitives graphiques	15
3.1.3	Clipping	17
3.2	Valeurs par défaut	21
3.3	Polymorphisme	21
3.3.1	Polymorphisme des données	21
3.3.2	Polymorphisme des fonctions	22
3.4	Classes et hiérarchie de widgets	23
3.4.1	Classes de widgets	23
3.4.2	Description des classes de widget demandées	24
3.4.3	Hiérarchie de widgets	25
3.5	Gestion de la géométrie : le placeur	25
3.5.1	Mécanisme général de gestion de géométrie	25
3.5.2	Algorithme du <i>placeur</i>	26
3.6	Gestion des événements	27
3.6.1	Principes	27
3.6.2	Interacteur actif	27
3.6.3	Traitants externes et internes	27
3.6.4	Exemple du déplacement	28
3.6.5	Exemple du redimensionnement	29
3.7	Gestion de l'affichage	29
3.8	Programme principal et boucle principale	30

3.8.1 Initialisation de l'application	30
3.8.2 Boucle principale	30
4 Travail à réaliser	33
4.1 Compilation	33
4.2 Code d'applications fournies	33
4.2.1 Minimal	34
4.2.2 Lignes	34
4.2.3 Cadre (frame)	34
4.2.4 Bouton simple (button)	34
4.2.5 Fenêtre hello world	34
4.2.6 Puzzle et 2048	35
4.3 Extensions	35
4.3.1 Gestion optimisée du clipping de ligne (***)	35
4.3.2 Gestion optimisée du clipping de polygone (****)	35
4.3.3 Widget bouton radio (**)	35
4.3.4 Widget champ de saisie (***).	36
4.4 Évaluation	36
4.4.1 Critères d'évaluation	36
4.4.2 Rendu des fichiers de votre projet	37
4.4.3 Soutenance	37
5 Consignes et conseils	39
5.1 Organisation du libre-service encadré	39
5.2 Documentation “Doxygen”	39
5.3 Cas de fraudes	39
5.4 Styles de codage	40
5.5 Outils	40
5.6 Évaluation de performances	41
A Étapes de progression	43
A.1 Dessin des primitives graphiques	43
A.2 Dessin de boutons en relief	43
A.3 Affichage de la fenêtre racine (root)	44
A.4 Création de la classe de widget “frame”	44
A.5 Mise en place d'un gestionnaire de géométrie	45
A.6 Bilan	45
A.7 Mise en place d'un gestionnaire d'événements dans l'application button.c	45
A.8 Généralité	45
Index	46

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 Langage C, Projet

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

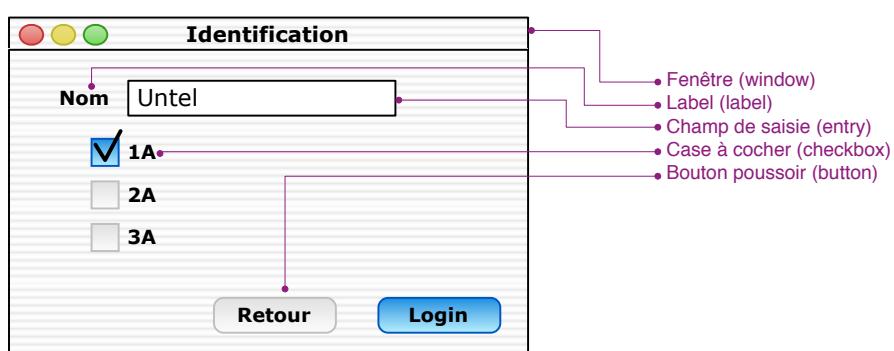


FIGURE 1.1 – Une fenêtre d'une interface utilisateur graphique.

Vous réalisez une bibliothèque logicielle qui facilite la programmation des Interfaces Utilisateur Graphiques (IUG). En utilisant cette bibliothèque, un programmeur pourra facilement créer une interface gra-

phique composée de fenêtres et d'interacteurs tels que boutons, champs de saisie, etc¹. 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,
- 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 dessin de primitives graphiques (dessin de lignes, de polygones),
- 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 utilisateurs (exécution des fonctions en réaction aux actions de l'utilisateur sur la souris et le clavier).

Le but du projet étant d'écrire du code en 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 réalisez offre les services suivants :

- **Dessin des primitives graphiques.** Vous réalisez une fonction de dessin de lignes brisées et une fonction de dessin de polygones *pleins* quelconques. Ces deux fonctions doivent être fortement optimisées car elles sont coûteuses en temps d'exécution, mais sont appelées de nombreuses fois par l'application.
- **Création et configuration des interacteurs.** Différentes fonctions permettent de créer différents interacteurs (fenêtres, boutons, labels) et de définir leurs attributs (texte ou image d'un bouton, par exemple).
- **Gestion de la géométrie.** La taille et la position des interacteurs dans leur fenêtre est calculée par un gestionnaire de géométrie. Cela permet de libérer le programmeur des calculs de géométrie quand un bouton s'agrandit, par exemple, suite à un changement de son label, ou à un redimensionnement de sa fenêtre.
- **Gestion des événements.** Votre bibliothèque doit faire en sorte, par exemple, qu'une fenêtre soit déplacée lorsque l'utilisateur clique sur son titre et déplace la souris. Clic et déplacement de la souris génèrent des "événements" que vous recevez et que vous devez traiter. De plus, le programmeur qui utilise votre bibliothèque peut lier une fonction "sauvegarde_document" à un bouton "Save". La bibliothèque est chargée d'appeler cette fonction quand l'utilisateur clique sur ce bouton.

1.2.2 Applications

Afin de tester votre bibliothèque, nous vous fournissons le code source de différents programmes (dessins de formes, affichage de bouton, de fenêtre, jeu de puzzle et jeu 2048). Ces programmes sont présentés dans la section 4.2.

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 sont décrits. Des explications plus détaillées pour chaque module seront présentées au chapitre suivant.

2.1 Programmation événementielle

2.1.1 Principe

Les applications interactives utilisent un modèle de programmation *évenementielle* alors que vous avez jusqu'ici programmé sur un modèle *séquentiel*. En séquentiel, l'exécution est linéaire avec des branchements prévus dans le programme. En événementielle, l'ordre d'exécution des différentes fonctions n'est pas connu au moment d'écrire le programme. Le programme principal se met simplement en attente d'*événements*, et *traite* ces événements au fur et à mesure qu'ils apparaissent.

Plutôt que d'avoir une seule fonction de traitement de tous types d'*événements*, la bibliothèque est chargée de *router* les événements vers des fonctions spécialisées de traitement d'*événement*, que vous appellerez *traitant* ("handler" ou "callback" en anglais). Par exemple, quand le programme reçoit un événement de type "l'utilisateur appui sur la touche C du clavier", la bibliothèque exécute le traitant des événements clavier.

La position du pointeur de la souris permet de router les *événements situés* : un événement de type "clic sur le bouton de la souris" intéresse tous les boutons graphiques. Mais quand l'utilisateur crée une *instance* de ce type d'*événement* en cliquant sur le bouton de la souris, cela ne concerne qu'un seul bouton graphique : celui situé sous le pointeur au moment du clic. C'est à la bibliothèque d'identifier cet interacteur et d'appeler son traitant et lui seul pour cet événement particulier. Nous détaillerons plus loin une technique générale pour identifier l'objet graphique situé sous le pointeur de souris, ce qui permet de réaliser le routage des événements situés.

2.1.2 Structure d'un programme événementiel

Avec la programmation par événement, le programme principal se résume à une phase d'initialisation et une boucle principale. À l'initialisation, les traitants sont enregistrés sur les types d'*événements* qui les intéresse. Dans la boucle principale, le programme s'endort en attente d'un événement, se réveille lorsqu'un événement survient, et route cet événement en appelant les traitants intéressés par les événements de ce type.

Dans le cas spécifique d'une interface utilisateur graphique, l'initialisation inclue aussi la création des interacteurs et leur placement à l'écran. Dans la boucle principale, le routage inclue l'identification de l'interacteur concerné dans le cas d'*événement situé*. Par ailleurs, la boucle principale inclue une étape de mise à jour de l'écran pour prendre en compte les modifications demandées par les traitants (déplacement de fenêtre, enfacement de bouton, etc.)

Le programme principal peut donc être résumé par le pseudo-code suivant :

- créer les interacteurs en définissant leurs attributs
- placer à l'écran les interacteurs grâce au gestionnaire de géométrie

enregistrer les traitants
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(s) traitant(s) associé(s)
 appeler le(s) traitant(s) associé(s)
fin tant que
finaliser le programme : libérer toutes les ressources utilisées.

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.

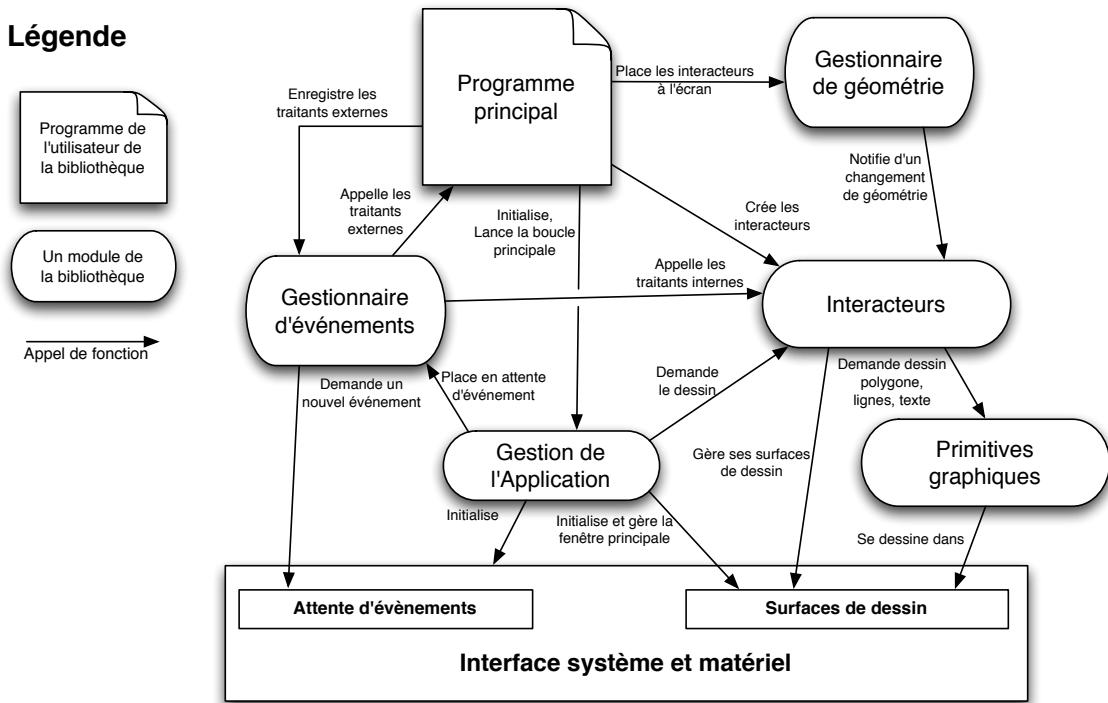


FIGURE 2.1 – Principaux modules et leurs interactions entre eux, avec l'interface matérielle, et avec le programme de l'utilisateur.

2.2.1 Interface système et matériel

Le module d'interface matériel, représenté par le cadre du bas dans la figure 2.1, vous donne accès à l'écran graphique et aux événements au travers. Dans ce projet, l'ensemble de l'application et des interacteurs sera géré dans une unique fenêtre du système d'exploitation. 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 :

- gestion des surfaces de dessin. Ce sont des zones mémoires où l'application “dessine” les interacteurs. La fenêtre système est une surface particulière affichée à l'écran. Les autres surfaces ne sont pas visibles, elles sont utilisées pour préparer les dessins avant de les copier sur la surface de la fenêtre système.
- attente des événements utilisateur (appui d'une touche, clic souris, etc.),
- création d'une surface contenant du texte,

— création d'une surface contenant une image chargée depuis un fichier (.png, jpg).

L'ensemble de ces fonctionnalités vous est fourni dans le cadre de ce projet sous la forme d'une bibliothèque ("libeibase.a" dont les fonctions sont déclarées dans le fichier "hw_interface.h") et ne fait donc *pas partie de ce que vous devez implémenter*.

2.2.2 Primitives Graphiques

Définitions

Une interface graphique doit pouvoir dessiner les interacteurs à l'écran et en modifier l'apparence en fonction des actions de l'utilisateur, comme donner l'aspect "enfoncé" à un bouton sur lequel on vient de cliquer. 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. La bibliothèque fournit donc des fonctions pour les tâches fréquentes de dessin : dessin de lignes, remplissage de polygones, et dessin de lettres formant un texte. À partir de ces trois primitives, il est relativement facile 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, c'est à dire redessiner ce qui était "en dessous", et redessiner la fenêtre au nouvel emplacement. Ceci est répété à chaque micro-déplacement de la souris, 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 centaines de milliers de pixels. Il en résulte que les algorithmes des primitives graphiques doivent être *extrêmement optimisées* sous peine d'avoir des ralentissements visibles dans les manipulations. Nous détaillons en section 3.1.2 les algorithmes classiques pour le tracé optimisé de lignes et de polygones pleins.

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.).

Les seules fonctions qui vous sont fournies dans ce projet sont l'allocation de mémoire pour le dessin (voir la section précédente) et le dessin de texte. C'est à vous de réaliser tous les autres services de dessin. Pour dessiner un interacteur, on appelle 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, leurs dessins viennent écraser en mémoire celui du parent, et donne l'impression qu'ils sont devant lui (par exemple, les boutons de la figure 2.3, à gauche).

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 (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 aux fonctions de dessin de tous les descendants qui 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). Nous détaillons en section 3.1.3 les différentes approches pour réaliser le clipping.

2.2.3 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 sont organisés dans une hiérarchie : 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.

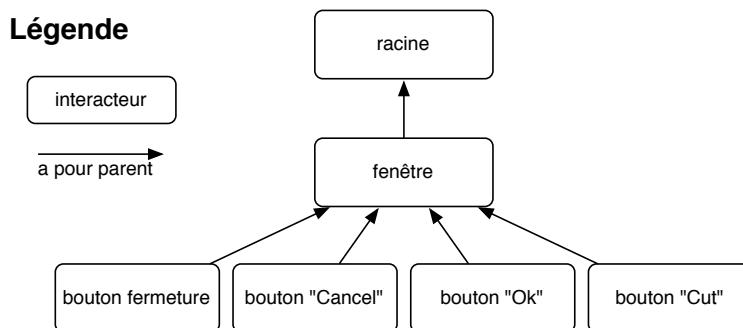


FIGURE 2.2 – Représentation de la hiérarchie de widgets de l’interface représentée à gauche sur la figure 2.3.

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.

Classes d’interacteur

Une interface utilisateur est composée de différents types d’interacteurs. Dans le projet, vous travaillez d’abord avec des boutons, des labels (champs de texte non éditable) et des fenêtres (ou “toplevel”). Tous les interacteurs partagent certaines caractéristiques et fonctionnalités : ils ont besoin par exemple de fonctions permettant d’allouer et de libérer leur espace mémoire, d’une fonction de configuration permettant au programmeur de modifier leur état, et d’une fonction qui se charge de les dessiner à l’écran.

Cependant, la réalisation de ces différentes fonctions dépend du type d’interacteur : 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.

Les bibliothèques de programmation d’interfaces graphiques organisent généralement les interacteurs en *classes* et s’appuient sur la programmation orienté objet. Les classes d’objet facilitent la programmation

des parties communes et de parties spécifiques des interacteurs. Le détail de l’implémentation des classes et des tables de fonction est donné au chapitre suivant en section 3.3.

2.2.4 Gestionnaire d’événements

Traitants internes et externes

Les traitants sont les fonctions associées à des type d’événement, tel que présenté en section 2.1. 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.

Chaque classe d’interacteur possède un pointeur vers une fonction de *traitant interne* qui réalise le comportement standard des interacteurs de cette classe. Par contre, 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.

Aiguillage d’événement par picking

Dans le cas des *événement situés*, le gestionnaire d’événement doit être capable d’identifier quel interacteur est positionné à la position (x, y) du pointeur sur l’écran. On appelle ça le “picking”. Il se peut également qu’il n’y ait pas d’interacteur à cet endroit là, hormis la “racine” définie à la section 2.2.3.

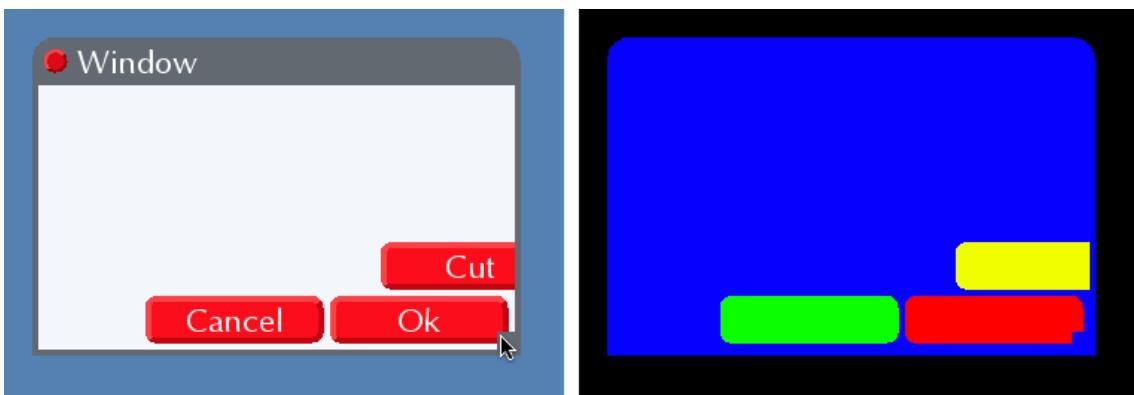


FIGURE 2.3 – Une interface affichée à l’écran (à gauche) et l’offscreen de picking correspondant (à droite). Dans l’offscreen, le fond (racine), la fenêtre toplevel et les boutons “Cancel”, “Ok” et “Cut” sont représentés, respectivement, en noir, bleu, vert, rouge et jaune. Le pointeur de souris, représenté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

Pour réaliser simplement le picking quelle que soit la forme des interacteurs, on utilise une technique dite de dessin hors-écran ou “offscreen”. Elle consiste à dessiner l’interacteur dans une surface dédiée appelée l’offscreen de picking. L’offscreen n’est jamais affiché à l’écran : il ne sert qu’au picking. Au lieu d’utiliser différentes couleurs pour dessiner 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.

2.2.5 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 "Ok" aux coordonnées (300, 200) et avec une taille de 80×30 pixels". Mais si la fenêtre qui contient ce bouton est redimensionnée, ces coordonnées ne sont sans doute plus correctes. 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 qui le contient.

C'est le rôle d'un *gestionnaire de géométrie* d'enregistrer des *contraintes* de géométrie du programmeur telles que "place le bouton "Ok" à l'angle en bas à droite de son parent, avec une marge de 10 pixels avec les bords du parent". Le gestionnaire doit ensuite traduire ces contraintes en position absolue à chaque fois que c'est nécessaire, comme par exemple quand le parent est redimensionné ou quand un widget est détruit et que la place qu'il libère doit être redistribuée aux autres widgets. Dans ce projet, vous implémentez un gestionnaire de géométrie appelé "placeur" (*placer* en anglais). C'est un gestionnaire simple qui permet de placer un interacteur en exprimant position et taille de façon *absolue* et/ou *relative au parent*. Davantage de détails sur ses spécifications seront donnés au paragraphe 3.5.

2.2.6 Gestion de l'application

Le module de gestion de l'application permet 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. 2.2.3). Le module initialise également les structures propres aux classes d'interacteurs.

Le module offre également :

- un point d'accès pour l'interacteur racine 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".

2.2.7 Programme principal

Votre bibliothèque ne contient pas de programme principal : c'est le programmeur d'application graphique qui écrit un programme principal pour son application. Dans ce projet, nous vous donnons plusieurs exemples de programmes principaux qui ne pourront compiler et s'exécuter qu'avec votre implémentation de la bibliothèque. Ces programmes sont présentés en section 4.2. Vous devrez en écrire d'autres pour tester tout ou une partie de vos modules.

Un programme principal appelle les fonctions du module de gestion de l'application pour initialiser, exécuter, puis terminer l'application interactive. 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 au paragraphe “Primitives Graphiques” de la section 2.2.2, c'est à vous de réaliser les fonctions qui réalisent les dessins de base, hormis le dessin de texte qui vous est fourni.

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_`”. La réalisation de ces fonctions vous est fournie dans la bibliothèque “`libeibase.a`” : vous n'avez pas à les réaliser. Les autres fonctions des services de dessins sont déclarées dans le fichier “`ei_draw.h`”, c'est à vous de les réaliser (à part `ei_draw_text(...)`).

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 2.2.4), ou bien à préparer un dessin qui sera ensuite copié sur une surface affichée à l'écran (voir le paragraphe “Clipping” en 3.1.3). 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 modifier directement les pixels. Il faut au préalable avoir un accès exclusif à ces pixels 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 additive¹. 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 Mac OS 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 4.2.6) 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.

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

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

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 :

$$S_R = (P_A * P_R + (255 - P_A) * S_R) / 255 \quad (3.1)$$

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. C'est à vous de la réaliser en utilisant la fonction `hw_text_create_surface` qui crée une nouvelle surface de dessin contenant le texte à afficher. Vous copiez ensuite 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.

Comme expliqué en 2.2.2, les algorithmes de dessin doivent être extrêmement optimisés car ils sont exécutés très fréquemment et que tout délai d'exécution sera perceptible par l'utilisateur de l'application. Nous présentons ci-dessous les algorithmes classiques pour le dessin optimisé des lignes et des polygones pleins.

Dessin de lignes

La fonction `ei_draw_polyline(...)` permet de dessiner une *ligne brisée* entre les points passés en paramètre. La réalisation de cette fonction nécessite de dessiner chaque segment de la ligne brisée. La figure 3.1 illustre le problème : quels pixels dessiner pour un segment mathématique allant du pixel (1, 0) au pixel (6, 2) ? On applique la stratégie suivante :

- On ne dessine qu'un seul pixel par colonne lorsque $|\Delta_x| > |\Delta_y|$ (ce type de segment est dit "dirigé par x "), un seul pixel par ligne lorsque $|\Delta_x| < |\Delta_y|$ (segment "dirigé par y ").
- On dessine le pixel le plus proche de l'intersection entre le segment mathématique et la colonne (segment dirigé par x) ou la ligne (segment dirigé par y).
- Pour une intersection exactement entre deux pixels, on dessine le pixel de coordonnée inférieure.

Le second problème à résoudre est la performance : le dessin de lignes est amené à être exécuté de nombreuses fois comme expliqué en section 2.2.2. Il faut optimiser sa réalisation, et en particulier les calculs réalisés dans la *boucle interne* de l'algorithme : ceux qui seront répétés pour chaque pixels. Plutôt que de recalculer l'intersection à chaque pixel, on utilise une approche itérative : quand x progresse de 1 (pixel), y devrait progresser de Δ_y/Δ_x . Mais comme les pixels sont dessinés uniquement sur des coordonnées

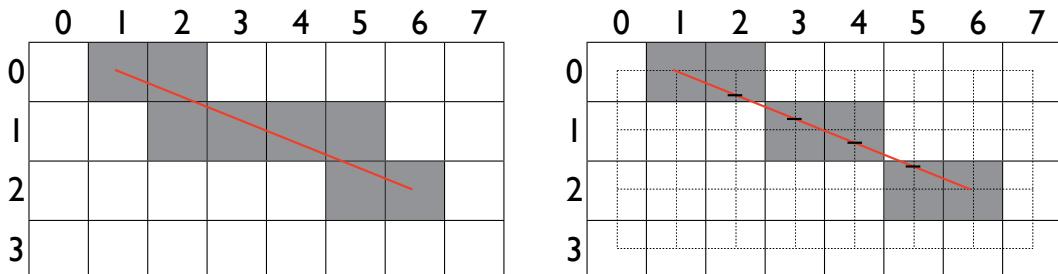


FIGURE 3.1 – Dessin d'un segment de droite. À gauche, tous les pixels qui touchent le segment mathématique sont dessinés, l'épaisseur n'est pas homogène. À droite, un seul pixel par colonne est dessiné. On choisit le pixel dont le centre est le plus proche de l'intersection entre le segment et le centre de la colonne.

entières, on conserve y tel quel, et on augmente une “erreur” $\varepsilon \leftarrow \varepsilon + \Delta_y / \Delta_x$. Lorsque l’erreur dépasse $1/2$ (la moitié d’un pixel), on doit augmenter y de 1, et enlever 1 à l’erreur :

```

 $x \leftarrow x + 1$ 
 $\varepsilon \leftarrow \varepsilon + \Delta_y / \Delta_x$ 
if  $\varepsilon > 1/2$  then
     $y \leftarrow y + 1$ 
     $\varepsilon \leftarrow \varepsilon - 1$ 
fin if
dessiner ( $x, y$ )

```

La boucle contient des opérations sur les nombres réels. Par soucis d’efficacité, on veut la transformer pour traiter uniquement des nombres entiers. Définissons $E = \varepsilon \times \Delta_x$, soit $\varepsilon = E / \Delta_x$. Alors quand on incrémente x , $E \leftarrow E + \Delta_y$. Pour incrémenter y , on teste $E > 1/2 \times \Delta_x$, ce qui est équivalent à $2 \times E > \Delta_x$. L’algorithme devient :

```

 $x \leftarrow x + 1$ 
 $E \leftarrow E + \Delta_y$ 
if  $2 \times E > \Delta_x$  then
     $y \leftarrow y + 1$ 
     $E \leftarrow E - \Delta_x$ 
fin if
dessiner ( $x, y$ )

```

L’algorithme ne manipule plus que des variables entières. C’est l’algorithme de “Bresenham”, du nom du Professeur qui l’a inventé. L’algorithme est donné ici dans sa version canonique : $\Delta_x > 0, \Delta_y > 0, |\Delta_x| > |\Delta_y|$. Il est facile de l’adapter pour les autres cas en inversant les signes, ou bien en inversant les variables lorsque $|\Delta_x| < |\Delta_y|$.

Dessin des polygones

La fonction `ei_draw_polygon(...)` permet de remplir l’intérieur d’un polygone défini par un ensemble ordonné de sommets, et avec une couleur donnée. Tout comme les autres dessins de primitives graphiques, ce remplissage doit être effectué d’une manière efficace, surtout qu’il concerne souvent des centaines de milliers de pixels.

La première optimisation consiste à dessiner le polygone ligne par ligne horizontalement. Les pixels sont ordonnés en mémoire de gauche à droite en commençant par la ligne du haut, et jusqu’à la ligne du bas. Pour remplir une ligne horizontale d’un polygone, il suffit donc d’écrire la valeur de la couleur dans la mémoire qui représente le pixel, d’incrémenter le pointeur de pixel, et de répéter ceci jusqu’à la fin de la ligne. On appelle une ligne horizontale de l’image une “scanline”. Le problème revient alors à déterminer les points d’intersections entre la scanline courante et les côtés du polygone, comme illustré sur la figure 3.2, à gauche, pour la scanline 8 de l’image.

Supposons que nous disposons de la liste ordonnée (de gauche à droite) des points d’intersection entre la scanline et les côtés du polygone. Il suffit de colorer tous les pixels entre les deux premières intersections, puis de sauter les pixels jusqu’à l’intersection suivante, et recommencer, jusqu’à atteindre la dernière

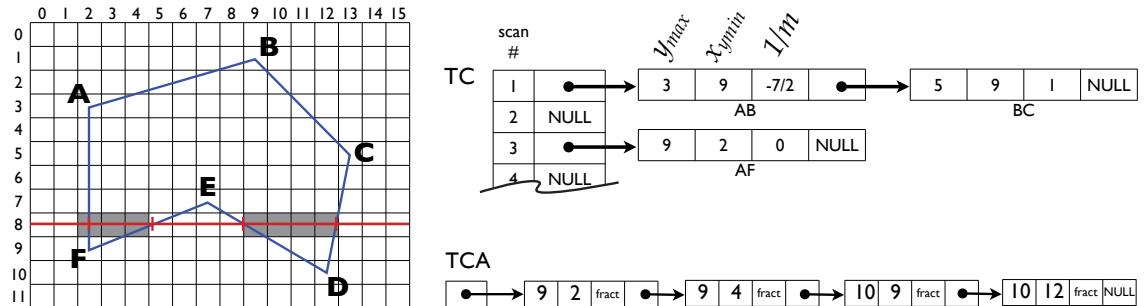


FIGURE 3.2 – Remplissage de polygones. À gauche : La scanline numéro 8 intersecte les 4 côtés du polygone (AF), (EF), (ED), (CD). À droite : les premières entrées de la TC (en haut), et la TCA au moment de remplir la scanline 8 (en bas).

intersection (à la scanline 8 : colorier entre (AF) et (EF), sauter entre (EF) et (ED), colorier entre (ED) et (CD). L'algorithme de remplissage des polygones consiste à calculer, pour toutes les scanlines, cette liste d'intersection qu'on appelle *la Table des Côtés Actifs* (TCA). La TCA de la scanline 8 est représentée sur la figure 3.2, en bas à droite.

Plutôt que de calculer cette TCA à chaque scanline en partant de rien, la deuxième optimisation consiste à la calculer incrémentalement à partir de la TCA de la scanline précédente. Calculer l'intersection entre une ligne horizontale de l'image (la scanline) et un segment de droite (un côté du polygone), c'est exactement ce que fait l'algorithme de Bresenham pour le dessin des lignes dans le cas $|\Delta_x| < |\Delta_y|$. On utilise donc Bresenham pour mettre à jour toutes les intersections entre la scanline et les côtés quand on passe à la scanline suivante. La seule différence par rapport à Bresenham, c'est que l'algorithme est toujours dirigé par y , même quand un côté a $|\Delta_x| > |\Delta_y|$. Ceci nécessite une petite adaptation.

Représentation des côtés La première étape de l'algorithme est de construire une *table des côtés* (TC). On représente chaque côté du polygone par une structure qui mémorise les informations suivantes :

- Le numéro de scanline à partir de laquelle le côté n'aura plus d'intersection, et pourra être oublié, c'est l'ordonnée maximale du côté y_{max} .
- L'abscisse de l'intersection du côté avec la première scanline qui intersecte ce côté (l'abscisse du côté à l'ordonnée minimale x_{ymin}). Elle sera mise à jour à chaque nouvelle scanline.
- Les paramètres du côté pour l'algorithme de Bresenham : E , $|\Delta_x|$, $|\Delta_y|$. Dans la figure 3.2, ces paramètres sont représentés par la réciproque de la pente du côté $(x_1 - x_0)/(y_1 - y_0)$. C'est ce qu'il faut ajouter à x quand y progresse de 1, c'est à dire quand on passe à la scanline suivante,
- un pointeur vers une autre structure représentant un côté, afin de pouvoir chaîner ces structures et former une liste.

La TC et 3 structures représentant 3 côtés différents sont illustrées sur la figure 3.2, à droite. La TC contient une entrée par scanline qui intersecte le polygone, chaque entrée pointe vers la liste chaînée des côtés du polygone qui "commencent" à cette scanline, c'est à dire dont le y_{min} est égal à cette scanline. Certaines entrées de la TC sont à NULL : celles des scanlines sur lesquelles ne commence aucun côté.

La TCA est simplement représentée par un pointeur vers un côté : c'est la tête d'une liste chaînée. La TCA est initialisée à la liste vide. À chaque nouvelle scanline, on va enlever de TC les côtés qui commencent à cette scanline, et les placer dans TCA. On va enlever de TCA les côtés qui ont fini à la scanline précédente. Et on garde, bien sûr, les côtés présents dans la TCA qui intersectent encore la nouvelle scanline.

L'algorithme On initialise donc TC avec tous les côtés du polygone mais *en ignorant les côtés horizontaux* car ils sont inutiles à l'algorithme. La TCA est initialisée à la liste vide, et la scanline courante y à la première scanline qui intersecte le polygone. On exécute la boucle suivante :

- tant que TC et TCA ne sont pas vides
- Déplacer les côtés de TC(y) dans TCA.
 - Supprimer de TCA les côtés tels que $y_{max} = y$.
 - Trier TCA par abscisse croissant des intersection de côté avec la scanline.
 - Modifier les pixels de l'image sur la scanline, dans les intervalles intérieurs au polygone (voir les règles de remplissage ci-dessous).
 - Incrémenter y .
 - Mettre à jour les abscisses d'intersections des côtés de TCA avec la nouvelle scanline.

Règles de remplissage On souhaite que des polygones qui partagent des côtés n'empiètent pas les uns sur les autres, comme illustré sur la figure 3.3. Pour cela, il faut appliquer les règles suivantes lors du remplissage des intervalles de pixels :

- La coordonnée de l'intersection est arrondie à l'entier supérieur en entrée d'un intervalle, à l'entier inférieur en sortie de l'intervalle.
- Sur coordonnée entière, on considère que le pixel fait partie du polygone sur une entrée d'intervalle, pas sur une sortie.

3.1.3 Clipping

Comme indiqué à la section 2.2.2 les *primitives graphiques* prennent en paramètre un *rectangle de clipping*, qu'on appellera à présent "le clippeur". Le clippeur définit la zone de l'écran auquel doivent se

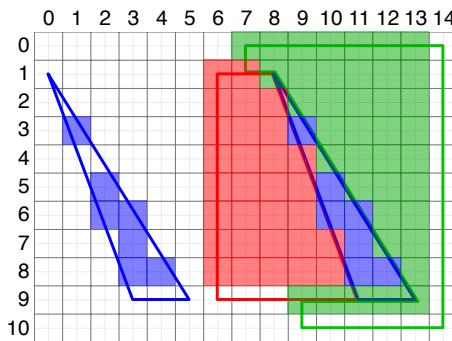


FIGURE 3.3 – Règles de remplissage.

limiter les dessins. Réaliser le *clipping* peut se faire de différentes façon. Nous détaillons ci-dessous 3 techniques : de la plus simple à programmer mais la moins efficace, à la plus complexe à programmer mais la plus efficace. Dans le projet, il vous est fortement conseillé de mettre en œuvre la première approche. Puis, si le temps vous le permet, vous pourrez mettre en œuvre les approches plus complexes mais plus efficaces.

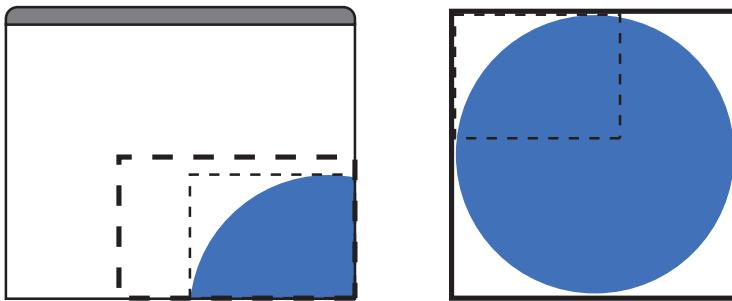


FIGURE 3.4 – Clipping par offscreen. La forme (le cercle) doit être dessinée sur la fenêtre (à gauche) dans les limites du clippeur (rectangle en gros pointillés). Un offscreen de taille suffisante pour contenir la forme est alloué (à droite), le cercle y est dessiné. L'intersection entre le clippeur et la forme (rectangle en petits pointillés) est copié depuis l'offscreen vers la fenêtre.

- Le *clipping par force brute* consiste à tester systématiquement, à chaque fois qu'il faut modifier un pixel en mémoire, si les coordonnées de ce pixel sont à l'intérieur du clippeur. Si ce n'est pas le cas, la modification du pixel n'est pas réalisée. Cette technique est très simple à réaliser : il suffit de faire 4 tests qui comparent les coordonnées du pixel avec les 4 bords du clippeur. C'est la technique la moins efficace car elle nécessite le calcul par la machine de 4 tests (4 soustractions) à chaque modification de pixel, et les calculs sont réalisés sur toute la forme à dessiner, même si le clippeur n'en conserve qu'une toute petite partie.
- Le *clipping hors-écran* consiste à allouer une surface hors-écran assez grande pour contenir la totalité de la forme à dessiner. On peut alors exécuter le dessin de la forme dans cette surface sans se poser le problème du clipping. Une fois le dessin réalisé, on copie le dessin depuis la surface hors-écran vers la surface de l'écran, mais en limitant la copie à la sous-partie qui correspond au clippeur. Cette approche est illustrée sur la figure 3.4. Par cette technique, on élimine le coût de calcul du clipping dans les algorithmes de dessin des formes (par exemple les 4 tests de la technique par force brute). Par contre, comme dans le clipping par force brute, le dessin de toute la forme doit être calculé, même si le clippeur n'en conserve au final qu'une petite partie.
- Le *clipping analytique* consiste à calculer l'intersection de la forme à dessiner avec le clippeur. Le résultat de ce calcul est une forme qui est complètement contenue dans le clippeur. On peut alors exécuter l'algorithme de dessin de cette forme sans se soucier du problème de clipping, donc de façon optimale. Par ailleurs, le dessin de la forme “clippée” nécessite les calculs uniquement sur la partie visible de la forme, ce qui est une optimisation très importante dans le cas d'un petit clippeur sur une grande forme. Cependant, le calcul de l'intersection d'une forme avec le clippeur peut être

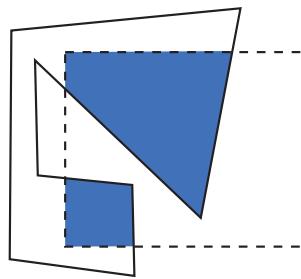


FIGURE 3.5 – Clipping analytique d'un polygone complexe. Le polygone est dessiné en traits pleins, le clippeur en pointillés. L'intersection résulte en deux polygones (les formes pleines).

complexe. Par exemple, la figure 3.5 montre que l'intersection d'un clippeur rectangulaire avec un polygone quelconque peut aboutir à plusieurs polygones dont il faut calculer les coordonnées de nouveaux sommets qui n'existaient pas dans le polygone initial.

Dans les deux sections suivantes, “Clipping analytique des lignes” et “Clipping analytique des polygones”, nous présentons les algorithmes de clipping analytique des lignes et polygones. Cependant, ces algorithmes concernent uniquement les groupes qui auront tout fini par ailleurs et qui souhaitent se lancer dans l'optimisation du clipping. Il n'est donc *pas nécessaire de lire ces deux sections au début du projet*. Ces algorithmes sont considérés comme des *extensions* au projet (voir 4.3 “Extensions”).

Clipping analytique des lignes

Le problème consiste à calculer, à partir d'un segment et d'un clippeur rectangulaire, l'intersection du segment avec le clippeur. Il faut traiter de nombreux cas : tester la position des deux extrémités du segment par rapport aux 4 côtés du clippeur. Pour faire ces tests de façon efficace, Cohen et Sutherland propose de coder la position d'un point par rapport au clippeur sur un code à 4 bits. Prenons un point (x, y) et un clippeur défini par ses deux sommets en haut à gauche (x_{min}, y_{min}) et en bas à droite (x_{max}, y_{max}). Le code du point (x, y) est défini ainsi : le premier bit est à 1 si le point est *sous* le clippeur, c'est à dire si $y_{max} - y < 0$, c'est donc le bit de signe de $y_{max} - y$. On procède ainsi pour les 3 autres côté du clippeur : on prend les bits de signe de $y - y_{min}$ (haut), $x_{max} - x$ (droit), $x - x_{min}$ (gauche). Les codes de Cohen-Sutherland sont représentés sur la figure 3.6, à droite.

Par des opérations logiques bit à bit, on peut rapidement traiter les cas “triviaux” :

- Un segment est *accepté* de façon triviale lorsque l'opération “ou logique” bit à bit entre les codes de ses deux sommets renvoie 0 : aucun des deux sommets n'est hors du clippeur, le segment est entièrement dans le clippeur.
- Un segment est *rejeté* de façon triviale lorsque l'opération “et logique” bit à bit entre les codes de ses deux sommets renvoie un nombre différent de 0 : les deux sommets ont un bit en commun, ils sont tous les deux extérieurs au clippeur par rapport à un de ses côtés, le segment entier est hors du clippeur.

Si un segment n'est ni accepté, ni rejeté trivialement, alors on va le couper par rapport à un des côtés du clippeur. On utilise à nouveau les codes des sommets pour choisir un sommet extérieur (son code est

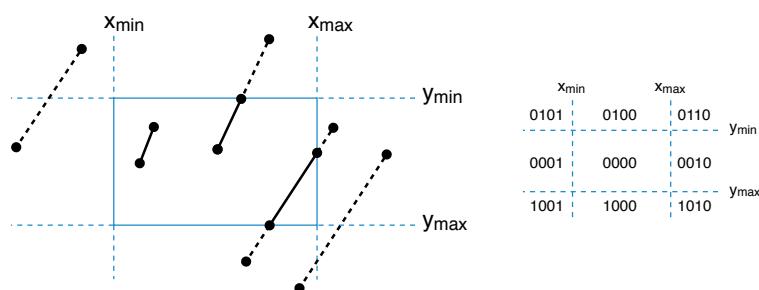


FIGURE 3.6 – Clipping analytique de ligne. Codes de Cohen-Sutherland.

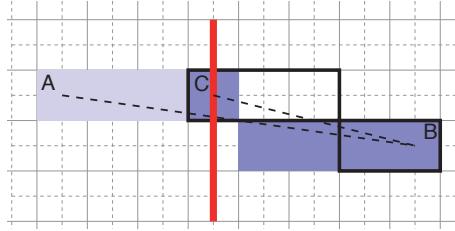


FIGURE 3.7 – Initialisation d'un segment coupé : on souhaite couper le segment AB au niveau d'un bord du clippeur (ligne rouge verticale). On calcule l'intersection entre le bord et le segment au niveau du pixel C. Un tracé de ligne de Bresenham CB dessinera les pixels encadrés en noir, alors que AB dessinera les pixels grisés.

différent de 0) et les bits du code de ce sommet permettent d'identifier le bord du clippeur sur lequel on va couper le segment. On calcule le code du nouveau sommet du segment qu'on vient de créer, et on peut reprendre l'analyse au début (acceptation ou rejet trivial, etc.).

Prenons l'exemple du segment en bas à droite sur la figure 3.6. Ses sommets ont les code 0010 (sommel à droite) et 1000 (sommel en bas). Le segment n'est ni accepté, ni rejeté trivialement. Les codes des deux sommets étant différents de 0, on peut choisir l'un ou l'autre pour couper le segment. Choisissons par exemple celui du bas. Son code 1000 indique qu'il faut couper le segment par rapport à y_{max} . On crée un nouveau segment avec l'intersection du segment avec y_{max} et en gardant le sommet de droite. Le nouveau sommet a pour code 0010, de même que l'ancien. Le nouveau segment est rejeté trivialement.

Il reste une difficulté illustrée sur la figure 3.7 : lorsqu'un segment est coupé par un clippeur, sa partie visible doit s'afficher exactement de la même façon que si le segment n'était pas clippé. Il faut donc penser, au moment où le segment est coupé, à mémoriser les informations permettant de "reprendre" l'algorithme de Bresenham (c.f. 3.1.2 "Dessin de lignes") à l'endroit de la coupure, c'est-à-dire calculer une erreur initiale E non nulle.

Clipping analytique des polygones

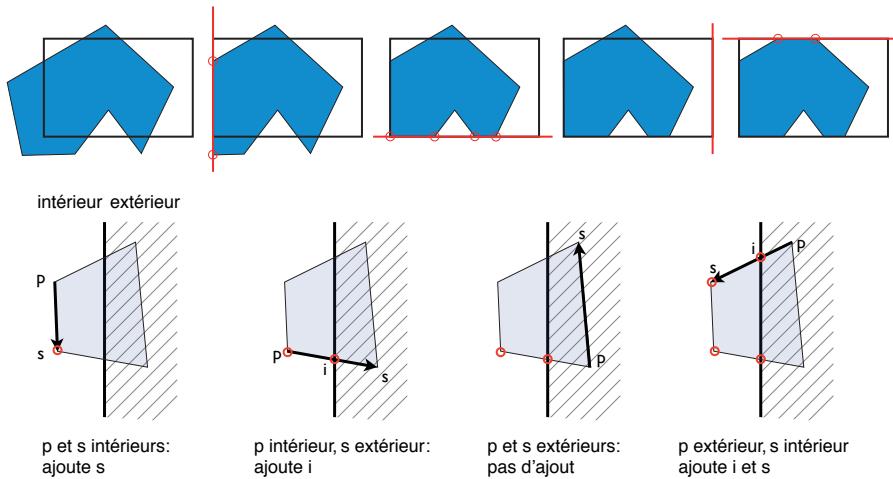


FIGURE 3.8 – Clipping analytique de polygone. En haut : le polygone (forme pleine) est coupé par les 4 bords du clippeur (rectangle noir) successivement : gauche, bas, droite, haut. En bas : les 4 cas de l'algorithme de Sutherland-Hodgman. Les cercles représentent les points ajoutés au polygone résultant.

Il s'agit de calculer l'intersection d'un polygone quelconque avec un clippeur rectangulaire. Pour simplifier le problème, Sutherland et Hodgman proposent de couper le polygone successivement avec les 4 bords du clippeur, comme illustré sur la figure 3.8 (en haut). Le problème se ramène donc à calculer un ou plusieurs polygones (voir figure 3.5) résultant du clipping d'un polygone par un demi-plan.

L'algorithme de Sutherland-Hodgman parcourt tous les côtés du polygone, en considérant, pour chaque

côté, le *sommet précédent* p et le *sommet suivant* s dans le sens du parcours. Selon 4 cas différents, illustrés sur la figure 3.8 (en bas), l’algorithme peut ajouter au polygone résultant le sommet précédent, suivant, ou l’intersection du côté avec le bord du clippeur.

De même que pour le clipping de ligne, la partie visible d’un polygone clippé doit s’afficher exactement de la même façon que si le polygone n’était pas clippé. Lorsqu’un côté d’un polygone est coupé, il faut prévoir l’initialisation du calcul de l’intersection du côté avec la scanline pour gérer le problème illustré sur la figure 3.7.

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);
```

3.3 Polymorphisme

Pour implémenter différentes *classes de widgets* 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

Un widget de la classe `toplevel` doit mémoriser 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` nécessite uniquement le texte à afficher sur le bouton². 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.

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

Le polymorphisme des données est implémenté en utilisant une *structure commune* qui regroupe tous les attributs universels :`ei_widget_t`.

```
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 sont ajoutés en créant une nouvelle structure de données dont le premier champ est du type de la *structure commune* (`ei_widget_t`). Par exemple, vous créez une nouvelle structure de donnée pour mémoriser les attributs d'un widget de type bouton :

```
typedef struct ei_button_t {
    ei_widget_t            widget;

    int                   specific_attribute1;
    char                  specific_attribute2[80];
} ei_button_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_button_t` peut être utilisé comme un pointeur vers une structure de type `ei_widget_t` parce que le premier champ de `ei_button_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, elle peut donc être utilisée pour tout type de widget. 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_button_t    mon_bouton;

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

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. Attention, `ei_widgetclass_t` ne représente pas un widget, mais la *classe* d'un widget. Pour les boutons par exemple, il n'y a dans le programme qu'un seul exemplaire la structure `ei_widgetclass_t` pour décrire cette *classe* de widget. Par contre, il y a un exemplaire de `ei_button_t` par *widget* de type bouton.

```
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;
ei_widgetclass_handlefunc_t     handlefunc;
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);
    ...
}

```

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.

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.2 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.

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 2.2.3, 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).

3.5 Gestion de la géométrie : le placeur

Dans ce projet, il vous est demandé de réaliser un gestionnaire de géométrie de type “placeur”. Son interface de programmation est définie dans le fichier “*ei_placer.h*”.

3.5.1 Mécanisme général 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 le placeur, le champ *placer_params* est à **NULL** dans la structure *ei_widget_t*, et le widget n'est pas encore affiché à l'écran.

Pour positionner un widget dans son parent, et donc le faire apparaître à l'écran, le programmeur appelle la fonction de configuration du placeur *ei_place*. Après cet appel, le champ *placer_params* du widget n'est plus **NULL** : il pointe vers une structure que vous devrez définir et qui décrit tous les paramètres de placement de ce widget.

Vous devez écrire le code de la fonction `ei_placer_run(...)`. Son rôle est de calculer ou recalculer la position et la taille d'un widget dans son parent. Vous devez appeler cette fonction chaque fois que nécessaire. Par exemple, en reconfigurant un widget de type `button` pour changer son label de "Ok" à "Cancel", il bouton peut nécessiter une taille plus grande. La fonction de configuration du widget doit donc appeler `ei_placer_run(...)` si le widget est placé à l'écran à ce moment-là. Autre exemple : une fenêtre `toplevel` qui a des descendants provoque le recalcule de la géométrie de ses descendants quand elle change de taille.

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 ces 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 `content_rect` du parent (définit dans la structure `ei_widget_t`). L'axe des abscisses croît vers la droite et l'axe des ordonnées croît vers le bas³.

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 d'un 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-Est (`ei_anc_northeast`), 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".

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.

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 Gestions des événements

3.6.1 Principes

Les différents types utilisés pour la mise en place des événements sont définis dans le fichier "`ei_event.h`".

Lorsqu'un événement est émis par l'utilisateur, la bibliothèque doit se charger d'appeler le traitant concerné. Un traitant renvoie toujours un booléen `ei_bool_t` qui indique s'il a traité l'événement ou non. Si l'événement n'a pas été traité, la bibliothèque peut appeler un autre traitant comme nous allons l'expliquer ci-dessous. Pour les événements situés (voir 2.1.1), le traitant concerné est le traitant de la classe de widget du widget qui est sous le pointeur de la souris. Pour les événements non situés, c'est au programmeur de l'application de définir le traitant par défaut par appel de la fonction `ei_event_set_default_handle_func(...)`. Outre les événements non situés, la bibliothèque communique au traitant par défaut les événements situés lorsqu'il n'y a aucun interacteur sous le pointeur de la souris, ou bien lorsque l'appel du traitant de l'interacteur concerné a renvoyé `EI_FALSE` (l'événement n'a pas été traité).

3.6.2 Interacteur actif

Il existe une exception à ce principe général de "routage" des événements. Ce routage exceptionnel a lieu lorsqu'un interacteur est en cours de manipulation par l'utilisateur. Par exemple, un utilisateur veut déplacer une fenêtre `toplevel` vers le haut de l'écran. Il appuie sur le bouton de la souris alors que le pointeur est sur la barre de titre de la `toplevel`. La bibliothèque utilise le picking (voir 2.2.4) pour identifier la `toplevel` à qui communiquer l'événement (appui de bouton de la souris). Le traitant des `toplevel` initialise le déplacement. L'utilisateur déplace alors le pointeur de la souris vers le haut ce qui génère un événement "déplacement de souris". Mais le pointeur est sans doute sorti des limites de la `toplevel`. Par picking, la bibliothèque ne pourra alors plus identifier la `toplevel` concernée par cet événement de déplacement : le picking identifiera l'interacteur situé au-dessus de la `toplevel` s'il y en a un, ou l'interacteur racine. Pour gérer ce problème, lorsqu'un utilisateur initie une interaction par appui sur le bouton de la souris, il faut considérer que l'interacteur choisi par ce click restera l'interacteur concerné par tous les événements suivants, jusqu'à l'arrivée d'un événement de relâchement du bouton de la souris. On dit que cet interacteur devient "l'interacteur actif" jusqu'au relâchement du bouton de la souris. Un traitant qui initialise une interaction de ce type appelle la fonction `ei_event_set_active_widget(...)` pour déclarer l'interacteur concerné comme *interacteur actif*. Le routage d'événement donne toujours la priorité à l'interacteur actif s'il y en a un : c'est son traitant qui sera appelé en premier.

Les types d'événements sont définis dans l'énumération `ei_eventtype_t`. Ils incluent : 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`).

3.6.3 Traitants externes et internes

Le programmeur d'application programme les *traitants externes* qui définissent le comportement spécifique de l'application en réaction aux actions de l'utilisateur. Par exemple, il crée une fonction `passe_au_suivant(...)` dont la signature correspond au type `ei_callback_t`. Il associe ce traitant au bouton correspondant en donnant son adresse comme paramètre `callback` à l'appel de la fonction `ei_button_configure(...)`.

La bibliothèque fournit également un ensemble de traitants internes qui permettent de gérer les comportements standards 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. Les traitants externes sont donc mémorisés dans la structure qui décrit chaque bouton (une version spécifique de `ei_widget_t` pour les widgets de type `button`). Par contre, il n'y a qu'un seul *traitant interne* qui gère tous les boutons, puisque le comportement standard ne varie pas en fonction des boutons (i.e. tous les boutons doivent apparaître enfouis quand on clique dessus). L'adresse du traitant interne d'une classe d'interacteur est donc mémorisé dans le champ `ei_widgetclass_handlefunc_t` de la structure qui décrit une classe : `ei_widgetclass_t`.

Lorsqu'un événement a lieu, la bibliothèque appelle le traitant concerné en lui 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_evkeyup`, la structure `ei_event_t` contient le code de la touche qui a été enfouie⁴ et un champ de bits qui décrit l'état enfoui 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 enfoui ou relâché.

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 traitant associé doit déclarer l'interacteur concerné comme “actif”, et prendre en compte les 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 traitant associé à l'interacteur ne tient plus compte des événements `ei_ev_mouse_move` et `ei_ev_mouse_buttonup`, le traitant déclare que l'interacteur n'est plus “actif”.

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 dans le traitant interne de la classe d'interacteur `toplevel`.

Début d'action

L'utilisateur vient de cliquer avec le pointeur de souris sur la barre d'en-tête de la `toplevel`. Le traitant interne associé à la classe `toplevel` reçoit un événement de type `ei_ev_mouse_buttondown` et localise le pointeur sur la barre d'en-tête. L'action de *déplacement* démarre. La position de la `toplevel` sera maintenant contrôlée par la souris. Le traitant déclare la `toplevel` comme “interacteur actif” (`ei_event_set_active_widget(...)`). Par ailleurs, il faut qu'à partir de maintenant le traitant s'intéresse aux événements `ei_ev_mouse_move` et `ei_ev_mouse_buttonup`.

Micro-déplacements de la fenêtre

Chaque mouvement élémentaire de la souris provoque la réception d'un événement de type `ei_ev_mouse_move` par le traitant interne de la classe `toplevel`. La position du curseur est utilisée pour calculer la nouvelle position de la `toplevel`. 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.

4. Les identificateurs des codes de touche clavier sont déclarés dans le fichier “`SDL_keycode.h`” de la bibliothèque `SDL`.

Fin d'action

L'utilisateur relâche le bouton de la souris. Le traitant interne reçoit un événement de type `ei_ev_mouse_buttonup`. Il déclare que l'interacteur n'est plus l'interacteur actif. Il note qu'il ne s'intéresse plus, à partir de cet instant, aux événements de déplacement de la souris et de relâché de son bouton. Par contre, il surveille le prochain appui sur le bouton de la souris pour initier un nouveau déplacement.

3.6.5 Exemple du redimensionnement

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

- Initialisation du redimensionnement lorsque la callback interne reçoit un événement de type `ei_ev_mouse__buttondown` alors que le pointeur est sur le bouton de redimensionnement. L'interacteur est déclaré comme "actif". Le traitant passe dans un état où il surveille 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 réception d'un événement de type `ei_ev_mouse_buttonup`. L'interacteur est déclaré comme non actif. La callback repasse dans un état où elle ne s'intéresse plus qu'aux événements `ei_ev_mouse_buttondown`.

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.

Il n'est pas efficace de mettre à jour l'écran de façon *immédiate* : par exemple, si le programmeur change le label d'un bouton de "Ok" à "Annuler", on pourrait dessiner immédiatement le bouton avec le nouveau label. Mais ce changement va modifier la taille du bouton, ce qui va entraîner une modification de la position et de la taille du bouton, et donc un nouveau dessin. Le premier dessin aura été exécuté pour rien. Les mises à jour à l'écran sont donc *différentes* : quand une mise à jour est nécessaire, on se contente d'appeler la fonction `ei_app_invalidate_rect(...)` et on lui passe en paramètre le rectangle de l'écran qui doit être mis à jour. La bibliothèque mémorise ainsi l'ensemble de ces rectangles. Quand les traitements sont finis, la bibliothèque se charge de dessiner tous les rectangles qui lui ont été communiqué. Ici, il est important de mettre en place des optimisations, notamment pour ne pas dessiner un rectangle qui est entièrement contenu dans un autre rectangle, i.e. éviter un dessin inutile. Le dessin effectif se déroule comme suit.

Dans la boucle principale, après avoir traité un événement, la bibliothèque bloque la surface de l'écran (3.1.1 "Surface de dessin"). Puis, pour le(s) rectangle(s) de l'écran à dessiner, la bibliothèque appelle la `drawfunc` de tous les widgets de la hiérarchie en utilisant le rectangle comme *rectangle de clipping* (2.2.2 "Primitives Graphiques"). 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 les surfaces et demande au système d'exploitation de copier ces pixels sur l'écran (`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. C'est un autre exemple de l'intérêt du dessin différé : sur le déplacement de quelques pixels d'une grande fenêtre, il y a une grande zone de chevauchement entre l'ancienne et la nouvelle position de la fenêtre comme illustré sur la Figure 3.9. Il serait très inefficace de dessiner deux fois cette zone : une fois pour effacer, puis une fois pour repositionner la fenêtre.

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.

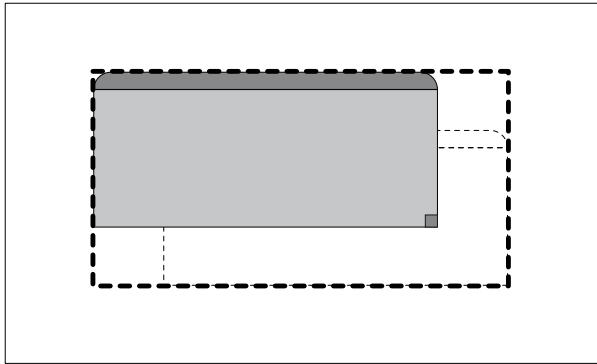


FIGURE 3.9 – Rectangles de re-dessin lors d'un déplacement de fenêtre. La fenêtre a été légèrement déplacée entre l'ancienne position (pointillés) et la nouvelle (fenêtre pleine). Deux rectangles de re-dessin sont générés : à l'ancienne et à la nouvelle position de la fenêtre. Il est préférable de ne redessiner qu'un seul rectangle (en vert) qui englobe les deux rectangles.

3.8 Programme principal et boucle principale

Un programme de type événementiel se décompose en deux grandes parties : l'initialisation et le lancement de la boucle principale (c.f. 2.1.2 “Structure d'un programme événementiel”).

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(...)`,
- 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.). Ces appels permettent en particulier de 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 lié 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 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 la bibliothèque `SDL2` pour accéder aux pixels de l'écran et aux événements utilisateur. `SDL2` est multiplateforme, vous pouvez donc développer le projet sur votre environnement préféré (Linux, Mac OSX, Windows). Par contre, les enseignants peuvent vous aider concernant Linux, c'est beaucoup moins sûr pour Mac OSX et Windows.

4.1 Compilation

L'archive de base du projet contient un fichier `CMakeLists.txt`. Pour compiler en dehors des sources du projet, l'archive contient un répertoire vide `cmake` dans lequel vous réaliserez la compilation. Par exemple :

```
cd cmake
cmake ..
make minimal
```

Cependant, il est fortement conseillé d'utiliser l'environnement de développement **CLion** pour bénéficier du débugger graphique, de la navigation entre les symboles, de la complétion automatique, de l'intégration avec Valgrind, et de nombreux autres outils inclus. Cette application est gratuite pour les étudiants, il suffit de demander une licence avec votre adresse e-mail `@grenoble-inp.fr`. L'archive contient un répertoire `clion`. Au premier lancement de **CLion**, choisissez l'option “ouvrir” et pointez sur ce répertoire. Pour vous aider à démarrer avec **CLion**, des aides sont disponibles sur ensiwiki¹. Davantage de détails sur CMake sont donnés en section 5.5.

4.2 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 : des lignes, un cadre, un bouton, une fenêtre et

1. <http://brouet.imag.fr/fberard/ProjetCLL/CLion>

un bouton et, enfin, les jeux puzzle et 2048. Ces deux dernières 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.

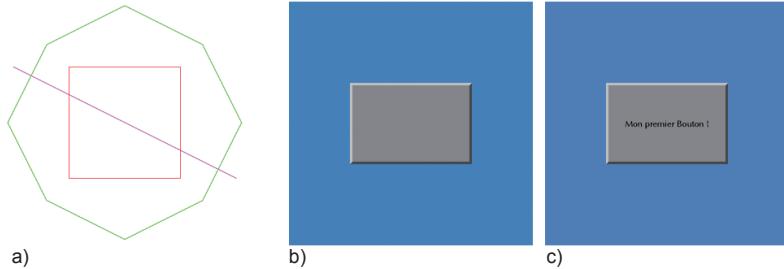


FIGURE 4.1 – Résultat des applications de test : lines (a), frame (b), et button (c).

4.2.1 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 blanc. Puis elle se met en attente d’un événement clavier (appui d’une touche) pour terminer.

4.2.2 Lignes

Cette application sert à tester le dessin des primitives graphiques telles que les lignes et les polygones illustrés sur la figure 4.1a. Elle fournit un ensemble simple de fonctions qui testent le dessin des lignes, et doit être étendue pour tester les polygones et le clipping. Pour vous aider dans ce développement, nous vous proposons un ensemble d’étapes dans l’annexe A.1.

4.2.3 Cadre (frame)

Cette application affiche un simple cadre (“frame” en anglais) tel qu’illustré sur la figure 4.1b. Elle est constituée d’un widget racine ayant pour descendant un widget de type `frame`. Le `frame` a une taille fixée. Son positionnement est défini 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.

4.2.4 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 4.1c). 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.7 des indications pour la réalisation de la gestion des événements.

4.2.5 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.

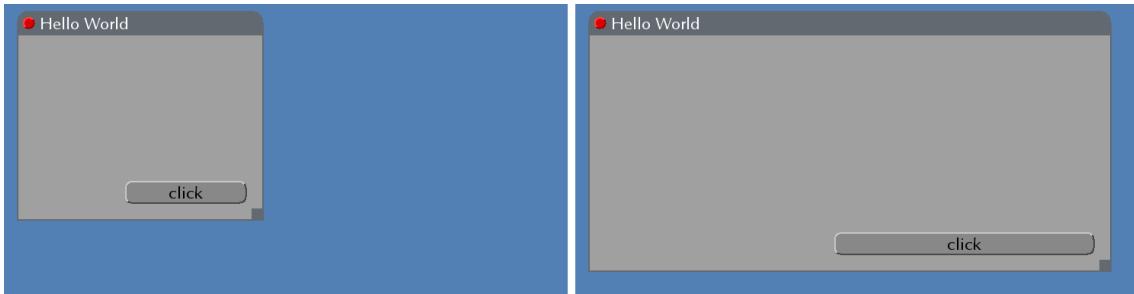


FIGURE 4.2 – Fenêtre redimensionnable et bouton de taille et de position relatives.

4.2.6 Puzzle et 2048

Ces deux applications ont une utilité réelle : il s’agit d’un jeu de taquin de 15 pièces carrées (figure 4.3 à gauche) et du jeu “2048” (figure 4.3 à droite). Pour le jeu de taquin, 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. Pour le jeu 2048, on utilise les flèches du clavier pour jouer. Plusieurs fenêtres de jeu peuvent être créées par le raccourci clavier <Command>-N. Une fenêtre est fermée par <Command>-W.

4.3 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 (***)�.

4.3.1 Gestion optimisée du clipping de ligne (**)

Cette extension consiste à réaliser le clipping analytique des lignes, tel que décrit en section 3.1.3.

4.3.2 Gestion optimisée du clipping de polygone (***)

Cette extension consiste à réaliser le clipping analytique des polygones, tel que décrit en section 3.1.3.

4.3.3 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 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 gauche.

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.

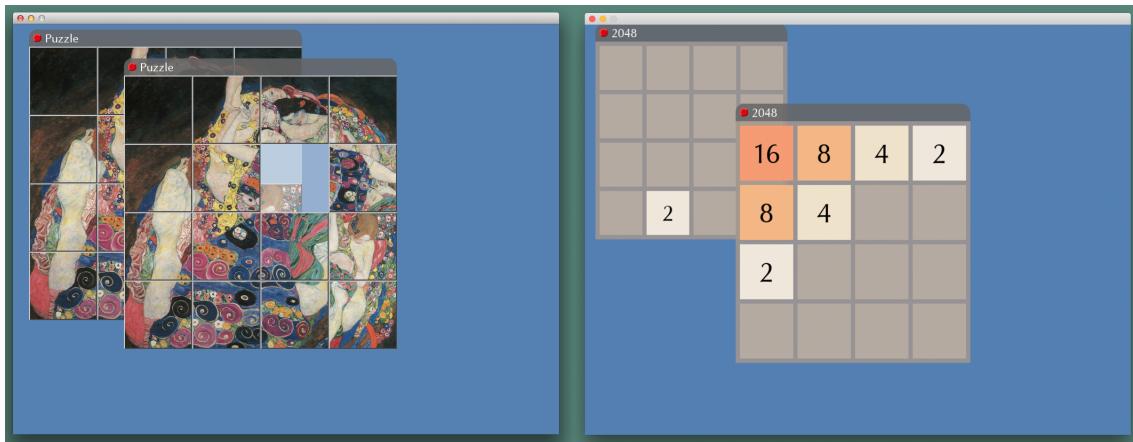


FIGURE 4.3 – Le jeu de taquin (à gauche) et le jeu 2048 (à droite). Chaque application contient deux toplevel. Les fenêtres de plus haut niveau à fond bleu sont des fenêtres système de l’application, ce ne sont pas des toplevel gérées par votre bibliothèque.

4.3.4 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 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 sont illustrés sur la figure 4.4, à droite. 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”.

4.4 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.

4.4.1 Critères d’évaluation

Par ordre d’importance :

- Exactitude : le projet fait ce qui est demandé.
- Qualité de la structure de votre code (modules, fonctions).

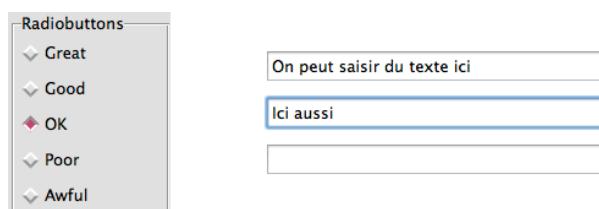


FIGURE 4.4 – Extensions : boutons radio (à gauche), champs de saisie (à droite).

- 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.

4.4.2 Rendu des fichiers de votre projet

L'archive de votre projet devra être déposée sur Teide. Pensez à supprimer tous les fichiers inutiles : le contenu des répertoires `clion` et `cmake`, tous les fichiers générés par la compilation (exécutables, `.o`) ou générés par vos outils (`.git`, `.vscode`, ...). L'archive doit contenir :

- 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 `CMakeLists.txt`. Ce fichier doit permettre au minimum de générer un fichier `Makefile` qui permet de nettoyer votre archive (`make clean`) et de régénérer la bibliothèque `make ei` et l'ensemble des exécutables (`make all`).

4.4.3 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 comment votre bibliothèque permet de réaliser des applications plus complexes que de simples exemples.
- 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 projet¹. Cette page contient les informations de dernière minute sur le déroulement et l'organisation du projet.

Pendant tout le projet, plusieurs salles machines de l'Ensimag sont réservées et des enseignants sont là pour vous aider (voir la page du projet pour le détail de la présence des enseignants). L'aide des enseignants porte sur :

- la programmation en langage C.
- l'environnement de développement (CMake, CLion, ligne de commande, 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.

5.2 Documentation “Doxygen”

Les fichiers d'en-tête qui vous sont fournis contiennent des commentaires qui documentent 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*². Ce système parcourt les fichiers de code, récupère les commentaires, et en fait une documentation structurée sous forme HTML. Pour générer la documentation, construisez la cible “doc” du projet (`make doc`). La documentation est générée dans les répertoires `docs/html`. Ouvrez le fichier “`docs/html/index.html`” dans un navigateur (tel que Firefox).

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 ratrappage 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.

1. <http://brouet.imag.fr/fberard/ProjetCLL/ProjetC>

2. <http://www.doxygen.org/>

- 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 4.4.3), 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

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, nous vous demandons de reproduire la présentation du code des fichiers fournis : tabulations de 8 caractères, style des identificateurs, placement des espaces et des accolades, etc. Faites des commentaires brefs qui explique *pourquoi* votre code est comme il est, et non *comment* il est. 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.

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 !

- Le projet utilise **CMake** comme outils de gestion de la compilation. La définition du projet pour `cmake` est contenue dans le fichier `CMakeLists.txt`. `cmake` est un méta-outil : il permet de générer différents types de fichiers pour différents outils de gestion de la compilation. Par défaut, `cmake` génère un fichier `Makefile` qui permet de compiler le programme avec la commande `make`. Sous Windows, nous vous conseillons de générer un projet pour **Visual Studio** avec `cmake`. Sur les autres plates-formes, nous vous conseillons d'utiliser l'environnement de développement **CLion** qui utilise directement `cmake` et son `CMakeLists.txt` pour définir le projet.
- `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). Vous pouvez l'utiliser en ligne de commande : `valgrind [options] <executable> <paramètres de l'exécutables>`

Ou plus simplement depuis **CLion** : <Run><Run with Valgrind Memcheck>.

Pour les fuites mémoires, vous constaterez qu'il en existe plusieurs dans libeibase et d'autres bibliothèques. Ces fuites proviennent de fonctions sur lesquelles nous n'avons pas le contrôle, nous ne pouvons donc pas les supprimer. Vous pouvez utiliser l'option --suppressions de valgrind pour cacher ces erreurs.

- Pour travailler à plusieurs en même temps, il est fortement conseillé d'utiliser un gestionnaire de versions. Le plus répandu est git (<https://ensikiwiki.ensimag.fr/index.php?title=Git>).
- 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 !³

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.

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  start, end;

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 utiliser 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.

3. 7g de CO₂ par requête

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.8).

A.1 Dessin des primitives graphiques

Le but de cette première étape est de développer et tester les primitives graphiques nécessaires au dessin des lignes brisées et des polygones pleins. Ces primitives constituent la base de la bibliothèque : il est donc important d'avoir écrit et testé rapidement une première version qu'il sera possible d'améliorer et d'optimiser par la suite.

- Écrivez le programme de la fonction `ei_draw_polyline` déclarée dans `ei_draw.h`. Dans un premier temps, cette fonction ne fait pas le clipping et dessine une ligne "canonique" de Bresenham (voir la section 3.1.2).
- Compilez et exécutez le programme de test `lines.c`, qui dessine une ligne simple.
- Étendez votre réalisation de `ei_draw_polyline` pour traiter les autres octants, les lignes horizontales et verticales, etc. Enlevez les commentaires sur les appels des autres fonctions dans `lines.c` pour tester votre programme.
- Réalisez et testez un algorithme de clipping pour l'affichage de lignes brisées. Enlevez les commentaires sur les définitions de `clipper` et `clipper_ptr` pour tester votre programme.
- De manière similaire, réalisez la fonction `ei_draw_polygone` qui est déclarée dans `ei_draw.h` et dont le principe est donné en section 3.1.2. Étendez `lines.c` pour tester le dessin de polygones, d'abord non clippés, puis clippés.

A.2 Dessin de boutons en relief

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

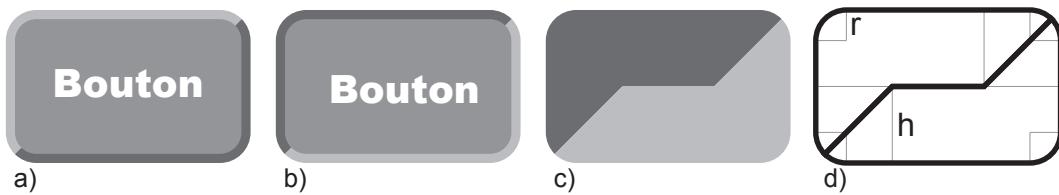


FIGURE A.1 – Dessin des boutons. a) effet “relevé”, b) effet “enfoncé”, c) les deux formes du fond, d) schéma : r est le rayon des arrondis, h est la moitié de la hauteur ou de la largeur du bouton (le plus petit des deux).

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.3 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()`¹. 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.4 Crédit de la classe de widget “frame”

Tout widget appartient à une classe. Le widget 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 `frame` dont on peut notamment changer la couleur de fond. Outre le widget racine, le programme crée également un autre widget `frame` ayant un effet de relief (voir la figure 4.1b).

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 2.2.3. 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.

1. Attention, sous Mac OS, `getchar()` bloque l’application et ne permet pas l’affichage de la fenêtre.

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.5 Mise en place d'un gestionnaire de géométrie

Il s'agit maintenant de faire une implémentation de la fonction `ei_place`. 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.

Implémentez les fonctions `ei_place(...)` et `ei_placer_run(...)`. 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.6 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.7 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 trois aspects principaux : la définition d'une fonction traitant interne et son association à la classe de widget `button`, la création d'un traitant externe et son association à l'interacteur (ceci est fait dans "`button.c`"), 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.

- 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). Si ce n'est pas un événement situé, le traitant concerné est celui qui a été défini par le programmeur par un appel à `ei_event_set_default_handle_func(...)`. 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 est 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.8 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 document, et qui ne sont pas présent dans le "Code d'applications fournies" (section 4.2). Par exemple, est-ce que votre bibliothèque est capable de gérer correctement le cas illustré sur la figure A.2 ? 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 ?

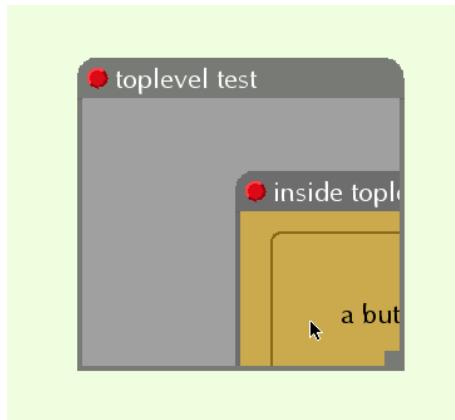


FIGURE A.2 – Un exemple non documenté (si ce n'est cette image).

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”, 27
API, 21, 33
application, 6, 12
attributs, 21

bibliothèque, *voir* bibliothèque logicielle, 6
bibliothèque logicielle, 6
boucle principale, 12, 30
bouton, 10, 24, 45
 dessin, 43
bouton radio, 35
Bresenham, 16
button, *voir* bouton
“button.c”, 34

cadre, 24, 43, 44
callback, *voir* traitant
champ de saisie, 36
classe, *voir* interacteur (classe d’interacteur)
clipping, 9, 10, 18
 analytique des lignes, 19
 analytique des polygones, 20
code de touche clavier, 28
critères d’évaluation, 36

descendant, *voir* interacteur (hiérarchie)
doxygen, 13, 39
drawfunc, *voir* ei_widget_class_t (drawfunc)
déplacement, 28

ei_app_create, 30, 44
ei_app_invalidate_rect, 29
ei_app_quit_request, 30, 45
ei_app_redraw, 30
ei_app_run, 30, 44
ei_axis_set_t, 24
ei_button_configure, 21, 24, 26, 27
ei_callback_t, 27
ei_color_t, 14
ei_copy_surface, 15
“ei_draw.h”, 13, 15, 24
ei_draw_polygone, 15
ei_draw_polyline, 15
ei_draw_text, 15
ei_ev_mouse_buttondown, 28
ei_ev_mouse_buttonup, 28
ei_ev_mouse_move, 28
ei_event_set_default_handle_func, 27

ei_event_t, 28
ei_eventtype_t, 27
 ei_ev_keydown, 27
 ei_ev_keyup, 27
 ei_ev_mouse_buttondown, 27
 ei_ev_mouse_buttonup, 27
 ei_ev_mouse_move, 27
ei_fill, 15
ei_font_t, 24
ei_frame_configure, 24, 44
ei_geometry_param_t, 21
ei_map_rgba, 14
ei_place, 25, 26, 45
 anchor, 26
“ei_placer.h”, 25
ei_relief_t, 24
ei_surface_t, 13
ei_toplevel_configure, 24
“ei_widget.h”, 24
ei_widget_create, 23, 30
ei_widget_t, 21
 children_head, 25
 children_tail, 25
 next_sibling, 25
 parent, 25
 placer_params, 25
 wclass, 23
ei_widgetclass_allocfunc_t, 23
ei_widgetclass_drawfunc_t, 23
ei_widgetclass_register, 23, 30, 44
ei_widgetclass_t, 22, 44
 allocfunc, 23, 44
 drawfunc, 29, 44
 releasefunc, 44
 setdefaultsfunc, 23, 44
ensiwiki, 39
évaluation de performances, 41
évaluation du projet, 36
événement, *voir* gestionnaire d’événements

fenêtre système, 13
fenêtre système, 8, 12, 44
fichiers, 37
focus clavier, 36
fonte, 24
frame, *voir* cadre
“frame.c”, 34, 43
“freq_counter.h”, 41

fréquence d'appel, 41
 frequency_counter_t, 41
 frequency_init, 41
 frequency_tick, 41

 gestion de l'application, 8, 12
 gestionnaire d'événements, 6, 8, 11, 27, 45
 événement non situé, 7
 événement situé, 7, 27
 picking, 11
 gestionnaire de géométrie, 6, 8, 12, 25, 34, 45
 absolu, 12, 26
 placeur, 12, *voir* placeur
 relatif, 12, 26

 handler, *voir* traitant
 "hello_world.c", 34
 hw_create_window, 13
 hw_event_wait_next, 30, 45
 hw_image_load, 13
 hw_init, 13, 30
 "hw_interface.h", 9, 13, 24
 hw_interface_update_rects, 13
 hw_now, 41
 hw_surface_create, 13, 30
 hw_surface_free, 13
 hw_surface_get_buffer, 14
 hw_surface_get_channel_indices, 14
 hw_surface_lock, 13, 14
 hw_surface_unlock, 13
 hw_surface_update_rects, 29
 hw_text_create_surface, 13

 image, 13, 24
 interacteur, 6, 8, 10
 classe d'interacteur, 10, 21, 23, 44
 clipping, 9
 hiérarchie, 10, 25
 identifiant, 11
 ordre, 9, 10, 25
 racine, 10, 12, 30, 44
 interacteur actif, 27
 interface système et matériel, 8
 IUG, 5

 "libei.a", 33
 "libeibase.a", 9, 13
 "lines.c", 34

 "minimal.c", 34
 modules, 8

 offscreen de picking, 11, 13, 30
 ordre, *voir* interacteur (ordre)

 paramètres d'événement, 28
 parent, *voir* interacteur (hiérarchie)
 picking, *voir* gestionnaire d'événement (picking)

 pixel, 9, 14
 alpha, 14
 représentation en mémoire, 14
 RGB, 14
 RGBA, 14
 RGBX, 14
 transparence, 14
 placeur, 25
 algorithme, 26
 ei_placer_run, 25
 polygone
 dessin, 16
 table des côté actifs, 16
 table des côtés, 17
 polylines, *voir* primitives graphiques (lignes brisées)
 polymorphisme, 21
 primitives graphiques, 6, 9, 15
 lignes brisées, 15
 polygones, 15, 16
 programmation événementielle, 7
 programme principal, 8, 12, 30
 "puzzle.c", 35

 racine, *voir* interacteur (racine)
 redimensionnement, 29
 relief, 24
 remplissage, 15
 root, *voir* interacteur (racine)

 scanline, 16
 SDL2, 33
 soutenance, 37
 surface de dessin, 8, 11, 13
 adresse en mémoire, 14
 allocation, 13
 blocage, 13, 29
 copie, 15
 cycle de mise à jour, 14
 déblocage, 13, 29
 fenêtre système, 13
 libération, 13
 mise à jour, 13, 29

 table des côté actifs, *voir* polygone
 table des côtés, *voir* polygone
 table des pointeurs, 23
 tests, 6, 33
 texte, 13, 24
 toplevel, 10, 24, 25, 28, 34
 touches spéciales, 28
 traitant, 7, 11, 25, 27, 28, 34
 externe, 11, 27
 interne, 11, 27
 traitant par défaut, 7, 27
 transparence, 14

 valeurs par défaut, 21
 widget, *voir* interacteur, *voir* interacteur