Projet Technogique L3
Interfaces Utilisateurs Graphiques

Licence 3 Informatique – Université de Bordeaux

Pierre Bénard, Aurélie Bugeau

2016-2017

Table des matières

1 Introduction **3**
1.1 Objectifs . . . 3
1.2 Fonctionnement . . . 3
1.2.1 Interface système . . . 3
1.2.2 Gestion de l'application . . . 4
1.2.3 Interacteurs . . . 4
1.2.4 Gestionnaire de géométrie . . . 5
1.2.5 Gestion des événements . . . 6

2 Étude des besoins **7**
2.1 Besoins fonctionnels . . . 7
2.2 Besoins non fonctionnels . . . 9

3 Cas d'utilisation et exemple de gestion des événements **10**
3.1 Cas d'utilisation . . . 10
3.2 Un exemple de gestion d'événements : le déplacement . . . 10

4 Architecture **12**

5 Consignes **14**
5.1 Bibliothèque minimale . . . 14
5.2 Extensions . . . 14

6 Organisation **16**

7 Introduction au C++ **17**

1 Introduction

1.1 Objectifs

L'objectif du projet est le développement d'une bibliothèque permettant à un utilisateur de créer facilement une interface utilisateur graphique (IUG). Cette bibliothèque est destinée à des programmeurs qui pourront ainsi facilement générer une interface graphique composée de boutons, zones de texte, zones de saisie, etc.

Le projet et ce document sont fortement inspirés de [1].

1.2 Fonctionnement

La gestion d'une interface utilisateur graphique repose sur la programmation événementielle. Contrairement à la programmation séquentielle où l'exécution est linéaire, la programmation événementielle repose sur des événements dont l'ordre n'est pas connu au moment de l'écriture du programme. Ces événements peuvent être le déplacement de la souris, les clics de la souris ou l'appui sur une touche du clavier. On parle alors d'**intéracteurs**. La réaction du programme par rapport à chacun des événements est gérée par des fonctions de traitement d'événements, définies par le programmeur, appelés **traitants** (*handler* ou *callback* en anglais). Après une phase d'initialisation, le programme principal attend qu'un événement intervienne. La bibliothèque communique alors cet événement au traitant. La bibliothèque se charge enfin de la mise à jour de l'écran à la suite du traitement de tout nouvel événement.

Un programme événementiel se décompose de la façon suivante :

1. Création des interacteurs et définition de leurs attributs (boutons par exemple).
2. Enregistrement de traitants d'événements utilisateur (fonctions de traitement spécifiques au programme).
3. Placement à l'écran des interacteurs via un gestionnaire de génométrie.
4. Boucle Principale – tant que pas de demande d'arrêt du programme :
   (a) Dessiner à l'écran les mises à jour nécessaires.
   (b) Attendre un événement.
   (c) Analyser l'événement pour trouver le ou les traitants associés.
   (d) Appeler le ou les traitants associés.
5. Libérer toute la mémoire.

Cet algorithme correspond au programme principal permettant la création d'une interface utilisateur graphique. C'est donc l'utilisateur de la bibliothèque qui est en charge d'écrire ce programme. Il fait appel à des fonctionnalités gérées par la bibliothèque graphique comme expliqué ci-après.

1.2.1 Interface système

L'accès à l'affichage graphique ainsi qu'aux événements de bas niveau se fait grâce à l'utilisation d'un module d'interface matériel. Pour des raisons de simplicité, l'ensemble de l'application et des interacteurs sera géré dans une unique fenêtre « système ». La bibliothèque dessinera des sous-fenêtres à l'intérieur de cette dernière. Les fonctionnalités de l'interface système/matérielle sont les suivantes :

— initialisation de la fenêtre graphique système,
— gestion des surfaces de dessin, c'est-à-dire des zones mémoires où l'application « dessine » les interacteurs, pour les copier finalement sur la fenêtre système,
— dessins élémentaires (lignes, polygones, textes),

— attente des événements utilisateur (appui sur une touche, clic souris, etc.),
— mesure du temps.

1.2.2 Gestion de l'application

Le module de gestion de l'application est en charge des principales étapes du cycle de vie de l'application. Le module offre des fonctions qui permettent d'initialiser l'application, de lancer la boucle principale, et finalement de libérer les ressources utilisées par l'application. En interne, le module alloue et initialise les structures de données que la bibliothèque gère pour toute 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. Le module offre également :

— un point d'accès pour l'interacteur racine de l'application. Ceci permet de donner un parent aux interacteurs de plus haut niveau créés par le programmeur de l'application,
— une fonction permettant de demander la terminaison de l'application, c'est-à-dire de sortie de la boucle principale. Le programmeur appelle en général cette fonction en réaction à une action spécifique de l'utilisateur, comme l'appui sur un bouton « Quitter ».

1.2.3 Interacteurs

Les interacteurs (ou *widgets*) sont les objets graphiques interactifs de l'application. La bibliothèque graphique doit pouvoir dessiner des interacteurs, modifier leur apparence en fonction des actions de l'utilisation, c'est-à-dire des événements, et appliquer les traitants correspondants.

Dessin des interacteurs Le dessin des interacteurs fait appel à des primitives graphiques : dessin de ligne, remplissage de polygones, dessin de texte, etc. Ces primitives sont utilisées à chaque fois qu'il faut redessiner une partie de l'écran. Par exemple, quand on déplace une fenêtre à l'écran, la bibliothèque doit redessiner ce qui était « en dessous » à son ancien emplacement, et la dessiner au nouvel emplacement, et ce à chaque micro-déplacement de la souris. Ce processus a lieu en général 60 fois par seconde pendant le déplacement de la souris. De plus, le dessin d'un polygone, même de taille modeste, peut nécessiter la modification de dizaines de milliers de pixels. Il en résulte que les algorithmes des primitives graphiques doivent être extrêmement optimisées. Dans le cadre de ce projet, les fonctions de dessin des primitives graphiques vous sont données.

Une interface graphique doit être capable de dessiner plusieurs types d'interacteurs. Ces différents types d'interacteurs partagent un certain nombre de caractéristiques et fonctionnalités communes. Chaque interacteur a notamment besoin d'attributs pour mémoriser son état, d'une fonction de configuration permettant de modifier l'état de l'interacteur, et d'une fonction de dessin qui se charge de dessiner l'interacteur à l'écran. Cependant, la réalisation de ces différentes fonctions dépend du type d'interacteur.

La figure 1a présente un exemple d'interface contenant des interacteurs de type bouton.

Étiquette des interacteurs La bibliothèque doit appeler le traitant correspondant au type d'événement qui a lieu sur un interacteur. Pour cela, chaque interacteur possède une étiquette (*tag*), qui peut être de deux types (« nom de sa classe » ou « all »). Le tag du nom de la classe permet de définir des comportements au niveau global d'une classe d'interacteurs plutôt qu'au niveau d'une instance d'un interacteur spécifique. Par exemple, quand on clique sur un interacteur de type bouton, on souhaite que le bouton apparaisse avec un relief enfoncé. Le tag « all » permet de réaliser des liens qui se déclenchent à chaque fois que l'événement associé intervient, qu'il y ait ou non un interacteur concerné par celui-ci. Par exemple, pour quitter l'application lorsque l'utilisateur appuie sur la touche « q ».

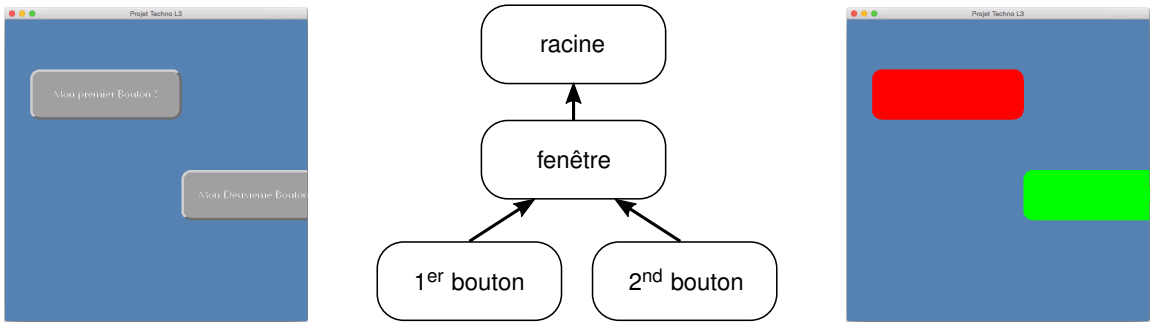

Figure 1 – (a) Un exemple d'interface contenant deux interacteurs de type « bouton ». (b) La hiérarchie des interacteurs correspondant. (c) l'*off-screen* de *picking*.

Hiérarchie d'interacteurs Les interacteurs ont la propriété fondamentale d'être organisés hiérarchiquement sous forme d'arbre. Un interacteur possède toujours un et un seul interacteur parent et peut avoir des interacteurs descendants. Par convention, chaque interacteur ne peut être dessiné qu'à l'intérieur des limites de son parent. C'est pour cela, 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 (*clipping*) sur ses limites (Figure 1a). Au sommet de la hiérarchie, on considère un interacteur racine (« root »).

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.

Les descendants d'un interacteur doivent 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 à 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.

Clipping Toutes les fonctions de dessin des primitives graphiques (rectangles, polygone, etc.) prennent en paramètre un rectangle de clipping : la primitive graphique est dessinée uniquement dans les limites de ce rectangle.

1.2.4 Gestionnaire de géométrie

Définir précisément la position et la taille des widgets (interacteurs) dans leur parent peut être une tâche complexe. Pour un programmeur, il est souvent préférable de préciser la position et la taille d'un widget en fonction de la position et de la taille de son parent. C'est le rôle d'un gestionnaire de géométrie d'enregistrer les contraintes de position et taille exprimées par le programmeur et d'être capable de les traduire en position et taille absolue dans le parent. C'est notamment nécessaire lorsque le parent est redimensionné ou lorsqu'un interacteur est détruit et que la place qu'il libère doit être redistribuée aux interacteurs restant dans le parent.

1.2.5 Gestion des événements

En programmation événementielle, le comportement du programme est réalisé par les fonctions qui traitent les événements générés par les actions de l'utilisateur. On distingue deux catégories de traitant. Les **trai-**

tants internes à la bibliothèque qui sont responsables du comportement standard des interacteurs et sont fournis par la bibliothèque (par exemple, l'apparence « enfoncée » lors de l'appui sur un bouton). Les **traitants externes** qui sont responsables du comportement de l'application et sont fournis par le programmeur qui utilise la bibliothèque (action à réaliser lors de l'appui sur chaque bouton). Le lien entre événement utilisateur et traitant est géré par un gestionnaire d'événement.

Picking Lorsque l'utilisateur appuie sur le bouton de la souris, ou simplement lorsqu'il déplace la souris, le système d'exploitation envoie un événement à l'application qui précise la position du pointeur. La bibliothèque doit alors vérifier quel interacteur est situé sous la souris afin de communiquer cet événement, dit situé, au traitant correspondant. Le gestionnaire d'événement doit donc choisir l'interacteur concerné. On appelle cette fonction le « picking ».

La réalisation du picking se fait en dessinant l'interacteur dans une surface dédiée, appelée l'*off-screen* de *picking*, qui ne sera jamais affichée à l'écran. Au lieu d'utiliser différentes couleurs dans le dessin de l'interacteur (pour le fond, le texte, etc.), on utilise une seule « couleur » qui correspond en fait à un numéro d'identifiant propre à l'interacteur (cf. les couleurs vives de la figure 1c). À toute opération de dessin à l'écran correspondra donc une opération de dessin dans l'off-screen de picking. L'opération de picking devient alors triviale : par construction, l'identifiant de l'interacteur concerné par un clic à la position (x, y) est simplement la valeur du pixel (x, y) dans l'off-screen 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, la première correspondant à l'écran, l'autre à l'off-screen de picking.

Couplage événement-traitant ou *binding* Le module de gestion d'événement offre des fonctions pour lier un couple événement-traitant (*bind*) ou supprimer ce lien (*unbind*). Ces fonctions seront appelées à l'initialisation de la bibliothèque pour l'enregistrement des traitants internes, elles sont aussi appelées depuis le programme principal où le programmeur de l'application enregistre les traitants externes. Ces fonctions peuvent également être appelées depuis les traitants eux-mêmes pour modifier dynamiquement le comportement de l'interface.

Fonction traitant d'événements ou *callback* Lorsqu'un événement a lieu, la bibliothèque est en charge d'appeler le ou les traitants liés, en leur passant en paramètre un événement. Il permet au traitant de savoir quel est le type de l'événement, mais aussi de recevoir des paramètres d'événements. Par exemple, 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 du bouton de souris qui a été enfoncé ou relâché.

2 Étude des besoins

Cette section décrit l'analyse fonctionnelle du projet. Elle définit donc les fonctionnalités que votre bibliothèque devra offrir et leurs caractéristiques.

2.1 Besoins fonctionnels

La liste des besoins fonctionnels de la bibliothèque à réaliser est donnée ci-dessous.

Interface système Les fonctionnalités de l'interface système/matérielle vous est fourni dans le cadre de ce projet sous la forme d'une bibliothèque (`libeibase.a`) et ne fait donc pas partie de ce que vous devez implémenter. Vous pourrez au besoin accéder directement aux fonctions offertes par la bibliothèque *Allegro* [1] sous-jacente.

Dessin des interacteurs Les interacteurs ont en commun leurs trois premiers attributs de présentation : leur taille (que le gestionnaire de géométrie peut satisfaire ou non, en fonction d'autres contraintes), leur couleur de fond et la taille de leur bordure en pixels. À minima, les différents interacteurs que devra dessiner la bibliothèque sont :

Fenêtre ou *Toplevel*
Un interacteur de type `toplevel` est constitué 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. Outre les 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 qui spécifie si la fenêtre est redimensionnable ou non, et si oui, sur quels axes (horizontal et/ou vertical),
— la taille minimale de la fenêtre, contrainte dont devra tenir compte le gestionnaire de géométrie en cas de redimensionnement.

Cadre ou *Frame*
Un interacteur de type `frame` est un cadre rectangulaire qui peut être utilisé pour dessiner un simple cadre, ou un cadre contenant du texte ou une image. Les attributs propres aux cadres sont :
— un type énuméré spécifiant le relief du widget. Le relief donne un aspect 3D (enfoncé, relevé, plat) au cadre. Le relief est dessiné sur le bord du widget. Si la taille de la bordure spécifiée est de largeur nulle, alors aucun relief n'est dessiné.
— un texte, ainsi que les attributs spécifiant son aspect,
— 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 l'interacteur au cas où l'image serait plus petite que l'interacteur.

Bouton
Un interacteur de type `button` sera dessiné comme un cadre rectangulaire interactif contenant un texte ou une image. L'appui sur un bouton doit lui donner l'aspect « enfoncé » et le retour à une apparence en relief quand le clic est terminé. Un bouton possède les mêmes attributs qu'un cadre :

1. http://liballeg.org/

relief, texte ou image pouvant être dessinés dans le bouton. Un seul attribut est spécifique aux boutons, le rayon des arrondis aux angles du bouton.

Hiérarchie des interacteurs Comme indiqué en section 1.2.3, les widgets sont organisés hiérarchiquement, et tout widget (sauf le widget racine) doit avoir un parent. La bibliothèque devra donc gérer la hiérarchie des interacteurs en maintenant pour chaque widget la liste triée de ses descendants. 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 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 sa 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.

Enfin, la gestion du clipping sera intégrée en respectant la hiérarchie des interacteurs.

Gestion des événements Un gestionnaire d'événement, basé sur les étiquettes, et le picking doivent être intégrés. Afin de pouvoir gérer à la fois l'exécution de traitants internes et de traitants externes, la bibliothèque devra être capable d'appeler plusieurs traitants par événements.

Les différents événements et traitants internes que devra gérer la bibliothèque sont :
- 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

Gestion de la géométrie La bibliothèque doit intégrer un gestionnaire de géométrie appelé « placeur » (ou *placer* en anglais). C'est un gestionnaire simple qui permet de placer un interacteur en exprimant sa position et sa taille de façon absolue et/ou relative à son parent. Un gestionnaire de géométrie peut avoir à résoudre des contraintes incompatibles ; le placeur doit donc gérer une priorité sur ces contraintes.

Gestion de l'application La bibliothèque doit intégrer un gestionnaire de l'application permettant d'initialiser l'application, de lancer la boucle principale, et finalement de libérer les ressources utilisées par l'application.

Programme principal La bibliothèque ne contient pas de programme principal : c'est le programmeur d'application graphique qui écrit un programme principal spécifiquement pour une application donnée. Afin de pouvoir tester la bibliothèque, plusieurs exemples de programmes principaux sont fournis. D'autres devront être ajoutés.

Documentation La bibliothèque doit être bien documentée à l'aide de l'utilitaire *Doxygen* [2].

2. http://www.stack.nl/~dimitri/doxygen/

2.2 Besoins non fonctionnels

Gestion de l'affichage Durant la durée de vie d'une application graphique, l'écran doit être mis à jour de nombreuses fois pour de multiples raisons. Par souci d'optimisation et de simplification, les mises à jour à l'écran ne sont pas réalisées immédiatement. Elles sont programmées pour être réalisées plus tard lors de la phase de re-dessin dans la boucle principale. Ainsi, lorsqu'une mise à jour est nécessaire, la bibliothèque mémorise le rectangle de l'écran qui doit être mis à jour. Dans la boucle principale, après avoir traité un événement, la bibliothèque se charge de mettre à jour tous les rectangles qui ont été enregistrés.

Langage de programmation Le langage de programmation du projet sera le C++. Le standard C++11 sera utilisé, notamment pour les fonctions *callback*.

Architecture logicielle La bibliothèque devra respecter l'architecture décrite en section 4 et les fichiers d'entête fournis.

3 Cas d'utilisation et exemple de gestion des événements

Plusieurs cas d'utilisation de la bibliothèque sont décrits dans cette section. Les programmes principaux associés vous seront fournis. Cette section présente également un exemple de gestion des événements.

3.1 Cas d'utilisation

minimal L'application `minimal` est la seule que vous pouvez compiler et exécuter dès le début du projet : elle utilise uniquement les fonctions qui vous sont fournies dans `libeibase`. L'application se contente de remplir la fenêtre système en rouge et de dessiner un polygone à l'intérieur.

frame L'application `frame` affiche un simple cadre. Elle est constituée d'un interacteur racine ayant pour descendant un interacteur de type frame, de taille fixe. Son positionnement est définit de façon absolue à l'intérieur du widget racine. Cette application ne gère pas les événements, il n'y a donc aucun moyen de la quitter, si ce n'est en tuant le processus.

button Cette application est similaire à la précédente, mais le widget frame est remplacé par un interacteur button. 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,
— sortie de l'application lors du clic sur la croix rouge.

toplevel Cette application introduit un interacteur de la classe `toplevel` : une fenêtre avec une barre d'en-tête qui permet de la déplacer sur l'écran. Elle possède un bouton en bas à droite. 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.

3.2 Un exemple de gestion d'événements : le déplacement

Le déplacement d'un interacteur nécessite les étapes suivantes :

1. **Lien entre appui sur la souris et déplacement**
   À l'initialisation, un lien entre l'événement « appui sur le bouton de la souris » au niveau d'une `toplevel` et le traitant « déplacement » est créé par *binding*.
2. **Démarrage de l'action déplacement**
   L'utilisateur vient de cliquer avec le pointeur de souris sur la barre d'en-tête de la `toplevel`. Le traitant associé (déplacement) à « appui sur le bouton de la souris » pour la classe `toplevel` est appelée. L'action de déplacement démarre. La position de l'interacteur sera maintenant contrôlée par la souris. Il est donc nécessaire de remplacer sa gestion de géométrie actuelle par une gestion en positionnement absolu.
3. **Enregistrement de deux nouveaux traitants**
   A partir de ce moment, l'interacteur doit être tenu au courant des événements de type « relachement du bouton » et « déplacement de la souris ». Deux nouveaux traitants doivent être associés à ces événements (à nouveau par *binding*). On pourra utiliser le tag « all », puisqu'aucune autre interaction ne peut avoir lieu pendant le déplacement de la fenêtre : nous sommes donc intéressés par ces événements quel que soit le widget concerné.

4. **Déplacement de la souris alors que le bouton est toujours enfoncé**
   Chaque mouvement élémentaire de la souris déclenche maintenant l'appel du traitant liée à l'événement « déplacement de la souris ». La position du curseur est utilisée pour calculer la nouvelle position de l'interacteur. 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 (`root`). Par contre, le positionnement absolu d'un interacteur, grâce au gestionnaire de géométrie, s'exprime dans le repère du parent de cet interacteur.
5. **Fin de l'action par détection de l'événement « relâchement du bouton »**
   L'utilisateur relâche le bouton de la souris. Le traitant associé à l'événement « relâchement du bouton » est appelée. Ce traitant doit simplement désabonner les traitants liés aux événements « relâchement du bouton » et « déplacement de la souris ».

4 Architecture

La bibliothèque développée **doit respecter** l'architecture présentée en figure 2. Un squelette de l'application vous est fourni dans le cadre de ce projet. Pour plus d'information, vous pouvez consulter la documentation associée. Les fichiers d'entête seront à compléter, mais le prototype des fonctions existantes **ne doit pas être modifié** sans l'accord préalable du client et après discussion argumentée.

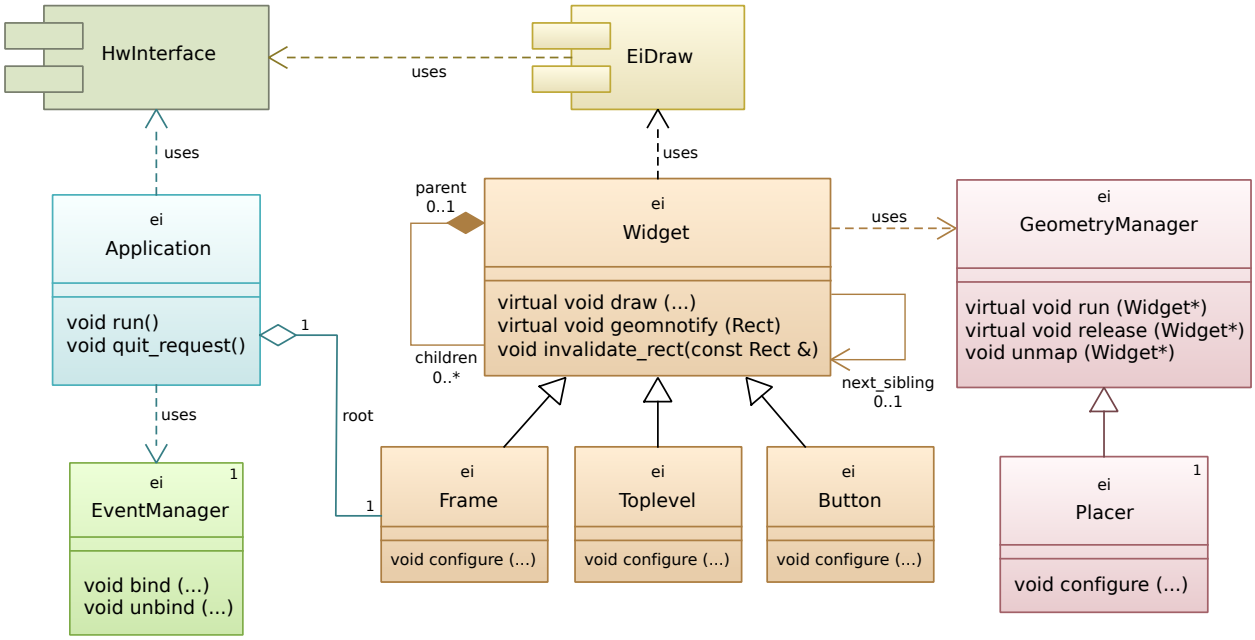

Figure 2 – Architecture du programme

Toutes les types et classes de la bibliothèque font partie de l'espace de nommage (*namespace*) `ei`.

Classe `Application` À sa construction, cette classe initialise les ressources matérielles et crée la fenêtre principale en utilisant `HwInterface`. Elle alloue également la racine (`root`) de la hiérarchie de widgets. Elle expose les méthodes principales : `run` pour lancer la boucle principale et `quit_request` pour demander sa terminaison. Elle définit également une méthode `invalidate_rect` permettant de spécifier un rectangle de l'écran qui doit être mis à jour, comme indiqué en section 2.2.

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

Le bon usage du rectangle de clipping est essentiel pour éviter les 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. Il se peut également que de grandes zones de l'écran soient redessinées plusieurs fois. Avant de parcourir tous les rectangles de re-dessin, la bibliothèque pourra appliquer des stratégies d'optimisation, comme par exemple fusionner les rectangles ayant une intersection importante.

Classe `Widget` et ses dérivées La classe `Widget` représente un intéracteur abstrait dans la hiérarchie. Tout widget – à l'exception de la racine – possède un parent (pointeur `parent` non-nul). Il peut également posséder des descendants sous la forme d'une liste chaînée (pointeurs `next_sibling`, `children_head` et `children_tail`). La classe définit en particulier la méthode `draw` pour dessiner le widget et ses éventuels descendants, et la fonction `geomnotify` pour mettre à jour sa position calculée par le gestionnaire de géométrie.

Les classes dérivées `Frame`, `Toplevel` et `Button` spécialisent `Widget` en surchargeant la méthode `draw` et en exposant une nouvelle méthode `configure` propre à chacune d'elles. Cette méthode peut accepter de nombreux paramètres afin de définir précisément chaque détail de l'aspect de l'intéracteur. Il serait fastidieux pour le programmeur d'applications de devoir spécifier ces paramètres à chaque fois qu'il veut modifier un seul aspect de l'intéracteur.

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.

Classe `EventManager` Il ne doit en exister qu'une instance, cette classe suit donc le patron de conception « singleton ». Elle expose les méthodes `bind` et `unbind` afin d'ajouter et de retirer un traitants (*callback*) à un widget ou à un tag pour un type d'événement donné. La classe `Application` et les différents programmes de test feront appel à l'instance d'`EventManager` pour enregistrer leurs traitants.

Classes `GeometryManager` et `Placer` La classe abstraite `GeometryManager` définit trois méthodes communes à tout gestionnaire d'événement : `run` pour exécuter le gestionnaire sur un widget donné (et ses descendants), `release` et `unmap` pour oublier un widget et ne plus l'afficher à l'écran.

La classe `Placer` surcharge la fonction virtuelle pure `run` et expose la méthode `configure` permettant de paramétrer un widget, avec le même mécanisme de valeurs par défaut que les intéracteurs. Cette méthode prend en paramètre les positions et tailles absolues désirées `(x, y, width, height)`, ainsi que les positions et tailles relatives désirées `(rel_x, rel_y, rel_width, rel_height)`, exprimés dans le repère du parent du widget, dont l'origine est l'angle en haut à gauche du parent.

Les paramètres relatifs sont représentés par des nombres flottants. Une ordonnée relative de 0.0 correspond au côté haut du parent, 1.0 à son côté bas, et 0.5 à son centre. Une hauteur relative de 0.5 correspond à la moitié de la hauteur du parent. Les paramètres de position `(x, y, rel_x, rel_y)` définissent la position ponctuelle d'un seul pixel du parent, mais le widget a une surface rectangulaire de plusieurs pixels. Il faut donc aussi spécifier comment le widget s'attache, ou s'ancre, sur cette position. C'est le rôle du paramètre `anchor` : une ancre Nord-Ouest, 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.

Le placeur doit également gérer une priorité sur les contraintes de taille d'un interacteur : les paramètres de `Placer::configure` sont prioritairement respectés. S'ils ne sont pas fournis, c'est la taille demandée qui est respectée (paramètre `requested_size` de l'appel à `Button::configure`). Si ce paramètre n'est pas non plus fourni, alors c'est la taille par défaut qui est respectée.

5 Consignes

L'objectif principal du projet est le développement de la bibliothèque permettant la programmation d'interface graphique.

5.1 Bibliothèque minimale

Dans un premier temps, il vous faudra compléter le squelette de l'application fourni de façon à répondre à l'analyse des besoins (section 2).

Le répertoire `tests` contient les programmes principaux permettant de tester les cas d'utilisation présentés dans la section 3.1. Nous vous conseillons de faire fonctionner ces programmes dans l'ordre où ils vous sont présentés en section 3.1.

Des tests unitaires de vos méthodes devront être réalisés avec *Catch* [3].

5.2 Extensions

Des extensions de la bibliothèque sont possibles. Vous en réaliserez au moins une. 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 ($\star$) à très compliquée ($\star\star\star$).

Interacteur bouton radio ($\star\star$) Cette extension consiste à ajouter une classe de widget « bouton radio » (ou *radio-button* 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.

À la différence des classes de widget `frame`, `button`, et `toplevel`, nous ne fournissons pas l'interface de programmation des `radiobutton`. 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.

Gestion des tags des widgets ($\star\star$) La gestion des événements utilise le concept de tag pour limiter l'appel des traitants aux widgets qui possèdent un tag particulier. Cette extension consiste à ajouter une méthode à la bibliothèque pour permettre au programmeur de définir des nouveaux tags et d'affecter lui-même les tags qu'il désire à un widget particulier. Cette méthode permet de donner des comportements à des widgets en leur donnant simplement un tag. Par exemple, le programmeur peut vouloir implémenter des infos bulle (« tooltips »).

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

3. https://github.com/philsquared/Catch

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

Interacteur 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`. 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 ».

Gestionnaire de géométrie en grille (⋆⋆⋆) Cette extension consiste à ajouter un nouveau gestionnaire de géométrie qui place et dimensionne les widgets dans une grille, le `griddeur`. Le gestionnaire de géométrie grille est le gestionnaire de géométrie le plus flexible et le plus facile à utiliser. Il divise logiquement le widget parent en lignes et colonnes dans un tableau à deux dimensions. Là aussi, c'est à vous de définir l'interface de programmation du `griddeur`. Vous pourrez vous inspirer de l'interface de programme du `placer`.

6 Organisation

— Date de rendu : rendu de la bibliothèque minimal en milieu de semestre. Rendu de la bibliothèque avec fonctionnalités optionnelles (extension) : fin du semestre 2
— Notation prévue : code, rapport et documentation du code, soutenance
— Travail en groupes : 5 groupes de 4
— Chaque groupe devra travailler sur un `git` partagé créé sous le gestionnaire de projet *Savane* du Cremi [4] ; le client devra y avoir accès.
— Des tests unitaires et du profiling du code devronts être réalisés tout au long du projet. Pour plus d'informations sur les bonnes pratiques de la programmation, vous pouvez vous référer aux transparents mis à disposition sur la page moodle du projet.

4. https://services.emi.u-bordeaux.fr/projet/

7 Introduction au C++

Le C++ est un langage de programmation orienté objet tel que le Java. Un programme en C++ a une syntaxe similaire à un programme en C, le C étant un langage de programmation impératif. On retrouve ainsi la fonction `main`, les fichiers *headers* et les pointeurs. Quelques particularités du C++ par rapport au Java et au C vous sont ici présentées. Pour plus de détails sur ces notions et sur le C++ en général, veuillez vous référer au site : http://en.cppreference.com/w/cpp/language.

Compilation Comme avec le C, la compilation génère un fichier binaire directement utilisable sur la machine sur laquelle il a été créé. Pour l'utiliser sur une autre machine, il faut généralement recompiler le programme. Avec Java, la compilation d'un code source produit un code intermédiaire (et non binaire) qui a besoin de la JVM (*Java Virtual Machine*) pour être interprété. Ce code est portable sur toute machine disposant de la JVM.

Allocation dynamique L'allocation dynamique consiste à réserver de l'espace mémoire sur le tas contrairement à l'allocation statique qui se fait sur la pile. Rappel : tout objet alloué dynamiquement par le programmeur doit être libéré. En C++, les opérateurs d'allocation et de désallocations sont `new` et `delete` (`new[]` et `delete[]` pour les tableaux). Ces opérateurs remplacent les fonctions `malloc` et `free` du C. Un exemple est montré dans le code 3.

Référence Le langage C permet le passage des arguments aux fonctions par **valeur** ou par **adresse** (ou par pointeur). Le C++ permet en outre le passage par référence.

Une référence est un synonyme d'une autre variable. Elle permet de manipuler une varibale sous un autre nom que celui sous lequel elle a été déclarée. Voici un exemple :

Code 1 – Référence

```
int i = 0;
int *pi = &i;   // Pointeur sur la variable i
*pi = *pi+1;    // Manipulation de i via pi
int &ri = i;    // Référence sur la variable i
ri = ri+1;      // Manipulation de i via ri
```

Une variable et sa référence ont la même adresse. Utiliser une référence pour manipuler un objet revient donc exactement au même que manipuler un pointeur constant sur cet objet. Ainsi employées, les références reviennent à utiliser les pointeurs mais avec une simplification dans l'écriture. Dans l'exemple précédent, la référence `ri` revient à manipuler la valeur contenue à l'adresse de la variable `i` et s'utilise comme l'expression `*pi`.

Une référence peut-être passée en paramètre d'une fonction. Elle permet la modification de la variable définie dans la fonction appelante, sans pour autant utiliser explicitement les pointeurs.

Code 2 – Référence

```
void f(int &i) {
    i=i+1;
}

int main(void) {
    int i=0;
    f(i);    // i vaut 1 après l'appel de la fonction f
    return EXIT_SUCCESS;
}
```

Pour des raisons de temps de calcul et de gain en espace mémoire, il est conseillé de passer par référence tous les paramètres dont la copie peut prendre beaucoup de temps ou d'espace.

Constructeur / destructeur Les constructeurs et destructeurs sont deux méthodes particulières qui sont appelées respectivement à la création et à la destruction d'un objet. Toute classe `T` a un constructeur `T::T()` et un destructeur `T::~T()` par défaut, fournis par le compilateur. Ce constructeur par défaut réserve l'espace mémoire nécessaire à l'objet mais n'initialise pas les valeurs de ses attributs et n'alloue pas la mémoire lorsque les attributs sont des pointeurs. Le destructeur par défaut libère la mémoire occupé par l'objet, mais pas celle alloué dynamiquement pour ses attributs. Il est donc souvent nécessaire de redéfinir les constructeurs et destructeurs par défaut afin de gérer certaines actions qui doivent avoir lieu lors de la création d'un objet et de sa destruction. Le compilateur défini également implicitement un constructeur par copie de la forme `T::T(const T&)`.

Un constructeur se définit comme toute méthode (fonction), mais il doit porter le nom de la classe et n'a pas de type de retour. Le corps du constructeur peut contenir, avant l'ouverture de l'accolade, une liste d'attributs à initialiser précédée du caractère « : » (cf. code 4, ligne 1). S'il n'est pas précisé, le constructeur par défaut est appelé automatiquement lors de l'instanciation de l'objet. Le destructeur est appelé automatiquement lors de sa destruction.

Plus d'info : http://en.cppreference.com/w/cpp/language/initializer_list

Un exemple de classe avec trois constructeurs et un destructeur :

Code 3 – Fichier d'entête

```
class Rectangle
{
public:
    // Constructors
    Rectangle();                    // overrides the default constructor
    Rectangle(double w, double h); // constructor with 2 parameters
    Rectangle(const rectangle &);  // overrides the implicitly-declared copy
        constructor

    // Destructor
    ~Rectangle();

private:
    double width, height;
};
```

Code 4 – Fichier source

```
Rectangle::Rectangle() : width(0), height(0) {}

Rectangle::Rectangle(double w, double h) {
    width = w;
    height = h;
}

rectangle::Rectangle(const Rectangle &r) {
    width = r.width;
    height = r.height;
}

~Rectangle() { }
```

Code 5 – Fonction main

```
int main(void) {
    Rectangle r(10,20);
    Rectangle* tabRect = new Rectangle[10];
    Rectangle* pRect = new Rectangle(10,20);
    delete[] tabRect;
    delete pRect;
}
```

Pointeur `this` Pour que les méthodes d'une classe puissent accéder à une instance de leur classe, le compilateur leur transmet implicitement comme premier argument un pointeur sur ces données : `this`. Le pointeur `this` représente l'objet lui-même (voir Code 9). C'est un pointeur constant ; il ne peut pas être modifié.

Plus d'info : http://en.cppreference.com/w/cpp/language/this

Méthode virtuelle et classe abstraite Un méthode déclarée comme virtuelle peut être surchargée dans une classe dérivée. Contrairement aux méthodes non virtuelles, la surcharge est préservée, même s'il n'y a pas d'information de typage à la compilation (cf. code 7). Il est par conséquent conseillé de définir le constructeur de toute classe mère comme publique et virtuel (ou privé et non-virtuel).

Les classes abstraites correspondent aux « interfaces » en Java. Une classe est abstraite si elle possède ou hérite d'une méthode virtuelle pure. Par exemple, la classe `Shape` du code 6 est abstraite, mais pas la classe `Rectangle` car elle surcharge la méthode `draw`.

Code 6 – Fichier d'entête

```
class Shape
{
public:
    Shape();
    virtual ~Shape();

    virtual void draw() = 0; // pure virtual function
}

class Rectangle : Shape
{
public:
    Rectangle();
    ~Rectangle();

    void draw(); // overrides Shape::draw
}
```

Code 7 – Fonction main

```
int main(void) {
    Rectangle r;
    Shape* s = &r;
    s->draw();      // internally, Rectangle::draw() is called
}
```

Plus d'info : http://en.cppreference.com/w/cpp/language/abstract_class

Surcharge d'opérateurs En utilisant la surcharge d'opérateurs, le C++ permet d'appliquer les opérateurs classiques (+, - , /, *, ==, etc.) aux nouvelles classes définies par le programmeur. Il suffit pour cela au programmeur de déclarer les opérateurs associés si les opérateurs fournis par défaut par le compilateur ne conviennent pas. Un exemple de surcharge vous est donné ci-dessous pour une classe rectangle.

Code 8 – Fichier d'entête

```
class Rectangle
{
public:
    Rectangle(double w, double h);
    ~Rectangle();

    // Operator overloading
    Rectangle &operator=(const Rectangle &);
    Rectangle &operator+=(const Rectangle &);

private:
    double width, height;
};
```

Code 9 – Fichier source (extrait)

```
Rectangle &Rectangle::operator=(const Rectangle &r) {
    width = r.width;
    height = r.height;
    return *this;
}

Rectangle &Rectangle::operator+=(const Rectangle &r) {
    width += r.width;
    height += r.height;
    return *this;
}
```

Plus d'info : http://en.cppreference.com/w/cpp/language/operators

Template Comme en Java, les *templates* (types génériques) permettent à une classe ou une fonction de pouvoir s'adapter à plusieurs types sans avoir besoin d'être recopiée ou surdéfinie. Ils permettent de spécialiser du code lors de la compilation, et ainsi optimiser les performances. En outre, ils augmentent considérablement la généricité du code produit, facilitant sa maintenance. En C++ la définition d'une classe ou fonction générique suit la syntaxe : `template < parameter-list > declaration`. À l'utilisation, l'instanciation du *template* peut être explicite ou déduit implicitement par le compilateur (cf. code 10).

Code 10 – Fonction générique

```
template<typename T>
void f(T s) {
    std::cout << s << '\n';
}

int main(void) {
    f<double>(1); // instantiates and calls f<double>(double)
    f<>('a');     // instantiates and calls f<char>(char)
    f(7);         // instantiates and calls f<int>(int)
}
```

Plus d'info : http://en.cppreference.com/w/cpp/language/templates

Containers La bibliothèque standard C++ définit un ensemble de *containers* (tableaux statiques et dynamiques, listes, tables de hachage, etc.) sous la forme de classes génériques. Le code 11 donne un exemple d'utilisation du container `vector`.

Code 11 – Containers

```
#include <iostream>
#include <vector>

int main(void) {
    // Create a vector containing integers
    std::vector<int> v = {7, 5, 16, 8};

    // Add two more integers to vector
    v.push_back(25);
    v.push_back(13);
}
```

Plus d'info : http://en.cppreference.com/w/cpp/container

Singleton Le patron de conception « singleton » permet de s'assurer qu'il n'existera qu'une seule instance d'une classe durant le cycle de vie de l'application. Pour s'en assurer, le constructeur de la classe est déclaré comme privé. En outre, en C++11, on peut empêcher que le programmeur appelle le constructeur par copie et l'opérateur d'affectation (=) en utilisant le mot-clé `delete` (cf. code 12).

Code 12 – Singleton

```
class Singleton
{
public:
    static Singleton& getInstance()
    {
        static Singleton instance; // guaranteed to be destroyed.
        // Instantiated on first use.
        return instance;
    }
private:
    Singleton(); // private constructor, prevents this method to be used

public:
    Singleton(Singleton const&)      = delete; // prevents this method to be used
    void operator=(Singleton const&) = delete; // prevents this method to be used
}
```

Références

[1] F. Bérard, P. Reignier, JS. Franco, E. Frichot, N. Gesbert, D. van Amstel, C. Ramisch, A. Shahwan., Interfaces Utilisateur Graphiques, Projet Logiciel en C, ENSIMAG, 2016