THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)

Discipline ou spécialité :

Informatique

Présentée et soutenue par :
Benjamin Tissoires
le : lundi 26 septembre 2011

Titre :

Conception (instrumenté), réalisation
et optimisation de graphismes
interactifs

Ecole doctorale :
Mathématiques Informatique Télécommunications (MITT)

Unité de recherche :
IRIT

Directeur(s) de Thèse :
Philippe Palanque
Stéphane Conversy (encadrant)

Rapporteurs :
Jean Vanderdonckt
Nicolas Roussel

Autre(s) membre(s) du jury

Jean-Pierre Jessel
François Bérard

Remerciements

Je tiens tout d'abord à remercier Stéphane Conversy, pour ses qualités d'encadrant, à la fois scientifiques et humaines, pour sa patience et pour la grande liberté qu'il m'a laissée au cours de cette thèse. Stéphane m'a beaucoup aidé tout le long de ces années de recherche en m'aiguillant sur des travaux innovants et intéressant, et en sachant me maltraiter pour que j'écrive mes différents articles. Merci donc de m'avoir emmené au bout de cette aventure.

Je tiens aussi à remercier mes collègues de bureaux et de thèse, la Conversy-team, à savoir Christophe Hurter, et Gilles Tabart, avec qui j'ai passé trois des plus belles années de ma vie. Merci particulièrement à Christophe qui par sa motivation sans limites nous a permis d'écrire un très beau logiciel, au nom un peu moins sexy : FromDaDy. Ce logiciel m'a permis d'obtenir une de mes trois meilleures publications, donc vraiment, merci Christophe. Que serais-je devenu sans la bonne ambiance qu'il régnait au sein de l'équipe PII de l'ex-cena ? Merci donc tout d'abord à Éric Blond, sans qui jamais je n'aurais suivi cette voie, Jean-Paul Imbert et ses commentaires toujours sarcastiques, Alexandre Bustico et ses commentaires non moins sarcastiques, Jean-Luc Vinot et ses idées toujours pleines de bon sens, Robert Parise et ses côtes fêlées, Roland Alonso et ses idées toujours très particulières, Philippe Ribet et ses photos splendides, Mathieu Serrurier, Philippe Averty, Gwenaël Bothorel, François-Régis Colin, Daniel Étienne, Géraldine Beboux, Sylvie Athènes. Merci aussi aux petits derniers de l'équipe que j'ai un peu moins côtoyés : Nicolas Courtel pour sa grande efficacité, Rémi Lesbordes pour son enthousiasme, Caroline Martin pour sa bonne humeur, François Marais pour ses splendides maquettes, et Maxime Cordeil qui a repris le flambeau sur la 3D. Bien entendu, je n'oublie pas mes collègues de l'autre côté de la passerelle de l'époque, l'ENAC (qui est devenu mon employeur au moment où je rédige ces lignes) : Stéphane Chatty qui me permet de travailler aujourd'hui sur un projet passionnant qu'est le noyau Linux, Yannick Jestin qui m'a beaucoup apporté par ses relectures et critiques constructives pour mes présentations, Anne-Marie Schaal, Hélène Gaspard-Boulinc, Daniel Prun, Mathieu Magnaudet, Catherine Lethondal, et bien sûr Fabien André et Raphaël Hoarau. Je n'oublie pas non plus Cécile Moura qui a bien voulu partager son bureau avec moi lors de ma phase finale de rédaction.

Je n'oublie pas de remercier les membres du jury de thèse qui m'ont fait confiance et m'ont permis de réussir ce dernier examen. Merci aux membres de l'équipe ICS de l'IRIT, et tout particulièrement Philippe Palanque qui fut un directeur de thèse très efficace et toujours disponible.

Enfin, merci à mes amis pour m'avoir supporté et encouragé tout au long de ma thèse : Pablo pour avoir soutenu avant moi et m'avoir remotivé à fond, Manu, Olivier, Romain, Mathieu. Merci aussi à ma famille qui m'a permis d'arriver finalement à l'ENAC en m'encourageant dans ma passion, l'aviation. Merci aussi à ma belle famille qui m'a toujours motivé pour la rédaction de ce mémoire.

Et finalement, je ne remercierais jamais assez mon épouse, Stéphanie, qui m'a supporté pendant la rédaction de sa thèse et pendant la mienne qui a duré une éternité en comparaison.

Table des matières

Introduction générale **1**

Du graphisme interactif à la compilation graphique

1 Problématique 4
1.1 L'activité de design des systèmes graphiques interactifs 4
1.2 Des technologies toujours en évolution 5
1.3 Des performances absolument nécessaires 5
2 Thèse 6
3 Méthodologie employée 6
4 Organisation du mémoire 7

Partie I État de l'art **9**

Chapitre 1
Méthodes et instruments de conception des systèmes graphiques interactifs

1.1 Introduction 12
1.2 Différents types d'applications 12
1.2.1 Les systèmes interactifs standard 12
1.2.2 Les systèmes interactifs post-WIMP 12
1.3 Les modèles et architectures utilisés par les systèmes graphiques interactifs 13
1.3.1 Arch / Slinky 13
1.3.2 MVC 14
1.3.3 MDPC 14
1.4 Méthodes de développement pour la conception et la programmation . 15

1.4.1 Les outils industriels 15
1.4.2 USIXML et les architectures dirigées par les modèles 16
1.4.3 IntuiKit 17
1.4.4 Artistic Resizing 18
1.5 Les boîtes à outils disponibles 18
1.5.1 Les moteurs de rendu graphique 18
1.5.2 Les boîtes à outils maximisant la performance de rendu 19
1.5.3 Les boîtes à outils maximisant la richesse du rendu 19
1.6 Conclusion 19

Chapitre 2
Le système à flot de données

2.1 Introduction 21
2.2 Le flot de données 22
2.3 Modèle d'exécution 22
2.3.1 Exécution dirigée par les données 22
2.3.2 Exécution dirigée par la demande 23
2.3.3 Les systèmes à flot de données synchrones 23
2.3.4 Utilisation dans le cadre de ces travaux 23
2.4 Le système de flot de données pour les applications graphiques interactives 23
2.4.1 SketchPad 24
2.4.2 Garnet / Amulet 25
2.4.3 Icon 25
2.4.4 Functional Reactive Programming 26
2.5 Le modèle à flot de données pour la création de visualisations 26
2.6 Conclusion 27

Chapitre 3
Compilation

3.1 Introduction 29
3.2 Principes généraux 30
3.2.1 Un ensemble processeur de langage 30
3.2.2 Description des étapes de compilation 30
3.3 Les principales optimisations réalisables 32
3.3.1 Transformations conservant la fonctionnalité 32

3.3.2 Optimisation des blocs de base 34
3.3.3 Optimisations durant l'édition des liens 34
3.3.4 Optimisation à la volée 34
3.3.5 Optimisations dépendant de l'architecture cible 34
3.4 Compilation et graphisme 35
3.4.1 Transformations et graphisme interactif : Indigo 35
3.4.2 Compilations de bas niveau et graphisme. 35
3.5 Conclusion 36

Partie II Maximisation instrumentée de la séparation entre description et implémentation des systèmes graphiques interactifs 37

Chapitre 1
Démarche de conception instrumentée

1.1 Exigences 40
1.1.1 Donner du contrôle au designer 40
1.1.2 Minimiser l'apprentissage 41
1.1.3 Abstraire les éléments graphiques 41
1.1.4 Développer rapidement différentes alternatives 41
1.1.5 Garder un contrôle fin sur la richesse du rendu final 41
1.1.6 Modularité 42
1.1.7 Conserver de bonnes performances de rendu 42
1.2 Hayaku : design et concepts 42
1.2.1 Idée générale 42
1.2.2 Abstraire pour contrôler les éléments graphiques 43
1.2.3 Production de l'application graphique interactive 44
1.3 Scénarios d'utilisation 45
1.3.1 Scenario n°1 : Une application multi-touch 45
1.3.2 Scenario n°2 : une image radar 50
1.3.3 Scenario n°3 : un clavier virtuel 51
1.3.4 Scenario n°4 : intégrer des composants dans des applications existantes 54

1.4 Conclusion 58

Chapitre 2
Réalisation d'un compilateur graphique

2.1 Introduction 62
2.2 Survol rapide du fonctionnement de la boîte à outils 62
2.3 Interprétation contre compilation 62
2.4 Exigences pour le compilateur graphique 65
2.4.1 Performances de l'exécutable final 65
2.4.2 Performances du temps de compilation 66
2.4.3 Portabilité du code produit 66
2.4.4 Modularité permettant une évolution 66
2.5 Les différents langages intermédiaires utilisés 66
2.5.1 Les langages sources 66
2.5.2 Les premières transformations 67
2.5.3 Simplifications graphiques 67
2.5.4 Génération de la description dans le langage du moteur de rendu 68
2.5.5 Génération du code final 68
2.5.6 Écriture du code final 69
2.6 Le système de flot de données 69
2.6.1 Implémentation du système de flot de données 70
2.6.2 Optimisations du système de flot de données 70
2.7 Implémentations et optimisations du compilateur graphique 71
2.7.1 Première implémentation 71
2.7.2 Deuxième implémentation : externaliser la boucle de rendu . . . 71
2.7.3 Implémentation finale 72
2.7.4 Optimisations 73
2.8 Génération de code statique ou semi-statique 74
2.8.1 Gérer des applications dont le nombre d'objets graphiques est variable 74
2.8.2 Pré-requis pour pouvoir générer du code embarquable 75
2.9 Génération automatique de code 75
2.10 Le support du picking 76
2.11 Conclusion 77

Chapitre 3
Discussion et analyse

3.1 Évaluations préliminaires . . . 79
3.1.1 Puissance d'expression . . . 79
3.1.2 Performances . . . 81
3.1.3 Utilisabilité . . . 81
3.2 Premiers retours des designers d'applications interactives . . . 84
3.3 Conclusion . . . 85

Partie III Maximisation non instrumentée de la séparation entre description et implémentation des systèmes graphiques interactifs 87

Chapitre 1
Réalisation d'applications avec comme contrainte principale la vitesse d'exécution

1.1 Description de l'application, des besoins . . . 90
1.1.1 La configuration visuelle . . . 90
1.1.2 La rotation "Rolling the Dice" . . . 91
1.1.3 Le brushing et la sélection incrémentale . . . 91
1.1.4 L'organisation des vues et le *"Pick'n Drop"* . . . 92
1.2 Implémentation . . . 92
1.2.1 La génération de la vue . . . 93
1.2.2 Dessiner des lignes ou des ronds . . . 93
1.2.3 La rotation "Rolling the Dice" . . . 96
1.2.4 Le Brush . . . 96
1.2.5 Le Pick and Drop . . . 97
1.2.6 Le rendu des différentes vues dans la grille . . . 97
1.3 Conclusion . . . 98

Chapitre 2
Cache de champ de force avec texture pour le placement d'étiquettes

2.1 Problème . . . 99

2.2 Description des besoins 100
2.3 Algorithmes existants 100
2.4 Principes de l'implémentation GPU 101
2.5 Gains 102
2.5.1 Simplicité 102
2.5.2 Performances 103
2.5.3 Utilisation de la mémoire 103
2.5.4 Le problème est-il toujours NP-complet ? 104
2.6 Détails techniques 104
2.6.1 Récupérer des données depuis la carte graphique 104
2.6.2 Shaders employés 104
2.7 Conclusion 106

Conclusion générale **107**

Annexe A
Détails de l'application multitouch réalisée avec Hayaku

A.1 Les classes graphiques 113
A.2 Le modèle conceptuel 116
A.3 Les connections entre le modèle et les classes graphiques 117
A.4 La scène 118
A.5 Le lanceur 119
A.6 Traces d'exécution 121

Annexe B
Détails de l'application clavier logiciel réalisée avec Hayaku

B.1 Les classes graphiques 123
B.2 Le modèle conceptuel 127
B.3 Les connections entre le modèle et les classes graphiques 128
B.4 La scène 130
B.5 Le lanceur 133

Annexe C
Calculs du gain de l'approximation du rendu des lignes dans FromDaDy

Bibliographie **143**

Introduction générale

Du graphisme interactif à la compilation graphique

Développer des applications graphiques interactives est une tâche difficile [Myers 94, Myers 08]. Cette tâche est particulièrement délicate pour les systèmes interactifs *riches*. Nous donnons la définition suivante d'un système interactif *riche* :

Un système interactif est **riche** lorsqu'il est graphiquement *complexe*, qu'il se *rafraîchit souvent*, et qu'il peut afficher un grand nombre d'entités graphiques. Le rendu graphique d'un tel système *complexe* puisqu'il présente des propriétés graphiques évoluées comme des gradients, des ombres portées, des dessins manuscrits, des effets de transparence, de flou, etc. Ils doivent aussi se rafraîchir souvent puisqu'ils présentent de l'animation et de l'interaction avec l'utilisateur final.

Ces systèmes *riches* regroupent donc les applications dont le design graphique apporte des effets lors de l'interaction qui permettent d'améliorer l'expérience utilisateur. Des projets comme Digistrips [Mertz 00] ou Aster [Merlin 08] (Figure 1) jouent sur les ombrages, les dégradés, les transparences, l'écriture manuscrite, l'inertie ou encore la physique pour aider l'utilisateur final à réaliser sa tâche. De même, les menus des plateformes iPhone et Android présentent des effets d'inertie, de notification de fin de liste (par un effet de "bump", ou un éclairage adéquat). Ces systèmes interactifs riches nécessitent des graphismes interactifs paramétrés de manière très précise [Hartmann 08]. La construction de ces graphismes peut amener le designer à travailler au pixel près [Tabart 07].

FIGURE 1 – Éléments graphiques d'Aster : écriture manuscrite, intégration de gestes et de menus avec transparence, objets graphiques riches.

Réaliser de telles applications graphiques interactives nécessite de travailler avec des langages, des boîtes à outils et des modèles graphiques spécifiques. En plus de cette étape de production et de spécification fine des graphismes, il faut ensuite trouver un compromis entre les designs produits par le designer graphique (à savoir les graphismes, les interac-

tions et les animations) et les pénalités de performances induites par ces designs. Lorsque l'équilibre entre ces parties est trouvé, les performances d'utilisation ainsi que la satisfaction de l'utilisateur final en sont améliorés [Mertz 00]. L'iPhone le démontre : les effets graphiques du dispositif sont une part importante de son succès. Ces effets graphiques sont épaulés par des widgets et des applications réactives.

1 Problématique

Dans cette section, nous effectuons des constats sur la façon dont les applications graphiques interactives sont produites afin d'élaborer notre problématique.

1.1 L'activité de design des systèmes graphiques interactifs

Concevoir de tels systèmes est une activité récente qui a été rarement explicitement décrite et assistée dans le passé. Les qualités de cette activité de production sont essentielles pour l'utilisabilité du produit. Malheureusement, concevoir des systèmes interactifs riches et performants nécessite des spécialistes formés à l'optimisation du graphisme interactif.

Comme le montre Artistic Resizing [Dragicevic 05], nous pensons que les outils techniques mis à la disposition du designer ne lui permettent pas aujourd'hui de participer pleinement à l'activité de production de l'application, et donc de produire exactement ce qu'il a conçu. De plus, ces techniques ne lui permettent pas de réaliser du design exploratoire, où le designer affine son design par essais-erreurs. Un article récent analyse l'activité de design et de programmation des systèmes interactifs [Myers 08]. Parmi les différentes interviews, les designers expriment que "le comportement conçu était complexe [...] et requérait des capacités de programmation élevées"; que "le design des comportements interactifs émerge au travers du processus d'exploration [...] et aujourd'hui, les outils tolèrent difficilement un processus itératif"; "les détails sont importants, et vous ne pouvez jamais vous en faire une image complète jusqu'à ce que l'application soit complètement achevée"; "Je peux me représenter de manière très précise l'apparence désirée. Cependant, Je ne peux qu'approximer le comportement du moteur d'exécution". Ce document souligne aussi le fait que les designers graphique veulent faire des "transitions et des animations complexes".

Ainsi, les processus de conception des applications graphiques interactives sont aujourd'hui inadaptés pour produire des applications riches. Au cours de ces travaux, nous nous sommes intéressé à cette dimension en la nommant **puissance d'expression de la description**. Cette puissance d'expression passe par la manipulation de concepts appropriés par le designer graphique : afin de réaliser une tâche de production de graphisme, le concepteur doit manipuler un outil de production de dessins. Ainsi, en manipulant des objets et des outils proches des concepts employés, le concepteur d'application interactive peut plus facilement exprimer exactement son idée sans avoir à contraindre son design à cause des outils à sa disposition.

1.2 Des technologies toujours en évolution

Un deuxième problème qui émerge, est le fait que de nouvelles technologies pour la création d'application interactives sont sans cesse créées. De nouveaux moyens de penser, concevoir et développer des interfaces émergent tous les ans. Par exemple, nous avons vu apparaitre Java2D, Adobe Flash, Adobe Flex, Microsoft Dot Net, XAML, SVG, WMF, les interfaces Web 2.0 programmées en javascript directement dans le navigateur web (grâce au Canvas d'HTML5), OpenGL, Cairo, Qt, Prefuse, Protovis, le SDK de l'iPhone, Android de l'Open Handset Alliance, WebOS, Qt-Quick avec QML, etc. Pour concevoir des applications interactives sur toutes ces plateformes (embarquées, web, machines de bureau, machines ARM), le designer d'interface a à sa disposition une pléthore de boîtes à outils, généralement incompatibles les unes avec les autres.

Ceci amène à un échec de la réutilisabilité, un concept considéré comme l'un des plus importants du génie logiciel : les designers doivent re-développer les logiciels déjà existants afin de les porter sur une nouvelle plateforme, avec le problème de ne pas pouvoir réutiliser le code déjà conçu et bien testé de l'application précédente. Par exemple, le sous-système des menus qui est aujourd'hui bien intégré dans les machines traditionnelles de bureau (comme Windows ou MacOSX), ne sont que pâlement imitées dans les interfaces Web 2.0, où l'utilisateur doit suivre un tunnel strict lorsqu'il doit naviguer dans un menu hiérarchique. Nous extrayons de cette situation un besoin réel de **réutilisation** des logiciels existants, tout particulièrement si nous considérons que de nouvelles plateformes continuerons à apparaitre dans le futur (WebGL en est un example).

1.3 Des performances absolument nécessaires

Les applications interactives riches nécessitent un taux de rafraîchissement élevé car elles présentent des animations et de l'interaction. Dans ces travaux, nous nous sommes intéressé à cette dimension en la nommant **performance du moteur de rendu**. Afin de mesurer cette performance, nous nous sommes basé sur le temps mis pour produire une image. Ainsi, plus ce temps est faible, plus le processeur central est déchargé du traitement du graphisme. Le rendu final est donc plus fluide car il peut être mis à jour plus souvent. Un rendu de 100 millisecondes (10 images par secondes) est la limite haute à ne pas dépasser puisque sinon cela perturbe la boucle perception-action de l'utilisateur.

Dans ce manuscrit, nous nous exprimons souvent ces performances sous la forme d'un nombre d'images par seconde. Ce nombre, quand il est supérieur au taux de rafraîchissement de l'écran (50 ou 60 images par secondes pour un écran LCD classique), n'a pas de sens en tant que tel. Nous avons effectué nos tests en forçant le réaffichage à la fin de la génération de chaque image afin d'obtenir ces chiffres. Ainsi, quand nous donnons une valeur de 500 images par seconde, cela veut dire que le temps mis pour générer une image est 10 fois plus rapide que le taux de rafraîchissement de l'écran. Le processeur central peut donc gérer des tâches annexes pendant les 9/10e du temps qui est alloué à l'application interactive.

2 Thèse

Les outils de production d'applications graphiques interactives sont inadaptés à un processus de design exploratoire. Ils ne conviennent pas non plus dans la démarche d'inclusion du designer graphique dans le développement. De plus, la notion de réutilisabilité des designs est inexistante avec les processus de développement habituels. Aussi, la thèse que nous défendons est la suivante :

> *Pour améliorer le processus de développement des applications graphiques interactives et leur réutilisabilité, il faut séparer la description de la partie interactive de son implémentation et de ses optimisations. Cette séparation passe par l'utilisation de l'ensemble des techniques et outils disponibles (méthodes de compilation ou utilisation de la carte graphique afin de simplifier la description par exemple).*

Séparer description, implémentation et optimisation permet de réorganiser les tâches entre les différents acteurs du processus tout en maximisant la simplicité de description, ce qui favorise les itérations dans le développement. Il est aussi possible d'instrumenter ce processus de conception grâce à de nouveaux outils comme un compilateur graphique.

3 Méthodologie employée

Dans cette thèse, nous avons tout d'abord cherché à comprendre les outils et les techniques mis à disposition des designers graphiques afin d'améliorer le processus. Ce travail passe par un état de l'art ainsi que différentes approches de réalisation et d'optimisation d'applications graphiques interactives.

À partir des besoins extraits ci-dessus (soucis de capacité à régler finement les graphismes, améliorer les performances générales des graphismes, et tenter de résoudre le problème de la réutilisabilité), et de notre état de l'art, nous avons fait le constat que l'activité de production des scènes interactives pouvait se rapprocher d'un mécanisme de compilation. Nous avons introduit Hayaku, une boîte à outils qui vise à s'attaquer à ce que Brad Myers appelle les *insides* d'une application [Myers 92]. Les *insides* d'une application graphique sont les éléments que souhaite faire apparaître le designer graphique mais dont les outils qu'il a à sa disposition ne lui permettent pas de le réaliser. Il doit donc faire appel à un programmeur pour coder manuellement son interaction.

Ces travaux s'intéressent aussi à la partie technique des optimisations réalisées sur les applications graphiques interactives. Ces recherches nous ont amené à co-écrire un logiciel d'exploration de données multi-dimensionnelles, FromDaDy. Ce logiciel présente à l'utilisateur plusieurs centaines de milliers de données et il est donc crucial qu'il soit optimisé afin de maximiser l'*expérience utilisateur*. Nous avons aussi amélioré tant du point de vue des performances d'exécution que de la maintenabilité une algorithme en utilisant la mémoire graphique (en deux dimensions) comme moyen d'accélération.

4 Organisation du mémoire

La première partie du mémoire présente un état de l'art contenant les bases théoriques sur lesquelles nous nous sommes appuyées.

- Pour cela, nous présentons tout d'abord, les outils et les techniques que le concepteur d'applications graphique interactive peut utiliser. Ces techniques nous permettent d'enclencher une réflexion sur l'implication du designer graphique dans le processus de développement.
- Ensuite, nous présentons le système de flot de données en général, puis appliqué aux graphismes. Ce chapitre nous permet de préciser les fondements théoriques nécessaires à la compréhension d'une partie des mécanismes internes du compilateur graphique que nous présentons, à savoir la gestion des dépendances.
- Enfin, nous présentons une brève introduction des concepts de compilation que nous avons utilisé. Cette introduction nous permet de montrer que le processus complet de développement des systèmes interactifs peut être apparenté à une chaîne de compilation. Ainsi, les optimisations et les techniques disponibles dans la théorie de la compilation peuvent être transposées dans le domaine du graphisme.

La deuxième partie présente nos travaux sur la démarche de conception des applications graphiques interactives.

- Tout d'abord, nous présentons la boîte à outils Hayaku [Tissoires 11]. Nous présentons son utilisation par un designer d'interaction au travers de quatre scénarios.
- Ensuite, nous présentons le fonctionnement interne de la boîte à outils. Cette dernière repose sur un compilateur graphique développé au cours de cette thèse.
- Enfin, nous discutons des résultats et des perspectives concernant notre démarche instrumentée de conception.

La dernière partie de notre thèse présente nos travaux d'optimisation non instrumentée d'applications graphiques interactives.

- Nous présentons les techniques que nous avons utilisé pour créer une application de visualisation de données multi-dimensionnelles.
- Nous montrons ensuite que modifier un algorithme existant en utilisation un cache de texture comme moyen de réaliser certains calculs coûteux permet un gain pour la maintenabilité du système et pour ses perspectives d'évolutions.

Enfin, nous concluons ce mémoire en résumant nos contributions et en présentant les perspectives de travaux de recherche permettant d'améliorer le processus de développement que nous présentons.

Première partie

État de l'art

Chapitre 1

Méthodes et instruments de conception des systèmes graphiques interactifs

Sommaire

1.1 Introduction 12

1.2 Différents types d'applications 12

1.2.1 Les systèmes interactifs standard 12

1.2.2 Les systèmes interactifs post-WIMP 12

1.3 Les modèles et architectures utilisés par les systèmes graphiques interactifs 13

1.3.1 Arch / Slinky 13

1.3.2 MVC 14

1.3.3 MDPC 14

1.4 Méthodes de développement pour la conception et la programmation 15

1.4.1 Les outils industriels 15

1.4.2 USIXML et les architectures dirigées par les modèles . . . 16

1.4.3 IntuiKit 17

1.4.4 Artistic Resizing 18

1.5 Les boîtes à outils disponibles 18

1.5.1 Les moteurs de rendu graphique 18

1.5.2 Les boîtes à outils maximisant la performance de rendu . . 19

1.5.3 Les boîtes à outils maximisant la richesse du rendu 19

1.6 Conclusion 19

1.1 Introduction

L'objectif de ce chapitre est d'introduire le problème de la conception des systèmes graphiques interactifs. On observe une grande diversité de tels systèmes, et les besoins en terme de développement sont différents. Nous allons d'abord aborder les différentes classes de ces systèmes, puis nous étudierons les moyens à disposition pour concevoir de tels systèmes interactifs.

1.2 Différents types d'applications

1.2.1 Les systèmes interactifs standard

Les **systèmes interactifs standard** regroupent l'ensemble des applications graphiques créées pour les machines grand public qui s'articulent autour du paradigme clavier/souris.

On peut recenser les types d'applications suivant [Preece 94, Dragicevic 04a] :

- Les **langages de commande** regroupent les interfaces textuelles du type console UNIX où toute l'interaction est réalisée au clavier.
- Les **systèmes de menus** permettent la présentation d'une liste de choix dont l'utilisateur final sélectionne son choix par pointage de la souris.
- Les **formulaires** étendent les systèmes de menus en offrant des *champs* de texte qu'il est possible de renseigner. Les *tableurs* en sont une version plus sophistiquée. Ce type d'interface est aussi couramment utilisé sur internet pour permettre une interaction avec l'utilisateur.
- La **manipulation directe** [Shneiderman 83, Hutchins 85] permet à l'utilisateur final la possibilité de travailler directement avec des objets de la tâche plutôt que de transmettre des commandes à une machine.
- Les **dispositifs WIMP** [1] reprennent le concept de la manipulation directe en se basant sur une combinaison des trois paradigmes d'interaction suivant :
  - Le **drag-and-drop** : les icônes représentant les fichiers sont déplacés et déposés sur un objet sensible pour exécuter une commande.
  - Le **fenêtrage** : les fenêtres sont des zones rectangulaires de l'écran qui sont ajustables et superposables. Dans ces rectangles se retrouvent les différents systèmes interactifs, ou programmes, qu'utilise l'utilisateur.
  - Les **widgets** [2] : ces items graphiques sont issus d'une bibliothèque générique en général propre au système. Ils regroupent les boutons poussoirs ou à bascule, les barres de défilement, les listes déroulantes, etc...

1.2.2 Les systèmes interactifs post-WIMP

L'interaction directe permet aux designers d'applications graphiques interactives de produire des applications plus naturelles pour les utilisateurs. Le succès de l'Iphone d'Ap-

1. WIMP : Windows, Icons, Mouse, Pull-down menus.
2. contraction de Windows Objects

ple montre combien les utilisateurs finaux sont sensibles à ces interfaces naturelles, ou intuitives. Ces applications graphiques interactives sont appelées applications **post-WIMP**.

Les contextes d'utilisation de ces nouveaux systèmes sont très différents et ciblent des besoins spécifiques. Par exemple, certains ciblent une utilisation au doigt (applications sur téléphones tactiles), tandis que d'autres disposent d'un stylet (outils de dessin des designers graphiques). Dans le laboratoire où ont été réalisé ces travaux, de nombreuses applications de ce style ont été produites. Nous pouvons citer Digistrip [Mertz 00], une application à destination des contrôleurs aériens.

Pour développer de telles applications, nous avons vu arriver différentes technologies : Java2D, Adobe Flash, Adobe Flex, Microsoft Dot Net, XAML, SVG, WMF. Les applications Web 2.0, Cairo, Prefuse, Protovis, iPhone SDK, Android, Palm WebOS, en sont quelques autres exemples. Tous ces systèmes permettent au développeur d'applications interactives de ne plus se contenter du paradigme WIMP.

1.3 Les modèles et architectures utilisés par les systèmes graphiques interactifs

Concevoir et programmer une application graphique interactive est une activité complexe. Pour réaliser la tâche de programmation, différents modèles et architectures d'applications graphiques interactives ont été créés. Nous allons présenter ici quelques uns de ces modèles.

1.3.1 Arch / Slinky

Un des modèles qui segmente le plus l'architecture des systèmes graphiques interactifs est le modèle Arch (et son méta-modèle Slinky) [Uims 92]. Ce modèle est présenté Figure 1.1.

Ce modèle introduit 5 composants :

- le composant d'**interaction** est responsable de l'implémentation physique avec l'utilisateur final. Ce sont généralement les objets graphiques présentés à l'utilisateur.
- le composant de **présentation** est chargé de faire le lien entre le composant de dialogue, et le composant d'interaction. C'est lui qui va faire le choix de la représentation utilisé pour l'objet abstrait. Par exemple, un item à choix multiples peut être soit représenté par un menu ou par un système de radio-boxes.
- le composant de **dialogue** gère le séquencement de l'interaction en entrée et en sortie. Il maintient aussi la consistance entre les vues multiples d'une application interactive.
- le composant d'**adaptation du domaine** permet de faire le lien entre le composant de dialogue et le noyau fonctionnel.
- le **noyau fonctionnel** est la partie de l'application qui n'est pas reliée à l'interaction.

Au cours de ces travaux, nous nous sommes focalisé uniquement sur les composants *présentation* et *interaction* de ce modèle. Cette partie présente l'inconvénient d'être en général le goulot d'étranglement de l'application interactive [Barboni 07, Blanch 05] en

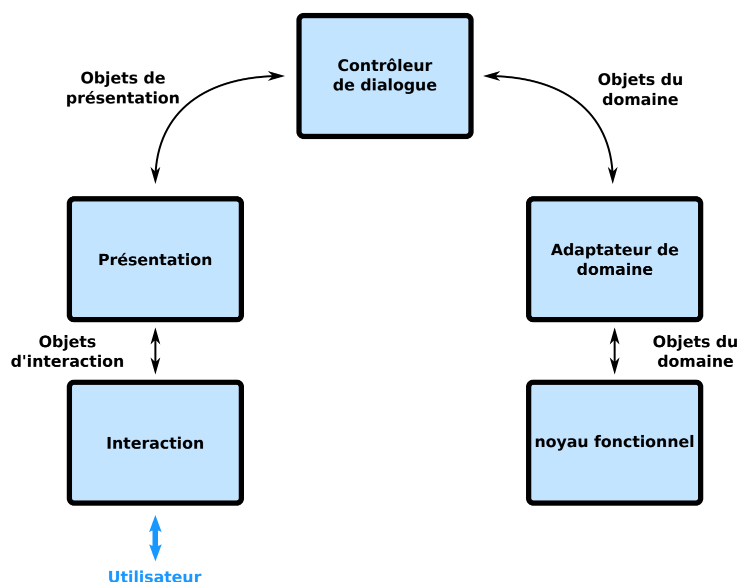

FIGURE 1.1 – Le modèle Arch.

terme de performances de rendu. Lorsque ces couches sont peu optimisées, l'application peut devenir inutilisable puisque non réactive.

1.3.2 MVC

Le modèle MVC [Schmucker 87, Krasner 88] décompose les systèmes interactifs en un **modèle**, une ou plusieurs **vues** et un ou plusieurs **contrôleurs**. Le modèle est son noyau fonctionnel. Les vues sont les représentations de ce qui est affiché à l'écran. Enfin, les contrôleurs permettent à l'agent d'interpréter et recevoir les évènements de l'utilisateur final pour mettre à jour le modèle.

L'inconvénient majeur de ce modèle est qu'il est relativement difficile à implémenter de manière propre. En effet, la séparation est délicate à trouver entre la vue et le contrôleur. Dans la plupart des cas, ces deux composants sont associés car ils sont trop intimement liés [3].

1.3.3 MDPC

MDPC est un modèle proposé par [Conversy 08] afin de répondre au problème posé par MVC concernant la vue et le contrôleur. Ce modèle décompose la vue en deux parties : la *vue d'affichage* et la *vue de picking*, d'où le nom de *Model-Display view-Picking view-Controller*.

Cette décomposition permet d'abstraire le contrôleur de la vue en se basant sur les transformations inverses du dispositif d'entrée. La vue de picking reçoit les évènements utilisateur et les transforme dans la représentation du contrôleur (par exemple, une valeur entre 0.0 et 1.0 pour un bouton rotatif). Le contrôleur agit ensuite sur le modèle en

3. La conception du langage Java ainsi que sa partie présentation (Swing) s'est basée sur une conception MVC. Cependant, la vue et le contrôleur ne sont pas séparés dans les composants swing.

connectant les modifications correctes à réaliser. Le graphe d'affichage est ensuite mis à jour afin de répercuter ces modifications.

MDPC considère que l'ensemble de l'application graphique peut être assimilé à une ou plusieurs transformations. La notion de sélection des items graphiques peut être exprimée comme la transformée inverse de la représentation de la souris dans la vue de picking pour remonter au modèle. Le problème d'un tel système est que toutes ces transformations coûtent cher en temps d'exécution. En effet, une implémentation naïve du système consiste à réaliser à chaque rafraîchissement toutes les transformations qui permettent de créer toutes les vues et d'obtenir les informations nécessaires. Seulement, il doit être possible d'améliorer ce système en conservant une interface de programmation du point de vue du développeur à base de transformations mais qui en interne réalise les optimisations nécessaires à chacun des niveaux de transformation.

1.4 Méthodes de développement pour la conception et la programmation

Si les applications graphiques interactives ont évolué, les méthodes de production de ces dernières aussi. La conception participative (CP) [Muller 07] permet d'impliquer au maximum l'utilisateur final dans la production de l'application interactive. Cette méthode repose sur l'utilisation de prototypes papiers en début de cycle [Snyder 03], puis évolue vers des maquettes de plus en plus réalistes. Cette méthode de développement est itérative et nécessite un certain nombre d'itérations. Il faut donc fournir aux concepteurs la possibilité de produire facilement et rapidement des prototypes afin de pouvoir les tester et les valider au plus tôt avec les utilisateurs. La *facilité* de prototypage peut s'exprimer en terme de temps d'apprentissage des outils par exemple. La *rapidité* peut se compter en lignes de codes nécessaire ainsi qu'en temps de développement. Enfin, ces prototypes amènent à rédiger un certain nombre de spécifications, et souvent l'application finale est recodée complètement pour résoudre les problèmes des performances et de robustesses induits par les boîtes à outils de prototypage. Cette dernière phase implique donc au minimum deux personnes : un *designer graphique* et un *codeur d'interactions*.

1.4.1 Les outils industriels

Dans le monde industriel, nous pouvons relever deux technologies permettant de réaliser des applications graphiques interactives où l'implication du designer graphique est forte. Tout d'abord, Microsoft présente un tel système avec Microsoft Expression Blend Suite. Cette suite utilise le langage XAML (un dérivé d'XML) pour décrire les parties graphiques de l'application. Comme dans l'approche d'Intuikit [Chatty 04], le but est ici de séparer la description graphique du noyau fonctionnel de l'application. Le designer graphique peut produire un visuel par classe C# qui a besoin d'être dessiné. Cet outil introduit le concept de "binding" entre les objets C# et les objets graphiques. Ceci est très utile pour les programmeurs puisque cela leur permet d'abstraire les liens entre les formes graphiques et les objets sources. Ils doivent cependant manipuler du code de bas niveau pour implémenter les interactions et les animations.

Adobe propose lui aussi une suite de logiciels pour développer des systèmes interactifs. L'utilisation combinée des outils Flex et Flash permet aux concepteurs d'applications de séparer la partie description graphique du noyau fonctionnel. Toutefois, même si le designer peut utiliser Flash pour construire ses composants graphiques, il est contraint dans le choix du langage d'implémentation du noyau fonctionnel de son application : il doit utiliser ActionScript. Ce point pose problème lors de la création d'applications critiques devant être embarquées par exemple. De plus, il n'y a pas de moyen d'extraire les graphismes du code afin de séparer la partie rendue de la partie interaction. Il n'y a pas non plus de possibilité d'exprimer les propriétés graphiques à l'aide d'un système de flot de données. Enfin, même si Flex est capable de s'exécuter sur diverses plateformes, il n'est pas possible d'embarquer les graphismes produits dans les scènes graphiques d'autres applications.

1.4.2 USIXML et les architectures dirigées par les modèles

USer Interface eXtensible Markup Language (USIXML) [Limbourg 05] est un langage basé sur XML qui permet aux designers graphique de tester différentes approches à partir du même modèle. Ce langage se base sur l'ingénierie dirigée par les modèles où il faut en premier lieu modéliser la tâche utilisateur. USIXML comprends un langage de description de l'interface, User Interface Description Language (UIDL), et une spécification de transformations de graphes (Figure 1.2).

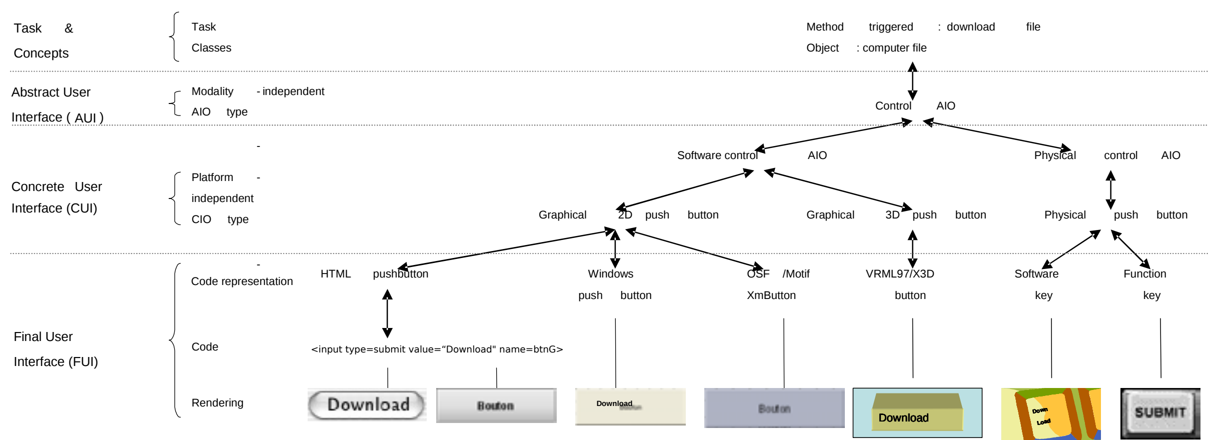

FIGURE 1.2 – Les transformations d'USIXML [Limbourg 05]

Plusieurs aspects d'USIXML sont proches de nos travaux. USIXML abstrait le graphisme pour générer plusieurs interfaces selon la plateforme cible. USIXML permet donc à moindre coût de porter une interface d'une technologie vers une autre. USIXML repose sur plusieurs modèles décrivant l'interface, et génère l'interface finale à partir de transformations sur ces modèles. Ceci permet d'abstraire l'interface pour le designer graphique et de ne travailler qu'avec des objets du modèle.

Cependant, nous ne nous situons pas exactement au même niveau : USIXML est un langage et une boîte à outils orientée application (modélisation de la tâche génération à l'aide d'un ensemble de widgets pré-définis), tandis que nos travaux sont situés plus bas

dans la couche présentation [4], ce qui nous permet de nous intéresser aussi aux interfaces post-WIMP.

1.4.3 IntuiKit

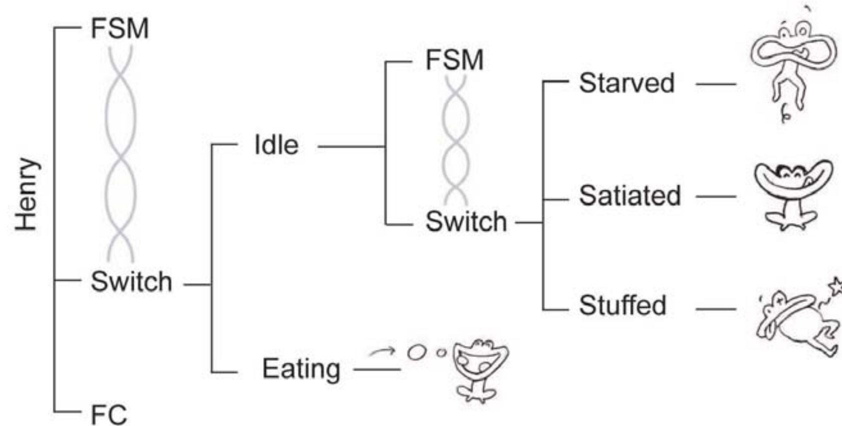

FIGURE 1.3 – L'arbre IntuiKit pour Henry le crapaud [Chatty 04].

[Chatty 04] présentent une méthode et des outils associés afin d'impliquer le designer graphique dans le développement et la conception graphique des systèmes interactifs. Les programmeurs et les designers se mettent tout d'abord d'accord sur une représentation simple et conceptuelle de la scène interactive. Ils produisent au cours de cette phase un fichier SVG (Scalable Vector Graphics) appelé représentation *basse fidélité* (les dessins des crapauds dans la Figure 1.3). Tandis que les programmeurs codent les interactions avec une représentation de bas niveau, les designers peuvent travailler de leur côté sur leur conception graphique. Puisque les designers et les programmeurs respectent le contrat précédemment conçu, la production du système final consiste simplement à remplacer la description basse fidélité par celle du designer.

Néanmoins, l'outil présenté oblige le designer graphique à utiliser une bibliothèque qui transforme la description de haut-niveau qu'il a fourni (le fichier SVG) en une description bas niveau (utilisable par un canvas proche de celui de Tk) avec un pouvoir d'expression moindre. De plus, le designer graphique doit manipuler des portions de code de bas niveau afin d'implémenter le comportement graphique. Ceci réfrène les explorations des designs alternatifs puisque changer ces graphismes implique beaucoup de manipulations pour voir l'impact dans le résultat final. De plus, lors de la phase d'optimisation du code, cette approche tombe à nouveau dans un processus séquentiel : les programmeurs doivent attendre les solutions du designer graphique avant de pouvoir optimiser le code de rendu. Les designers doivent de leur côté attendre les optimisations des programmeurs pour valider l'utilisabilité de leur conception.

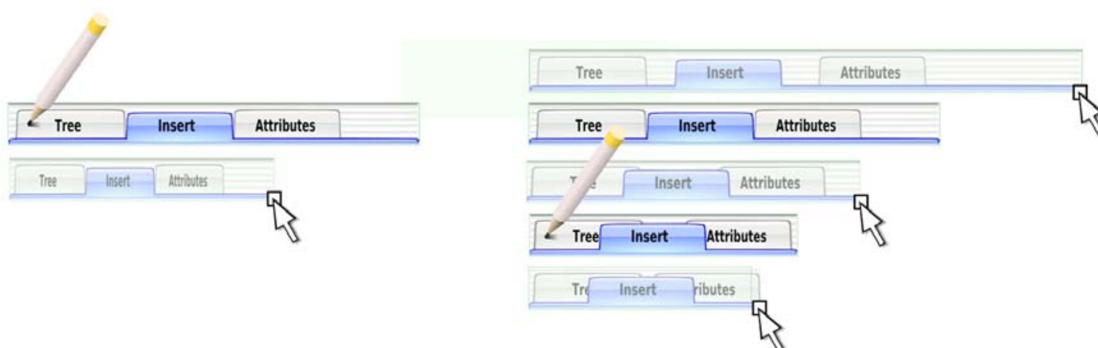

FIGURE 1.4 – Illustration d'Artistic Resizing [Dragicevic 05].

1.4.4 Artistic Resizing

Artistic Resizing [Dragicevic 05] est une technique de développement d'applications graphiques interactives qui a pour objectif d'améliorer le pouvoir d'expression du designer graphique en lui donnant la possibilité d'exprimer le comportement des items graphiques lors du redimensionnement de l'application. Le designer fournit pour cela plusieurs version de l'objet graphique selon différentes tailles clés (Figure 1.4, celles pointées par le crayon). Le système réalise ensuite une interpolation entre les caractéristiques des objets graphiques pour déterminer le comportement lors du redimensionnement. On passe ainsi d'un redimensionnement inadéquat sur la partie gauche de la Figure 1.4 (les polices de caractères sont tassées) à celui qui offre un meilleur rendu sur la partie droite (les largeurs des onglets sont conservées).

Artistic Resizing donne la possibilité au designer graphique de gérer lui-même et de manière fine le redimensionnement. Ce système montre la nécessité de donner le moyen au designer de l'application de contrôler toutes les parties de l'application, même des parties comme le redimensionnement qui sont habituellement codées en *dur* dans la boîte à outils. Il montre enfin que le designer graphique peut configurer de manière fine l'interface si les outils qui lui sont proposés sont adaptés. Cependant ce système mériterait d'être employé en dehors du redimensionnement seul.

1.5 Les boîtes à outils disponibles

Pour développer des applications graphiques interactives, les développeurs utilisent des boîtes à outils. Parmi les nombreuses boîtes à outils disponibles, certaines tentent de répondre au problème des faibles performances de rendu : Prefuse [Heer 05], Jazz [Bederson 00], Piccolo [Bederson 04], et Infoviz Toolkit (IVTK) [Fekete 04] par exemple.

1.5.1 Les moteurs de rendu graphique

Les boîtes à outils que nous allons décrire reposent sur d'autres bibliothèques logicielles pour fonctionner. Il existe dans les systèmes interactifs développés aujourd'hui deux caté-

4. La couche présentation est définie dans le modèle Arch dans la partie 1.3.1

gories de telles bibliothèques, ou moteurs de rendu :

- les moteurs de rendu purement **logiciel** qui reposent sur un traitement entièrement réalisé par le processeur central,
- les moteurs de rendu accélérés **matériellement** par la carte graphique.

Lorsque les performances de rendu final sont critiques, les développeurs peuvent écrire directement leurs applications dans de telles bibliothèques.

Les moteurs de rendu accélérés matériellement utilisent aujourd'hui OpenGL ou Direct3D. Si ces moteurs sont très efficaces, ils présentent l'inconvénient d'être "éloignés" du graphisme riche puisque la primitive de base de dessin est le triangle. Le programmeur doit en effet transformer ses graphismes en un *maillage* de triangles afin de pouvoir les afficher.

1.5.2 Les boîtes à outils maximisant la performance de rendu

Les performances sont maximisées par l'utilisation de structures ad-hoc explicites (des tables pour Prefuse et IVTK), ou cachées (des structures de données spatiales pour Piccolo). La première limitation vient de la forme de la description graphismes qui est à la fois inapproprié, car verbeuse, et trop pauvre, car décrite sous forme textuelle. De plus, les graphismes riches créés avec des outils classique de dessin comme Inkscape ou Adobe Illustrator ne peuvent généralement pas être utilisés directement par ces boîtes à outils.

Les canvas des boîtes à outils standard utilisent de plus en plus des moteurs de rendus accélérés matériellement pour leur affichage. Cairo ou le canvas de Qt sont implémentés en interne en utilisant la carte graphique. Du point de vue de l'utilisateur, il conserve son interface de programmation. Cependant, quand ces boîtes à outils sont accélérées, elles ne le font que d'une seule façon. Ces accélérations sont soit ad-hoc, soit trop générales, et ne sont pas forcément appropriées pour toutes les applications.

1.5.3 Les boîtes à outils maximisant la richesse du rendu

Les boîtes à outils graphiques gérant des *graphismes riches* existent. SVG permet en effet l'utilisation de filtres comme le flou gaussien ou encore l'illumination d'éléments par une source lumineuses. Peu de boîtes à outils interprètent ces éléments. Une bibliothèque d'interprétation et de rendu de fichiers SVG comme Batik [5] les interprète. Cependant, de telles implémentations sont incompatibles avec la boucle d'interaction car le moteur de rendu utilisé est trop lent.

1.6 Conclusion

Au cours de ce chapitre, nous avons présenté les différents systèmes interactifs existants. Nous avons vu les différences entre les applications WIMP et post-WIMP. Les architectures et les boîtes à outils graphiques utilisées pour produire ces applications semblent toujours devoir effectuer un compromis entre deux notions : la richesse du rendu et les performances d'exécutions.

5. `http://xmlgraphics.apache.org/batik/`

Les boîtes à outils WIMP sont très performantes, car très simples. Il est très délicat pour le designer graphique de sortir des widgets de base. Implémenter un pie-menu, ou un range-slider sont deux challenges en Swing par exemple.

Les boîtes à outils post-WIMP essaient plutôt de maximiser la richesse et la qualité du rendu final (Intuikit, Artistic resizing et Batik par exemple). Néanmoins, ces outils ne peuvent être utilisées que pour des besoins de prototypage en raison des piètres performances d'exécutions du moteur de rendu.

D'autres boîtes à outils post-WIMP (Prefuse, IVTK ou Piccolo) obtiennent des performances de rendu acceptables mais ne permettent pas au designer graphique d'avoir un pouvoir d'expression maximal.

Enfin, ces boîtes à outils soulèvent le problème de la non portabilité du code. Quand bien même elles seraient efficaces en termes de performances, elles forcent toujours le concepteur de l'application à travailler avec un langage et un moteur d'exécution spécifique (celui de la boîte à outils), et ne gèrent qu'un seul moteur de rendu. Par exemple, il n'est pas possible d'utiliser Prefuse dans une application écrite en C ou en C++.

Toutes ces considérations nous amènent à penser qu'il manque dans l'écosystème du rendu graphique un ensemble d'outils qui permette de favoriser la richesse de rendu sans trop réduire les performances de l'application finale. Nous allons voir au cours des chapitres suivants les techniques qui nous permettent de construire de tels outils.

Chapitre 2

Le système à flot de données

Sommaire

2.1 Introduction **21**
2.2 Le flot de données **22**
2.3 Modèle d'exécution **22**
2.3.1 Exécution dirigée par les données 22
2.3.2 Exécution dirigée par la demande 23
2.3.3 Les systèmes à flot de données synchrones 23
2.3.4 Utilisation dans le cadre de ces travaux 23
2.4 Le système de flot de données pour les applications graphiques interactives **23**
2.4.1 SketchPad 24
2.4.2 Garnet / Amulet 25
2.4.3 Icon 25
2.4.4 Functional Reactive Programming 26
2.5 Le modèle à flot de données pour la création de visualisations **26**
2.6 Conclusion **27**

2.1 Introduction

Le nom **système à flot de données** provient de la notion conceptuelle qu'un programme dans un système à flot de données est un **graphe dirigé** et que les **données** transitent entre les **instructions**, le long des arcs [Arvind 86, Davis 82, Dennis 74, Dennis 75, Johnston 04].

Les travaux que nous avons effectués utilisent le système de flot de données de deux manières :

- de manière **interne**, pour pouvoir contrôler les dépendances entre le code produit et les entrées de la partie rendu du compilateur graphique
- de manière **externe**, pour aider le designer d'interaction à spécifier sa conception.

Nous allons ici présenter ce système de flot de données au travers des travaux précédents, et nous verrons ensuite quelles sont les utilisations qui en sont faites dans les systèmes graphiques interactifs.

2.2 Le flot de données

Ces systèmes ont été créé vers 1975 alors que certains chercheurs trouvaient que les machines de Von Neumann (les processeurs classiques) étaient “de manière inhérente inadaptées à l'exploitation du parallélisme” [Dennis 75]. Le modèle de flot de données présente en effet par nature une affinité avec le parallélisme. En effet, les instructions disponibles sont celles d'un langage fonctionnel sans effets de bord. Un autre avantage du système à flot de données est son *déterminisme*. À chacune des configurations en entrée correspondra une et une seule valeur de sortie [Arvind 86, Davis 82, Kahn 74]. Il est donc possible de réaliser des preuves et des optimisations sur ces programmes.

2.3 Modèle d'exécution

Comme nous l'avons dit plus tôt, un programme exprimé sous forme de flot de données est conceptuellement un graphe dirigé. Les nœuds de ce graphe sont les instructions primitives du langage, et les arcs entre ces nœuds représentent les dépendances entre ces instructions [Kosinski 73, Johnston 04]. Un arc qui est dirigé vers un nœud est appelé **entrée** de ce nœud, et un arc qui en ressort est appelé **sortie** de ce nœud.

2.3.1 Exécution dirigée par les données

Lorsque le programme démarre, les feuilles du graphes sont *activées*. On place pour cela dans ces feuilles un jeton d'activation. Lorsqu'un nœud possède une ou plusieurs entrées activées, il est alors *activable*. Quand un nœud est activable, il réalise son instruction, et enlève un jeton dans chacune de ses entrées. Il place ensuite un jeton d'activation dans sa (ou ses) sortie(s). Il arrête ensuite son traitement et attend d'être de nouveau activable.

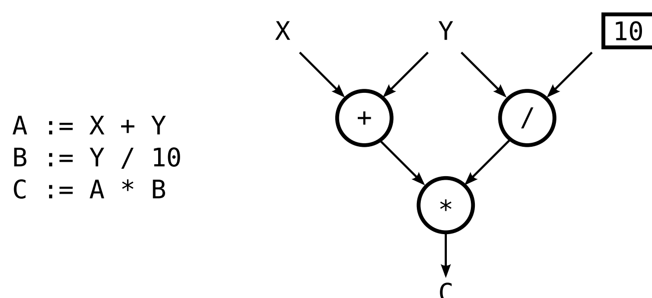

FIGURE 2.1 – Un programme simple et sont équivalent en système de flot de données (tiré de [Johnston 04])

La Figure 2.1 présente un exemple de tel programme. Sur un processeur classique, le code de gauche est exécuté en trois instructions séquentielles. Sur un système de flots de

données, les deux premières instructions peuvent être exécutées en parallèle, et le temps total du système ne prendra plus que deux pas de temps.

2.3.2 Exécution dirigée par la demande

Une autre approche de l'exécution d'un système à flot de données est celle dirigée par la demande [Davis 82, Kahn 74]. Dans l'approche précédente, chaque fois qu'une donnée est mise à jour, tout le graphe de dépendance de cette dernière est à son tour mis à jour, et cela même si certaines des variables de sorties ne sont jamais utilisées.

L'approche dirigée par la demande consiste à n'effectuer les mises à jour des données qu'une fois que le programme effectue une requête sur une variable. Le graphe de dépendance est alors vérifié, et les données sont mises à jour le cas échéant.

Si le système dirigé par la demande présente l'avantage de ne calculer que les nœuds qui ont été effectivement mis à jour, il présente toutefois l'inconvénient d'être plus coûteux pour l'accès aux éléments puisqu'il est nécessaire de parcourir le graphe de dépendance à chaque accès à une variable (tandis que dans l'autre approche, on est sûr que le contenu des variables est à jour puisqu'à chaque modification d'une variable, le graphe de dépendance est parcouru).

2.3.3 Les systèmes à flot de données synchrones

Les systèmes à flot de données synchrones [Lee 87] sont un sous-ensemble du système de flot de données présenté ci-dessus où les jetons d'activation qui sont consommés à chaque activation sont connus à la compilation [Bhattacharyya 96]. Ainsi, la tâche du compilateur est de convertir le système à flot de données en un ensemble d'instructions séquentielles qui ne requièrent pas d'ordonnancement dynamique.

2.3.4 Utilisation dans le cadre de ces travaux

Dans nos travaux, nous utilisons les systèmes à flot de données pour traduire la notion de dépendances entre les spécifications fournies par le designer d'interaction et l'application interactive finale. Nous utiliserons un *système de flot de données synchrone dirigé par les données*. Ce système à flot de données nous permet de réaliser des optimisations internes. Il nous permet aussi de traiter de manière efficace les entrées de l'utilisateur final. Cette facilité de description des dépendances est aussi exportée au designer d'interaction. Dans ce cas, le flot de données est utilisé comme un langage de spécification.

2.4 Le système de flot de données pour les applications graphiques interactives

Les applications graphiques interactives peuvent utiliser pour leur spécifications une description proche d'un système de flot de données [Appert 09, Dragicevic 04b, Huot 04, Esteban 95]. Dans certains cas, ce système est à base de *contraintes* [Vander Zanden 94, Sutherland 63]. Une contrainte présente l'avantage d'être bidirectionnelle : le designer peut

contraindre deux lignes entre elles en exprimant leur parallélisme par exemple. Néanmoins, un tel système peut être jugé comme confus par l'utilisateur [Esteban 95].

2.4.1 SketchPad

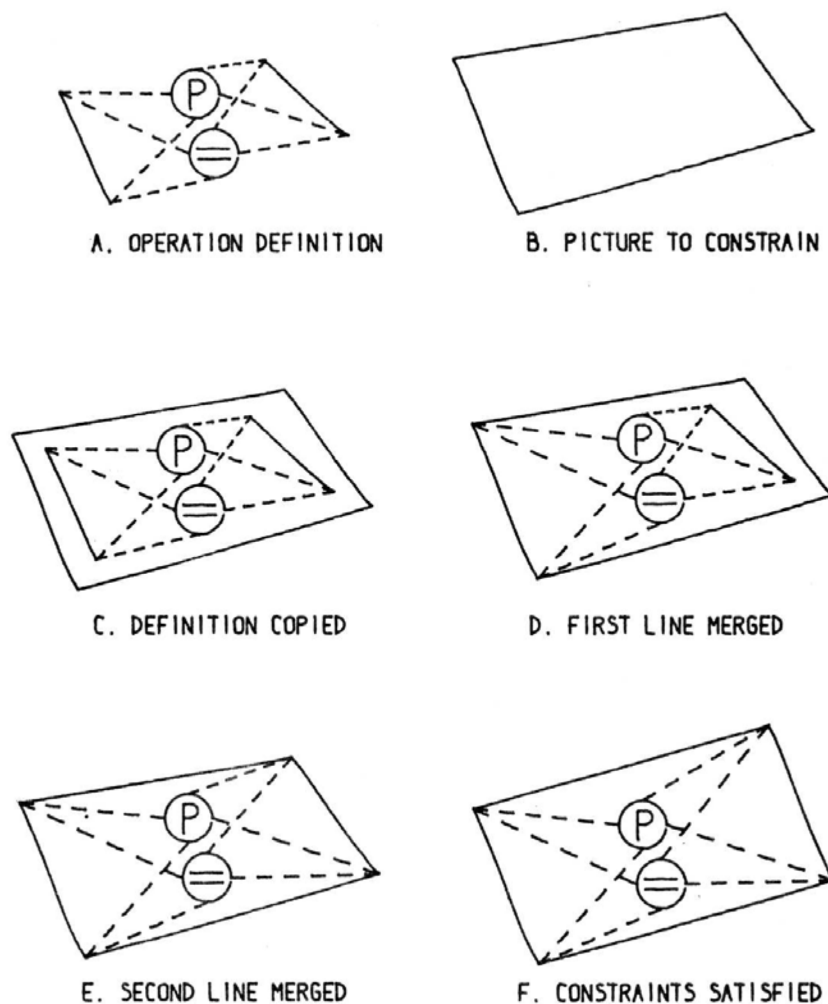

FIGURE 2.2 – Sketchpad [Sutherland 63].

SketchPad [Sutherland 63] fut le premier logiciel de dessin à utiliser un système de contraintes. SketchPad autorise la création de lignes contraintes les unes par rapport aux autres (Figure 2.2). Les contraintes de SketchPad sont bidirectionnelles. Elles permettent

au designer d'indiquer que telle ligne est parallèle ou perpendiculaire avec telle autre sans notion de priorité de l'une par rapport à l'autre.

2.4.2 Garnet / Amulet

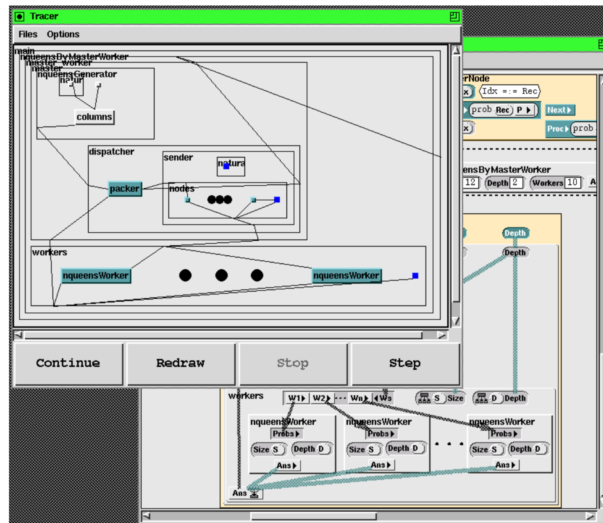

FIGURE 2.3 – Amulet (`http://www.cs.cmu.edu/afs/cs/project/amulet/www/amulet-all-pictures.html`).

Garnet [Myers 90, Vander Zanden 94, Vander Zanden 01a] présente un modèle de contraintes mono-directionnelles (Figure 2.3). Ce système est plus proche du flot de données, sans toutefois exiger que le graphe résultant soit acyclique. Les contraintes permettent de décrire de manière simple les dépendances entre les composants de l'interface. Elles permettent de spécifier l'application interactive avec un langage descriptif (exemple tiré de [Vander Zanden 94]) :

```
circle.left = self.object_over.right + 10
circle.object_over = rect34
```

Cet exemple traduit le fait que le cercle considéré doit être à 10 unités à gauche de la droite de l'objet qui se trouve au dessus de lui. La position du cercle est alors automatiquement recalculée quand cet objet change de forme ou de position. Garnet introduit la notion de *pointeur*, au sens des pointeurs dans le langage C, dans les variables. Cela leur permet, comme dans l'exemple précédent, de ne pas reconfigurer à chaque fois la contrainte si celle-ci est relative à un autre objet qui est susceptible d'être modifié au cours de l'exécution. Dans notre exemple, il faudrait reconfigurer la contrainte chaque fois que l'objet situé au dessus du cercle change.

2.4.3 Icon

Icon [Dragicevic 04b] gère les dispositifs en entrée d'une machine afin de pouvoir configurer de manière dynamique et visuelle le comportement de ceux ci. Le langage utilisé est un langage visuel pour la programmation du flot de données.

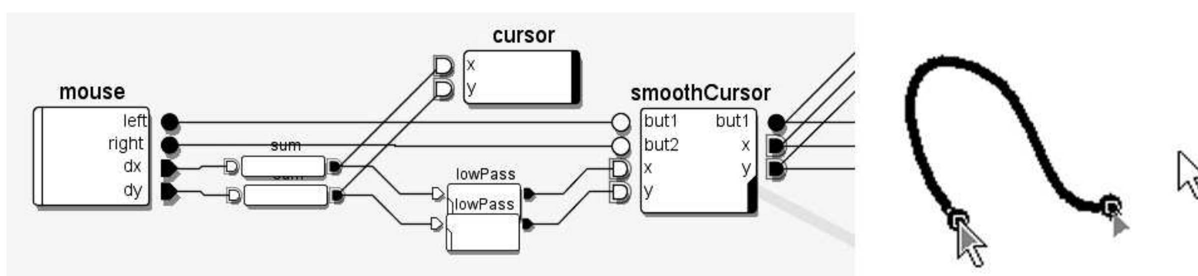

FIGURE 2.4 – Une configuration d'Icon [Dragicevic 04b].

2.4.4 Functional Reactive Programming

La Programmation Fonctionnelle Réactive (FRP) [Nilsson 02] est une adaptation du système de flot de données pour la programmation des systèmes interactifs. La FRP introduit la notion de *signaux* et de *fonctions* de ces signaux vers d'autres signaux. Les signaux peuvent être discrets ou continus permettant de gérer différents types de flux en entrée : périphériques d'entrée, capteurs de robots, etc. La FRP permet d'envisager de manière simple et compréhensible par les programmeurs des problèmes complexes ayant différentes sources d'entrées très dynamiques.

2.5 Le modèle à flot de données pour la création de visualisations

[Card 99] propose un modèle pour décrire les visualisations comme une séquence de flot de données depuis des données brutes vers les vues (Figure 2.5). Ce modèle de flot de données est toujours utilisé dans beaucoup de logiciels de visualisation (SpotFire [Ahlberg 96], VQE [Derthick 97], InfoVis Toolkit [Fekete 04], ILOG Discovery [Baudel 04], nVizN [Wilkinson 99]...).

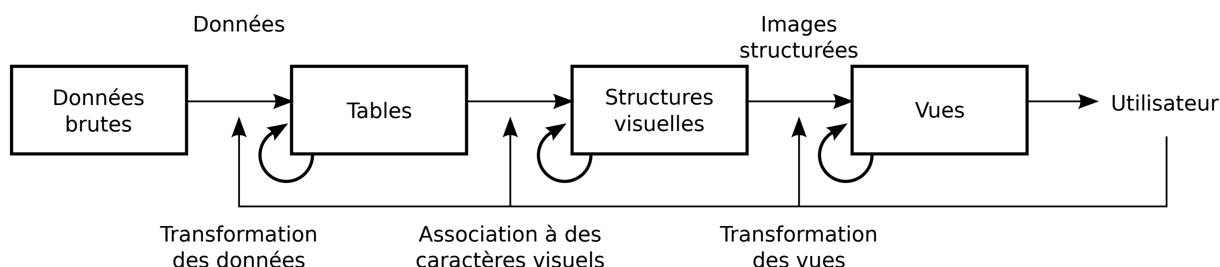

FIGURE 2.5 – Le modèle de flot de donné pour l'exploration de données [Card 99].

Le modèle de flot de données permet de faire la correspondance entre les données à afficher et leur représentation [Hurter 10]. Dans ce modèle, les données sont transformées vers une représentation au travers de trois étapes paramétrables par l'utilisateur. Les données sont tout d'abord formatées dans des tables. Ceci permet d'enrichir les données en injectant de nouvelles données calculées en fonction des données brutes. Ensuite, ces

tables sont utilisées pour faire la correspondance avec des éléments visuels, par exemple les variables visuelles de [Bertin 83]. Enfin, les entités visuelles sont transformées pour être présentées à l'écran. L'utilisateur peut ensuite interagir avec n'importe laquelle de ces étapes pour modifier le visuel, changer de représentation, zoomer, etc....

2.6 Conclusion

Au cours de ce chapitre, nous avons présenté les systèmes à flots de données. Ces systèmes possèdent de bonnes propriétés (pas d'effets de bord, exécution contrôlée) et peuvent être parallélisés. Ceci a permis leur utilisation dans des outils de production d'applications graphiques interactives. Nous avons aussi abordé le concept de contrainte qui est un proche voisin du système à flot de données.

Dans le cadre de ces travaux, nous utilisons le flot de données pour gérer les dépendances entre les différents graphes. Ces graphes subissent en effet diverses transformations. Nous allons voir dans la partie suivante les analogies que l'on peut faire entre ces transformations et la théorie de la compilation.

Chapitre 3

Compilation

Sommaire

3.1 Introduction **29**
3.2 Principes généraux **30**
3.2.1 Un ensemble processeur de langage 30
3.2.2 Description des étapes de compilation 30
3.3 Les principales optimisations réalisables **32**
3.3.1 Transformations conservant la fonctionnalité 32
3.3.2 Optimisation des blocs de base 34
3.3.3 Optimisations durant l'édition des liens 34
3.3.4 Optimisation à la volée 34
3.3.5 Optimisations dépendant de l'architecture cible 34
3.4 Compilation et graphisme **35**
3.4.1 Transformations et graphisme interactif : Indigo 35
3.4.2 Compilations de bas niveau et graphisme. 35
3.5 Conclusion **36**

3.1 Introduction

Nous avons vu au cours du Chapitre 1 quelques méthodes de production des systèmes graphiques interactifs. Nous avons pu constater que le schéma d'implémentation de tels systèmes était relativement constant quelque soit la méthode employée. Il est en effet nécessaire d'exprimer les éléments graphiques dans une certaine représentation (que se soit du code ou du dessin). Puis des transformations sont appliquées, avant de pouvoir dessiner les items graphiques dans l'application finale.

Au cours de ces travaux, nous faisons le rapprochement entre ces méthodes de production et la notion de compilateur :

> Un compilateur est un programme, ou un ensemble de programmes qui traduit un texte écrit dans un langage de programmation - le langage *source* - en un autre langage de programmation - le langage *cible*.[Aho 86]

Nous allons voir dans ce chapitre les principes généraux de la compilation ainsi que différentes techniques d'optimisation à notre disposition. Enfin, nous verrons les techniques "manuelles" employées aujourd'hui pour optimiser les applications graphiques interactives.

3.2 Principes généraux

L'objectif de cette section n'est pas de fournir une analyse exhaustive de la théorie de la compilation, mais de présenter les définitions et les principes généraux nécessaires à la compréhension de ces travaux.

3.2.1 Un ensemble processeur de langage

Tout d'abord, un compilateur prend en entrée un **langage source** et sort un code produit dans un **langage cible**. Le programme source est généralement écrit par un humain, et le langage cible est généralement de l'assembleur. Néanmoins, le terme générique de "compilateur" regroupe en fait plusieurs passes (Figure 3.1).

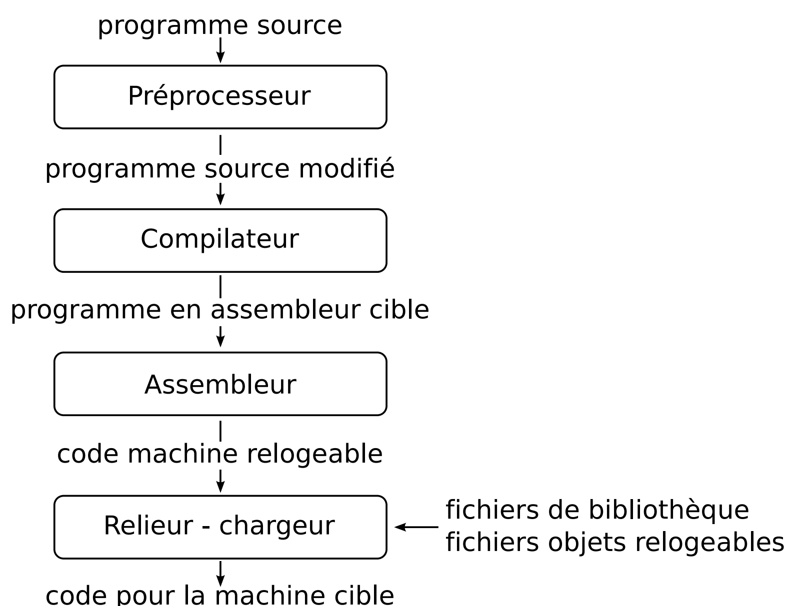

FIGURE 3.1 – Schéma général d'un compilateur (tiré de [Aho 86])

Tout d'abord, certains langages comme le C admettent une phase de **pré-processing** qui permet d'assembler un ensemble de fichiers et de faire des remplacements d'abbréviations (les *macros*). Ensuite, vient la phase de génération de code **assembleur** puis la phase de génération du **code machine relogeable**. Enfin, ce code machine relogeable est lié aux autres fichiers de bibliothèque relogeable pour former l'**exécutable cible**.

3.2.2 Description des étapes de compilation

Le traitement interne du compilateur est lui aussi scindé en plusieurs passes (Figure 3.2).

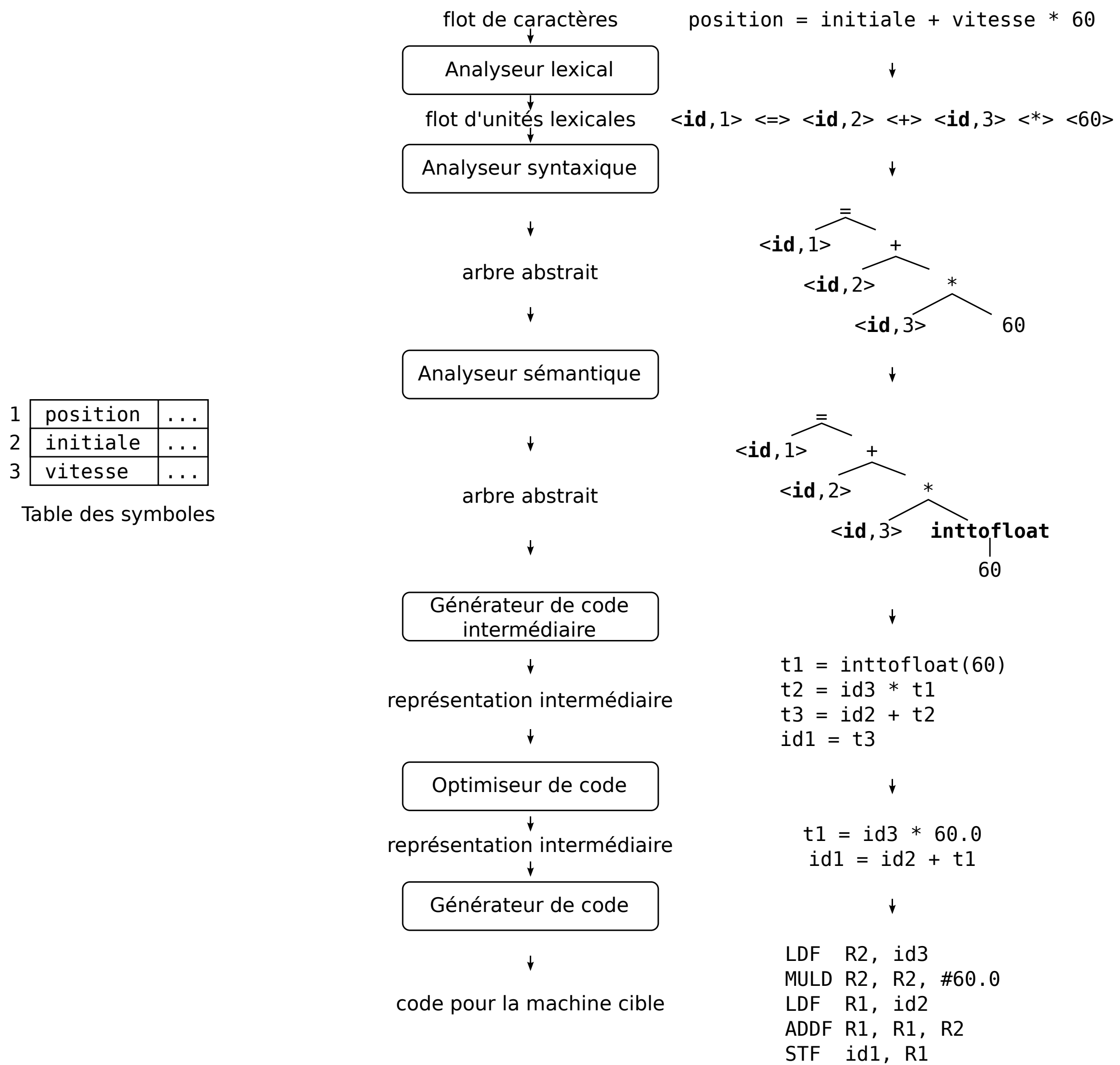

FIGURE 3.2 – Phases d'un compilateur (tiré de [Aho 86])

Le compilateur effectue d'abord l'**analyse lexicale** du programme source. Cette étape consiste à scinder les différents groupes de caractères en un ensemble de **lexèmes**. À chacun des lexèmes est associé un **identificateur** (la suite de caractères qui le compose) et une position dans la *table des symboles* si le lexème n'est pas un mot-clé du langage.

Une fois cette étape de séparation des unités lexicales réalisée, l'étape d'**analyse syntaxique** produit le graphe qui va dépeindre la structure *grammaticale* du programme source. Cette étape permet de vérifier que le programme est conforme à sa grammaire et permet de produire le premier arbre de la chaine.

Cet arbre est ensuite validé par l'étape de l'**analyse sémantique**. Cette analyse vérifie que chacune des opérations contenues dans l'arbre est valide, et permet aussi de *contrôler les types*.

Ensuite, l'arbre abstrait est traduit en un **code intermédiaire**. Dans l'exemple que nous déployons ici, le compilateur n'utilise qu'une seule représentation intermédiaire, mais il est fréquent qu'il y en ait plusieurs. Ceci permet d'effectuer différentes optimisations liées soit à l'architecture cible, soit au langage source par exemple.

Ce code intermédiaire est ensuite **optimisé**. Nous reviendrons plus en détail sur ces optimisations dans la section 3.3.

Enfin, le code optimisé est converti en **code assembleur**, code proche du binaire du processeur, mais sous forme textuelle. Le code binaire, appelé **code machine**, est produit par le couple assembleur - relieur.

3.3 Les principales optimisations réalisables

Nous nous baserons sur [Aho 86] et [Burke 93] pour la partie concernant la compilation statique, et sur [Krall 98], [Ishizaki 03] et [Lattner 04] pour celle sur la compilation dynamique.

3.3.1 Transformations conservant la fonctionnalité

Élimination de code mort

Lors de l'analyse du code, il arrive fréquemment que ce dernier comporte des instructions qui ne seront jamais évaluées ou qui ne devraient pas être ré-évaluées ou dont le résultat n'est jamais utilisé. Ces instructions peuvent donc être éliminées. Le travail du compilateur va donc être de détecter ces calculs inutiles (mais aussi les opérations qui ne seront jamais appelées) et de les remplacer par des instructions moins coûteuses (stockage dans des variables intermédiaires ou suppression par exemple). Ce travail s'appelle de l'élimination de code mort (ou *dead-code elimination*).

L'objectif est ici double : d'une part cela réduit la taille de l'exécutable final, et d'autre part, le code est plus efficace car les instructions exécutées sont plus proches les unes des autres, ce qui a pour effet de diminuer les défauts de cache du processeur.

Élimination des sous-expressions communes

Il arrive fréquemment que le code généré comprenne des sous-expressions communes, c'est à dire des productions de code identique (figure 3.3 à gauche, termes en "$d+e$"). Ces sous-expressions ralentissent les temps de calcul puisque qu'en mémorisant le résultat on peut éviter de tout recalculer (figure 3.3 à droite).

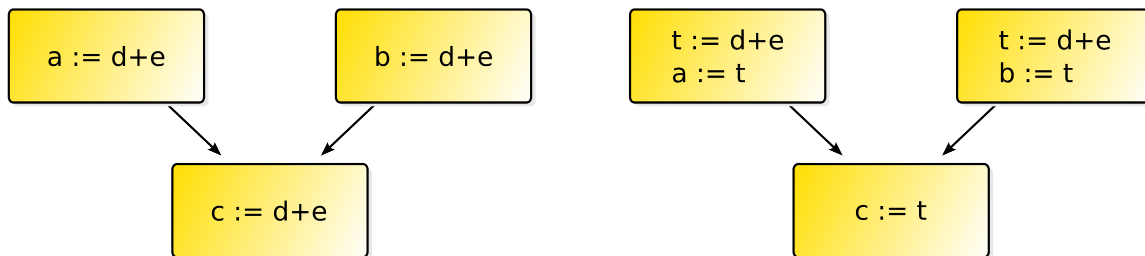

FIGURE 3.3 – Résultat de l'élimination de sous-expressions communes (exemple tiré de [Aho 86])

Propagation des copies

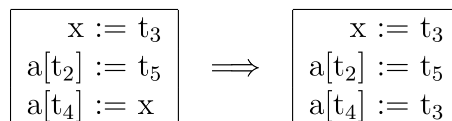

FIGURE 3.4 – Résultat de la propagation des copies

Le deuxième point abordé par [Aho 86] concerne la propagation des copies. Le principe de cette optimisation consiste à éliminer les références vers certaines variables dans le but de pouvoir les supprimer plus tard. La figure 3.4 illustre ce principe. Dans la partie gauche, la variable "x" est affectée puis réutilisée à la troisième ligne. En remplaçant ce "x" comme dans la partie droite de la figure par "t_3", la ligne 1 correspondant à l'affectation de "x" est alors inutile et pourra être supprimée si elle n'est pas utilisée plus loin dans le code.

Propagation des constantes

La propagation des constantes consiste à analyser le code et à extraire les valeurs constantes de leur registre.

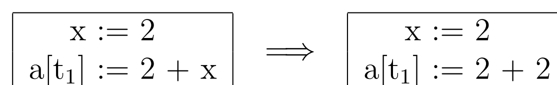

FIGURE 3.5 – Résultat de la propagation des constantes

La figure 3.5 illustre cette optimisation et l'intérêt que l'on peut en retirer : on peut à la fois simplifier l'affectation de la case mémoire "a[t_1]" et supprimer l'affectation de "x".

Optimisation des boucles

Dans la théorie des langages, la boucle tient une place très importante. De par sa nature répétitive, les compilateurs cherchent à optimiser au maximum ces portions de code afin de réduire le temps de calculs. On trouve dans ce type d'optimisation du déplacement de code (pour factoriser des calculs) ou de l'optimisation des variables d'induction [6].

3.3.2 Optimisation des blocs de base

Un **bloc de base** est une séquence d'instructions consécutives dans laquelle le flot de contrôle est activé au début de celle-ci, et inhibé à la fin, sans possibilité d'arrêt ou de branchement autre qu'à la fin de la séquence.

Un bloc est donc une suite d'instructions continue et sans possibilité de branchements. Les optimisations faites sur les blocs peuvent être, par exemple, de l'ordre des optimisations sur le flot de contrôle, de l'élimination de code inutile ou de l'échange d'instructions pour rendre l'ensemble du bloc plus performant (dans un système où l'addition est plus performante que la multiplication par exemple, l'instruction "2 * a" sera potentiellement remplacée par "a + a" ou "a < < 1").

3.3.3 Optimisations durant l'édition des liens

Effectuer des optimisations inter-procédurales consiste à regarder l'enchaînement des différents appels fonctionnels afin de trouver de nouvelles réductions de code. Un exemple de ce type d'optimisation consiste en l'élimination de fonctions non appelées : le compilateur construit un graphe d'appels des fonctions à partir de celles appelées dans le "main" ; toutes les fonctions non appelées sont alors éliminées.

3.3.4 Optimisation à la volée

Les compilateurs à la volée (Just In Time compilers, ou JIT) sont réellement nés avec LISP en 1960 [Aycock 03]. Un JIT permet de réaliser lors de l'exécution des optimisations et des nouvelles compilations afin d'améliorer la vitesse d'exécution du programme. Ceci est réalisable puisqu'à l'exécution, le compilateur dispose de nouvelles informations (le contexte d'exécution notamment) qui lui permettent de réduire la taille du code produit.

Smalltalk [Deutsch 84], Self [Hölzle 94], ou encore Java incluent aussi un JIT. Si les JIT font des optimisations à l'exécution [Krall 98], d'autres, comme LLVM [Lattner 04], peuvent même effectuer ces optimisation de manière *offline* après avoir récupéré des informations. LLVM est aussi capable de réaliser des optimisations interprocédurales comme présentées par [Burke 93].

3.3.5 Optimisations dépendant de l'architecture cible

Le dernier type d'optimisations abordé ici concerne celles dépendant de l'architecture cible. Les processeurs modernes présentent en effet des jeux d'instructions spéciaux

6. Une variable d'induction est une variable qui ne dépend que du compteur d'incrément de la boucle.

(SSE, MMX et autres) conçu pour améliorer l'efficacité du code dans certains cas précis. Par exemple, SSE permet de traiter en parallèle plusieurs instructions, et est donc particulièrement bien adapté à des opérations vectorielles.

3.4 Compilation et graphisme

3.4.1 Transformations et graphisme interactif : Indigo

L'utilisation de transformations qui s'appliquent sur une description de haut niveau a été étudiée dans le projet Indigo [Blanch 05].

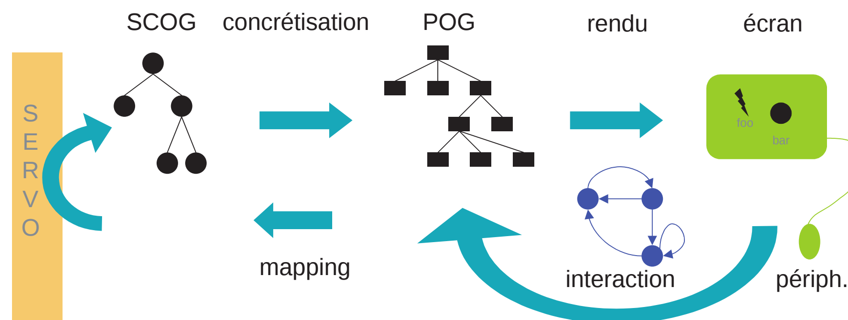

FIGURE 3.6 – Le fonctionnement général d'Indigo [Blanch 05].

Le fonctionnement général d'Indigo est présenté Figure 3.6. Le Servo contient le graphe unique représentant le modèle conceptuel de la scène (le COG). Ensuite le Servir communique avec le Servo au moyen d'un protocole pour gérer les entrées et les sorties. C'est dans le Servir qu'est contenu le graphe des objets perceptuels (le POG) et où est réalisé l'interaction. Une fois l'interaction traitée, le POG (et le COG) sont mis à jour puis les modifications sur l'affichage sont répercutées. Réaliser une telle architecture permet de placer le COG sur une machine distante et d'avoir plusieurs POG sur plusieurs clients distantes. À la différence du serveur graphique X11, le rendu et l'interaction sont à la fois gérés par le *Servir*, le serveur de l'architecture Indigo (Figure 3.7).

Cette idée de transformations a ensuite été étendu par l'implémentation d'un ensemble de widgets de la norme ARINC 661 [Barboni 07], et plus tard par le modèle MDPC [Conversy 08] que nous avons déjà présenté.

3.4.2 Compilations de bas niveau et graphisme.

La notion de compilation dans les graphismes a été introduite par Nitrous, un générateur pour des graphismes interactifs [Draves 96]. Cependant, comme dans [Peercy 00], le compilateur présenté n'est qu'orienté pixel, et ne gère pas les dispositifs d'entrée ou les entrées provenant de l'application elle-même.

LLVM [Lattner 04], avec la pile OpenGL développée par Apple, peut efficacement abstraire la description de l'interface avec le matériel cible. Le compilateur JIT inclus avec LLVM peut optimiser les différents shaders disponibles afin d'avoir l'implémentation la plus efficace possible.

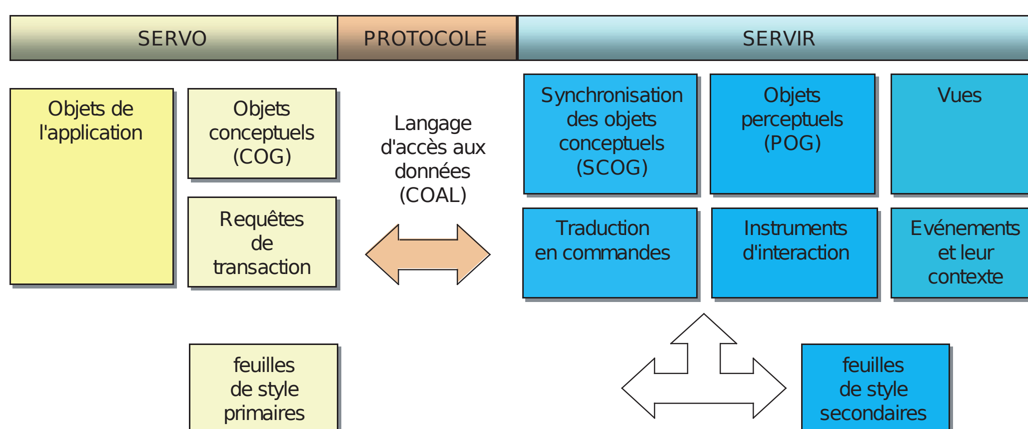

FIGURE 3.7 – L'architecture d'Indigo [Blanch 05].

D'autres travaux utilisent des optimisations semblables à celles réalisées par les compilateur pour produire un code efficace (Java3D, LLVM associé à Gallium3D [7] dans Mesa). Cependant, ces outils ne sont accessibles seulement qu'aux programmeurs de bas niveau (ceux capable de programmer les cartes graphiques directement). Ils ne sont pas supposés être utilisés par l'ensemble des programmeurs d'applications graphiques interactives.

Enfin, Protovis [Heer 10] a aussi récemment présenté des résultats sur l'intégration d'une chaine de compilation dans leur boîte à outils pour produire des applications sur différentes plateformes.

3.5 Conclusion

Dans ce chapitre, nous nous sommes intéressé à la théorie de la compilation. Cette théorie a pour base des transformées de langages, et donc de graphes. Ceci est proche des concepts présentés plus tôt pour la partie rendu des applications graphiques interactives. Nous pensons qu'il est possible d'adapter ces concepts au domaine des rendus graphiques afin de résoudre l'antinomie soulevée au cours du chapitre 1 : les applications ont du mal a être à la fois performantes et graphiquement riches.

Envisager le concept de rendu graphique sous l'aspect de la théorie de la compilation nous permet d'espérer des gains sur les optimisations automatiques à réaliser. La théorie de la compilation est riche d'une bibliothèque d'optimisations qui sont applicables pour partie au domaine du rendu graphique : sous-expressions communes, propagation des copies, propagation des constantes, optimisation des boucles, des blocs de base, ou encore optimisations inter-procédurales.

7. `http://wiki.freedesktop.org/wiki/Software/gallium`

Deuxième partie

Maximisation instrumentée de la séparation entre description et implémentation des systèmes graphiques interactifs

Chapitre 1

Démarche de conception instrumentée

Sommaire

1.1 Exigences 40
1.1.1 Donner du contrôle au designer 40
1.1.2 Minimiser l'apprentissage 41
1.1.3 Abstraire les éléments graphiques 41
1.1.4 Développer rapidement différentes alternatives 41
1.1.5 Garder un contrôle fin sur la richesse du rendu final 41
1.1.6 Modularité 42
1.1.7 Conserver de bonnes performances de rendu 42
1.2 Hayaku : design et concepts 42
1.2.1 Idée générale 42
1.2.2 Abstraire pour contrôler les éléments graphiques 43
1.2.3 Production de l'application graphique interactive 44
1.3 Scénarios d'utilisation 45
1.3.1 Scenario n°1 : Une application multi-touch 45
1.3.2 Scenario n°2 : une image radar 50
1.3.3 Scenario n°3 : un clavier virtuel 51
1.3.4 Scenario n°4 : intégrer des composants dans des applications existantes 54
1.4 Conclusion 58

Au cours de l'état de l'art présenté précédemment, nous avons vu que les boîtes à outils graphique interactives sont confrontées au problème de la maximisation de deux dimensions a priori antagonistes : les *performances de rendu* et le *pouvoir d'expression du designer*. Les *performances de rendu* concernent la vitesse d'exécution de l'application finale (présentée à l'utilisateur final). Il faut que cette vitesse soit compatible avec la boucle perception/action de l'utilisateur. Le *pouvoir d'expression du designer* est la possibilité offerte au designer de pouvoir réaliser sans contraintes le design graphique qu'il souhaite. Cette possibilité passe par un langage graphique lui permettant de représenter son idée, mais aussi par une méthode de production lui permettant d'obtenir "rapidement" un résultat afin de l'affiner.

Dans ce chapitre, nous présentons une méthode instrumentée de conception des interfaces homme-machine qui implique un *designer d'interactions*. Cette méthode s'appuie sur un ensemble de fonctionnalités spécifiques d'Hayaku, la boîte à outils graphique réalisée lors de cette thèse [Tissoires 11]. Ce chapitre ne porte que sur l'utilisation de l'outil par le *designer d'interactions*. La partie technique de la boîte à outils sera donnée dans le chapitre suivant.

1.1 Exigences

Tout d'abords, nous exprimons les exigences issues des analyses que nous avons présentées dans notre état de l'art.

1.1.1 Donner du contrôle au designer

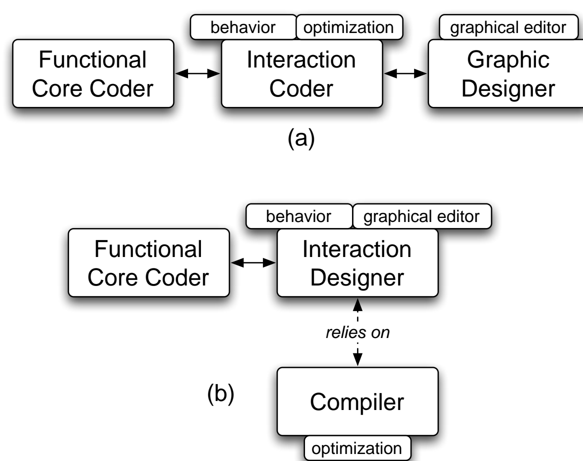

FIGURE 1.1 – La répartition des rôles avec Intuikit et XAML (a) et la répartition avec Hayaku (b).

Développer une interface homme-machine de qualité requiert d'introduire dans la boucle de développement un acteur chargé de la conception du graphisme, le *designer graphique*[Chatty 04]. Aujourd'hui, le designer intervient lors du cycle de développement

en amont de la phase de prototypage. En effet, ne pouvant exprimer directement dans l'application exécutable ses interactions et ses graphismes, il ne peut que fournir des spécifications. Il existe donc un autre acteur entre le designer et le reste de l'équipe, le programmeur d'interaction (Figure 1.1 a). Cependant, permettre au designer graphique de s'impliquer plus dans le développement du code final permet un gain en terme d'utilisabilité et donc de satisfaction de l'utilisateur final[Chatty 04, Dragicevic 05].

La première exigence de la méthode proposée est donc de fournir au designer des outils lui permettant d'intervenir directement dans le processus de création de l'application.

1.1.2 Minimiser l'apprentissage

Afin d'augmenter l'implication du designer graphique dans l'activité de développement de l'application interactive finale, il est nécessaire de lui fournir de nouveaux outils. Cependant, ces outils ne doivent pas transformer son métier actuel en un métier de codeur. En effet, le designer graphique doit voir son pouvoir d'expression maximisé. On ne peut lui demander de réaliser ses graphismes et ses interactions sous forme de lignes de code. Cela poserait le problème qu'on lui changerait son métier, et qu'il risque de se limiter lui-même compte tenu de ses propres capacités de codage. La deuxième exigence que nous pouvons énoncer est donc que le designer continue de travailler avec des outils de création graphique pour la partie graphique de l'application.

1.1.3 Abstraire les éléments graphiques

Si l'on souhaite inclure le designer graphique dans le développement et si l'on ne souhaite pas qu'il code la partie graphique, il est nécessaire d'abstraire les éléments graphiques pour pouvoir définir un langage commun entre les différents modules en charge du projet. Cette abstraction permet en plus de ne plus se soucier du rendu graphique lors du codage du reste de l'application. Il devient alors possible de tester différents graphismes sans avoir besoin de les coder réellement (comme dans [Hartmann 08]).

1.1.4 Développer rapidement différentes alternatives

Afin de réaliser une interface de qualité, les designers graphique ont besoin d'explorer différents designs lors du processus de création [Green 89]. Une bonne abstraction ainsi que l'utilisation d'outils de dessin permettent de tester rapidement le rendu de l'application finale. Ceci permet au designer graphique de se rendre compte très tôt de ses choix et donc de les modifier de manière incrémentale très rapidement.

1.1.5 Garder un contrôle fin sur la richesse du rendu final

Un des métiers du designer graphique, est de valider que le rendu final est conforme aux spécifications [Tabart 07]. Il est donc nécessaire de fournir au designer graphique la possibilité de travailler de manière très fine sur le rendu final. Il sera peut être amené à faire certaines concessions (en terme de performances du rendu final par exemple), mais ce sera son choix, et non un choix imposé par l'implémentation.

1.1.6 Modularité

Développer une application interactive est coûteux. Le design de cette application prend une part importante de ce coût. Il est donc nécessaire de permettre de réutiliser plus tard ces investissements dans de nouvelles applications. Cette approche passe par un développement modulaire. Grâce à un tel développement, les différents modules décrivant l'application graphique peuvent être réutilisés.

1.1.7 Conserver de bonnes performances de rendu

Enfin, le problème majeur des boîtes à outils graphiques post-WIMP à l'heure actuelle est que le résultat final est peu optimisé et souffre donc d'une vitesse de rendu faible. Il existe des boîtes à outils spécialisées comme Prefuse [Heer 05] ou Piccolo [Bederson 00] qui sont performantes, mais pas généralistes. En effet, elles vont maximiser la performance au détriment du pouvoir d'expression du designer graphique.

1.2 Hayaku : design et concepts

Cette partie présente les services offerts par la boîte à outils graphique développée afin de répondre aux différents problèmes posés précédemment. Nous présentons ici *Hayaku*[Tissoires 09], une boîte à outils qui instrumente l'activité de design des applications graphiques interactives. De nombreux environnements de développements tentent de maximiser l'expérience utilisateur, donc le pouvoir d'expression du designer. Cependant, ils ne maximisent pas toutes les exigences ci-dessus, et notamment celle relative aux contraintes de performance. Hayaku répond à ces exigences, notamment en se reposant sur un compilateur graphique [Tissoires 08].

1.2.1 Idée générale

L'idée générale de notre approche diffère de l'approche proposée par [Chatty 04] : au lieu d'accepter le fait que le designer graphique et le programmeur d'interactions sont deux acteurs distincts, et donc de maximiser la communication entre ces deux personnes, nous avons donné au designer les poignées nécessaires pour contrôler le design graphique. Ceci nous permet de transformer le designer graphique en *designer d'interactions*. Comme le montre la Figure 1.1, Hayaku donne le pouvoir d'expression nécessaire au designer, tandis qu'il transfère le problème des optimisations vers le compilateur graphique. Ainsi, le designer d'interaction contrôle mieux le rendu final. Ceci lui permet de tester et valider les interactions plus tôt dans le processus de développement. Il peut donc les contrôler de manière plus fine. Cette approche est similaire à Artistic Resizing[Dragicevic 06] : au lieu de décrire par du code le comportement graphique des éléments dont la taille change, Artistic Resizing permet au designer graphique d'exprimer le comportement au travers d'exemples des différents états. Cette méthode est plus proche de son métier puisqu'il peut exprimer dans son langage le comportement qu'il souhaite sans biais introduit par un langage de programmation.

1.2.2 Abstraire pour contrôler les éléments graphiques

Nous fournissons au designer d'interaction une chaîne d'outils qui utilisent en premier lieu un format de dessin vectoriel standard : SVG, Scalable Vector Graphics. Ce format est utilisable par des outils standards de production de graphismes comme Inkscape ou Adobe Illustrator. Le designer peut ainsi se reposer sur son expérience avec ces outils.

Le designer définit par l'intermédiaire des dessins SVG une représentation d'un type d'objet (un bouton, un item de l'application interactive). Ces dessins SVG sont l'équivalent de "classes" graphiques. Durant l'exécution du programme, des "instances" de ces "classes" vont être crées, chacune de ces "instances" ayant ses propres états.

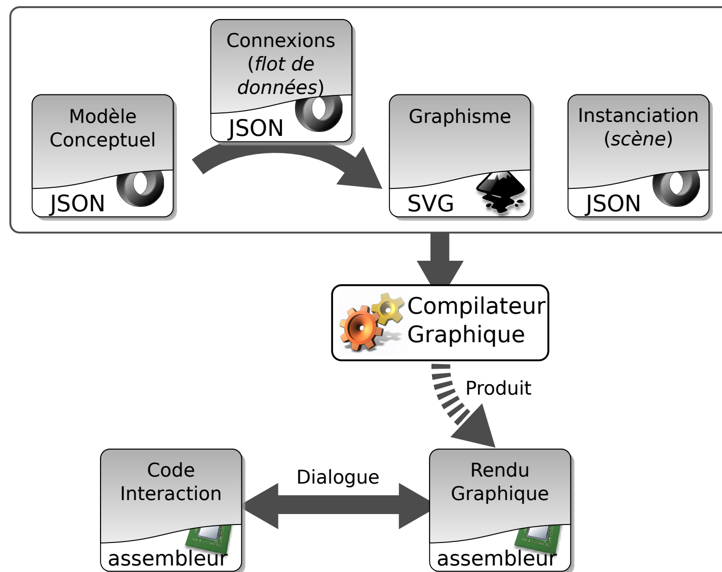

FIGURE 1.2 – Schéma de principe d'Hayaku.

Pour pouvoir utiliser ces objets SVG, le designer d'interaction doit fournir à Hayaku trois descriptions, toutes écrites dans le langage JSON [8] (Figure 1.2). La première, appelée *modèle conceptuel*, sert à définir un langage commun entre le designer d'interactions et les programmeurs du noyau fonctionnel. Ce langage est utilisé comme une passerelle entre le noyau fonctionnel et les graphismes interactifs. Dans le cas de la description d'un bouton poussoir, le fichier SVG contient le graphisme du bouton, et son fichier de modèle contient une représentation abstraite de sa structure : position (x,y), état (enfoncé, relâché, survol), largeur, etc... De même que dans [Chatty 04], les designer d'interactions et le reste de l'équipe doivent alors se mettre d'accord pour produire ces *modèles*. Ils définissent un contrat (ou une sorte d'interface au sens Java) entre la partie graphique et le noyau fonctionnel. Au cours de la phase de compilation, ces modèles sont traduits en classes.

Avec Hayaku, ce modèle est fortement lié au graphisme. Ceci pose problème puisque nous pensions séparer les objets du modèle abstrait des objets graphiques. Il est nécessaire de décrire beaucoup de propriétés graphiques dans ce modèle. Par exemple, dans le cas du bouton poussoir, le modèle idéal ne contiendrait que la propriété "enfoncé". Cependant,

8. Javascript Object Notation

Hayaku oblige à inclure dans ce modèle des propriétés graphiques (x, y, taille, taille de l'ombre portée, couleur, couleur de l'ombre, etc...) afin de pouvoir effectuer le rendu de l'application.

Pour faciliter l'écriture souvent redondante des propriétés des objets du modèle, nous offrons au designer d'interaction un mini-langage mathématique qui s'apparente à du flot de données. Ainsi, il lui est possible de décrire des attributs du modèle comme relatifs les uns aux autres. Dans notre exemple, la taille de la police du texte employé peut être liée à la largeur de l'item.

La deuxième description fournie par le designer d'interaction, la description des *connexions*, décrit comment le modèle définit précédemment va être relié aux graphismes SVG. Cette description fournit des connexions entre les différents champs du modèle conceptuel vers les différents nœuds ou attributs de ces nœuds de l'arbre SVG. Le designer nomme les groupes graphiques SVG et se sert de ces identifiants pour réaliser la liaison dans le fichier de connexions. Avec l'exemple du bouton, le fichier de connexions stipulera que le champ X du modèle, sera lié à l'attribut *translation* du groupe de plus haut niveau de l'objet SVG. Ainsi, X et Y seront attachés à la position réelle de l'objet dans le graphe de scène produit. En un sens, cette description est similaire aux feuilles de styles que l'on peut trouver dans HTML par exemple.

Enfin, la troisième description qui doit être fournie par le designer d'interaction est celle de la *scène*. Ceci lui permet d'instancier les différentes instances des classes précédemment décrites, afin de composer l'application graphique finale. Ainsi, dans ce fichier, il stipulera les positions des différents boutons de l'application finale, ainsi que leur label si ces paramètres font partie du modèle.

Même si cette modélisation conceptuelle des applications graphiques peut paraître compliquée, elle ne l'est en réalité pas plus que les moyens existants pour produire le code de ces applications : la première description fournie en JSON peut être considérée comme la définition de classes, la seconde comme une feuille de style graphique, tandis que la dernière peut correspondre à la phase d'instanciation des différentes classes lors du début de l'exécution du programme. Le seul ajout est la description SVG, qui correspond à une définition des "*classes graphiques*".

1.2.3 Production de l'application graphique interactive

La structure présentée ci-dessus permet de garantir une séparation entre la description des formes graphique et son implémentation. Une implémentation naïve de la boîte à outils consisterait à conserver ces différents graphes en mémoire lors de l'exécution. Lors de la modification d'une propriété d'élément, il suffit alors de ré-interpréter ces graphes pour redessiner la scène.

Cependant, la vitesse d'exécution d'une telle implémentation est lente. Il est donc nécessaire de réaliser des transformations pour convertir ces quatre langages en un langage admissible par le moteur de rendu. De telles transformations sont très coûteuses. En se reposant sur un compilateur graphique, Hayaku permet au concepteur de contourner le problème de la description et de l'optimisation de ces transformations. Ce compilateur prend en entrée une description SVG et les trois fichiers écrits en JSON, et génère une application.

1.3 Scénarios d'utilisation

Nous allons présenter quatre scénarios d'usage afin d'illustrer les concepts que nous avons présentés plus haut.

Le premier scénario introduit la chaine d'outils. Les scénarios suivants sont plus complexes et montrer la diversité d'usage d'Hayaku.

1.3.1 Scenario n°1 : Une application multi-touch

Afin de présenter notre approche, nous allons tout d'abord présenter une application simple. Cette application permet à un utilisateur déplacer, orienter et retailler des objets graphiques à l'aide d'un écran multi-touch [9]. Cette première application peut sembler pauvre en termes de graphismes et d'interactions, mais sa simplicité permet une introduction à Hayaku, et permet de montrer des extraits de codes courts.

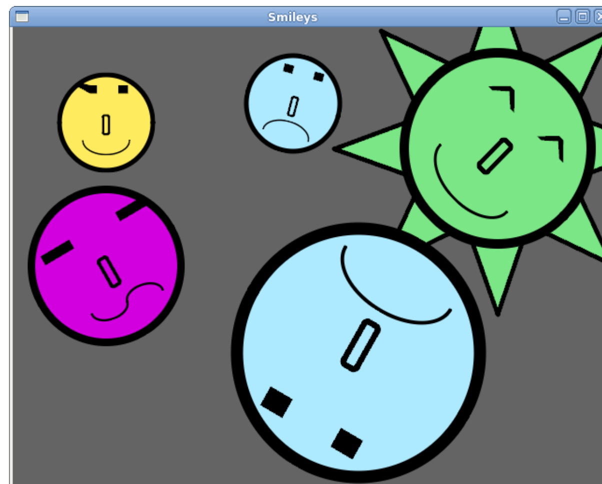

FIGURE 1.3 – Une application multi-touch simple.

Ce scenario d'utilisation concerne le développement d'une application qui supporte les interactions de type "multi-touch" (Figure 1.3) pour l'utilisateur final. L'interaction consiste à contrôler de manière indépendante (grâce au paradigme de la manipulation directe) chacun des composants visuels présents sur la figure 1.3. Les propriétés manipulables par l'utilisateur final sont donc la position de l'objet, sa rotation et sa mise à l'échelle. L'ensemble du code nécessaire à l'exécution de ce programme est fourni dans l'annexe A p. 113.

La première étape du processus consiste à définir les quatre *classes graphiques* (les différents "visages" de la figure 1.3) à l'aide du logiciel Inkscape, et de les enregistrer au format SVG (le code est fourni dans l'annexe A. Les formes sont centrées autour du point (0,0) afin de conserver des paramètres de rotation et de mise à l'échelle cohérents entre les

9. Nous avons choisi ici et dans le reste de ce document de conserver le terme anglais multi-touch puisque c'est le terme le plus communément utilisé, "multi-touches" ou "multi-touchés" pouvant être les versions françaises.

différentes parties qui composent les objets. Ce centrage permet aussi de simplifier le code des transformations à appliquer lors du traitement des données d'entrée de l'utilisateur.

La deuxième étape consiste à définir le *modèle conceptuel* du composant graphique à l'aide d'un fichier JSON. Puisque cette application ne nécessite que la rotation et la mise à l'échelle des différentes formes graphiques précédemment décrites, le modèle conceptuel est relativement simple : chacun des objets graphiques a besoin d'un paramètre de position, de rotation et de mise à l'échelle. Le listing 1.1 résume cette description en utilisant la syntaxe JSON. Toutes les formes graphiques partagent la même structure, et les mêmes transformations graphiques. La boîte à outils ne permet qu'une correspondance de type "un vers un" entre les objets du modèle et les objets graphiques (à chaque classe du modèle ne correspond qu'un seul visuel). Il faut donc sous-classer la description principale des objets en quatre sous-classes, une pour chaque représentation graphique (une seule de ces sous-classes est montrée dans le listing 1.1 : Object_0).

Listing 1.1 – Exemple d'implémentation des items multi-touch à l'aide de la syntaxe JSON. Les "v" devant les types (vint ou vfloat) spécifient que les symboles sont des variables dont le contenu peut changer lors de l'exécution.

```
{
    "model": "SMILEYS",
    "classes": [ {
        "name":"PastilleCommune",
        "extends":null,
        "attributes": {
            "X0":"vint",
            "Y0":"vint",
            "SCALE":"vfloat",
            "ROTATION":"vfloat",
            "Z-ORDER":"vfloat",
            "Picking_Key":"vint"
        }
    },
    {
        "name":"Object_0",
        "extends":"PastilleCommune",
        "attributes": {}
    }]
}
```

La troisième étape de conception consiste à définir les connexions entre le modèle conceptuel et les éléments graphiques (Listing 1.2), toujours dans un fichier utilisant la syntaxe JSON. Ces connexions consistent en une suite de paires clés-valeurs dans la partie "connexion" de la description JSON. Les autres éléments de ce fichier permettent de faire le lien entre le modèle utilisé (mot clé *model*), quels sont les objets de ce modèle à représenter graphiquement (mot clé *conceptualClass*), dans quel fichier SVG se trouvent les items graphiques (mot clé *graphicFile*) et quel est le groupe graphique dans ce fichier se rattachant à l'objet conceptuel (mot clé *graphicClass*). Enfin, L'utilisateur de Hayaku a la possibilité de spécifier si une ou plusieurs des variables du modèle conceptuel recevront

les évènements de notification lorsque l'un des doigts sera sur la forme graphique.

Listing 1.2 – Exemple de fichier décrivant la connexion entre les éléments du modèle conceptuel et leurs duaux graphiques (le groupe nommé smiley_svg) à l'aide de la syntaxe JSON.

```
{
  "model":"SMILEYS",
  "objects": [
  {
    "conceptualClass":"Object_0",
    "graphicFile":"demo.svg",
    "graphicalItems": [
    {
      "graphicClass":"smiley_svg",
      "connections":
      {
        "X0":"smiley_svg.transform.tx",
        "Y0":"smiley_svg.transform.ty",
        "SCALE":"smiley_svg.transform.scale",
        "ROTATION":"smiley_svg.transform.rotation",
        "PRIORITY":"smiley_svg.transform.priority"
      },
      "picking":
      {
         "Picked_Key":"smiley_svg"
       } } ]
  }]
}
```

Enfin, la quatrième et dernière description de l'application utilisant la syntaxe JSON consiste à fournir au système les instanciations des différents composants graphiques de la scène finale (Listing 1.3).

Listing 1.3 – Exemple de l'instanciation des éléments multi-touch à l'aide de la syntaxe JSON.

```
{   "name":"Smileys",
    "model":"SMILEYS",
    "content": [
    {   "type":"Object_0",
        "attributes": {
            "ID":0,
            "ParentID":0,
            "X0":100,
            "Y0":100,
            "SCALE":0.5,
            "ROTATION":0.0,
            "Picked_Key":-1
        } } ]
```

```
}
```

Le designer graphique doit aussi fournir au système la partie interaction, c'est à dire la connexion entre les différents évènements d'entrée et la réaction associée aux composants graphiques. Hayaku ne fournit pas de support explicite aux dispositifs d'entrée. C'est donc au concepteur de décrire à l'aide d'une plateforme cible (Python + OpenGL ou Java + Swing), dans le langage cible et à l'aide de la gestion des entrées à sa disposition comment ces évènements vont affecter le modèle conceptuel. Ces modifications seront ensuite transmises vers le rendu graphique de manière automatique. Cependant, lors de la génération du code correspondant au modèle conceptuel, Hayaku offre la possibilité d'ajouter du code arbitraire défini par l'utilisateur, ce qui permet au concepteur d'abstraire le comportement (voir le listing 1.4). De plus, Hayaku fourni un mécanisme de désignation d'items graphiques (le terme anglais étant le *picking*), qui peut être appelé depuis le code utilisateur.

Listing 1.4 – Le code python des trois commandes qui permettent de contrôler les objets graphiques.

```
class Smiley(object):
    # ....
    def mouseMotionCallback(self, idMouse,x,y):
        if self.state == 'move':
            dx = x - self.old_x
            dy = y - self.old_y
            self.old_x, self.old_y = x, y
            self.translate(dx,dy)
        elif self.state == 'rotate':
            dy = y - self.old_y
            self.old_y = y
            self.rotate(dy)

    def translate(self,dx,dy):
        self.x0.set(self.x0.eval() + dx)
        self.y0.set(self.y0.eval() + dy)

    def rotate(self, dr):
        self.rotation.set(self.rotation.eval() + dr)

    def zoom(self, z):
        if self.scale.eval() + z >= 0.1:
            self.scale.set(self.scale.eval() + z)
```

Afin de tester et lancer l'application, l'utilisateur édite un script python qui contient un appel vers la fonction *load* (Listing 1.5, partie *main*). Cette fonction prend en paramètre les trois fichiers JSON précédemment décrits (le modèle conceptuel, les connexions du modèle vers le SVG, et la scène finale). L'utilisateur peut ensuite lancer la commande *hayaku* avec ce script en paramètre. Si la phase de compilation réussit, Hayaku lance l'application générée.

Listing 1.5 – Le code python du script de lancement de l'application interactive.

```
def addObject(keyclass, o, connections):
    # on ajoute l'objet o a la liste interne des objets connus
    global objects
    s = keyclass(o.ID)
    objects.append(s)
    # et on ajoute des callbacks eventuelles (picking ici)
    connections.append((s.GraphicObject.Picked_Key,
                                          s.picking_callback))

def parseScene(parent,scene,connections):
    # Construction de la "scene"
    # ajout des callbacks sur la souris
    for b in mouse().buttons:
        connections.append((b, Demo_Object_mouseButton))
    connections.append((mouse().x, Demo_Object_mouseMotion))
    connections.append((mouse().y, Demo_Object_mouseMotion))
    # traitement de la scene
    for o in scene:
        if o.getClass() in ['Object_0','Object_1','Object_2',
                                    'Object_3']:
            addObject(Demo_Object, o, connections)

    return connections

def onScene(scene):
    # callback appelee une fois le la scene generee
    Connects = []
    Connects = parseScene(None,scene,Connects)
    gc.connect(Connects) # binds the callbacks to the dataflow

## main
if __name__ == "__main__":
    import graphical_compiler.gc as gc
    gc.load(conceptualModel = "smileys_model.json",
            graphicLass = "smileys_modelToSVG.json",
            scene = "smileys.json",
            callback = onScene,  width = 640, height = 480,
                backColor = (100/255.0,100/255.0,100/255.0))
    glvm.wait()
```

Sur un core 2 duo cadencé à 2,4 GHz, le temps de compilation pour cet exemple est de 2,2 secondes lors du premier lancement. Les recompilations suivantes nécessitent moins de temps (1,9 secondes) puisque le compilateur graphique n'a plus besoin de recompiler des parties de l'application qui ne changent pas entre deux lancements. En effet, le compilateur graphique compile séparément les différents fichiers sources et les composants graphiques

(mais aussi les *shaders*[10], les polices de caractère ou certaines fonctions utilitaires). Ces fichiers sont générés dans un sous-dossier “BUILD”, et le compilateur se base sur l’heure de création entre les sources et les binaires pour savoir si les résultats de la compilation sont encore valides.

L’application prend ensuite moins d’une seconde pour se lancer, et tourne à 515 images par secondes en moyenne (voir le tableau 3.2 p. 81). La machine de test est équipée d’une carte NVIDIA GeForce 8800M. Encore une fois, cette application graphique est simple et peu coûteuse en termes de puissance de calcul. Même si la partie graphique est simple, cet exemple montre que la boîte à outils est suffisamment réactive pour traiter une fréquence de données d’entrée importante avec une faible latence.

1.3.2 Scenario n°2 : une image radar

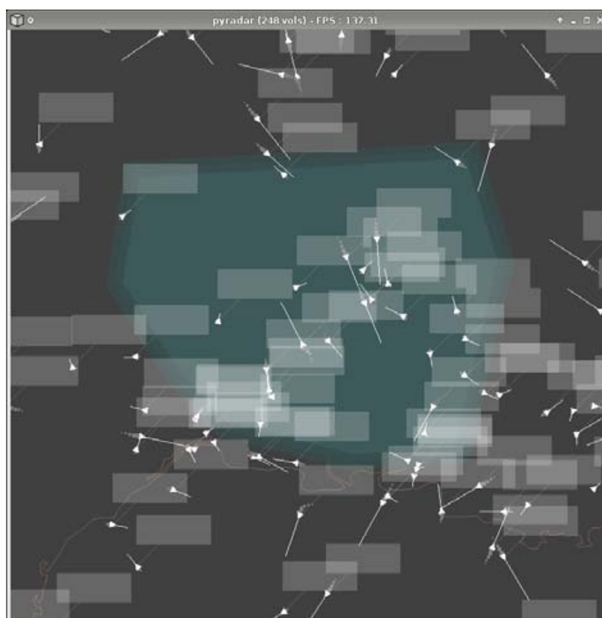

FIGURE 1.4 – Exemple d’une application fournissant une image radar rendue avec Hayaku.

L’objectif du deuxième scénario que nous présentons ici est de vérifier la performance du rendu graphique de Hayaku dans le cadre d’une scène complexe qui est connectée à un serveur de données externes.

Cette deuxième application est une image radar affichant les avions au contrôleur aérien (Figure 1.4). Elle est plus gourmande en termes de puissance de calculs. Cette application doit être capable d’afficher jusqu’à 500 avions, chacun étant composé de 10 éléments graphiques. Chacun de ces avions représente donc 20 triangles. Les textes des étiquettes des avions ne sont pas affichés afin de ne pas influencer sur le nombre de triangles dessinés (le rendu des fontes est entièrement vectoriel avec Hayaku, et peut

10. Un shader est un programme exécuté sur le processeur graphique et nécessaire pour le rendu d’une scène. Depuis les dernières versions d’OpenGL et DirectX, le processeur graphique est entièrement programmable et il est obligatoire de définir de tels shaders.

potentiellement être très coûteux). Ici, Hayaku affiche plus de 10000 triangles (500 * 20 triangles) à chaque image sans impacter sur les performances de rendu : le taux de rafraichissement est supérieur à 500 images par secondes sur notre machine de test.

1.3.3 Scenario n°3 : un clavier virtuel

Le troisième scenario présenté ici est une démonstration du pouvoir d'expression offert par la boîte à outils. Nous avons expérimenté avec un designer graphique la réalisation d'une application, sans imposer de contraintes autres que celles introduites par le support partiel de la norme SVG par Hayaku. En effet, pour des choix d'implémentation les cubiques de Bézier, et les filtres notamment ne sont pas gérés par la boîte à outils. Les exigences relatives à ce scénario sont les suivantes : tout d'abord, ce scénario est un test grandeur nature de l'application. Il faut donc que le designer d'interactions puisse développer de manière autonome (une fois la phase d'apprentissage de la boîte à outils réalisée) la partie graphique de son application. Il faut ensuite que la qualité du rendu soit suffisamment correcte pour que le designer puisse s'en satisfaire. Hayaku doit aussi favoriser la conception incrémentale de l'application interactive. Enfin, il faut que l'application finale soit réactive (< 100 ms), pour ne pas perturber la boucle perception-action. L'ensemble du code est fourni dans l'annexe B.

FIGURE 1.5 – L'application test clavier en mode "expanding keyboard".

Description de l'application test

L'application choisie consiste à réaliser un clavier logiciel de 40 touches redimensionnables (cf Figure 1.5). Pour offrir un jeu de caractères étendu, 2 touches de fonction permettent de permuter les touches en modes *capitale* et *minuscule* (avec accentuations), ou en mode *numérique*. Un afficheur permet de visualiser le texte saisi. Le clavier bénéficie aussi d'un redimensionnement adapté à la taille de la fenêtre qui le contient. Enfin, le clavier intègre une fonction *"expanding keyboard"*[Raynal 07], d'agrandissement des touches (de type *fish-eye*) lié à la position de la souris. Ceci permet de supporter des affichages de faible résolution.

Le design graphique du clavier (type 2D avec des couches, avec des effets 3D simulés : ombrage, gradients, etc...) utilise une description uniquement vectorielle des composants, pour garantir une meilleure qualité au redimensionnement, et met en œuvre des capacités graphiques "riches" : dégradés, transparence, antialiasing, ombres portées... Concernant les interactions, des feedbacks visuels accompagnent les événements du curseur associés aux objets de représentation des touches, et sont graphiquement réalisés par la modification des propriétés SVG des composants.

Ce travail de conception avait été réalisé auparavant par le designer graphique pour la réalisation d'une autre application.

Les Étapes de la Réalisation

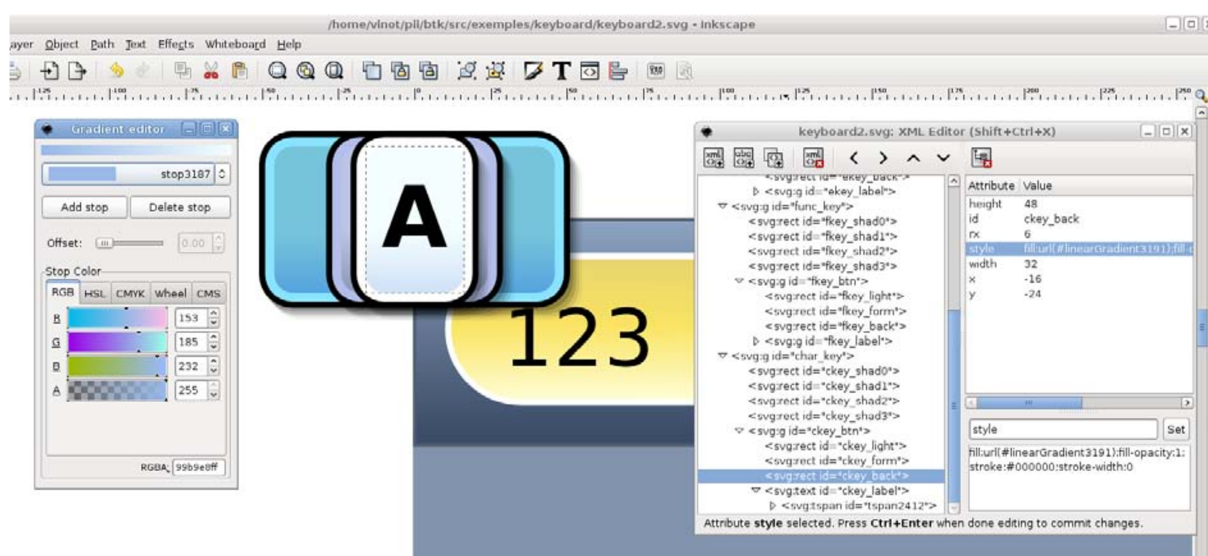

FIGURE 1.6 – Le fichier SVG de description des éléments graphiques du clavier, réalisé avec Inkscape.

Le travail d'élaboration graphique a été réalisé avec Inkscape (cf Figure 1.6).

Dans un premier fichier SVG, le designer a créé tout d'abord une touche du clavier. Elle est composée de 8 couches graphiques superposées, dont un calque *lumière* activé lors de l'appui de touche, et un groupe de calques pour réaliser l'ombre portée (Figure 1.7).

Ces calques sont nommés et groupés en un unique composant SVG. Le design de chaque touche "construit" l'apparence globale du clavier lorsque les touches sont assemblées. Durant le processus de conception, le designer peut être amené à corriger/modifier le design d'une touche afin de corriger l'apparence globale. Aussi, il passe par une phase de construction de maquette de principe : il les duplique, les organise et les assemble dans un nouveau fichier. La création visuelle de la zone supérieure, incluant l'afficheur texte, une touche "backspace" et un fond dégradé, complète cette maquette.

Pour passer à l'implémentation logicielle, 3 exemples différenciés des touches *char_key*, *func_key* et *enter_key*, ainsi que les blocs décrivant l'afficheur et l'arrière plan sont sauvegardés dans un nouveau fichier SVG. Pour faciliter l'utilisation des composants touches

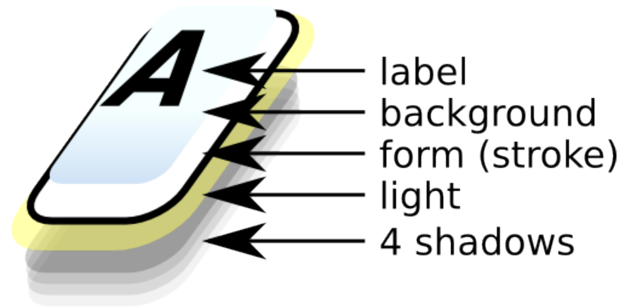

FIGURE 1.7 – Les 8 couches d'une seule touche du clavier.

et leurs transformations géométriques lors de l'écriture des fonctions de comportement graphique, les groupes ont été positionnés aux coordonnées (0,0). Ces composants graphiques SVG nommés correspondent aux modèles et aux classes objet (Python) gérant la partie comportementale de l'application (Figure 1.8). Une super classe *Key* a été définie pour prendre en charge les propriétés et méthodes communes aux touches.

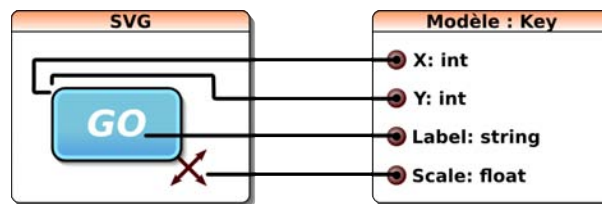

FIGURE 1.8 – Correspondance entre les classes graphiques SVG (à gauche) et le modèle (à droite).

L'un des aspects les plus intéressants du travail réalisé ici par le designer, et rendu accessible par la toolkit, a été de penser les objets SVG formant chaque composant en intégrant le comportement graphique à mettre en oeuvre durant l'interaction. Ce comportement a été défini en ciblant les propriétés graphiques à *connecter* au travers des fichiers modèles, en spécifiant leur type et leur adressage dans la scène SVG, et en décrivant l'évolution de leurs valeurs : pré-définie ou relative à d'autres paramètres (semblable aux contraintes de Garnet [Vander Zanden 01b]) ou calculée par méthodes. Par exemple, la gestion de la taille des différentes couches graphiques (largeur des ombres ou centrage du texte) est relative au paramètre *width* de la touche en elle-même. La description relative est particulièrement efficace pour adapter automatiquement l'ensemble des éléments graphiques à un changement de taille de la touche lors de l'effet fish-eye. De même, L'ajout des "connexions" permet de construire facilement des comportements complexes de l'application comme le couplage entre redimensionnement de la fenêtre et transformation géométrique du clavier.

La description JSON de la scène (ensemble des composants graphiques de l'interface) a été jugé "fastidieuse" par le designer et a rendu nécessaire l'écriture d'un script Python générant cette description.

L'écriture du fichier principal python, regroupant l'ensemble des classes et méthodes et la fonction principale de création de l'application, a été réalisée en s'inspirant de la

structure d'un autre fichier exemple.

1.3.4 Scenario n°4 : intégrer des composants dans des applications existantes

Le compilateur graphique peut générer soit une application autonome, soit du code embarquable dans le code d'une application existante. Générer du code embarquable permet au designer graphique d'utiliser le compilateur graphique comme un outil de traduction entre le langage SVG et l'application exécutée. Plus généralement, cet outil nous permet de dire que notre boîte à outils permet de construire des composants graphiques. On peut alors parler d'une boîte à outils pour boîtes à outils (l'utilisation du terme anglais *toolkit* sied mieux ici : Hayaku est une *toolkit de toolkit*).

Les exigences du dernier scénario sont de maximiser la modularité pour pouvoir utiliser le design produit dans une autre application. Reposer sur un compilateur graphique prend ici une importance toute particulière, puisque cela permet de produire du code pour différents moteurs de rendu graphique, comme OpenGL ou Cairo.

Afin de montrer que Hayaku peut être utilisé dans des applications existantes, nous avons implémenté un menu circulaire (ou pie-menu en anglais) générique (Figure. 1.9). L'objectif de ce scénario était de fournir une description du pie-menu qui soit indépendante de l'implémentation finale, et qui puisse être donc réutilisée, notamment dans d'autres applications déjà écrites et complexes (Figure 1.10).

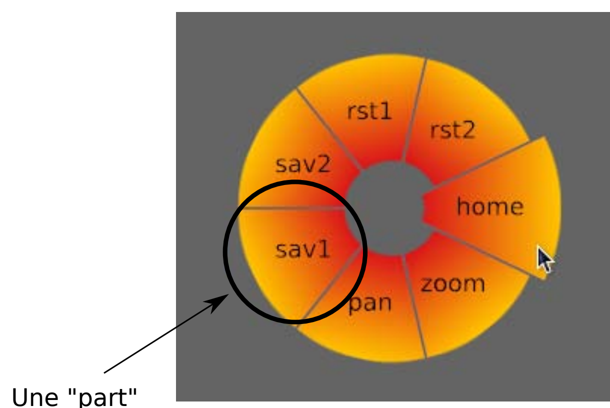

FIGURE 1.9 – Le menu circulaire en action. Il comporte 7 “parts”

La version du pie-menu que nous avons conçu inclut un retour visuel lors du survol d'un entrée du menu : l'entrée survolée a ses rayons internes et externes élargis tels que présenté à la figure 1.9. De ce fait, la description graphique du menu ne peut pas se réduire à un simple cercle. Il faut séparer le visuel en un ensemble de portions de cercles, chacune se rapportant à une entrée. Une autre fonctionnalité du menu requise est de paramétrer le nombre exact de sections (d'éléments de menu).

Réalisation

Le design des éléments du pie-menu en lui même ressemble à celui du clavier logiciel : nous avons spécifié le pie-menu comme étant un ensemble de “parts” de cercle. Ainsi chaque section de menu comporte 7 paramètres graphiques principaux : une position, un texte, un angle, un rayon interne, un rayon externe, un angle de rotation, et un gradient coloré. Afin de décrire la scène, nous avons dû recourir à un script similaire à celui utilisé par le clavier logiciel pour générer les différentes touches. Ce script génère les différentes sections de menu et leurs paramètres en fonction du nombre de sections souhaité.

La description du comportement se construit en attachant aux variables de *picking* une fonction d'activation qui modifie le rayon interne, le rayon externe, et la couleur lorsque nécessaire. Des évènements de plus haut niveau, comme “le secteur 7 a été sélectionné”, doivent être générés par la partie comportementale de l'application, puisque Hayaku ne gère que la partie graphique.

Embarquer le code dans une application existante

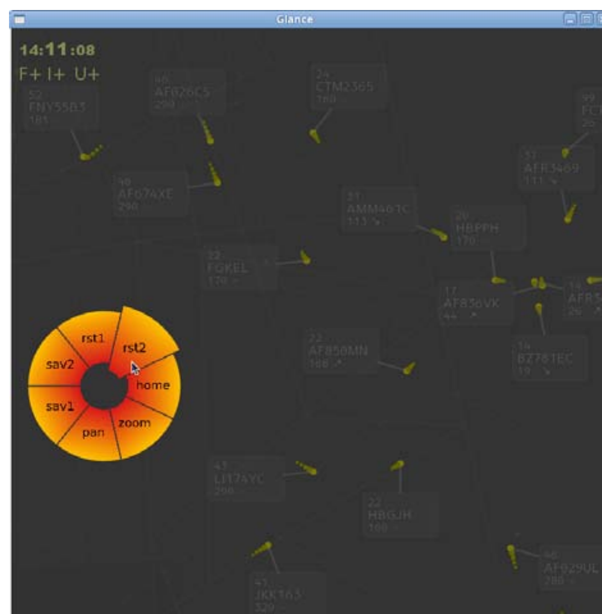

Figure 1.10 – Le pie-menu dans une application existante.

Nous avons incorporé le pie-menu précédemment décrit dans une application existante. Cette application hôte est une visualisation d'image radar à destination des contrôleurs aériens (voir la Figure 1.10). L'application du scénario n°2 en est une imitation. Elle est écrite en C++ et utilise OpenGL pour sa partie rendue graphique. Elle a été conçue comme extensible et prévoit un mécanisme pour charger des objets dynamiques externes. Nous avons utilisé et adapté ce mécanisme pour y insérer notre pie-menu. En l'absence d'un tel mécanisme, il aurait été nécessaire (et suffisant) d'ajouter un appel à la fonction de dessin Hayaku à la fonction de re-dessin de l'application hôte. La gestion du picking et

réalisée à chaque passe de dessin (voir la section 2.10 de la partie suivante). Nous n'avons donc pas mélangé les mécanismes de picking d'Hayaku et de l'application hôte.

Nous avons procédé comme suit : tout d'abord, il a fallu écrire une classe C++ qui permet d'interagir avec les objets chargés dynamiquement et générés par Hayaku. Cette classe est en quelque sorte la "glue" qui permet de faire le lien entre l'application hôte et les composants graphiques générés. De même, cette classe permet de factoriser l'ensemble du code nécessaire au chargement et à l'initialisation de tous les composants graphiques produits par Hayaku.

Listing 1.6 – Le code spécifique au pie menu dans RadarGL.

```
PieMenu::PieMenu (const char* filename):
    DynLibItem(filename)
{
    display = false;
    int(*get_Nb)() = (int(*)())GET_ELEM_IN_LIB(lib, "get_Menu_0_Nb");
    if (!get_Nb)
        return;
    nb = get_Nb();
    pies_functions = new pieFunctions[nb];
    set_X0 = (void (*)(double))GET_ELEM_IN_LIB(lib, "set_Menu_0_X0");
    set_Y0 = (void (*)(double))GET_ELEM_IN_LIB(lib, "set_Menu_0_Y0");
    for (int i = 0 ; i < nb ; i++) {
        // mise en place des callbacks attachees au flot de donnees
        {
            ostringstream ss;
            ss << "set_callback_Menu_0_Pie_" << i << "_Picked_Key";
            pies_functions[i].set_callback_pie_PICKED =
              (void (*)(void (*)(void*), void*)) GET_ELEM_IN_LIB(
                                              lib, ss.str().c_str());
        }{
            ostringstream ss;
            ss << "get_Menu_0_Pie_" << i << "_Picked_Key";
            pies_functions[i].get_values_PICKED =
              (int (*)()) GET_ELEM_IN_LIB(lib, ss.str().c_str());
        }{
            ostringstream ss;
            ss << "set_Menu_0_Pie_" << i << "_R_int";
            pies_functions[i].set_R_Int =
              (void (*)(double)) GET_ELEM_IN_LIB(lib,
                                            ss.str().c_str());
        }{
            ostringstream ss;
            ss << "set_Menu_0_Pie_" << i << "_R_ext";
            pies_functions[i].set_R_Ext =
              (void (*)(double)) GET_ELEM_IN_LIB(lib,
                                            ss.str().c_str());
        }
```

```
        pies_functions[i].data.pie = this;
        pies_functions[i].data.which = i;
    }
}

void PieMenu::setUpCallbacks()
{
    for (int i=0 ; i < nb ; i++)
    {
        pies_functions[i].set_callback_pie_PICKED(
          &PieMenu::callbackPicking,
          &(pies_functions[i].data));
    }
}

void PieMenu::callbackPicking(void* args) {
    struct dataCB* data = (struct dataCB*) args;
    PieMenu* o = (PieMenu*)data->pie;
    pieFunctions* pie = &(o->pies_functions[data->which]);
    bool selected = pie->get_values_PICKED() >= 0;

    if (selected) {
        pie->set_R_Int(20.0);
        pie->set_R_Ext(110.0);
    } else {
        pie->set_R_Int(30.0);
        pie->set_R_Ext(100.0);
    }
}

void PieMenu::onMouseMotion(int x, int y)
{
    set_X0(x);
    set_Y0(y);
}

void PieMenu::onMouseButton(
  int button, bool state, int x, int y)
{
    if (button == 0) {
        display = !state;
    }
}
```

Ensuite, nous avons écrit une sous-classe plus spécifique (Listing 1.6) au pie-menu afin de gérer correctement le comportement de ce pie-menu conformément aux actions utilisateur. Cette sous-classe comporte 112 lignes de codes et est une transcription en C++ du code précédemment en python au cours du scénario n°4. Le constructeur et la

fonction setUpCallbacks permettent de se lier à la bibliothèque dynamique générée par Hayaku. Ce code n'est pas destiné à être écrit par le designer d'interaction mais par un programmeur qui connaît le code de l'application hôte.

Les trois fonctions *callbackPicking*, *onMouseMotion* et *onMouseButton* montrent l'intégration simple du code produit par Hayaku. Le code C++ ne contient pas de références au graphisme et ne fait référence qu'à l'objet du modèle conceptuel pour mettre à jour le graphisme. Dans l'autre sens (du graphisme vers le code C++), ce lien est réalisé par l'appel de la fonction *callbackPicking* lorsque le curseur de la souris passe au dessus d'une forme graphique notifiant son picking au noyau fonctionnel.

Comme le montre la figure 1.10, le pie menu s'insère sans modifier le rendu de l'application hôte. De plus, cette insertion ne fait pas chuter le taux de rafraîchissement de cette dernière. En effet, l'insertion du pie menu dans le graphe de scène préexistant de l'application hôte ne rajoute que le temps de dessin de ce dernier, soit environ 2,5 ms.

Ce cas d'utilisation nous a permis de montrer qu'il était possible d'externaliser la création des composants graphiques et de les réutiliser dans d'autres applications. Toutefois, il est rare que des systèmes gèrent l'exécution d'extensions arbitraires avec un chargement dynamique. Dans ce cas, le code généré par Hayaku doit être incorporé de manière plus fine au niveau du source de l'application hôte. La glue entre le code original et le composant créé est alors plus simple puisqu'on supprime le chargement dynamique et qu'on le remplace par un "#include" au début. Les fonctions de dessins consistent alors juste à appeler la fonction *draw* du composant exporté (qui elle-même appelle la fonction de *picking*).

1.4 Conclusion

Au cours de ce chapitre, nous avons présenté comment la boîte à outils que nous proposons s'intègre dans le cycle de création de l'application. Hayaku nous permet d'introduire le concept de *designer d'interaction*. Le designer d'interaction est un designer graphique qui peut aussi contrôler et spécifier les interactions de l'application finale.

Au cours du scénario numéro 1, nous avons vu un exemple simple d'utilisation de Hayaku. Ceci nous a permis d'illustrer le fait que Hayaku repose sur un compilateur graphique qui est en charge d'optimiser le code produit. L'application finale ne souffre pas de problèmes de performances même si elle doit faire face à un grand nombre d'évènements.

Le scénario numéro 2 nous a permis de valider le fait que reposer sur un compilateur graphique nous permet de réaliser des applications graphiques interactives avec un grand nombre de formes graphiques. D'autre part, l'application est connectée à un serveur de données externes et ne souffre pas de ralentissements non plus lié à cette connexion.

Le scénario numéro 3 nous a permis de valider que nous ne restreignons pas le pouvoir d'expression du designer graphique. Nous lui avons donné la boîte à outils, et il nous a produit une application graphique riche. Ce scénario nous a permis aussi de valider la performance de Hayaku puisque le nombre d'objets graphique est grand.

Enfin, le dernier scénario nous a permis de voir l'intégration de la production de Hayaku avec une application tierce. Façades [Stuerzlinger 06] présente une technique de composition des éléments de applications graphiques au moyen du gestionnaire de fenêtres.

Notre résultat est plus intégré puisque la technique utilisée n'est pas de la composition, mais de l'inclusion de code. Un chargeur de bibliothèque produit par Hayaku nous permet de faire dessiner par l'application hôte C++ un élément graphique interactif entièrement décrit dans des langages de haut niveau. Ceci nous amène à dire qu'Hayaku est une boîte à outils qui permet de construire des composants pour une autre boîte à outils.

Chapitre 2

Réalisation d'un compilateur graphique

Sommaire

2.1 Introduction 62
2.2 Survol rapide du fonctionnement de la boîte à outils . . . 62
2.3 Interprétation contre compilation 62
2.4 Exigences pour le compilateur graphique 65
2.4.1 Performances de l'exécutable final 65
2.4.2 Performances du temps de compilation 66
2.4.3 Portabilité du code produit 66
2.4.4 Modularité permettant une évolution 66
2.5 Les différents langages intermédiaires utilisés 66
2.5.1 Les langages sources 66
2.5.2 Les premières transformations 67
2.5.3 Simplifications graphiques 67
2.5.4 Génération de la description dans le langage du moteur de rendu 68
2.5.5 Génération du code final 68
2.5.6 Écriture du code final 69
2.6 Le système de flot de données 69
2.6.1 Implémentation du système de flot de données . . . 70
2.6.2 Optimisations du système de flot de données . . . 70
2.7 Implémentations et optimisations du compilateur graphique 71
2.7.1 Première implémentation 71
2.7.2 Deuxième implémentation : externaliser la boucle de rendu 71
2.7.3 Implémentation finale 72
2.7.4 Optimisations 73
2.8 Génération de code statique ou semi-statique 74
2.8.1 Gérer des applications dont le nombre d'objets graphiques est variable 74

2.8.2 Pré-requis pour pouvoir générer du code embarquable . . . 75
2.9 Génération automatique de code 75
2.10 Le support du picking 76
2.11 Conclusion 77

2.1 Introduction

Après avoir présenté la méthode d'utilisation de notre boîte à outils Hayaku, nous expliquons plus en détails les aspects techniques du compilateur graphique sur lequel s'appuie Hayaku.

2.2 Survol rapide du fonctionnement de la boîte à outils

La commande *hayaku* appelle automatiquement le compilateur graphique (CG). Ce comportement est similaire à l'exécutable Python : l'utilisateur pense lancer un interpréteur, mais les fichiers sont en réalité compilés. Le compilateur graphique crée un dossier *BUILD* dans lequel il placera toutes ses productions. Les fichiers JSON fournis sont transformés en fichiers python, et un ensemble de fichiers sources écrits en C et leurs fichiers d'entête sont ensuite produits. Enfin, le CG appelle le compilateur C *gcc* afin de compiler ces fichiers C et de produire les différents fichiers objets. Ces objets peuvent ensuite être embarqués dans des applications ayant des modules avec une interface (ABI) C. Le compilateur génère aussi une bibliothèque dynamique qui peut soit être liée à l'exécutable de Hayaku, soit être embarquée directement dans une application existante (par un appel à *dlopen*). Afin de réduire les temps de compilation, le compilateur graphique est capable de détecter s'il y a eu des modifications entre deux compilations successives, et est capable de ne recompiler que les parties modifiées. Ce comportement est encore similaire à Python.

2.3 Interprétation contre compilation

Dans cette section, nous expliquons pourquoi le mécanisme de rendu graphique peut être considéré comme une chaîne de compilation. Nous définissons ensuite la notion du compilateur graphique (CG) et des langages graphiques intermédiaires.

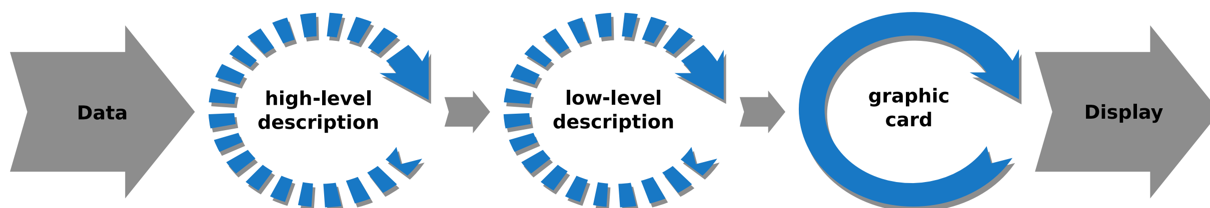

FIGURE 2.1 – Le mécanisme dit “classique” du rendu graphique.

Écrire une scène interactive requiert un certain nombre d'étapes pour produire l'application finale. La Figure 2.1 illustre la production informatique de graphismes : avant de pouvoir commencer à construire et afficher des formes graphiques, il est nécessaire de recueillir un certain ensemble de données (graphiques, de modélisation, etc...). Ensuite, ces données sont transformées en une description de haut niveau (code source utilisable par une boîte à outils de rendu graphique par exemple). Ce langage de haut niveau peut directement être utilisé pour l'affichage dans une boucle de rendu qui analyse ce langage entre deux rafraîchissements au cas où un changement a eu lieu. Il y a ainsi une interprétation du langage de haut niveau vers les appels contrôlant la carte graphique (première boucle en pointillés).

Lorsque des performances de rendu graphique sont plus critiques, le programmeur va coder des transformations de ce langage de haut niveau en une description de plus bas niveau (par exemple en utilisant la bibliothèque de rendu OpenGL), qui à son tour est utilisé pour l'affichage. Ceci requiert que le programmeur implémente un mécanisme de synchronisation car il est nécessaire de conserver une synchronisation entre la description de plus haut niveau et celle de plus bas niveau. Cette situation est représentée sur la Figure 2.1 par la deuxième boucle en pointillés. La boucle en ligne pleine symbolise la boucle de rendu utilisée systématiquement par le contrôleur vidéo qui doit parcourir la mémoire vidéo à chaque rafraîchissement de l'écran. Les deux boucles en pointillés symbolisent le fait que la boucle de rendu logicielle peut être soit placée sur la description de haut niveau, soit sur la description de bas niveau.

Écrire une application à l'aide d'une boîte à outils de haut niveau (un canvas) s'apparente donc à interpréter le graphe de scène fourni en entrée. En effet, la boucle de lecture de la description se place autour de la description de haut niveau, et la boîte à outils l'interprète en commandes compréhensibles par la carte graphique (que se soient du remplissage de texture ou des appels OpenGL ou Direct3D).

Par contre, écrire une application interactive performante consiste à effectuer une chaîne de transformations, ce qui est l'objectif d'un compilateur :

> Un compilateur est un programme, ou un ensemble de programmes qui traduit un texte écrit dans un langage de programmation - le langage *source* - en un autre langage de programmation - le langage *cible*.[Aho 86]

Une fois cette opération réalisée manuellement, le graphe de scène est proche de l'assembleur compréhensible par la carte graphique, et on supprime l'étape d'interprétation précédente.

Le rendu graphique est assimilable à la compilation de langages : les données décrites précédemment peuvent être considérées comme le langage source, et les fonctions de dessin comme le langage cible. Pour expliquer la structure du CG (Figure 2.3), nous pouvons la comparer à la structure d'un environnement de programmation en Java (Figure 2.2). La chaîne de compilation graphique consiste en une suite de transformations entre différent langages. La description de haut niveau de la scène graphique - passée par l'intermédiaire des fichiers produits en SVG et en python - est l'équivalent du code Java écrit par le programmeur. La description de bas niveau est quant à elle fortement liée au matériel et au type de rendu employé (nous avons implémenté les transformées vers OpenGL et Cairo), et est l'équivalent du code intermédiaire (le *bytecode*) généré par le compilateur

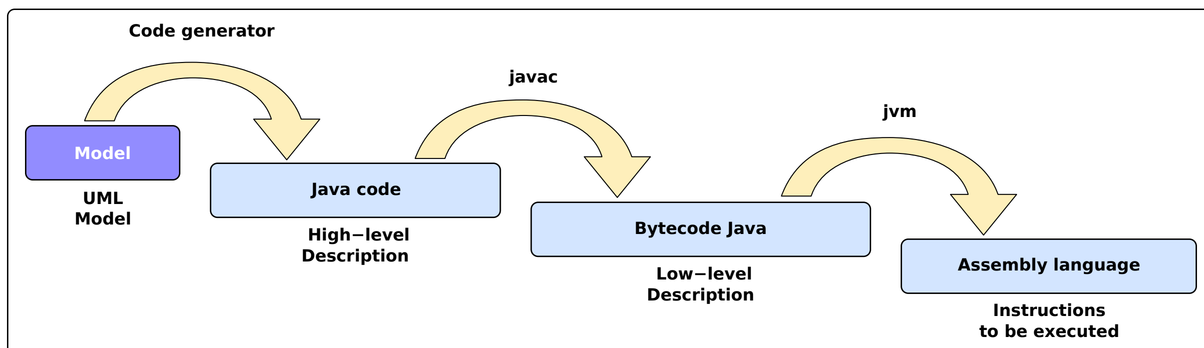

FIGURE 2.2 – La chaîne de compilation utilisée par le langage Java lorsque l'on part depuis un fichier écrit en UML...

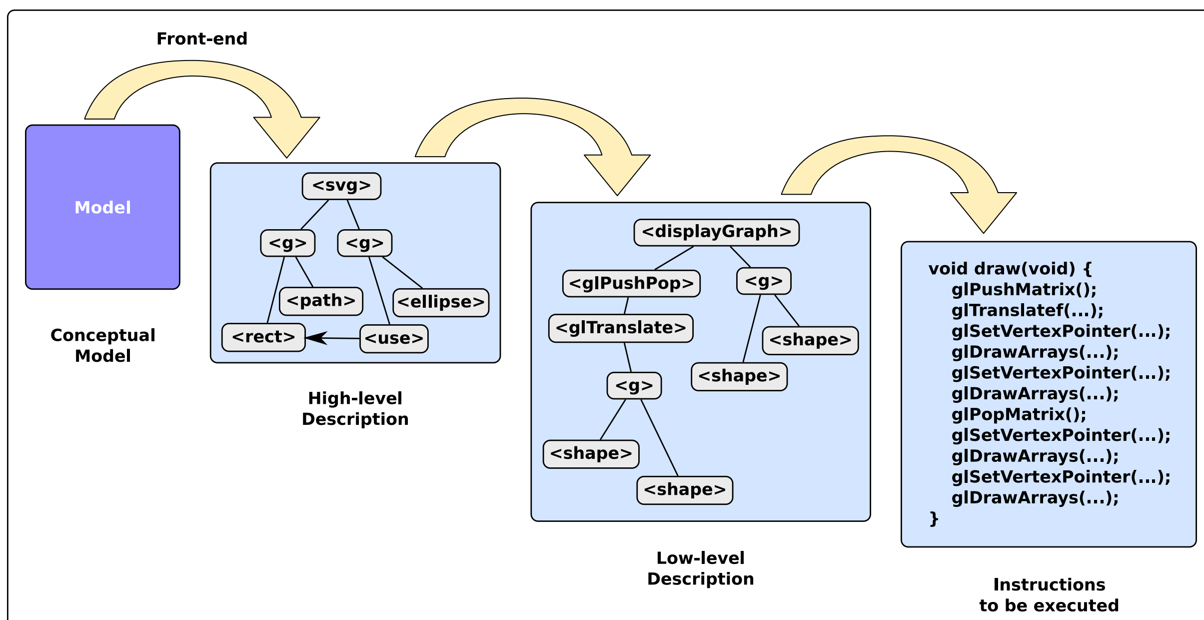

FIGURE 2.3 – ...et son équivalent lors du rendu d'applications.

javac. A la fin de la chaîne, un dernier processus (la Java Virtual Machine) interprète ou compile (dans le cas d'un compilateur natif java comme gcj) les instructions qui doivent être exécutées directement sur le matériel. En plus, cette machine virtuelle peut embarquer un JIT et donc encore améliorer la vitesse d'exécution du programme.

Le CG ajoute un autre étage dans la chaîne de compilation, à savoir la génération du code python à partir des descriptions JSON. On peut dire que cet étage est équivalent aux environnements qui génèrent du code Java depuis des descriptions UML. Nous verrons que cet étage supplémentaire permet au CG de traiter les transformations d'une manière uniforme, de telle sorte que les optimisations sont appliquées à toutes les étapes de la production du programme final.

Si la structure du CG est proche d'un compilateur de langage, le fait de travailler avec des scènes graphiques interactives impose certaines distances avec le modèle d'un langage classique. Tout d'abord, si la vitesse d'exécution du code produit est une contrainte forte, comme dans tout langage compilé, le temps de compilation du code source impacte aussi sur le développement de l'application finale. Nous avons vu au cours du premier chapitre que les designers d'interaction utilisent Hayaku pour tester et valider rapidement leurs conceptions. Un temps de compilation trop long empêche un tel développement itératif. Un autre problème lié au graphisme est qu'il est susceptible d'être modifié au cours de l'exécution de l'application. Un éditeur de dessin en est un exemple : le graphe de scène des objets à dessiner est potentiellement complètement modifié à chaque action de l'utilisateur final. Idéalement, le CG doit donc être embarqué dans l'application finale afin de pouvoir palier à de tels changements.

Considérer le processus de rendu de scènes graphiques comme une chaîne de compilation nous permet d'escompter les bénéfices suivants : l'architecture ainsi décrite permet de séparer la description du graphisme et des optimisations, et donc de rendre plus efficace la modification de la description ; les concepts tels que les optimisations qui ont été déjà bien étudiés dans la littérature relative au problème de la compilation peuvent être transposés aux problème du rendu graphique ; la description de haut niveau est suffisamment abstraite pour être indépendante du moteur de rendu finalement employé ; enfin, la sémantique des transformations utilisées est suffisamment claire pour permettre des vérifications sur le rendu, voire des preuves.

2.4 Exigences pour le compilateur graphique

Nous pouvons maintenant exprimer les exigences requises pour la création d'un compilateur graphique.

2.4.1 Performances de l'exécutable final

Une des premières exigences est que le code produit doit être le plus optimisé possible afin d'être le plus rapide possible en vitesse de rendu. En effet, l'intérêt d'utiliser un compilateur graphique réside dans le fait que le code produit sera plus performant qu'en utilisant une boîte à outils graphique riche, mais peu optimisée.

2.4.2 Performances du temps de compilation

Compte tenu du cycle de développement itératif des applications graphiques, le temps de compilation ne doit pas être trop long (de l'ordre de quelques secondes) afin de pouvoir essayer rapidement plusieurs designs, voire de l'embarquer directement en tant que compilateur juste-à-temps (JI).

2.4.3 Portabilité du code produit

Un des bénéfices escomptés du compilateur graphique est de pouvoir réutiliser les sources dans différents contextes. Ainsi, le code produit devra être portable, ou le compilateur doit permettre de modifier facilement les dernières couches de production de code afin de pouvoir produire du code pour différents moteurs de rendus, langages ou plateformes.

2.4.4 Modularité permettant une évolution

Enfin, l'exigence précédente nous amène à souhaiter une modularité dans le compilateur pour pouvoir cibler différents langages, plateformes, moteurs de rendu graphique. Cette modularité permet aussi d'actualiser plus facilement le compilateur graphique aux futures architectures afin de mettre à jour les exécutables produits sans forcément reconcevoir l'application entièrement.

2.5 Les différents langages intermédiaires utilisés

2.5.1 Les langages sources

Nous avons déjà présenté les différents langages sources utilisés dans le chapitre précédent. Nous rappelons ici rapidement leur nom et leur fonctions.

Le premier langage en entrée de la chaîne de compilation est une abstraction des composants graphiques. Ce langage permet d'abstraire la couche de présentation de l'application (la partie graphique), du reste de l'application, i.e. le noyau fonctionnel. Cette description est fournie pour Hayaku en JSON et représente les classes de la partie graphique. Nous appelons cette description le *modèle conceptuel*.

Le deuxième langage nécessaire au CG est la représentation graphique de ces classes. Cette description est fournie en SVG et peut être produite à partir d'un logiciel habituellement utilisé par les designers graphique, tel qu'Inkscape ou Adobe Illustrator. Ce fichier forme la définition des *classes graphiques*.

Le troisième langage fourni par le designer graphique est une description des connexions entre les classes définies plus haut et les classes graphiques du SVG. Ce langage respecte la syntaxe JSON. Ce fichier est simplement appelé le fichier de *connexions*.

Enfin, il ne reste plus qu'au designer graphique à fournir une première instanciation de ses objets (toujours au format JSON) afin d'être capable d'initialiser l'état de son application graphique au démarrage. Ici, le designer d'interaction fourni la description de sa *scène*.

2.5.2 Les premières transformations

Une fois les fichiers sources fournis, le CG commence à convertir ces sources en langage Python pour pouvoir effectuer des transformations. La transformation des fichiers de classe écrits en JSON est relativement simple : à chacune des classes JSON fournies on crée une classe Python. Chacun des champs de la classe JSON est traduit en un champ Python (c'est ici que l'on initialise les variables du système de flots de données). Enfin, quelques fonctions d'introspection sont ajoutées pour faciliter les transformations futures (principalement des fonctions permettant de remonter à la chaîne de caractères brute fournie par l'utilisateur).

Le chargement du fichier de description des graphiques est classique (lecture d'un fichier XML), et la gestion des connexions (fournies en JSON) consiste essentiellement à rechercher des éléments dans la description graphique précédemment chargée. Cette recherche se base sur les identifiants que peut donner le designer à n'importe quel item SVG.

Une fois ces fichiers chargés et analysés, le compilateur graphique lance les compilations de chacun des objets du modèle conceptuel. Ceci permet de réaliser un cache de compilation.

Une fois l'ensemble de la chaîne de compilation réalisée, le compilateur graphique charge le fichier de description de la scène. Il assemble alors les différents caches de compilation réalisés pour générer l'exécutable final.

2.5.3 Simplifications graphiques

L'utilisation de langages intermédiaires n'est pas nouvelle dans le domaine de la compilation. Elle est utilisée dans presque tous les compilateurs standards. Cependant, il est rare de voir de tels langages intermédiaires employés dans les différentes boîtes à outils graphiques disponibles. Nous avons adapté ce principe à notre compilateur graphique.

Le premier étage de transformation, après la lecture et la mise en forme sous forme de graphes des langages d'entrée, effectue des simplifications graphiques. En effet, la sémantique SVG est complexe : cela permet de faciliter l'écriture et la lecture du fichier XML par un opérateur humain, de diminuer la taille finale du document, et cela permet aussi d'optimiser le rendu dans certains cas (spécifier un rectangle est intéressant tant pour l'utilisateur puisque des propriétés graphiques seront respectées - parallélisme, orthogonalité - mais aussi pour l'interpréteur SVG qui peut accélérer le traitement), et il est possible d'exprimer n'importe quelle forme de haut niveau décrite en SVG (comme un rectangle, une ellipse, un rectangle à coin arrondi et avec une bordure de 5 pixels) en un ensemble de formes élémentaires, les chemins (*path*). Pour cela, une figure contenant un *stroke* (une bordure) et un élément de remplissage peut être éclatée en deux chemins, l'un contenant la bordure, et l'autre le remplissage (Figure 2.4). Aussi, le compilateur transforme le langage de description SVG en un langage que nous appelons *mini SVG*, qui contient l'ensemble minimum des éléments SVG nécessaires à la description de n'importe quel dessin SVG.

Les avantages de ce langage sont multiples. Tout d'abord, cela permet de convertir le graphe cyclique et dirigé [11] du langage SVG en un arbre. Les transformations résultantes

11. SVG prévoit l'utilisation du *use* : ceci permet de référencer un autre élément graphique par son lien

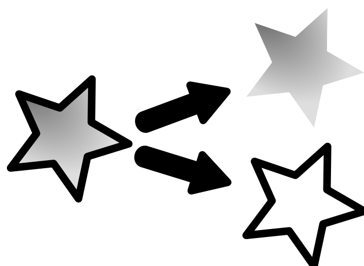

Figure 2.4 – Une forme avec un fond plein et un contour peut être sous-divisée en deux formes graphiques élémentaires

sont plus simples, et plus sûres puisque celles-ci sont réduites. De plus, cela permet d'optimiser le code lors de la génération. Par rapport à un compilateur de langage, cette étape est similaire à une étape transformant divers types de boucles (*for*, *repeat*, *while*) en un code utilisant uniquement des *JMP*.

Cependant, nous remarquons que l'on perd de l'information de haut-niveau (typiquement un rectangle possède des propriétés qui sont perdues ici - 4 angles droits, côtés parallèles, etc...). Un autre problème est que l'on risque d'obtenir une explosion du nombre d'items graphiques. Néanmoins, nous jugeons les avantages comme plus importants, puisqu'une implémentation judicieuse permet de contourner en partie les inconvénients (reformulation des informations de haut niveau sous forme de contraintes, cache de compilation, etc).

2.5.4 Génération de la description dans le langage du moteur de rendu

Le CG convertit ensuite cet arbre en un arbre d'appels OpenGL, Cairo, ou tout autre moteur de rendu. Cet arbre ne contient pas à proprement parler les appels eux-mêmes, mais une abstraction de ceux-ci. Un arbre d'appel OpenGL contiendra des nœuds *glBegin*, *glPushMatrix* par exemple. Ces nœuds ne contiennent pas l'appel en lui-même (la chaîne de caractères le représentant), mais la sémantique de l'appel. En C, le code pour un *glPushMatrix* est effectivement *glPushMatrix()*, mais en Java, on obtiendra un *GL.PushMatrix*. Cela permet de générer différents langages cibles qui peuvent utiliser le moteur de rendu (actuellement C et Java). Cet arbre est appelé le *graphe d'affichage*.

La structure résultante reste toujours arborescente puisque cela permet de factoriser des points d'implémentation. Dans notre exemple, à chaque *glPushMatrix* est associé un *glPopMatrix* qui sera effectué en remontant le nœud de l'arbre.

2.5.5 Génération du code final

L'étape suivante consiste à générer le code final dans le langage cible. Le code est produit sous forme de graphe (donc une représentation interne), afin de réaliser des optimisations relatives à l'état du système. Les optimisations sont réalisées en exécutant un

(nom ou url) et d'en surcharger les propriétés.

automate sur le code généré qui modélise le fonctionnement du moteur de rendu final. Cet automate connait les états dans lesquels se trouve le système à chaque instruction et peut donc savoir si telle ou telle instruction est nécessaire, ou si elle peut être simplifiée. Par exemple, dans le cas du moteur de rendu OpenGL, des shaders sont employés pour réaliser certains traitements. Cet automate connaissant l'état du système (quel shader est activé et quels sont ses variables) peut transformer l'instruction générée (ou l'ensemble d'instruction) consistant à activer tel shader et modifier ses variables en une nouvelle instruction (ou un ensemble d'instructions) qui va juste modifier les variables qui ne sont plus à jour.

2.5.6 Écriture du code final

La dernière étape de la chaîne de compilation consiste à transformer la représentation interne du code final en un fichier texte. Ceci est réalisé par un parcours d'arbre.

2.6 Le système de flot de données

Nous avons vu que le compilateur graphique réalisait un certain nombre de transformations pour générer le code final. Le designer d'interactions interagit avec les éléments du modèle pour modifier la scène produite. Ces modifications sur les sources sont ensuite répercutées sur le code produit par l'intermédiaire d'un système de flot de données.

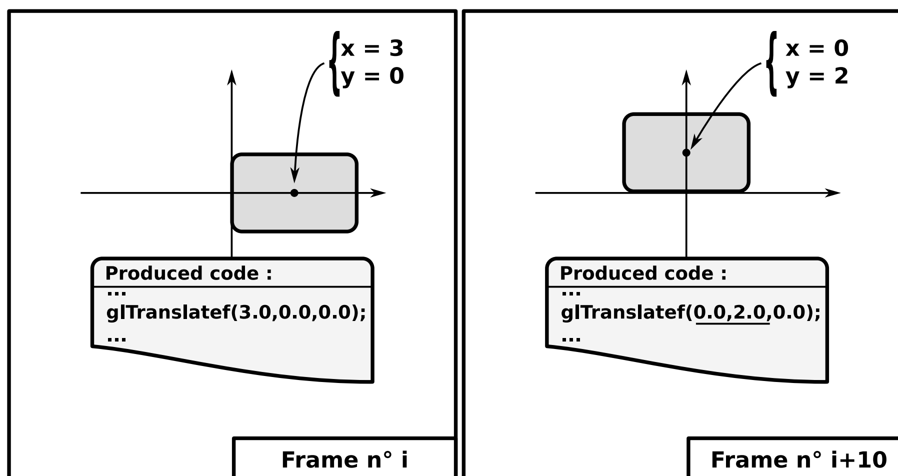

FIGURE 2.5 – Impact du changement d'une propriété d'un objet dans le code produit

Parallèlement à la production du code de rendu dans le langage cible, le CG produit un système de flot de données pour garantir la mise à jour des propriétés graphiques. Ce code est aussi écrit dans le langage cible et bénéficie donc des optimisations agressives du compilateur natif de ce dernier. Ces optimisations sont très performantes puisque les fonctions produites ne sont que des fonctions mathématiques et des écritures mémoire.

Grâce à ce mécanisme, le code produit n'a pas besoin d'être réinterprété chaque fois qu'une propriété du modèle conceptuel est modifiée : le système de flot de données met simplement à jour le code (en fait une zone mémoire) produit (Figure 2.5).

Le programmeur précise lui-même les champs des classes de son modèle conceptuel qui sont variables à l'exécution. Ceci se traduit dans notre implémentation par un "v" devant le type (*vint* par exemple) lors de la déclaration du *modèle conceptuel* en JSON. À l'opposé, une variable typée sans le "v" sera considérée comme constante par le compilateur graphique. En résumé, cette étape est l'inverse du *const* des langages C et C++ où l'on spécifie quels symboles seront constants. Ceci permet au compilateur graphique de sortir du flot de données toutes les données statiques et de les mettre de manière littérale dans le code final.

2.6.1 Implémentation du système de flot de données

Le langage utilisé pour le système de flot de données est un langage arithmétique auquel est ajouté la fonction de tests. L'utilisateur dispose des fonctions suivantes : $if, not, +, -, *, /, <, cos, sin, acos, asin, sqrt, min, max$. L'utilisateur du CG spécifie ses variables et peut en déclarer de nouvelles à l'aide de formules. Pour cela, le CG surcharge les opérateurs utilisés par Python pour éviter la tâche d'analyse syntaxique. L'idée est ici de remplacer cette analyse syntaxique par l'introspection de Python. Par exemple, l'utilisateur peut écrire :

x0 = var('x0',5)
y0 = var('y0',10)
x1 = var('x1',x0+200)
y1 = var('y1',y0+250)

Cet extrait de code produit deux variables d'entrée $x0$ et $y0$ et deux variables dépendantes, $x1$ et $y1$. Le système de nommage des variables permet de les référencer plus tard dans la description de la scène. Par exemple, $x0$ et $y0$ pourraient être les points d'ancrage d'une forme graphique, et $x1$ et $y1$ le point d'ancrage (calculé de manière automatique à la mise à jour de $x0$ et $y0$) d'une autre forme qui est déplacée de $(200, 250)$ par rapport à la première.

2.6.2 Optimisations du système de flot de données

Comme le système de flot de données est un langage mathématique et fonctionnel (donc sans effets de bord), nous pouvons appliquer divers types d'optimisation. Ces optimisations peuvent être par exemple relatives à la sémantique des fonctions elles même. Générer "$x + x + x + x + x + x$" peut alors être transformé en "$6 * x$", ce qui permet de réduire le nombre d'opérations de cinq à une seule (à condition de considérer qu'effectuer une multiplication est moins coûteuse que six additions). Comme nous l'avons dit plus haut, d'autres optimisations sont laissées au compilateur natif puisque le code généré est ensuite recompilé par *gcc* dans le cas d'un langage cible comme le C par exemple.

2.7 Implémentations et optimisations du compilateur graphique

Si le système de flot de données est resté relativement stable du point de vue de l'implémentation au cours de la réalisation de Hayaku, il n'en est pas de même pour la chaine de transformations des langages sources vers le code à exécuter. Nous allons détailler ici les différentes implémentations réalisées, leurs qualités, et les défauts qui nous ont poussés à expérimenter une autre réalisation.

2.7.1 Première implémentation

L'objectif premier de cette réalisation était de tester rapidement les concepts présentés ci-dessus. Nous avons écrit le compilateur graphique dans un langage à prototypage rapide : Python. La première réalisation utilise une boucle de rendue basée sur des appels de fonctions Python encapsulant les appels à OpenGL (en C).

Les avantages ici sont les premières raisons qui nous ont poussé à rester avec un langage interprété comme Python : rapidité de développement, simplicité générale du code. Cependant, utiliser Python pour faire du code OpenGL impacte grandement les performances globales et la réactivité du système. En effet, dans la version de base de l'interpréteur Python, les variables d'appel de fonction ne sont pas fortement contraintes à un type (on ne déclare pas une fonction en typant ses paramètres). Si le choix du langage Python parait judicieux pour permettre un développement rapide des fonctions de transformations, ce choix ne convient pas à une boucle de rendu OpenGL. Il faut que l'interpréteur teste si un appel de fonctions s'applique aux paramètres qui lui sont fournis. L'interpréteur ne met pas en cache ces tests réalisés à chaque appel de fonction. Pour une utilisation habituelle telle qu'un script, ceci ne gène pas l'exécution. Cependant, une boucle de rendu OpenGL nécessite de repasser plusieurs fois (autant de fois que l'on affiche une image) par la fonction de rendu. Ainsi, nous surchargeons l'interpréteur par un grand nombre de tests inutiles (puisqu'ils ne changent pas d'une exécution sur une autre).

Cette surcharge de Python introduit par conséquent un problème au niveau des performances : l'interpréteur perd beaucoup de temps à traiter la boucle de rendu OpenGL. Ainsi, le traitement des évènements du flot de données sont ralentis puisque l'interpréteur est surchargé. Les évènements utilisateurs sont donc traités trop tard (délai > 20 ms). Une solution pourrait être d'utiliser des threads pour séparer le flux des instructions OpenGL du flot de données. Cette solution ne marche pas pour ce langage précis puisque Python introduit dans ses versions actuelles (avant la version 3.0 tout du moins) un verrou global sur les threads.

2.7.2 Deuxième implémentation : externaliser la boucle de rendu

Nous avons donc cherché à résoudre ce problème de performances en externalisant la gestion de la boucle de rendu. Cette externalisation a consisté à écrire un module python asynchrone (basé sur des threads posix et une synchronisation ad-hoc). Ce module prenait en entrée une liste d'instructions OpenGL (une liste de pointeurs vers des fonctions) et

exécutait ces instructions.

Cet interpréteur OpenGL externe présentait quelques avantages. Tout d'abord, les performances du moteur de rendu ont considérablement augmenté. Ensuite, nous avons pu nous en servir comme d'un JIT (un Just-In-Time) compiler pour pouvoir réaliser des optimisations dynamiques (voir section 2.7.4). Néanmoins, nous avons trouvé deux problèmes majeurs à cette réalisation. Tout d'abord, l'implémentation d'un tel interpréteur OpenGL est coûteux en développement. Les sources d'erreurs lors de l'écriture du compilateur graphique sont nombreuses puisque l'implémentation consiste essentiellement à manipuler des pointeurs de fonctions (donc à écrire des pointeurs de pointeurs de fonctions).

Ensuite, il reste le problème que la chaîne de compilation reste fortement liée à Python et à OpenGL. Modifier cette chaîne pour cibler un autre moteur de rendu est faisable (Cairo pourrait être implémenté de la même manière par exemple), néanmoins, on ne peut pas se passer de l'exécutable Python qui contrôle la boucle principale du programme. Ainsi un des principaux avantages du compilateur qui est sa capacité à générer du code pour des environnements différents est perdu.

2.7.3 Implémentation finale

Nous avons conservé les phases de transformations de langage en Python afin de conserver un développement le plus rapide possible. Néanmoins, l'exécutable final ne peut être contraint à ce langage pour des raisons de performances et de portabilité.

Le compilateur graphique est capable de produire différents types de sorties, que ce soit en termes de langage cible et d'exécutable (pour l'instant en C et en Java), qu'en termes de moteur de rendu graphique (actuellement OpenGL et partiellement Cairo). Pour cela, nous avons séparé la partie compilation de la partie exécution. Le compilateur génère du code dans le langage cible avec le module de rendu choisi, et embarque ce code dans une bibliothèque dynamique.

Ainsi, lancer le compilateur graphique sur un fichier source compile bien le code et produit l'exécutable final, mais il ouvre aussi une fenêtre et affiche directement la scène graphique résultante. Ceci permet de tester au fur et à mesure l'application produite, alors que le code produit est entièrement externalisé.

Afin de maximiser la réutilisation du code des transformations utilisées, nous avons dû implémenter notre propre mécanisme de classes partielles. Nous séparons ainsi la description des langages intermédiaires et les transformations entre ces langages. Au début de la chaîne de compilation, le CG choisit quels langages et transformations seront nécessaires pour produire le code final. Ce choix permet donc d'accrocher aux éléments de description des langages intermédiaire les fonctions de transformation vers le langage suivant. L'arbre ainsi généré peut alors être transformé en visitant chacun de ses nœuds. Ce mécanisme nous permet de modulariser au maximum le CG et donc de remplacer n'importe quel étage pour en changer le comportement.

2.7.4 Optimisations

Les optimisations réalisées par le compilateur graphique ont lieu sur presque tous les langages intermédiaires. Ces optimisations ont donc lieu soit sur la sémantique des graphismes en eux-même, soit sur la sémantique et l'état du langage utilisé. Ces optimisations peuvent être statiques ou dynamiques.

Les optimisations statiques

Les différents langages intermédiaires employés ont été écrits en utilisant des patrons de conception. Le patron décorateur permet au CG de construire l'arbre et les transformations suivantes en évitant les tests au fur et à mesure du parcours de l'arbre. Par exemple, si un élément ne contient aucune transformation de type mise à l'échelle (un *scale*) à l'intérieur même de la description graphique de l'objet, le compilateur n'inclut simplement pas le décorateur *scale* sur l'élément. L'arbre résultant contient donc le minimum d'éléments nécessaires pour se représenter la scène dans chacun des langages intermédiaires.

La deuxième possibilité offerte par cette approche est que le compilateur peut factoriser les différents éléments en détectant les sous-expressions communes. Par exemple, si une même transformation a lieu deux fois entre deux groupes avec les mêmes paramètres, le compilateur peut la factoriser en encapsulant ce groupe dans un groupe de niveau supérieur qui contient la transformation commune. Lors de l'exécution, la transformation n'est exécutée qu'une fois.

Nous avons vu que le compilateur graphique compilait tout d'abord chacune des classes du modèle conceptuel afin de faire un cache de compilation. Une fois cette étape réalisée, il charge le fichier de scène, et peut alors réaliser des optimisations inter-procédurales puisqu'il connait maintenant l'ensemble de la scène graphique.

Les optimisations dynamiques

Les deux premières versions du compilateur graphique proposait un exécutable en Python. Nous avons déjà vu que cela posait un problème de performances. Nous avions dû alors recourir à un compilateur dynamique (un JIT) afin d'améliorer de manière significative les performances. Ce JIT réalisait essentiellement une tache de mise en cache du graphe d'affichage vers une liste d'appels OpenGL qui était directement exécutée en C natif. De même, ce JIT réalisait de la *suppression de code mort* puisque certaines branches de l'arbre d'affichage n'étaient pas forcément affichées à l'exécution.

Les versions du CG qui ont suivi ont supprimé cet exécutable et exportent maintenant directement du code C (ou java selon la cible choisie) qui est à son tour exécuté. Le mécanisme de compilation dynamique est réduit à la seule gestions des différents caches pour éviter de surcharger l'application produite. Ces caches sont nécessaires, tant pour le système de flot de données que pour la tessellation des différentes formes graphiques.

Autres optimisations

Nous avons vu que le compilateur graphique est capable de réaliser des optimisations sur le graphe d'affichage. Ces optimisations peuvent soit être locales, soit être inter-

procédurales puisque le CG connaît l'ensemble de la scène interactive. Nous listons ici les différents types d'optimisation que le CG est capable de réaliser pour optimiser la vitesse de rendu ou le temps de compilation :

- *Recherche de sous-expressions communes* : Le CG est capable de détecter si un schéma d'affichage ou de code se répète, et peut le factoriser.
- *Propagation des constantes* : Lorsque l'utilisateur ne spécifie pas que tel paramètre de telle forme graphique sera variable, le compilateur propage la constante tout au long de la chaîne de compilation.
- *Aides de l'utilisateur* : Nous avons souhaité proposer au designer d'interaction la possibilité de spécifier manuellement des optimisations non triviales. Cependant, par manque de temps dans le développement de la boîte à outils, ces optimisations n'ont pu être implémentées.

2.8 Génération de code statique ou semi-statique

2.8.1 Gérer des applications dont le nombre d'objets graphiques est variable

La plupart des scénarios présentés au chapitre précédent sont des application où le nombre d'objets graphiques interactifs n'est pas variable (le clavier logiciel, ou le pie menu par exemple). Toutefois, pour d'autre type d'application interactives, tel que l'image radar présentée où le nombre d'avions n'est en théorie pas limité, l'architecture présentée (avec un fichier de scène) n'autorise a priori pas une création d'objets dynamique. Dans ce cas, Hayaku, et donc le compilateur graphique, présente deux stratégies.

La première stratégie est de fixer une limite arbitraire au nombre maximum d'objets graphiques à afficher. Une fois cette limite fixée, la scène consiste à instancier ce nombre maximal d'objets et de spécifier de ne pas les afficher au lancement. L'utilisateur créé donc une réserve d'objets graphiques, qu'il pourra activer à souhait au cours de l'exécution. En pratique cette stratégie marche pour une classe d'applications : pour notre exemple, on sait que le nombre maximal d'avions qui passeront dans un secteur donné est limité par une autorité de régulation afin que les contrôleurs aériens puissent gérer le trafic.

Quand bien même cette stratégie est relativement simple à mettre en œuvre, du point de vue de la compilation et de l'exécution elle pose quelques problèmes. Le principal problème est le temps nécessaire pour la compilation et pour l'exécution, puisque la boucle de rendu doit alors gérer un grand nombre d'objets interactifs qui ne seront peut-être pas affichés. Cependant, on évite de gérer la mémoire à l'exécution (puisque tout est déjà pré-instancié). De plus, toute la scène étant statique, on peut déterminer le temps maximal de rendu de la scène. Cette approche est donc adapté à des systèmes très contraints comme des systèmes embarqués.

La deuxième stratégie consiste à générer des objets graphiques interactifs dynamiques. Ces objets peuvent être instanciés dynamiquement par l'application hôte, et la gestion de ces objets incombe donc à cette dernière. Dans le scénario de l'image radar, ceci revient à générer un objets graphique *vol* et générer le code spécifique à son affichage. Nous perdons ici des optimisations inter-procédurales sur l'ensemble de la scène graphique,

mais le temps de compilation est théoriquement réduit (on passe de n compilations des n objets à une compilation par classe). Ensuite, ce code généré est embarqué dans une autre application (écrite à la main cette fois ci puisque l'implémentation actuelle d'Hayaku ne gère pas ces objets dynamiques). L'application doit gérer la création/destruction des objets graphiques, de même qu'elle doit gérer l'affichage de ces derniers en appelant leur fonction *draw*. L'intérêt de cette technique est que l'on supprime la limite au nombre d'objets affichables. Cependant il est plus difficile d'estimer le temps maximal d'exécution du code sur le matériel cible puisque cela requiert des allocations dynamiques.

2.8.2 Pré-requis pour pouvoir générer du code embarquable

La production de code semi-statique (tel que décrit dans la deuxième stratégie ci-dessus), nécessite de créer des objets dynamiques et embarquables. Nous explicitons ici les principales propriétés requises pour pouvoir générer de tels objets.

Tout d'abord, le code généré ne doit pas interférer avec l'état courant de l'application. Chaque fois qu'une modification de cet état doit être réalisée, il faut au préalable sauvegarder l'état courant pour pouvoir le restaurer une fois la commande terminée. Cette recommandation est particulièrement importante lorsque l'on travaille avec des moteurs de rendu graphiques. Ceux-ci sont pour la plupart basés sur un ensemble d'états graphiques. Toutefois, avec les dernières versions d'OpenGL, ces états sont en train de disparaître du fait de la tendance à développer des architectures parallèles, et donc de travailler avec plusieurs processus légers (des *threads*), fortement incompatibles avec ces états.

La deuxième recommandation que nous ferons ici consiste à produire du code qui soit facilement lisible par l'humain. S'il est tentant de générer du code sans signification lorsque l'on réalise un compilateur quelconque, il ne faut pas oublier ici que l'on souhaite que l'utilisateur puisse assembler le code produit avec d'autres applications. La notion d'interface ainsi générée se doit donc d'être la plus compréhensible possible [Ko 04] : par exemple, la fonction *set_component0_key25_backgroundColor_red* est beaucoup plus compréhensible que *set0_25_2_1* pour l'utilisateur.

2.9 Génération automatique de code

Nous avons réalisé un système de surveillance sur les fichiers sources fournis en entrée au compilateur graphique. Le système déclenche une recompilation automatiquement lorsqu'un fichier est modifié et sauvegardé. Le changement est automatiquement impacté dans l'application générée alors qu'elle est en train de s'exécuter. Par exemple, changer la couleur d'un composant d'une application graphique à l'aide d'un éditeur SVG tel qu'Inkscape, et sauvegarder ce changement alors que l'exécutable de Hayaku s'exécute met automatiquement à jour l'affichage des composants graphiques de cette classe. Ce comportement permet de développer dans un système “vivant”, à l'instar de la boucle REPL de LISP [Steele 93], ou de l'environnement Smalltalk Squeak [Ingalls 97]. Ceci illustre les avantages de séparer la description graphique de la description du comportement et d'utiliser un système de flot de données : puisque le périmètre de la partie graphique de l'application est clairement délimité, Hayaku est capable de remplacer le graphisme à

n'importe quel moment, et cela sans affecter le reste de l'application. Un tel outil permet de réduire le temps nécessaire entre la formulation d'une idée et son essai.

2.10 Le support du picking

L'écriture d'un système interactif exige que l'utilisateur puisse désigner dans son rendu final des items graphiques. Par exemple, le click sur un bouton représenté à l'écran n'est possible que si le curseur est sur le bouton. Une fois que le système sait quel est l'élément sous le curseur de la souris, il peut alors informer l'utilisateur final de cette information en affichant un feedback visuel sur l'élément en question. Dans le cas du scénario relatif au clavier logiciel, ce feedback correspond à changer le dégradé de la touche en question pour simuler une sorte de mise en surbrillance. Le système doit donc déterminer l'élément désigné par le curseur. Le terme employé pour la recherche d'éléments graphiques dans une scène est le terme de *picking*.

Dans Hayaku, puisque nous disposons d'un système de flot de données, nous pouvons l'utiliser tant comme un dispositif d'entrée, pour déplacer des formes graphiques ou changer des propriétés, que comme un dispositif de sortie en attachant des fonctions de retour (des *callbacks*) lorsque une donnée du flot est modifiée. Nous avons attaché à ce système de callbacks le système de picking. Lors de la définition de la scène, le designer d'interactions spécifie quelles seront ses variables de picking. Il peut ensuite y attacher une callback. Ainsi, il stipule les modifications graphiques (l'éclairage de la touche) dans la callback rattachée à la variable de picking de la touche.

Toutefois, la gestion du picking dans une scène complexe peut être très coûteuse. En effet, le picking classique employé dans les applications de type WIMP (tester individuellement chacun des éléments de la scène pour voir si leur boîte englobante intercepte la coordonnée souhaitée) n'est pas assez performant dès lors que l'on conçoit des interfaces avec des formes graphiques quelconques. Ces formes quelconques nécessitent des calculs complexes pour intercepter de manière analytique la position de la souris. Une autre solution consiste à réaliser une passe par objet graphique et à utiliser la rastérisation pour voir si le curseur intercepte l'objet, ce qui est aussi trop coûteux. Il est nécessaire d'optimiser ce mécanisme.

Par exemple, OpenGL fourni un mécanisme de picking par l'intermédiaire du mode de rendu *select*. Ce mécanisme consiste à nommer chacune des formes graphiques lors de la passe de rendu, et vérifier si la forme intercepte la coordonnée de picking. Cette technique présente l'inconvénient de devoir effectuer un rendu par picking. Cependant, lorsque l'on a besoin de générer un certain nombre de picking entre deux images, cette technique peut devenir très coûteuse : dans le cas d'une application multi-touch où 10 doigts sont posés sur l'écran, on devra donc effectuer 11 rendus de la scène (un d'affichage et dix pour le picking).

Nous avons donc choisi de ne pas utiliser cet algorithme et plutôt d'utiliser un algorithme de picking par couleur unique[Hanrahan 90]. Nous créons donc un buffer non affiché dans laquelle nous rendons les différentes formes de picking avec des pseudos-couleurs, chacune représentant un objet graphique. La requête de picking est donc triviale puisqu'il s'agit d'aller lire dans la texture et de déterminer la couleur (et donc l'objet associé à

cette couleur) situé à la coordonnée du curseur. Cette méthode de picking nous permet donc de contrôler le coût du picking puisqu'il est indépendant du nombre de “curseurs”, ou de requêtes. Un inconvénient de cette méthode (mais qui pour les cas d'utilisations que nous avons rencontré ne nous a pas posé de problème) est que le système ne peut pas connaitre l'ensemble de la pile des formes graphiques sous le curseur de la souris. Notre technique ne donne en effet que l'élément le plus proche du curseur, alors que le mode de rendu *select* d'OpenGL transmets la liste des objets dans l'ordre. Dans ce cas, il est nécessaire d'utiliser l'algorithme plus coûteux cité plus haut (GL_SELECT).

2.11 Conclusion

Nous avons abordé dans ce chapitre comment nous avons réalisé notre compilateur graphique et la boîte à outils graphique associée. Nous avons repris les concepts de la théorie de la compilation et les avons appliqués au domaine des applications graphiques interactives.

Avec le recul, nous pensons que beaucoup des choix que nous avons fait se sont révélés intéressants : séparer les différentes transformations afin d'être capable d'interchanger ces transformations et donc les langages cibles, réaliser des dépendances à l'aide de la théorie du flot de données, utiliser des langages intermédiaires afin de réaliser des optimisations graphiques au niveau le plus approprié en sont de bons exemples.

Cependant, si nous devions refaire le compilateur graphique aujourd'hui, certains points méritent d'être changés. Tout d'abord, le choix d'un langage à prototypage rapide comme Python peut sembler juste au premier abord mais ne l'est pas en réalité. Dès lors que l'on essaie de couvrir une norme comme SVG et que l'on essaie de maintenir une certaine qualité de codage (au moins vérifier que l'ensemble du code ne comporte pas d'erreurs), ce choix se retourne contre le programmeur puisque les erreurs ne sont levées qu'à l'exécution. Un autre écueil lié à la manière dont nous avons conçu le compilateur graphique est lié au fait que nous souhaitions réaliser à tout prix des optimisations inter-procédurales, donc sur toute la scène. Ceci nous a entravé dans le développement de certaines parties (comme la gestion dynamique de l'insertion d'objets graphiques dans le graphe de scène) pour au final ne pas pouvoir réaliser pleinement ces optimisations interprocédurales pour des questions de mémoire lors de l'application des transformations sur la scène globale.

Chapitre 3

Discussion et analyse

Sommaire

3.1 Évaluations préliminaires **79**
3.1.1 Puissance d'expression 79
3.1.2 Performances 81
3.1.3 Utilisabilité 81
3.2 Premiers retours des designers d'applications interactives 84
3.3 Conclusion **85**

3.1 Évaluations préliminaires

Comme beaucoup de méthodes qui visent à aider la démarche de design, évaluer l'utilisabilité d'une boîte à outils demande de réaliser des expérimentations contrôlées, avec différentes équipes de designers et sous différentes conditions. Réaliser une telle expérimentation est une tâche lourde et n'a pu être réalisée au cours de ces travaux de thèse. Nous proposons d'évaluer Hayaku selon trois aspects : sa puissance d'expression et son utilisabilité par l'utilisation des dimensions cognitives [Green 89], et ses performances lors des trois scénarios présentés au chapitre 1.

3.1.1 Puissance d'expression

Afin d'évaluer la puissance d'expression de la boîte à outils, nous proposons deux dimensions d'analyse : l'étendue des applications que l'on peut réaliser avec Hayaku, et la simplicité de la description de telles applications. Une cible d'applications trop restreinte indique en effet que la boîte à outils est trop spécialisée et que les bénéfices attendus ne seront pas vraiment significatifs. À l'opposé, étendre cette cible a souvent pour coût de diminuer la simplicité de description.

L'ensemble des applications possibles

Le compilateur graphique (et donc Hayaku) est capable de générer des interactions simples de type WIMP (tels que des ascenseurs) et des scènes graphiques plus complexes avec un grand nombre d'objets graphiques, telle que l'image radar. Hayaku est aussi capable de générer et gérer des interacteurs post-WIMP (pie-menus), des applications graphiques riches en terme de design (le clavier logiciel avec effet loupe), et peut supporter des applications multi-touch.

Cependant, comme nous l'avons expliqué dans le chapitre relatif à l'implémentation du compilateur graphique, nous n'avons présenté ici que des applications dont le nombre d'objets est connu à l'avance. Dans le cas où le nombre d'objets n'est pas connu au lancement, Hayaku fourni deux stratégies possibles : gérer un ensemble borné d'objets graphiques invisibles, ou bien générer des composants dynamiques qui restent à la charge de l'application hôte.

Enfin, nous n'avons pas conçu Hayaku comme étant capable de produire des applications très dynamiques, comme un éditeur graphique, puisque nous pensons que Hayaku n'est pas fait pour de telles applications. En effet, nous suspectons que le fait de vouloir écrire de telles applications avec Hayaku nécessiterait de “tordre” son modèle conceptuel et rendrait la réalisation trop pénible par rapport aux bénéfices escomptés.

Simplicité

Malgré nos recherches dans la littérature, nous n'avons pu trouver une définition claire du terme *simplicité*[Letondal 10]. Nous avons donc fait le choix de l'exprimer en termes de compacité du code nécessaire à la description des graphismes interactifs. Nous fournissons donc le nombre de lignes de code (LDC) des scénarios d'usage décrits plus tôt (table 3.1). Comme indiqué dans le chapitre 1, la fourniture de la description de la scène de certains des exemples est fastidieuse. Un palliatif à consisté à recourir à un script pour la produire. Ceci correspond à la colonne “générateur”.

application	modèle conceptuel	modèle vers SVG	scène	générateur de la scène
multi-touch	43 LDC	90 LDC	54 LDC	∅
clavier	129 LDC	199 LDC	890 LDC	210 LDC
pie menu	40 LDC	42 LDC	102 LDC	46 LDC

Table 3.1 – Le nombre de lignes de codes (LDC) nécessaires à l'écriture des différents scénarios présentés.

Le nombre de lignes de code par fichier est raisonnable du point de vue d'un programmeur (de l'ordre de 200 au maximum). Cependant, Hayaku s'adresse à des designers d'interactions, et il est légitime de se demander si cela n'est pas trop long. Pour ce public, un éditeur graphique abstrayant le langage permettrait une meilleur appropriation du langage. La description du modèle consiste à réaliser une modélisation des classes du modèle, et une notation type UML serait approprié. Pour le fichier de connexions, Un éditeur soulagerait la redondance qu'il y a entre le fichier du modèle et les classes graphiques

(SVG). Enfin, placer directement les éléments graphiques dans la scène permettrait de s'affranchir du fichier de scène pour le designer.

3.1.2 Performances

La table 3.2 présente les différentes performances obtenues dans ces trois cas d'utilisation. Les performances sont calculées en générant la scène avec Hayaku et en utilisant le moteur de rendu OpenGL. Nous avons différencié le "temps de première compilation" du "temps de recompilation" puisque Hayaku réalise un cache entre deux compilations successives (compilation des fontes par exemple). Le temps le plus significatif est donc le temps de recompilation puisque c'est ce temps que le designer graphique obtiendra le plus souvent : les designers graphiques passent plus de temps à réaliser des petits incréments à leur description, et relancent donc souvent la compilation.

application	temps de rendu	temps de première compilation	temps de recompilation
multi-touch	∼1.9 ms	2.2 sec	1.9 sec
keyboard	∼7.5 ms	29.1 sec	8.6 sec
pie menu	∼2.5 ms	10.2 sec	2.9 sec

TABLE 3.2 – Les performances obtenues avec les différents scénarios.

Nous voyons ici que pour notre application graphique interactive la plus riche en graphismes, le clavier logiciel, les temps de rendu donnés sont bons. En effet, ce clavier tourne à plus de 130 images par secondes, ce qui est compatible avec la boucle de perception de l'humain (25 images par secondes).

Les temps de compilations sont eux plus gênants pour le designer d'interactions. Une dizaines de secondes pour la mise à jour du graphisme n'est pas acceptable du moment que l'on fait du design exploratoire. Cependant, Hayaku n'est qu'un prototype et le temps de compilation peut être réduit avec une implémentation plus optimisée.

3.1.3 Utilisabilité

Évaluer l'utilisabilité d'une boîte à outils est un problème de recherche toujours ouvert [Myers 00] et nécessite de réaliser des expérimentations contrôlées, avec différentes équipes de designers sous différentes conditions expérimentales. Nous proposons donc de discuter de l'utilisabilité de Hayaku en se basant sur les *dimensions cognitives* des notations[Green 89] qui explicitent les qualités et les défauts d'une notation (ou un langage). Les dimensions cognitives sont basées sur les activités typiques de l'usage des systèmes interactifs. Nous avons choisi d'évaluer les activités suivantes : l'incrémentation, la transcription, la modification et le design exploratoire; selon les dimensions suivantes : proximité de la représentation, dépendances cachées, choix prématurés, évaluation progressive, abstraction, viscosité, et visibilité.

Proximité de la représentation (Closeness of Mapping)

Le designer d'interactions créé (de manière incrémentale) ses dessins directement dans un éditeur graphique : le résultat obtenu est alors très proche du produit final, en tout cas bien plus proche qu'avec une description textuelle du graphisme. Ceci permet donc l'utilisation d'outils de dessins *exploratoires* (tels qu'Inkscape), et permet donc de maximiser cette activité. La *modification* des graphismes est facile puisque l'éditeur modifie le fichier SVG en conservant les différentes propriétés déjà renseignées (par exemple le nommage). La *modification* sur l'application finale est faite automatiquement grâce au compilateur graphique. Enfin, porter l'application sur différentes plateformes est peu coûteux du fait que l'on utilise un compilateur qui peut recevoir plusieurs langages sources et plusieurs langages cibles (*transcription*).

Le langage source du compilateur est le modèle conceptuel de l'application écrit au format JSON. Puisque l'utilisateur conçoit lui-même ce modèle conceptuel, il peut le rendre le plus proche possible du domaine modélisé. On peut donc dire que la proximité de la représentation est maximisée.

Dépendances cachées (est-ce que des liens importants entre les entités sont cachés)

Effectuer les liens entre les graphismes, le modèle conceptuel et le système de flot de données nécessite de changer de notation (un éditeur graphique d'un côté, et une notation textuelle de l'autre). Ainsi, il est difficile pour le designer d'interaction de se représenter les liens entre ses modèles et ses graphismes puisqu'il doit faire la synthèse de trois fichiers pour y arriver.

Le système de flot de données produit par le CG n'est pas entièrement visible. En effet toutes les dépendances entre les données du modèle et le graphisme sont traduites par un flot de donnée dans le code généré. Il est donc difficile d'inférer exactement quelles modifications vont être réalisées une fois que la description du modèle et des transformations ont été données. Cependant, le designer est plus spécifiquement intéressé par le flot de données qu'il a lui-même renseigné. Cette partie étant uniquement située dans le fichier du modèle conceptuel, la synthèse des conséquences d'une modification d'une variable est simple à envisager (à condition que le fichier ne soit pas trop gros). La partie du flot de données générée et implémentée par le compilateur est quant à elle moins susceptible d'être lue et comprise par l'utilisateur, sauf pour déverminer l'application ou le compilateur. Ainsi, si le designer d'interaction ne sait pas exactement comment le flot de données est généré et quelles seront les étapes intermédiaires pour arriver au résultat, il est sûr que son calcul sera bien effectué et respecté.

Choix prématurés (est-il nécessaire de faire des choix dont la modification est coûteuse)

Utiliser un compilateur graphique permet d'éviter par nature des choix prématurés. Par exemple, changer l'environnement d'exécution peut être effectué à n'importe quel moment tout au long du processus de développement. De plus, la modification d'une propriété des objets graphiques ne requiert qu'une recompilation pour être impactée sur l'application

finale. Hayaku reposant sur un système de feuilles de styles pour lier le modèle graphique des formes graphiques en elle-même, nous pouvons aussi dire que le design graphique de l'application peut être ré-écrit plusieurs fois sans impacter sur la partie comportementale de l'application. Cependant, la structure des graphismes ne doit pas changer trop souvent puisque les différentes descriptions fournies se reposent en grande partie dessus (voir la partie viscosité pour plus de détails).

Évaluation progressive

L'évaluation d'une modification sur les graphismes est immédiate ce qui permet de modifier de façon incrémentale. Cependant, évaluer la modification du comportement de l'application nécessite de re-lancer le programme généré. Pour pallier ce problème, on pourrait envisager d'externaliser le comportement dans un fichier annexe, et l'on pourrait avoir le même mécanisme pour le comportement que pour le graphisme : lors d'un changement, le fichier de comportement est relu et les modifications sont automatiquement impactées.

Abstraction (types et disponibilité des mécanismes d'abstraction)

Hayaku repose sur un ensemble de fichiers écrits en JSON pour abstraire le modèle graphique, les connexions avec les graphismes, et la scène graphique finale. Cependant, si ce langage est bien adapté pour représenter des données abstraites, et permet aux utilisateurs d'abstraire leurs applications, ce langage est mal adapté pour l'humain qui doit l'écrire dans son éditeur de texte. En particulier, les connexions entre le modèle conceptuel et les graphismes fournis mériteraient clairement d'être traitées par un éditeur graphique puisqu'aujourd'hui cette tâche est fastidieuse étant donnée qu'il faut maintenir les mises à jour de part et d'autre à la main.

Viscosité (résistance aux changements)

Le modèle conceptuel requiert que tous les objets graphiques soient déclarés dans le fichier JSON. Si un élément graphique est utilisé plusieurs fois (comme une touche dans le clavier logiciel), une modification du "prototype" va obliger à propager la modification dans toutes les instances de cet élément. Une solution à ce problème de viscosité est d'écrire des petits scripts qui vont générer toutes les instances d'un prototype décrit au format JSON. Ce problème peut être considéré comme une autre étape de la chaîne de compilation, et aide donc à abstraire les concepts du modèle conceptuel des applications à concevoir.

Un changement dans le modèle conceptuel doit être propagé dans la description des connexions et dans la description de la scène. Un programmeur d'application rencontre ce même problème lorsqu'il ajoute un attribut dans une classe C++ : il faut modifier les arguments du constructeur si l'attribut doit être paramétré à l'instanciation. De nombreux mécanismes pour éviter ce type de problème existent (par exemple l'introduction d'une valeur par défaut, ou différentes techniques de refactorisation), mais Hayaku ne le supporte pas encore. De même, une modification dans la structure des graphismes (une modification

de la hiérarchie des éléments SVG) peut aussi avoir un impact non négligeable sur la description de la connexion entre le modèle et le graphisme.

Visibilité (possibilité de voir facilement les composants)

Actuellement, la visibilité de la boîte à outils est limitée. Par exemple, la description des éléments au format JSON tend à être relativement verbeuse et longue, ce qui a tendance à embrouiller la compréhension exacte du fichier.

3.2 Premiers retours des designers d'applications interactives

Le temps d'écriture de l'application multi-touch fut très court (une demi journée pour l'application entière) et fut grandement facilitée par le mécanisme d'abstraction. Le comportement multi-touch est simple et ne nécessite pas beaucoup de lignes de code. Nous pouvons noter que Hayaku ne propose pas de bibliothèque de reconnaissances de gestes, et que nous avons dû nous baser directement sur les évènements bruts transmis par le périphérique.

Concernant le clavier logiciel, nous avons fourni Hayaku au designer original de ce clavier. Nous lui avons alors demandé de le re-créer. Le designer en question est habitué à concevoir des graphismes mais aussi à écrire du code d'interaction.

Selon cet utilisateur, le premier point remarquable de la boîte à outils est la fiabilité du rendu. La toolkit utilisant un compilateur graphique, le code de rendu final souffre moins de compromis entre rapidité et puissance d'expression que les toolkits qu'il utilise habituellement. Le rendu graphique de l'application interactive est ainsi très proche du rendu statique produit par un éditeur vectoriel comme Inkscape ou Illustrator. Grâce à l'utilisation de SVG et du compilateur graphique, le pouvoir d'expression du designer graphique n'est potentiellement pas limité.

Le designer a ainsi beaucoup apprécié la puissance de description du modèle, à savoir la possibilité de pouvoir accéder à toutes les propriétés graphiques de tous les objets et de décrire les variables et leurs transformations. Ceci lui a permis de contrôler finement le comportement graphique des objets d'interface qu'il concevait. L'évolution de ces propriétés peut être décrite soit "relativement" à une autre à l'aide d'une expression mathématique (similaire au concept de contraintes dans Garnet [Vander Zanden 01b]), soit peut être transmise par la partie comportementale. Par exemple, le point d'ancrage d'une touche dépend de sa largeur (`"FORM_X0": "(self.WIDTH - 100) / -2"`). Puisque la largeur de la touche dépend de la distance au curseur, ce point d'ancrage est modifié automatiquement à chaque mouvement du curseur.

Les entrées-sorties de Hayaku ont été simplifiées par le fait de tout considérer comme un flot de données, ce qui simplifie la gestion des entrées-sorties pour le concepteur. Par exemple, l'implémentation de la mise à l'échelle globale du clavier n'a pas pris plus de 10 minutes, le temps de comprendre et implémenter la solution qui consiste à connecter les deux variables *screen_width* et *screen_height* à l'application. Ces connexions permettent

au designer graphique de construire rapidement des interactions complexes tel que l'effet "fish-eye" des touches.

Cependant, si le compilateur graphique et la boîte à outils associée permettent de faciliter l'écriture d'une application hautement interactive, il existe encore des problèmes : tout d'abord, l'écriture à la main des fichiers sources concernant la description abstraite des éléments graphiques est fastidieuse. L'intégrité et la cohérence entre les différents fichiers doit être traitée à la main et introduit actuellement une lourdeur sur le développement de l'application. Ensuite, le compilateur dans sa version actuelle ne supporte qu'un nombre limité de formes graphiques (de l'ordre de quelques milliers) dû à une mauvaise gestion de la mémoire dans les phases intermédiaires de compilation.

3.3 Conclusion

Comme les premiers retours le montrent (surtout celui du clavier logiciel), le temps de compilation est encore trop long pour favoriser la création. Hayaku est développée en Python afin d'être rapidement développée, et beaucoup de portions de codes ne sont pas optimisées (notamment les parcours de graphes). Beaucoup d'optimisations pourraient être réalisées afin d'améliorer ces lacunes de la boîte à outils.

Les performances actuelles du code généré par Hayaku est jugé comme bon. Compte tenu de la qualité produite, des temps de l'ordre de la dizaine de millisecondes ne nous paraissent pas rédhibitoires. Si je devais créer une application industrielle plus complexe que les exemples présentés ici, je choisirais peut être de ne produire que certaines parties de mon interface avec Hayaku et les parties trop critiques pourraient être codées à la main. Beaucoup de travail pourrait être réalisé pour améliorer encore les performances de rendu.

Enfin, les premiers retours des utilisateurs sont très engageants et nous permettent de penser que nous sommes dans la bonne voie pour le développement d'applications graphiques interactives. Cependant, ces mêmes retours sont critiques sur certains points, et notamment la syntaxe fastidieuse du langage JSON utilisé pour décrire les différents fichiers d'entrée. Nous pensons régler ce problème dans le futur en cachant ces fichiers à l'utilisateur grâce à une interface graphique. Il est possible d'imaginer un éditeur de modèle conceptuel qui se rapprocherait de la description d'un diagramme de classes, les *connections* peuvent être représentées visuellement par des liens entre ces objets du modèle et les objets graphiques SVG, et la description de la scène globale pourrait être aussi représentée en temps réelle.

Cette boîte à outils n'est aujourd'hui qu'un prototype, une preuve de concept, mais néanmoins elle permet déjà de réaliser des applications graphiques riches. Pour pouvoir conquérir le monde des boîtes à outils graphiques, Hayaku devrait tout d'abord finir d'implémenter l'ensemble de la sémantique de SVG, améliorer encore ses performances tant dans le temps de compilation que dans l'efficacité du code produit, et aussi ajouter de nouveaux langages et plateformes de sortie.

Troisième partie

Maximisation non instrumentée de la séparation entre description et implémentation des systèmes graphiques interactifs

Chapitre 1

Réalisation d'applications avec comme contrainte principale la vitesse d'exécution

Sommaire

1.1 Description de l'application, des besoins 90

1.1.1 La configuration visuelle 90

1.1.2 La rotation "Rolling the Dice" 91

1.1.3 Le brushing et la sélection incrémentale 91

1.1.4 L'organisation des vues et le *"Pick'n Drop"* 92

1.2 Implémentation 92

1.2.1 La génération de la vue 93

1.2.2 Dessiner des lignes ou des ronds 93

1.2.3 La rotation "Rolling the Dice" 96

1.2.4 Le Brush 96

1.2.5 Le Pick and Drop 97

1.2.6 Le rendu des différentes vues dans la grille 97

1.3 Conclusion 98

Les travaux présentés dans la partie précédente nous ont montré qu'il était possible de réaliser à faible coût des applications graphiques interactives grâce à l'utilisation d'un compilateur graphique. Néanmoins, si l'utilisation du compilateur graphique permet de s'affranchir de nombreuses contraintes et permet un gain de temps non négligeable, il reste des cas où l'optimisation manuelle reste plus efficace. Nous avons abordé ce problème au cours de la section 1.3.4 de la partie précédente. Nous avions cherché à intégrer des composants (un pie-menu) dans une application OpenGL écrite et optimisée à la main. Dans ce chapitre, nous présentons des exemples de cas où l'optimisation manuelle ne peut être remplacée par le compilateur graphique.

1.1 Description de l'application, des besoins

Nous avons réalisé un logiciel de visualisation et d'exploration de grande quantité de données multidimensionnelles : FromDaDy [Hurter 09]. L'objectif de l'application est d'afficher et de manipuler un grand nombre de trajectoires (> 100000) en temps interactif. Le logiciel est capable d'afficher plusieurs millions de données, sans gêner la boucle perception/action. Cette section détaille les principes de l'application FromDaDy qui permet de faire de l'exploration de grande quantité de données.

1.1.1 La configuration visuelle

FromDaDy utilise le mécanisme de flot de données au travers d'un outil qui permet à l'utilisateur de dessiner des connexions entre les champs des données et les variables visuelles [Bertin 83] et ainsi créer des *configurations visuelles*. Par exemple, dans la partie gauche de la figure 1.1, l'utilisateur a connecté la longitude avec l'axe X de la visualisation, et la latitude avec l'axe Y de la visualisation.

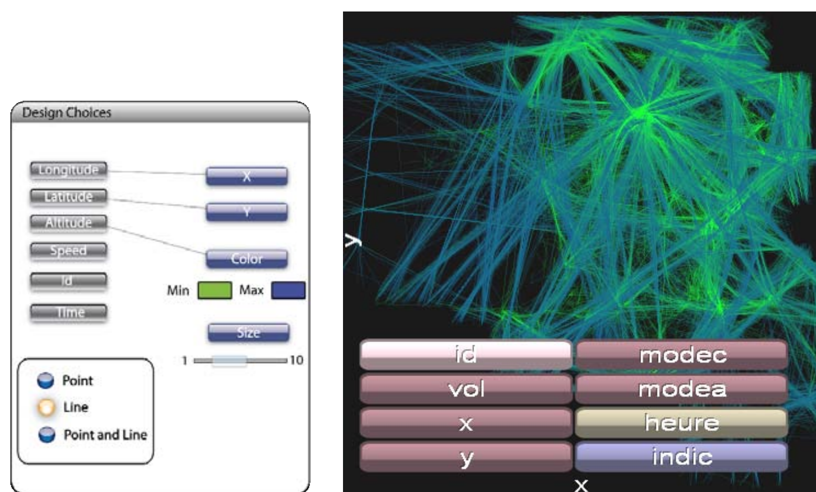

FIGURE 1.1 – L'outil de configuration pour le design visuel (à gauche) et le résultat (à droite).

L'utilisateur a aussi connecté le champ altitude de la base de données sur la couleur des lignes. La visualisation résultante produit une représentation cartésienne d'une journée de trafic au dessus de la France, avec l'altitude codée par la couleur (Figure 1.1 à droite). Ainsi, les lignes de couleur bleu correspondent aux avions qui ont volé dans des hautes altitudes, et les lignes vertes aux avions qui ont volé dans les couches basses. L'utilisateur peut aussi choisir de cliquer sur l'axe X ou Y directement sur la vue pour faire apparaître un menu qui permet de changer la correspondance entre l'axe visuel et le champ de la base de données.

1.1.2 La rotation "Rolling the Dice"

Les changements brutaux des axes d'une vue sont perturbants car ils empêchent l'utilisateur de garder son attention sur une entité graphique. De ce fait, FromDaDy utilise une animation similaire à celle présentée dans "Rolling the Dice" [Elmqvist 08]. En d'autres termes, une dimension de la vue est conservée lorsque l'autre change. Au lieu de simplement interpoler les différentes positions de chacun des points qui constituent la vue, FromDaDy réalise une rotation autour de l'axe correspondant à la dimension à conserver, en utilisant une troisième dimension dont l'axe est associé à la nouvelle dimension à visualiser (Figure 1.2). Cette animation donne de la sémantique au mouvement des points, ce qui permet à l'utilisateur d'interpréter l'animation comme une rotation, et ainsi lui permet de garder le focus sur une ou plusieurs entités visuelles pendant l'animation.

L'utilisateur peut contrôler manuellement la transition en cliquant sur un axe et en bougeant verticalement (ou horizontalement selon l'axe choisi) le curseur de la souris (Figure 1.2).

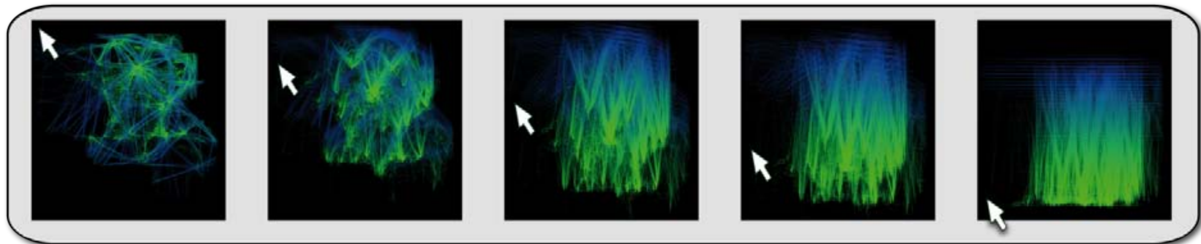

FIGURE 1.2 – L'utilisateur contrôle la transition entre la vue de dessus (Latitude, Longitude) et la vue verticale (Altitude, Longitude) en déplaçant le curseur le long de l'axe vertical.

1.1.3 Le brushing et la sélection incrémentale

L'utilisateur peut sélectionner des sous-ensembles sur sa vue au moyen d'une technique de pinceau (le *brushing*). La technique du brushing est une interaction qui permet à l'utilisateur de "peindre" les entités graphiques en utilisant un outil dont la taille et la forme sont paramétrables par l'utilisateur [Becker 87]. Chaque trajectoire qui intercepte la zone peinte lors du mouvement de la souris passe en mode *sélectionnée*, et devient grise. La sélection peut aussi être complétée a posteriori (la touche "contrôle" étant enfoncée), ou peut aussi être élaguée grâce au mode d'effacement du brush (la touche "majuscule" étant pressée). Notre implémentation laisse sur la vue finale le dessin de la zone peinte afin que l'utilisateur puisse voir et se rappeler plus facilement comment la sélection a été faite. Toutes les trajectoires qui coupent la zone peinte sont sélectionnées : modifier la sélection revient à compléter ou effacer la zone peinte (Figure 1.3, premières vignettes).

Lorsque les touches "contrôle" ou "majuscule" sont enfoncées, la taille de l'outil de brush peut être ajustée à l'aide de la molette de la souris. Si aucune de ces deux touches n'est appuyée, la molette de la souris permet d'utiliser la fonction de zoom sur la vue. La combinaison du changement rapide entre le mode d'ajout et le mode de suppression de la

sélection, la visualisation de la zone peinte, la sélection rapide du choix de la taille de la brosse et le zoom centré souris permet à l'utilisateur d'opérer une sélection incrémentale rapide.

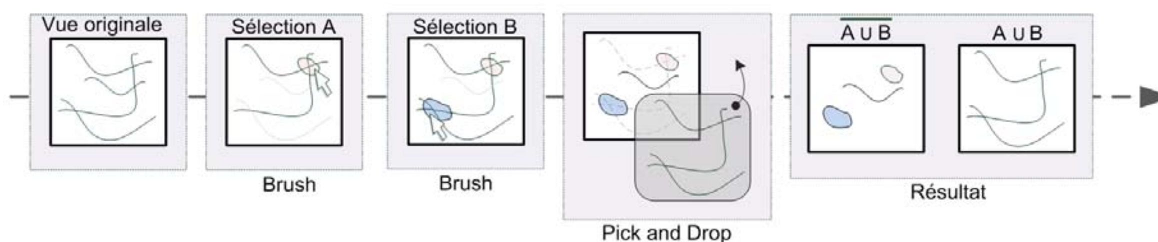

Figure 1.3 – Le brushing et le pick and drop de FromDaDy.

1.1.4 L'organisation des vues et le *"Pick'n Drop"*

Une session d'utilisation de FromDaDy commence par afficher une vue avec l'ensemble des données. Cette visualisation est paramétrée par défaut pour chacun des ensemble de données (au premier lancement). Cette vue est incluse dans une fenêtre et occupe alors une cellule dans la grille virtuellement infinie qui s'étend à partir des quatre côtés de la cellule.

Comme nous l'avons vu, l'utilisateur peut sélectionner par la technique du brush une sous-partie de ses données. En appuyant sur la barre d'espace, il aspire (*"Pick"*) les données dans une nouvelle vue rattachée au curseur de la souris (Figure 1.3). Les données aspirées restent sous le curseur tant que l'utilisateur n'appuie pas une nouvelle fois sur la barre d'espace. Il peut continuer d'interagir avec FromDaDy, et peut donc effectuer des opérations de zoom et de translation sur les cellules disponibles.

En appuyant à nouveau sur la barre d'espace, l'utilisateur lâche (*"Drop"*) ses données sur la vue sous le curseur, ou sur une nouvelle si la cellule ne contenait pas de vue.

Grâce au brushing et au paradigme du Pick'n Drop[Rekimoto 97], l'utilisateur peut donc créer de nouvelles vues afin d'afficher d'autres représentations de ses données. L'utilisateur détruit une vue lorsqu'il extrait toutes les données contenues dans une cellule.

1.2 Implémentation

L'implémentation d'un logiciel comme FromDaDy n'a pu être réalisé par notre compilateur graphique. En effet, une des premières difficultés de FromDaDy est d'afficher un grand nombre de trajectoires dynamiques, dans le sens ou le choix des dimensions est fait à l'exécution. Un deuxième problème est que les transformations en trois dimensions (comme celle utilisée par la rotation "Rolling the dice") ne sont pas supportées par notre compilateur graphique puisque le modèle graphique est basé sur SVG qui est fait pour des dessins en deux dimensions. Enfin, la sélection des trajectoires par picking demanderait un gros travail sur le compilateur graphique et rien n'aurait garanti des performances suffisamment élevées.

Travailler avec Hayaku nous aurait contraint à implémenter de nouvelles fonctionnalités de bas niveau, et il aurait été difficile d'en développer de nouvelles au fur et à mesure de l'avancement du projet. Pour toutes ces raisons, FromDaDy repose sur un moteur d'exécution ad-hoc entièrement déporté sur la carte graphique. Le choix de déporter le moteur d'exécution vers la carte graphique est dû à l'objectif principal de FromDaDy : présenter le plus rapidement possible les données à l'utilisateur pour qu'il puisse interagir avec.

1.2.1 La génération de la vue

Dessiner les données en entrée à FromDaDy le plus rapidement possible requiert d'effectuer le moins d'échanges possible entre le processeur central et le processeur graphique. En effet réaliser à chaque rafraîchissement ces transferts d'informations nécessite une bande passante non disponible. FromDaDy étant capable d'afficher plusieurs millions de données (sous forme de *float*), on ne peut se permettre de transférer des centaines de mega-octets à chaque image.

Nous avons donc décidé de ne transférer qu'une seule fois ces informations vers la carte graphique, au chargement de la base de données. Afin de s'abstraire des notions d'échelle entre les différentes données, nous normalisons tous les champs de la base entre 0 et 1. Nous créons donc une zone mémoire sur la carte graphique dans laquelle nous transférons l'ensemble de la base de données normalisée.

Pour afficher les paramètres visuels corrects, nous utilisons un *vertex shader* pour affecter les champs de la base qui iront dans les paramètres graphiques. La description des paramètres d'entrée du vertex shader est générée automatiquement en fonction des données disponibles. Cette description permet de spécifier la forme des données insérées dans la carte graphique et cette structure dépend des données à afficher (principalement du nombre de champs par donnée).

Les paramètres graphiques sont commandés depuis le processeur central en utilisant des variables de type *uniform*. En sortie du vertex shader, le pipeline graphique reçoit les coordonnées des points à dessiner (les vertices) dans l'espace écran, ainsi que les différents paramètres graphiques utilisé pour le représentation (la taille et la couleur) (Figure 1.4).

1.2.2 Dessiner des lignes ou des ronds

En sortie du vertex shader, n'est émis que le point courant dont les coordonnées et les paramètres graphiques ont été extrait depuis la base de données. L'étage suivant utilisé par FromDaDy est de générer d'autres vertices en fonction de ces paramètres de configuration.

Dans le cas où l'utilisateur choisi de dessiner des points circulaires, le *geometry shader* génère alors 4 vertices (V_0 à V_3 sur la Figure 1.5) par vertex en entrée pour dessiner deux triangles qui formeront la boite englobante du rond à dessiner. Pour dessiner le rond, nous utilisons le dernier étage programmable du pipeline graphique (le *pixel shader*). La carte graphique calcule pour chacun des pixels à dessiner la distance par rapport au centre de la coordonnée, et ne dessine le pixel que si la distance est supérieure à 1.

Dessiner des lignes épaisses avec des raccords nécessite un traitement plus complexe de la part du geometry shader. Il s'agit cette fois de générer des triangles (figure 1.6) entre

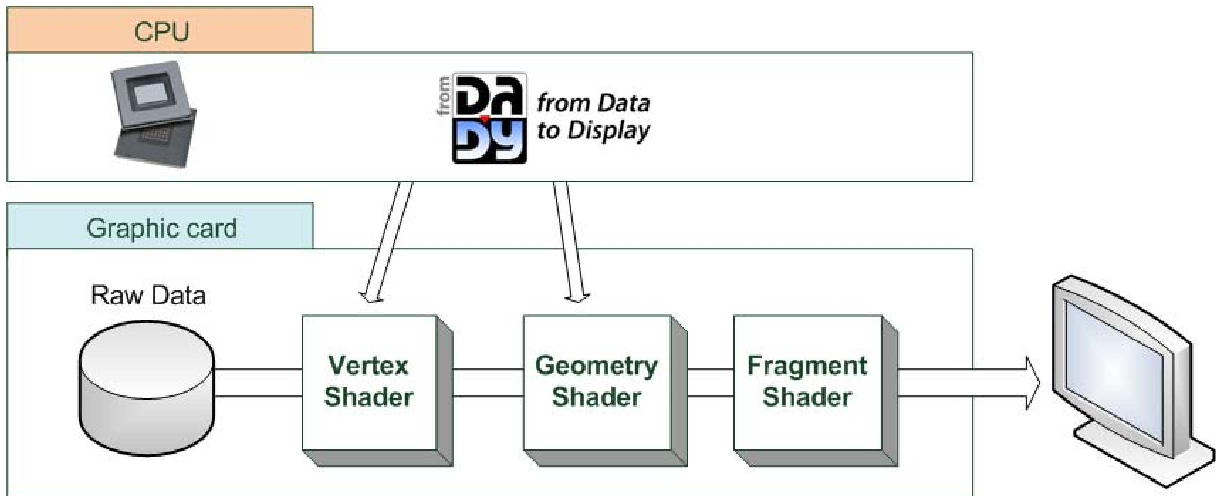

FIGURE 1.4 – Répartition du travail dans FromDaDy : le CPU ne fait que du contrôle sur les shaders embarqués dans le processeur graphique.

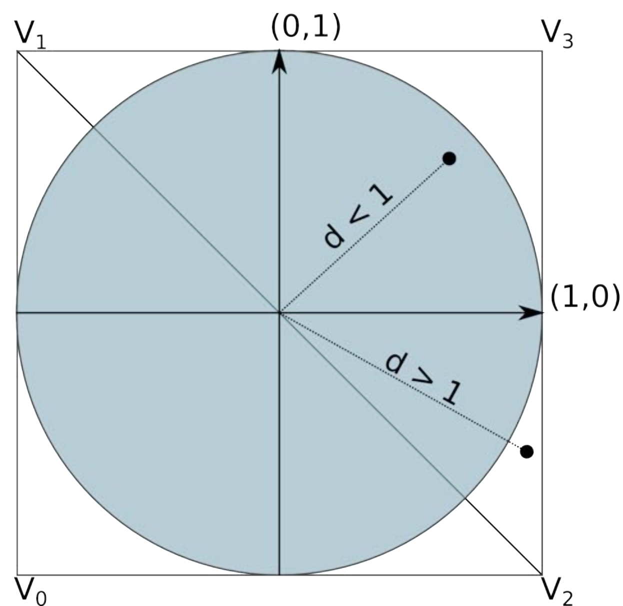

FIGURE 1.5 – La technique de dessin des points dans FromDaDy.

deux coordonnées fournies. Pour réaliser les raccords, nous nous sommes reposés sur la version *points* de l'affichage. Cependant, pour réduire les calculs de tessellation effectués par cette étape, nous avons décidé de tracer des segments de trajectoire indépendamment les uns des autres. Ceci pose un problème lorsque l'utilisateur choisit une représentation avec de la transparence. En effet, pour être sûr qu'il n'y ait pas de "trous" entre les segments de chaque trajectoire, le geometry shader écrit à chaque nœud de la trajectoire un point de la taille souhaitée (avec le mode *points*). Le rajout des polygones formant le segment ajoute des artéfacts visuels puisque l'on "fonce" la transparence (la figure 1.6 donne un bon exemple de ce type d'artéfact). Le test de *stencil* permet de résoudre ces artéfacts visuels qui apparaissent. l'idée consiste à ne pas redessiner sur un pixel qui correspond à la trajectoire courante.

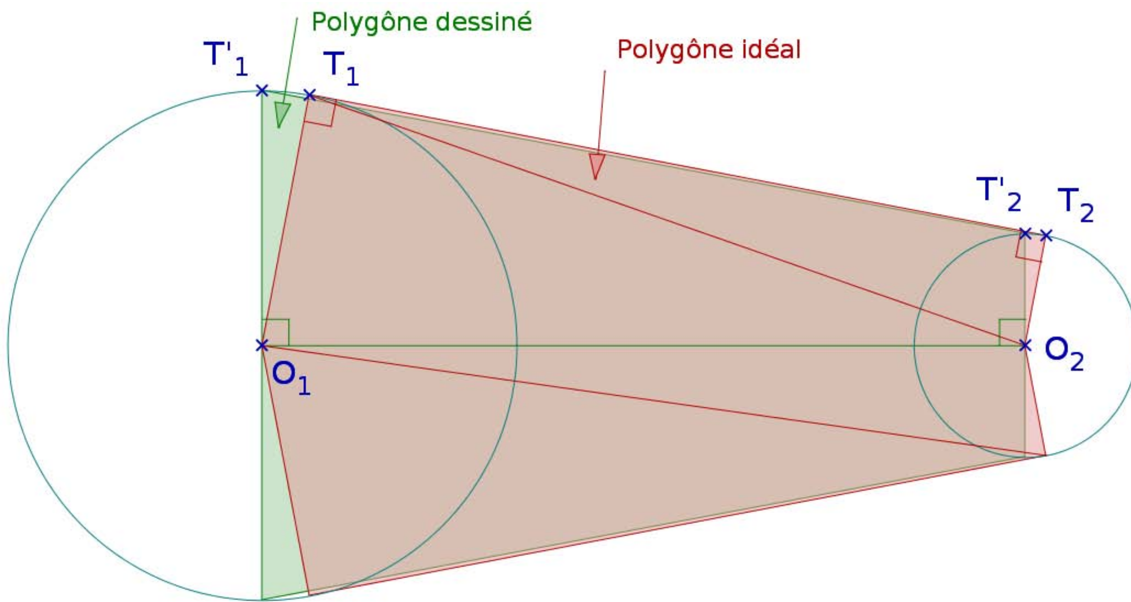

FIGURE 1.6 – Différence entre le polygone idéal et le polygone dessiné par FromDaDy.

Le deuxième problème que nous avons rencontré se situe au niveau de la détermination des coordonnées du polygone à afficher pour combler le segment. Puisque la taille de chacune des extrémités du segment peut varier, la recherche des points de contact des deux cercles et de leurs tangentes est trop coûteuse (voir l'annexe C pour un détail des calculs). Nous avons donc simplifié le problème en approximant le polygone idéal (en rouge sur la figure, passant par T_1 et T_2) par un polygone moins coûteux à calculer (le trapèze vert passant par T_1' et T_2'). Les coordonnées du polygone idéal sont les suivantes :

$$\boxed{\begin{cases} \vec{O_1T_1} = R_1 \frac{(R_1-R_2)}{\bar{O_1O_2}} \vec{O_t} \pm R_1 \frac{\sqrt{\bar{O_1O_2}^2-(R_1-R_2)^2}}{\bar{O_1O_2}} \vec{O_n} \\ \vec{O_2T_2} = R_2 \frac{(R_1-R_2)}{\bar{O_1O_2}} \vec{O_t} \pm R_2 \frac{\sqrt{\bar{O_1O_2}^2-(R_1-R_2)^2}}{\bar{O_1O_2}} \vec{O_n} \end{cases}} \tag{1.1}$$

D'un autre côté, les calculs de T_1' et T_2' sont plus simples :

$$\begin{cases} \vec{O_1T_1'} = \pm R_1\vec{O_n} \\ \vec{O_2T_2'} = \pm R_2\vec{O_n} \end{cases} \tag{1.2}$$

On passe de 10 opérations par point de tangence (dont deux racines carrées) à seulement une multiplication. La figure 1.6 présente cette optimisation. Le polygone idéal (en rouge sur la figure) doit être tangent aux deux cercles. Nous avons choisi de tracer un polygone plus simple (un trapèze dont les droites parallèles sont perpendiculaires à la direction de la ligne, en vert sur la figure). Cette optimisation crée des artéfacts visuels (lorsque la différence de taille entre les deux extrémités est trop importante), mais pour le cas général, cela ne gêne pas l'utilisateur.

1.2.3 La rotation "Rolling the Dice"

Implémenter la rotation "Rolling the Dice" est relativement simple. Les calculs d'affichage sont réalisés sur la carte graphique. Cette rotation ayant lieu dans un espace à 3 dimensions, la technique classique de rotation d'un objet dans l'espace fonctionne. Le choix des variables à affecter à chacun des axes (X,Y,Z) est réalisée par le vertex shader décrit dans la sous-section 1.2.1.

1.2.4 Le Brush

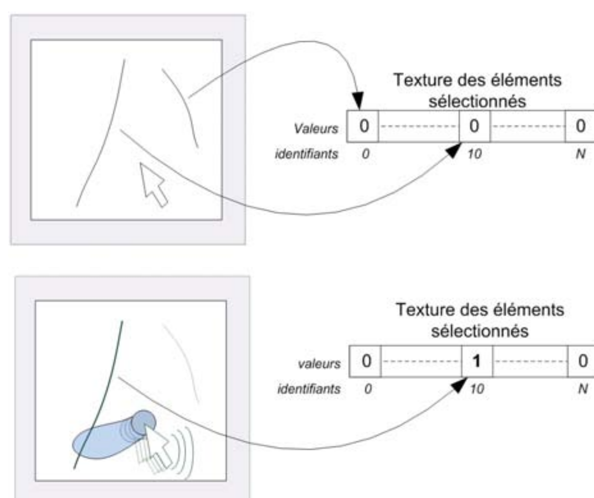

FIGURE 1.7 – Fonctionnement interne du brush dans FromDaDy

La fonction de brush est implémentée en trois passes (Figure 1.7). La première passe consiste à mettre à jour la zone peinte sur l'écran. Pour cela, on met à jour une texture dans laquelle est stockée l'ensemble des pixels qui forment la trace dessinée par l'utilisateur. Pour avoir une sélection incrémentale, on n'efface pas la texture d'une image sur l'autre. Ceci permet de mémoriser les différents tracés de l'utilisateur.

La deuxième passe consiste à déterminer pour chacune des trajectoires (ou des points dans le mode point) si elle (ou ils) intercepte la zone peinte dans le rendu final de la vue. Chaque trajectoire (et chaque point) a un identifiant unique. Pour réaliser l'opération de sélection, il faut changer la texture de rendu (la *render target*) pour la remplacer par une texture qui va contenir les états de sélection des lignes et des points. Chaque identifiant traduit une coordonnée et une seule dans cette texture. Les shaders mis en œuvre vont écrire un booléen à la coordonnée liée à l'identifiant (de trajectoire ou de point), qui stipule si le tracé intercepte la zone de brush. Une trajectoire sélectionnée est donc une trajectoire dont l'un au moins de ses points intercepte la texture de brush.

Enfin, il faut garantir que la trajectoire n'est pas sélectionnée/désélectionnée au fur et à mesure des tests qui ont lieu le long de la trajectoire. Si le test est positif, on écrase la valeur précédente (initialement à 0), et si le test est négatif, on ne modifie pas l'état courant (instruction *discard*).

La troisième et dernière passe de la sélection consiste à rendre la vue finale en appliquant un style différent (assombri ou pas) selon que la texture produite à l'étape précédente indique si la trajectoire a été sélectionnée ou pas.

Le brush tel que décrit ci-dessus permet d'éviter de réaliser des transferts entre le processeur central et la carte graphique. Ainsi, le processeur ne connaît pas l'état *brushé* ou non des différentes trajectoires, mais le rendu en tient compte. L'utilisateur peut donc voir en temps réel le résultat de son brush, sur plusieurs centaines de milliers de trajectoires.

1.2.5 Le Pick and Drop

Les données étant en permanence disponibles sur la carte graphique, le choix de dessiner ou non dans une vue des trajectoires (ou des points de trajectoires) est réalisé par l'utilisation d'un index (en mémoire centrale) sur ces données. Le pick ou le drop consiste à mettre à jour cet index en fonction des trajectoires sélectionnées. Il faut pour cela récupérer la texture des index des trajectoires générée par l'étape de brush (l'extraire de la carte graphique pour la faire remonter dans la mémoire centrale), et faire les opérations logiques associées au pick ou au drop (respectivement exclusion ou union) sur l'index des trajectoires de la vue courante.

1.2.6 Le rendu des différentes vues dans la grille

Pour que l'utilisateur puisse utiliser son espace de travail comme il le souhaite, il est nécessaire d'avoir un rendu rapide quel que soit le nombre de vues affichées. Pour ne pas pénaliser les performances, nous utilisons un système de rendu indirect (sauvegarder dans une texture chacun des rendus). En effet, rendre directement chacune des vues de la grille réduit la vitesse de rendu au fur et à mesure que le nombre de vues augmente. Nous avons donc opté pour une mise en cache de chacune des vues de la grille. Ainsi, une vue n'est mise à jour que lorsqu'elle subit des modifications. Le reste du temps, on applique simplement la texture dans laquelle le résultat a été stocké.

1.3 Conclusion

Au cours de ce chapitre, nous avons décrit l'implémentation d'une application capable d'afficher et de manipuler de grandes quantité de données. Pour que l'utilisateur puisse jouir d'une utilisation en temps interactif, nous avons eu recours à l'utilisation au maximum de la carte graphique. Les principes mis en place ont été d'effectuer le maximum de traitements en mémoire graphique, et de laisser le processeur central ne réaliser que des tâches annexes et ponctuelles.

Nous pouvons aussi nous demander si un tel logiciel est réalisable par une boîte à outils comme Hayaku. Compte tenu de l'avancement de l'état actuel de Hayaku, la réponse est négative. En effet, Hayaku ne gère pas encore des optimisations graphiques très agressives, ni ne gère les multiprocesseurs. Cependant, nous ne pensons pas que de gros efforts sur Hayaku permettraient de réaliser de telles performances. Les choix mis en place dans FromDaDy sont trop fins et spécifiques pour pouvoir être détectés automatiquement.

Cependant, comme décrit dans la section 1.3.4 du chapitre 1 de la partie relative à Hayaku, il est théoriquement possible d'intégrer Hayaku avec FromDaDy. Hayaku permettrait de réaliser les modules d'interface non critiques en terme de performance (les boutons, menus, affichages de textes, et autres). Cela soulagerait le moteur d'exécution écrit à la main de FromDaDy. Le principal point nous empêchant de réaliser ceci est un problème de ressources pour coder : Hayaku ne contient pas encore de module de sortie Direct3D. Il faudrait soit l'implémenter, soit basculer FromDaDy en OpenGL, ce qui est un travail énorme.

Chapitre 2

Cache de champ de force avec texture pour le placement d'étiquettes

Sommaire

2.1 Problème **99**
2.2 Description des besoins **100**
2.3 Algorithmes existants **100**
2.4 Principes de l'implémentation GPU **101**
2.5 Gains **102**
2.5.1 Simplicité 102
2.5.2 Performances 103
2.5.3 Utilisation de la mémoire 103
2.5.4 Le problème est-il toujours NP-complet ? 104
2.6 Détails techniques **104**
2.6.1 Récupérer des données depuis la carte graphique 104
2.6.2 Shaders employés 104
2.7 Conclusion **106**

2.1 Problème

Ayant réalisé cette thèse dans le laboratoire de l'aviation civile française, nous nous sommes attelés au problème du placement des étiquettes sur l'image radar présentée aux contrôleurs aériens. Sur cette image, de nombreuses informations sont présentes [Tabart 07]. À chacun des vols que le contrôleur a en charge, sont associées des informations grâce à une étiquette attachée à cette représentation du vol (Figure 2.1). Le principal problème inhérent à cette juxtaposition d'information est qu'il ne faut pas que l'étiquette en elle-même masque un vol, y compris le sien, et qu'elle ne masque pas non

plus une quelconque autre étiquette. Un autre problème vient du fait que l'image présentée aux contrôleurs doit être *continue* d'une image sur une autre. Le contrôleur ne regarde pas en permanence son image radar, et si les étiquettes sont déplacées de manière trop rapide, il risque de perdre ses repères, et cela augmentera sa charge cognitive.

FIGURE 2.1 – La comète radar présentée aux contrôleurs aériens.

2.2 Description des besoins

Dans ce chapitre, nous présentons un algorithme alternatif à celui aujourd'hui utilisé qui ne nécessite pas d'optimisations pour être efficace en terme de complexité. Compte tenu des exigences métiers, le placement des étiquettes doit satisfaire aux besoins suivants :

- il ne doit pas y avoir de recouvrement vis à vis des autres étiquettes ni des autres éléments graphiques de l'image,
- le calcul du nouveau placement doit être réalisé en temps interactif,
- le placement des étiquettes d'une image sur une autre doit être continu,
- la distance entre l'étiquette et le point d'intérêt doit être fixe,
- les étiquettes doivent être placées en priorité sur certains axes, et plutôt derrière l'avion,
- enfin la description de l'algorithme doit être simple.

2.3 Algorithmes existants

Dans la littérature, ce type de problème se nomme "*point based labelling*", ou placement d'étiquettes rattachées à un point. Ce problème est connu pour être NP-complet, et une solution classique consiste à utiliser un recuit simulé [Christensen 95]. Le recuit simulé étant une approche stochastique et basée sur un grand nombre d'itérations, elle ne peut pas s'effectuer en un temps acceptable pour l'interaction.

Une solution en temps plus acceptable est proposé par [Luboschik 08]. Cette solution est basée sur l'utilisation du processeur graphique pour le calcul des positions des étiquettes, mais présente l'inconvénient de reposer sur un algorithme glouton et ne garanti rien quant à la continuité de l'affichage si les points d'attache des étiquettes bougent, ce qui est le cas pour les avions que les contrôleurs ont en charge.

Pour pallier le problème de la continuité de placement, les algorithmes plus communément utilisés reposent sur une "simulation physique". Par exemple, un système de simulation à base de ressorts vérifie deux à deux chacune des paires d'étiquettes pour voir

si elles ne se superposent pas. Cependant, un calcul de complexité montre que l'algorithme sans optimisations est en $\mathcal{O}(n!)$, n étant le nombre d'étiquettes. Ceci pose un problème lors du passage à l'échelle : les temps de calculs augmentant exponentiellement, il est impératif de recourir à des optimisations afin de conserver un temps de calcul raisonnable. Les optimisations utilisées permettent d'affiner la recherche des étiquettes environnantes à l'aide de structures locales comme des quadtree ou des Bounded Volume Hierarchy (BVH). Ces optimisations sont plus couramment utilisées pour la technique du lancer de rayons [Gourmel 10].

Nous soutenons la thèse que ces optimisations complexifient la description de l'algorithme ce qui amène à des difficultés pour maintenir l'algorithme mais aussi pour le certifier afin de prouver qu'il ne risque pas de corrompre le système. Enfin, améliorer le système actuel en autorisant l'utilisateur à définir des zones d'attraction de forme quelconque pour les étiquettes comme des zones militaires qui ne le concernent pas, ou en empêchant les étiquettes de recouvrir certains objets graphiques est une opération complexe car l'algorithme des masses-ressorts est adapté à des formes rectangulaires. En écartant l'aspect quelconque de ces objets d'attraction/répulsion, on pourrait les modéliser sous la forme d'un ressort, mais on augmente alors le n de la complexité temporelle de l'algorithme.

2.4 Principes de l'implémentation GPU

Nous avons implémenté un algorithme de placement d'étiquettes sur le processeur graphique afin de simplifier la description, ce qui permet une adaptabilité à des formes quelconques. Le portage sur la carte graphique permet aussi d'améliorer les performances. En effet, le processeur de calcul de la carte graphique d'un ordinateur (GPU) est plus puissant en termes de puissance de calcul qu'un processeur central : les processeurs Intel core i7 tournent actuellement autour de (en 2011) 50 GFlops (Giga FLoating point OPerations per Second) [12], tandis que les produits Nvidia Tesla C2050 sont autour du Tera-Flops pour les calculs en simple précision et 500 GFlops pour les calculs en double précision [13]. Ceci est du au fait qu'un GPU est optimisé pour réaliser des traitements parallèles. Les technologies CUDA, OpenCL, ou encore Direct Compute permettent aux applications non graphiques d'utiliser cette puissance.

Le problème de travailler avec la carte graphique est qu'elle n'est efficace que pour réaliser des traitements massivement parallèles, et que l'algorithme à base de masses ressort ne l'est pas.

Implémenter une simulation physique utilisant la gravité et des étiquettes ayant une forme et un poids est réalisable, mais gourmande en terme de performances même pour un processeur graphique [Liu 10]. Nous avons choisi de travailler avec des **champs de force locaux aux étiquettes** [Hirsch 82].

Le principe général est résumé par la Figure 2.2. La réalisation se fait en trois passes :

- La première passe consiste à écrire dans une texture les différents champs de forces de chacune des étiquettes (les grands rectangles sur la figure, les flèches noires représentent le champ de force associé à chacune des étiquettes). Dans cette texture, nous

12. `http://www.intel.com/support/processors/sb/cs-023143.htm`
13. `http://www.nvidia.com/object/product_tesla_C2050_C2070_us.html`

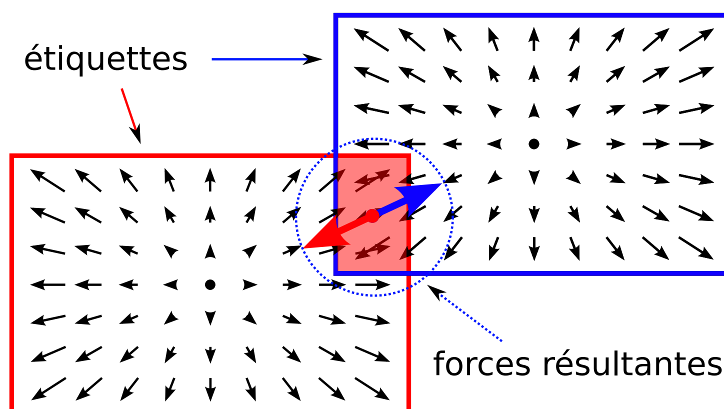

FIGURE 2.2 – Calcul des forces de répulsions entre étiquettes par rastérisation.

codons la composante en x des forces dans les canaux rouge et vert, et la composante en y dans les canaux bleu et alpha. Ces champs de forces sont dessinés les uns après les autres avec une opération de composition additive. Lors d'un rendu normal, la composition sert à réaliser des opérations de transparence. Ici, l'opération locale utilisée (une addition) permet de réaliser la somme vectorielle des différentes forces.

- La deuxième passe consiste ensuite à relire pour chacune des étiquettes la valeur totale du champ de force compris sous elle-même. Étant donné que la valeur totale du champ de force de chacune des étiquettes est nulle, si cette somme est non nulle, cela signifie qu'une autre étiquette est venue se superposer à la courante (zone en rouge sur la figure). On obtient ainsi une force résultante (flèche rouge pour l'étiquette rouge, et bleue pour l'étiquette bleue) qui va permettre de repousser l'étiquette vers une zone moins chargée. À cette force résultante s'ajoute la force de rappel (modélisée par un ressort) vers le point d'amarrage.
- La troisième passe est la phase d'affichage : après la mise à jour des positions des différentes étiquettes, le processeur graphique trace les nouvelles étiquettes.

2.5 Gains

Dans cette section, nous présentons ici les gains théoriques et expérimentaux réalisés par notre algorithme.

2.5.1 Simplicité

Notre algorithme, tout comme l'algorithme à base de masses-ressorts, est simple puisqu'il est réalisé en 3 passes de calculs seulement. Ces deux algorithmes ne se différencient pas dans leur pseudo code. Par contre, dès que l'utilisateur final souhaite des formes quelconques comme zones attractives ou pour les étiquettes elles-mêmes, notre algorithme est plus simple. Il suffit en effet de calculer le champ de force de l'étiquette une fois pour toutes au début de l'exécution. Par contre, pour les algorithmes n'utilisant pas un cache de texture, les calculs d'intersection sont réalisés par le processeur central, ce qui est coûteux

en calculs et compliqué à mettre en œuvre.

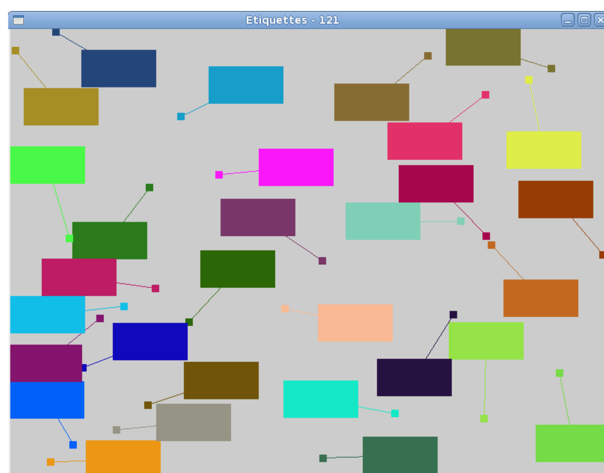

Figure 2.3 – Capture d'écran d'une configuration de 30 étiquettes, sur un écran de 800x600 à 121 images par secondes.

2.5.2 Performances

Tout d'abord, le point déterminant est de savoir si l'algorithme "fonctionne". Nous avons affiché 300 étiquettes sur un écran ayant une résolution de 1600x1050 sur une machine comportant une carte graphique NVIDIA 8800M GTX (la Figure 2.3 présente une réalisation avec moins d'étiquettes). Le taux de rafraîchissement est de 95 images par secondes. Nous voyons ici qu'une carte graphique haut de gamme de 2008 est suffisante pour réaliser une telle implémentation.

Le principe de l'algorithme est d'utiliser la mémoire du processeur graphique qui est disponible en grande quantité et qui est très rapide. Ceci nous permet d'obtenir un algorithme de complexité linéaire (en $\mathcal{O}(n)$, n étant le nombre d'étiquettes). Cette technique utilise la rasterisation comme moyen de calcul [Cohen 93]. L'algorithme à base de masses-ressorts est en $\mathcal{O}(n!)$, ce qui nous permet de dire que notre algorithme est plus efficace en calculs.

2.5.3 Utilisation de la mémoire

Les performances de calculs passent en $\mathcal{O}(n)$, mais ceci se traduit par une utilisation beaucoup plus importante de la mémoire. En effet, la mémoire requise est $\mathcal{O}(cte)$, la constante étant nombre de pixels affichés par l'application. Dans le cas précédent, la mémoire utilisée est de l'ordre du nombre d'étiquettes, en $\mathcal{O}(n)$ donc. Cependant, la constante utilisée par notre algorithme est très grande devant le nombre d'étiquettes (plusieurs millions contre plusieurs centaines).

2.5.4 Le problème est-il toujours NP-complet ?

Nous proposons un algorithme linéaire en fonction du nombre d'étiquettes, mais cela ne remet en rien la complexité NP-complet du problème. En effet, les exigences de production de l'algorithme diffèrent des algorithmes cherchant *la* meilleure solution pour une configuration donnée. Dans notre cas, puisque nous travaillons sur une image interactive, il nous importe peu d'obtenir cette meilleure solution. Tout se passe comme si l'algorithme de recuit simulé affinait en permanence son minimum local, tout en injectant de nouvelles variables depuis l'extérieur. La recherche du minimum est continue tout au long de l'interaction, et l'utilisateur peut même aider la machine en déplaçant manuellement certaines étiquettes.

2.6 Détails techniques

Dans cette section, nous détaillons les différents points qui nous ont semblé être intéressants concernant l'implémentation technique.

2.6.1 Récupérer des données depuis la carte graphique

Le premier point critique de ce travail est que l'on doit travailler avec la carte graphique comme co-processeur. Pour cela, nous avons besoin de récupérer les résultats des différents calculs réalisés par cette dernière. La littérature aborde déjà ce sujet [Harris 05], et nous étudierons ces différentes techniques et leurs avantages et inconvénients vis à vis de notre approche.

Tout d'abord, la première possibilité offerte consiste à faire un rendu dans une texture et ensuite lire cette texture. Cette technique présente l'intérêt d'être facile à mettre en œuvre, puisque c'est l'utilisation normale du pipeline graphique. Cependant nous avons rencontré des problèmes en utilisant cette technique lors de la mise au point des différents programmes, puisqu'elle exige un placement exact des différents fragments de sortie. Si ce placement est fait de manière approximative, il n'y a aucun moyen de vérifier que le processeur graphique réalise correctement ces calculs puisque l'unité de rasterisation, bien qu'elle soit désactivable, introduit un biais dans les valeurs de sortie.

La deuxième approche possible consiste à ne plus utiliser le pipeline graphique pour réaliser ces calculs, mais une autre API de programmation telle que CUDA, OpenCL ou Direct Compute. Ce choix avait été fait pour la première implémentation de l'algorithme, qui avait été entièrement codé en CUDA 1.0. Pour des questions de portabilité de code (OpenCL n'était pas encore disponible) sur différents matériels, nous n'avons pas continué nos travaux dans cette direction.

2.6.2 Shaders employés

Afin de réaliser l'algorithme en version pipeline graphique, nous avons dû utiliser plusieurs shaders (un groupe de shaders par passe de l'algorithme). Le shader principal sur lequel repose l'algorithme est le geometry shader réalisant le calcul du déplacement.

Ce shader est fournit dans le listing 2.1.

Listing 2.1 – Le code du geometry shader permettant de calculer l'offset de chacune des étiquettes

```
#version 150
#extension GL_EXT_geometry_shader4 : enable

uniform uniformData {
  mat4 projection_matrix;
  mat4 modelview_matrix;
  vec2 screenSize;
  vec2 headSize;
  vec2 labelSize;
} Uniforms;

in inputGeomT {
  vec4 trueLabelPosition;
  vec4 trueHeadPosition;
  vec4 head;
  vec4 color;
  vec2 size;
  vec2 headSize;
  vec4 pickingColor;
  vec4 pickingHeadColor;
  float name;
} inputGeom[];

uniform int lenVBO = 100;
uniform sampler2D backbuffer;
uniform float dRessort = 100;

out outputT {
  vec4 color;
  vec4 pickingColor;
} output;

void main(void)
{
  vec2 offset;
  for(int i = 0; i < gl_VerticesIn; ++i)
  {
    float resultX = 1.0;
    float resultY = 1.0;
    ivec2 sizeBackBuffer = textureSize(backbuffer, 0);

    for (int dx = -int(Uniforms.labelSize.x) / 2 + 1; dx <= int(Uniforms.labelSize.x) / 2 - 1; dx++)
    {
      for (int dy = -int(Uniforms.labelSize.y) / 2 + 1; dy <= int(Uniforms.labelSize.y) / 2 - 1; dy++)
      {
        vec2 coord = clamp(inputGeom[i].trueLabelPosition.xy + vec2(dx,dy) + sizeBackBuffer / 2,
              vec2(0,0), sizeBackBuffer - 1);

        coord = coord / sizeBackBuffer;

        vec4 color = textureLod(backbuffer, coord, 0);

        coord = coord / sizeBackBuffer;

        vec4 color = textureLod(backbuffer, coord, 0);

        resultX += color.r;
        resultX -= color.g;
        resultY += color.b;
        resultY -= color.a;
      }
    }

    float d = dRessort - distance(inputGeom[i].trueLabelPosition.xy, inputGeom[i].trueHeadPosition.xy);
    if (d*d < 4){
      d = 0;
    }
    d /= 2.0f;

    vec2 ressort = vec2(1.0, 1.0);

    // which side ?

    ressort += sign(inputGeom[i].trueLabelPosition.xy - inputGeom[i].trueHeadPosition.xy) * d;

    gl_Position = vec4( (2.0 * inputGeom[i].name + 1.0) / (2.0 * lenVBO) * 2.0 - 1.0, -0.5, 0, 1);
    output.color = vec4( resultX, resultY, ressort.x, ressort.y) / 2.0;
    EmitVertex();
    EndPrimitive();
  }
  EndPrimitive();
}
```

2.7 Conclusion

Nous avons présenté ici un algorithme original de placement d'étiquettes qui utilise la rastérisation pour effectuer ces calculs. Notre démarche a consisté à repenser la description de l'algorithme plutôt que de travailler les optimisations. Les intérêts de notre algorithme sont multiples :

- l'algorithme est simple à comprendre, donc à maintenir et à faire évoluer,
- l'algorithme est linéaire en fonction du nombre d'étiquettes, ce qui lui permet de passer facilement à l'échelle en rajoutant de nouveaux points d'intérêts,
- l'algorithme peut être utilisé pour des formes d'étiquettes quelconques : il suffit de calculer la somme totale des forces d'une étiquettes seule, et de la retrancher lors du calcul de la force s'exerçant dessus,
- enfin, les étiquettes peuvent aussi avoir subi des transformations quelconques.

Nous pensons que cet algorithme trouvera sa place dans un logiciel comme FromDaDy, et nous pensons l'incorporer dans une prochaine version. En effet, un tel algorithme permet d'augmenter l'interactivité du placement des étiquettes, en ajoutant par exemple des contraintes manuelles sans pénaliser les performances.

Conclusion générale

Conclusion et perspectives

Le domaine de la conception d'applications graphiques interactives est encore peu instrumenté et pose des problèmes concernant les évolutions des futurs systèmes interactifs. Les travaux que nous avons menés portent tant sur la démarche de conception de ces applications que sur des procédés techniques d'optimisation.

Rappel de la problématique

La première partie de ce manuscrit propose une description des outils et des méthodes mises à la disposition des concepteurs d'applications graphiques interactives. Nous avons décrit les problèmes d'intégration du designer graphique dans le cycle de conception de l'application. Nous en avons conclu que le designer ne pouvait pas réaliser directement des conceptions et qu'il devait pour cela reposer sur un programmeur d'interaction externe. Ceci est en contradiction avec le développement itératif qu'il est nécessaire de mettre en œuvre pour développer une application interactive.

La démarche

Notre démarche concernant la méthode de conception des applications graphiques interactives à consisté à comparer le processus de développement à une chaîne de compilation. Nous avons émit le postulat que ces deux processus étaient proches et qu'il était par conséquent possible de produire un compilateur graphique. L'avantage principal d'utiliser un compilateur graphique est de séparer *description*, *implémentation* et *optimisations*. Nous avons produit une preuve de concept d'un compilateur graphique et l'avons inclus dans une boîte à outils, Hayaku. Nous avons ensuite validé cette idée sur différents scénarios de manière à tester les différents points de cette approche. Lors du développement de ces scénarios, nous avons demandé à un designer graphique habitué à travailler avec des boîtes à outils graphique et à produire du code d'interaction de tester Hayaku. Ces tests nous ont permis d'évaluer notre approche et d'en voir les limites.

La deuxième partie de nos travaux à consisté à appliquer la séparation de la description des optimisations au travers de deux cas d'utilisations. Dans le premier, nous avons cherché à optimiser au maximum l'affichage d'un gros volume de données. Dans le deuxième, nous avons amélioré un algorithme en simplifiant la description de la problématique.

Les apports

Le principal résultat de nos travaux est le fait qu'il est possible, dans une certaine mesure, de séparer *description*, *implémentation* et *optimisations* pour le graphisme interactif. Cette séparation des concepts permet d'obtenir différents effets positifs sur le processus de développement des applications graphiques interactives. Tout d'abord, le designer graphique peut s'impliquer dans ce processus puisqu'il dispose d'outils lui permettant de produire rapidement son application finale. Ainsi, il peut contrôler lui-même la production des interactions. Son métier devient alors celui du *designer d'interactions*. Ensuite, puisque le travail de réalisation des optimisations est réalisé une fois pour toute par des experts au sein du compilateur graphique, le rendu final peut être amélioré par des techniques plus coûteuses exploitant les capacités du moteur de rendu. Ainsi, Hayaku, la boîte à outils que nous avons développé comme preuve de concept présente un rendu quasi-identique à celui réalisé statiquement par les logiciels de dessins des designers. La réalisation de ce compilateur permet aussi une plus grande réutilisation des designs produits. En changeant le module de sortie du compilateur graphique, il est possible de changer le langage mais aussi le moteur de rendu. Porter une application sur différentes plateformes est ainsi possible facilement. Ce principe nous permet aussi d'envisager une meilleur robustesse aux changements technologiques.

Nos travaux sur les optimisations manuelles nous ont amené à produire un logiciel d'exploration de grande quantité de données multidimensionnelles. Nous avons produit un logiciel capable d'afficher et de manipuler plusieurs millions d'informations par un humain. Optimiser ce logiciel a permis aux utilisateurs finaux de pouvoir interagir de manière naturelle avec leur données.

Nous avons aussi repensé un algorithme dont la complexité semblait devenir de plus en plus difficile à gérer. Cet algorithme est le placement des étiquettes sur une image radar. Nous avons montré qu'en changeant la description du problème et en utilisant la mémoire graphique, nous sommes capables de passer d'un algorithme en $\mathcal{O}(n!)$ en calculs à une version en $\mathcal{O}(n)$. En outre notre description du problème est plus simple pour l'intégration d'autres éléments extérieurs impactant la position des étiquettes, et nous permet d'imaginer de nouveaux paradigmes d'interaction pour les utilisateurs finaux, comme par exemple, la définition de zones sans étiquettes avec une interaction de type "*brushing*".

Limites et perspectives

Nos travaux sur la démarche de conception des systèmes graphiques interactifs nous ont montré que notre approche permettait une bonne implication du designer dans la description de l'interaction. Cependant, l'implémentation utilisée, et notamment le langage mis à la disposition de l'utilisateur final est difficilement accessible pour un novice. En effet, il y a un couplage fort entre les fichiers de définition des *classes graphiques*, du *modèle conceptuel*, et des *connexions*. Ce couplage doit être traité "à la main" lors de l'écriture de ces trois fichiers ce qui est fastidieux pour le designer d'interaction. De plus, ces fichiers devant être écrits dans le langage JSON, cette étape est d'autant plus difficile en raison

des messages d'erreurs peu explicite lors de l'analyse lexicale ou syntaxique.

Une solution permettant de contourner ces deux problèmes (couplage fort et langage nouveau pour le designer d'interaction) serait de concevoir un atelier intégré de conception sous la forme d'une application graphique faisant une interface entre le designer d'interaction et la production de ces fichiers.

Un deuxième problème relevé lors de la création des fichiers de *scènes* est lié à l'écriture fastidieuse (car répétitive) des différents objets et de leur placement sur la scène globale. Ce problème a été contourné au cours de ces travaux par l'écriture de scripts générant ces fichiers. Cependant, il est très délicat de demander à un designer d'interaction d'écrire de tels scripts, et une solution telle que présentée précédemment (l'atelier de conception) permettrait de résoudre aussi ce problème. La principale difficulté de cette partie de l'atelier réside entre autre dans la facilité avec laquelle le designer d'interaction sera capable de spécifier la répétition des objets (par exemple les touches du clavier logiciel). Il semble obligatoire de réaliser une étude approfondie des besoins de l'utilisateur final dans ce cas.

Une perspective introduite par la séparation de la description des graphismes vis à vis des règles de production consiste à pouvoir réaliser des preuves formelles sur les transformations. Ceci est très intéressant dans des contextes très exigeants tels ceux des applications embarquées dans les cockpit d'avion.

Un autre point que nous pouvons relever avec notre démarche d'instrumentisation de la conception des systèmes graphiques interactifs est le fait que nous ayons laissé volontairement de côté la spécification du code décrivant l'interaction. En effet, l'interaction est aujourd'hui exprimée au travers de fonctions d'un programme informatique, et cela nécessite donc des compétences non nécessairement mobilisables par un designer d'interaction. Il préfèrera sans doute spécifier le comportement de manière visuelle (soit comme dans Aritistic Resizing [Dragicevic 05], soit comme dans l'outil Flash d'Abobe), et cela vient donc s'ajouter à la liste des exigences de l'atelier de conception.

Finalement, nos expériences d'optimisations de systèmes graphiques interactif nous laissent supposer qu'il reste toujours des optimisations qui devront être réalisées à la main par un programmeur expert. Nous pensons qu'un logiciel comme FromDaDy ou l'implémentation de l'algorithme de placement des étiquettes proposé ne peuvent être réalisés que manuellement. Par contre, certaines parties de l'interface de FromDaDy pourraient être produites par une boîte à outils comme Hayaku. Ceci permettrait aux concepteurs de se focaliser uniquement sur les points critiques du système tout en bénéficiant d'une interface graphiquement riche et facilement modifiable. De même, on peut imaginer une généralisation de notre algorithme de placement des étiquettes qui serait implémentée dans Hayaku et qui permette de spécifier des contraintes de non-recouvrement entre des items graphiques.

Annexe A

Détails de l'application multitouch réalisée avec Hayaku

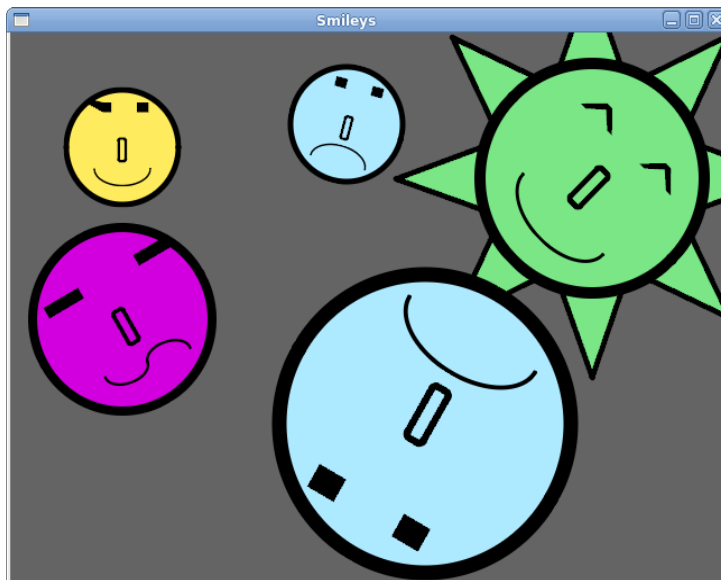

FIGURE A.1 – Une application multi-touch simple.

Dans cette annexe, nous fournissons l'ensemble des fichiers permettant de faire tourner notre application multitouch dont le visuel est présenté Figure A.1.

A.1 Les classes graphiques

Tout d'abord, le designer d'interaction produit les classes graphiques de la Figure A.2. Chacun des groupes est nommé grâce au champ *id* de SVG.

Listing A.2 – le source du fichier demo.svg.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- Created with Inkscape (http://www.inkscape.org/) -->

<svg
```

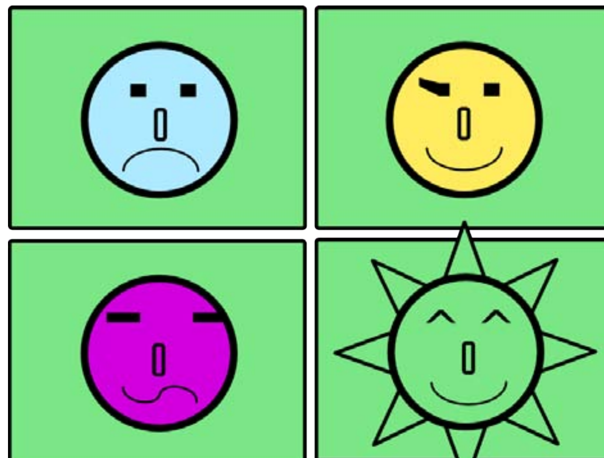

FIGURE A.2 – Les classes graphiques de l'application multitouch.

```
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:cc="http://creativecommons.org/ns#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:svg="http://www.w3.org/2000/svg"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:sodipodi="http://sodipodi.sourceforge.net/DTD/sodipodi-0.dtd"
    xmlns:inkscape="http://www.inkscape.org/namespaces/inkscape"
    width="810"
    height="610.00012"
    id="svg2"
    sodipodi:version="0.32"
    inkscape:version="0.47␣r22583"
    sodipodi:docname="demo.svg"
    inkscape:output_extension="org.inkscape.output.svg.inkscape"
    version="1.0">
  <g inkscape:label="Calque␣1"
     inkscape:groupmode="layer"
     id="layer1"
     transform="translate(231.91373,168.79994)">
    <g id="g3244"
       transform="translate(380.58627,-16.299943)">
      <rect ry="3" rx="3" y="-150" x="-200" height="295" width="395" id="rect3242"
         style="fill:#7ae685;fill-opacity:1;stroke:#000000;stroke-width:5;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1" />
      <g id="happy"
         transform="translate(-2.1213203,0)">
        <path sodipodi:type="arc"
           style="fill:#ffeb5e;fill-opacity:1;stroke:#000000;stroke-width:10;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
           id="path2419"
           sodipodi:cx="0" sodipodi:cy="0"
           sodipodi:rx="100" sodipodi:ry="100"
           d="M␣100,0␣A␣100,100␣0␣1␣1␣-100,1.2246064e-14␣A␣100,100␣0␣1␣1␣100,-2.4492127e-14␣z" />
        <g transform="matrix(0,1,-1,0,0,0)"
           id="text3191">
          <path d="m␣-49.547943,-47.228588␣17.514648,0␣0,21.083984␣-17.514648,0␣0,-21.083984␣m
0,66.821289␣17.514648,0␣0,14.277344␣-13.613281,26.5625␣-10.708008,0␣6.806641,-26.5625␣0,-14.277344"
             id="path2859"
             style="fill:#000000;fill-opacity:1;stroke:none;stroke-opacity:1" />
        </g>
        <rect style="fill:none;fill-opacity:1;stroke:#000000;stroke-width:5;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
           id="rect3195"
           width="12.374369" height="39.59798"
           x="-6.5996556" y="-14.999011"
           rx="3" ry="3" />
        <path sodipodi:type="arc"
           style="fill:none;fill-opacity:1;stroke:#000000;stroke-width:3;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
           id="path3197"
           sodipodi:cx="-17.147339" sodipodi:cy="90.890663"
           sodipodi:rx="50.027805" sodipodi:ry="29.698484"
           d="M␣32.880466,90.890663␣A␣50.027805,29.698484␣0␣0␣1␣-67.155763,91.717248"
           transform="translate(17.088421,-53.563769)"
           sodipodi:start="0" sodipodi:end="3.1137565"
           sodipodi:open="true" />
      </g>
    </g>
    <g id="g3280"
       transform="translate(-29.413732,-16.299943)">
```

```
      <rect style="fill:#7ae685;fill-opacity:1;stroke:#000000;stroke-width:5;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
         id="rect3258"
         width="395" height="295"
         x="-200" y="-150" rx="3" ry="3" />
      <g id="sad">
        <path d="M␣100,0␣A␣100,100␣0␣1␣1␣-100,1.2246064e-14␣A␣100,100␣0␣1␣1␣100,-2.4492127e-14␣z"
           sodipodi:ry="100" sodipodi:rx="100"
           sodipodi:cy="0" sodipodi:cx="0"
           id="path3208"
           style="fill:#aeeaff;fill-opacity:1;stroke:#000000;stroke-width:10;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
           sodipodi:type="arc" />
        <g transform="matrix(0,1,-1,0,0,0)"
           id="text3210">
          <path d="m␣-48.275871,21.663769␣17.514648,0␣0,21.083984␣-17.514648,0␣0,-21.083984␣m
0,-66.821289␣17.514648,0␣0,21.083984␣-17.514648,0␣0,-21.083984"
             id="path2867"
             style="stroke:none;fill:#000000;fill-opacity:1" />
        </g>
        <rect ry="3" rx="3" y="-13.726933" x="-4.6707234"
           height="39.59798" width="12.374369"
           id="rect3214"
           style="fill:none;fill-opacity:1;stroke:#000000;stroke-width:5;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1" />
        <path sodipodi:open="true"
           sodipodi:end="3.1137565" sodipodi:start="0"
           transform="matrix(1,0,0,-1,19.017353,159.18812)"
           d="M␣32.880466,90.890663␣A␣50.027805,29.698484␣0␣0␣1␣-67.155763,91.717248"
           sodipodi:ry="29.698484" sodipodi:rx="50.027805"
           sodipodi:cy="90.890663" sodipodi:cx="-17.147339"
           id="path3216"
           style="fill:none;fill-opacity:1;stroke:#000000;stroke-width:3;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
           sodipodi:type="arc" />
      </g>
    </g>
    <g id="g3271"
       transform="translate(-29.742159,295.55802)">
      <rect ry="3" rx="3" y="-150" x="-200"
         height="295" width="395"
         id="rect3260"
         style="fill:#7ae685;fill-opacity:1;stroke:#000000;stroke-width:5;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1" />
      <g id="question">
        <path sodipodi:type="arc"
           style="fill:#d200df;fill-opacity:1;stroke:#000000;stroke-width:10;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
           id="path3220"
           sodipodi:cx="0" sodipodi:cy="0"
           sodipodi:rx="100" sodipodi:ry="100"
           d="M␣100,0␣A␣100,100␣0␣1␣1␣-100,1.2246064e-14␣A␣100,100␣0␣1␣1␣100,-2.4492127e-14␣z" />
        <g id="text3222">
          <path d="m␣-82.169037,-54.697338␣44.741211,0␣0,13.613281␣-44.741211,0␣0,-13.613281"
             id="path2862"
             style="fill:#000000;fill-opacity:1;stroke:none" />
          <path d="m␣33.377838,-54.697338␣44.741211,0␣0,13.613281␣-44.741211,0␣0,-13.613281"
             id="path2864"
             style="fill:#000000;fill-opacity:1;stroke:none" />
        </g>
        <rect style="fill:none;fill-opacity:1;stroke:#000000;stroke-width:5;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
           id="rect3226"
           width="12.374369" height="39.59798"
           x="-6.5996556" y="-10.999011" rx="3" ry="3" />
        <g id="g3361">
          <path sodipodi:type="arc"
             style="fill:none;fill-opacity:1;stroke:#000000;stroke-width:3;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
             id="path3357"
             sodipodi:cx="-17.147339" sodipodi:cy="90.890663"
             sodipodi:rx="25" sodipodi:ry="15"
             d="M␣7.8526611,90.890663␣A␣25,15␣0␣0␣1␣-42.137654,91.308152"
             transform="matrix(0.9800085,0.1989558,-0.1989558,0.9800085,11.341165,-35.933771)"
             sodipodi:start="0" sodipodi:end="3.1137565"
             sodipodi:open="true" />
          <path sodipodi:open="true"
             sodipodi:end="3.1137565" sodipodi:start="0"
             transform="matrix(0.9800085,0.1989558,0.1989558,-0.9800085,24.150267,152.28376)"
             d="M␣7.8526611,90.890663␣A␣25,15␣0␣0␣1␣-42.137654,91.308152"
             sodipodi:ry="15" sodipodi:rx="25"
             sodipodi:cy="90.890663" sodipodi:cx="-17.147339"
             id="path3359"
             style="fill:none;fill-opacity:1;stroke:#000000;stroke-width:3;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
             sodipodi:type="arc" />
        </g>
      </g>
    </g>
    <g id="g3262"
       transform="translate(380.58627,293.70016)">
      <rect ry="3" rx="3" y="-150" x="-200"
```

```
          height="295" width="395"
          id="rect3255"
          style="fill:#7ae685;fill-opacity:1;stroke:#000000;stroke-width:5;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1" />
      <g id="sun">
        <path sodipodi:type="star"
           style="fill:#7ae685;fill-opacity:1;stroke:#000000;stroke-width:5;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
           id="path3365"
           sodipodi:sides="8"
           sodipodi:cx="611.6474" sodipodi:cy="455.85083"
           sodipodi:r1="174.00287" sodipodi:r2="87.001427"
           sodipodi:arg1="0.7796511" sodipodi:arg2="1.1723502"
           inkscape:flatsided="false" inkscape:rounded="0" inkscape:randomized="0"
           d="M 735.39108,578.1803 L 645.40279,536.037 L 612.6474,629.85083 L 578.81588,536.41968 L
489.31793,579.59451 L 531.46123,489.60622 L 437.6474,456.85083 L 531.07855,423.01931 L
487.90372,333.52136 L 577.89201,375.66466 L 610.6474,281.85083 L 644.47892,375.28198 L
733.97687,332.10715 L 691.83357,422.09544 L 785.6474,454.85083 L 692.21625,488.68235 L
735.39108,578.1803 z"
           transform="translate(-612.5,-455.42903)" />
        <path d="M 100,0 A 100,100 0 1 1 -100,1.2246064e-14 A 100,100 0 1 1 100,-2.4492127e-14 z"
           sodipodi:ry="100" sodipodi:rx="100"
           sodipodi:cy="0" sodipodi:cx="0"
           id="path3232"
           style="fill:#7ae685;fill-opacity:1;stroke:#000000;stroke-width:10;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
           sodipodi:type="arc" />
        <g id="text3234">
          <path d="m -30.349972,-55.979565 16.96875,17.40625 -6.28125,0 -13.75,-12.34375 -13.75,12.34375
-6.28125,0 16.96875,-17.40625 6.125,0"
             style="font-size:64px;fill:#000000;fill-opacity:1;stroke:none"
             id="path2870" />
          <path d="m 43.650028,-55.979565 16.96875,17.40625 -6.28125,0 -13.75,-12.34375 -13.75,12.34375
-6.28125,0 16.96875,-17.40625 6.125,0"
             style="font-size:64px;fill:#000000;fill-opacity:1;stroke:none"
             id="path2872" />
        </g>
        <rect ry="3" rx="3" y="-11.999011" x="-2.5996556"
           height="39.59798" width="12.374369"
           id="rect3238"
           style="fill:none;fill-opacity:1;stroke:#000000;stroke-width:5;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1" />
        <path sodipodi:open="true"
           sodipodi:end="3.1137565" sodipodi:start="0"
           transform="translate(21.088421,-50.563769)"
           d="M 32.880466,90.890663 A 50.027805,29.698484 0 0 1 -67.155763,91.717248"
           sodipodi:ry="29.698484" sodipodi:rx="50.027805"
           sodipodi:cy="90.890663" sodipodi:cx="-17.147339"
           id="path3240"
           style="fill:none;fill-opacity:1;stroke:#000000;stroke-width:3;stroke-linecap:round;
stroke-linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
           sodipodi:type="arc" />
      </g>
    </g>
  </g>
</svg>
```

A.2 Le modèle conceptuel

Le modèle conceptuel est présenté dans le listing A.3. Les quatres objets *Object_0* à *Object_3* héritent tous de la classe *PastilleCommune*.

Listing A.3 – Implémentation des items multi-touch à l'aide de la syntaxe JSON (fichier smileys_model.json).

```
{
    "model": "SMILEYS",
    "classes": [ {
        "name":"PastilleCommune",
        "extends":null,
        "attributes": {
            "X0":"vint",
            "Y0":"vint",
            "SCALE":"vfloat",
            "ROTATION":"vfloat",
            "Z-ORDER":"vfloat",
            "Picking_Key":"vint"
        }
    },{
        "name":"Object_0",
        "extends":"PastilleCommune",
        "attributes": {}
```

```
    },{
        "name":"Object_1",
        "extends":"PastilleCommune",
        "attributes": {
        }
    },{
        "name":"Object_2",
        "extends":"PastilleCommune",
        "attributes": {
        }
    },{
        "name":"Object_3",
        "extends":"PastilleCommune",
        "attributes": {
        }
    }]
}
```

A.3 Les connections entre le modèle et les classes graphiques

Voici le listing contenant les connexions entre les classes graphiques et les attributs du modèle.

Listing A.4 – Fichier de connexion des items multi-touch à l'aide de la syntaxe JSON (fichier smileys_modelToSVG.json).

```
{
    "model":"SMILEYS",
    "objects": [
    {
        "className":"Object_0",
        "file":"demo.svg",
        "graphicalItems": [
        {
            "name":"happy",
            "connections":
            {
                "X0":"happy.transform.tx",
                "Y0":"happy.transform.ty",
                "SCALE":"happy.transform.scale",
                "ROTATION":"happy.transform.rotation",
                "PRIORITY":"happy.transform.priority"
            },
            "picking":
            {
                "Picked_Key":"happy"
            }
        }]
    },
    {
        "className":"Object_1",
        "file":"demo.svg",
        "graphicalItems": [
        {
            "name":"sad",
            "connections":
            {
                "X0":"sad.transform.tx",
                "Y0":"sad.transform.ty",
                "SCALE":"sad.transform.scale",
                "ROTATION":"sad.transform.rotation",
                "PRIORITY":"sad.transform.priority"
            },
            "picking":
            {
                "Picked_Key":"sad"
            }
        }]
    },{
        "className":"Object_2",
        "file":"demo.svg",
        "graphicalItems": [
        {
            "name":"sun",
            "connections":
            {
                "X0":"sun.transform.tx",
                "Y0":"sun.transform.ty",
                "SCALE":"sun.transform.scale",
```

```
                "ROTATION":"sun.transform.rotation",
                "PRIORITY":"sun.transform.priority"
            },
            "picking":
            {
                "Picked_Key":"sun"
            }
        }]
    },{
        "className":"Object_3",
        "file":"demo.svg",
        "graphicalItems": [
        {
            "name":"question",
            "connections":
            {
                "X0":"question.transform.tx",
                "Y0":"question.transform.ty",
                "SCALE":"question.transform.scale",
                "ROTATION":"question.transform.rotation",
                "PRIORITY":"question.transform.priority"
            },
            "picking":
            {
                "Picked_Key":"question"
            }
        }]
    }]
}
```

A.4 La scène

Enfin, l'instanciation des différents objets utilisés dans l'application interactive finale est réalisée par l'intermédiaire du fichier de scène.

Listing A.5 – Le fichier de *scène* des items multi-touch à l'aide de la syntaxe JSON (fichier smileys.json).

```
{
    "name":"Stantum",
    "model":"SMILEYS",
    "content": [
    {
        "type":"Object_0",
        "attributes": {
            "ID":0,
            "ParentID":0,
            "X0":100,
            "Y0":100,
            "SCALE":0.5,
            "ROTATION":0.0,
            "Picked_Key":-1
        }
    },{
        "type":"Object_1",
        "attributes": {
            "ID":1,
            "ParentID":0,
            "X0":300,
            "Y0":80,
            "SCALE":0.5,
            "ROTATION":15.0,
            "Picked_Key":-1
        }
    },{
        "type":"Object_2",
        "attributes": {
            "ID":2,
            "ParentID":0,
            "X0":500,
            "Y0":300,
            "SCALE":1.0,
            "ROTATION":45.0,
            "Picked_Key":-1
        }
    },{
        "type":"Object_3",
        "attributes": {
            "ID":3,
            "ParentID":0,
            "X0":100,
            "Y0":250,
```

```
            "SCALE":0.8,
            "ROTATION":-30.0,
            "Picked_Key":-1
    },{
        "type":"Object_1",
        "attributes": {
            "ID":4,
            "ParentID":0,
            "X0":350,
            "Y0":40,
            "SCALE":0.3,
            "ROTATION":-15.0,
            "Picked_Key":-1
        }
    }]
}
```

A.5 Le lanceur

Voici le code complet de l'application Python gérant les interactions. La récupération des évènements multitouch est réalisé au travers du bus logiciel Ivy. Le traitement de ces évènements est réalisé dans la méthode *stantumEvent* de la classe *Demo_Object*.

Le lancement du compilateur graphique est effectué par l'appel à *gc.load* dans le *main* du fichier (à la fin du listing).

Listing A.6 – Le code python permettant de contrôler les objets graphiques.

```
#! /usr/bin/python
# -*- coding: utf-8 -*-
## Projet STANTUM
## mainfile stantum.py

import sys,os
sys.path.append(os.path.abspath(os.path.join(os.getcwd(),os.pardir,os.pardir)))
import re
from graphical_compiler import *
from math import atan, degrees, sqrt
from ivy_class import *
from get_opt import getparams

stantum = None

def getAngle(self,x0,y0,x1,y1):
    if x0 == x1:
        return 90.0
    return degrees(atan(float(y1-y0)/float(x1-x0)))

class Stantum(ivy):
    def __init__(self, bus):
        self.setObjects([])
        ivy.__init__(self,'GLVM_STANTUM',bus)
        IvyBindMsg(self.onCursorMove,
          "^DEVICE_EVENT id=(.*) SERIAL=(.*) ABS_X=(.*) ABS_Y=(.*) TIME=(.*)")

    def setObjects(self,objects):
        self.objects = objects
        self.pickingVar = {-1:None}
        for o in objects:
            self.pickingVar[o.picked.id] = o

    def onCursorMove(self,*larg):
        id,serial,x,y,time = larg
        id = int(id)
        serial = int(serial)
        x = int(float(x)*WIDTH)
        y = int(float(y)*HEIGHT)
        time = float(time)
        picked = glvm.pick(x,y,serial)
        object = self.pickingVar[picked]
        print object, serial,x,y
        glvm.postRedisplay()

    def mouseCallback(self,who, pressed, idMouse, x, y):
        for o in self.objects:
            o.mouseCallback(who, pressed, idMouse, x, y)

    def mouseMotionCallback(self, idMouse,x,y):
        for o in self.objects:
            o.mouseMotionCallback(idMouse, x, y)
```

```
class Demo_Object(object):
    def __init__(self, id):
        self.GraphicObject = getObject(id)
        self.scale = self.GraphicObject.SCALE
        self.x0 = self.GraphicObject.X0
        self.y0 = self.GraphicObject.Y0
        self.rotation = self.GraphicObject.ROTATION
        self.priority = self.GraphicObject.PRIORITY
        self.picked = self.GraphicObject.Picked_Key
        self.picked_key = False
        self.state = None
        self.old_x, self.old_y = None, None
        self.cursors = {}
    def picking_callback(self, id):
        value_picking = glvm.evalVar(id)
        self.picked_key = value_picking > -1
    def mouseCallback(self, who, pressed, idMouse, x, y):
        if self.state == None and self.picked_key  and who == 0 and pressed:
            self.state = 'move'
            self.old_x = x
            self.old_y = y
        elif self.state == 'move' and not (who == 0 and pressed) :
            self.state = None

        elif self.state == None and self.picked_key  and who == 2 and pressed:
            self.state = 'rotate'
            self.old_x = x
            self.old_y = y
        elif self.state == 'rotate' and not (who == 2 and pressed) :
            self.state = None

        if who == 3 and self.picked_key:
            self.zoom(0.1)
        elif who == 4 and self.picked_key:
            self.zoom(-0.1)

    def translate(self,dx,dy):
        self.x0.set(self.x0.eval() + dx)
        self.y0.set(self.y0.eval() + dy)

    def rotate(self, dr):
        self.rotation.set(self.rotation.eval() + dr)

    def zoom(self, z):
        if self.scale.eval() + z >= 0.1:
            self.scale.set(self.scale.eval() + z)

    def mouseMotionCallback(self, idMouse,x,y):
        if self.state == 'move':
            dx = x - self.old_x
            dy = y - self.old_y
            self.old_x, self.old_y = x, y
            self.translate(dx,dy)
        elif self.state == 'rotate':
            dy = y - self.old_y
            self.old_y = y
            self.rotate(dy)

    def stantumEvent(self,who,idMouse,x,y, picked, time):
        for cursor, (cx,cy,ct) in self.cursors.items():
            if time - ct > 0.05:
                self.cursors.pop(cursor)
        if idMouse not in self.cursors.keys() :
            if picked != self :
                return
            self.cursors[idMouse] = (x,y,time)
        oldx,oldy,oldtime = self.cursors[idMouse]
        self.cursors[idMouse] = (x,y,time)

        dx = x - oldx
        dy = y - oldy
        if len(self.cursors.keys()) == 1:
            self.translate(dx,dy)
        elif len(self.cursors.keys()) == 2:
            otherCursors = self.cursors.keys()
            otherCursors.remove(idMouse)
            otherCursor = otherCursors[0]
            cx0,cy0, ct0 = self.cursors[otherCursor]
            d = sqrt((x-cx0)**2 + (y-cy0)**2)
            oldD = sqrt((oldx-cx0)**2 + (oldy-cy0)**2)
            deltaZoom = ( d - oldD ) / 100
            angle = self.getAngle(x,y,cx0,cy0)
            oldAngle = self.getAngle(oldx,oldy,cx0,cy0)
            deltaRotation = angle - oldAngle
            if abs(deltaRotation) > 90.0:
                deltaRotation -= 180.0
            self.zoom(deltaZoom)
            self.translate(dx/2,dy/2)
            self.rotate(deltaRotation)

def Demo_Object_mouseButton(id, idMouse = 0):
    pressed = glvm.evalVar(id)
```

```
    who = mouse().getButton(id)
    x = glvm.evalVar(mouse().x.id)
    y = glvm.evalVar(mouse().y.id)
    stantum.mouseCallback(who,pressed,idMouse,x,y)

def Demo_Object_mouseMotion(id, idMouse = 0):
    x = glvm.evalVar(mouse().x.id)
    y = glvm.evalVar(mouse().y.id)
    stantum.mouseMotionCallback(idMouse,x,y)

def addObject(keyclass, o, objects, connections):
    s = keyclass(o.ID)
    objects.append(s)
    connections.append((s.GraphicObject.Picked_Key, s.picking_callback))

## fonction parseScene
## Construction de la scene Stantum
def parseScene(parent,scene,connections):
    for b in mouse().buttons:
        connections.append((b, Demo_Object_mouseButton))
    connections.append((mouse().x, Demo_Object_mouseMotion))
    connections.append((mouse().y, Demo_Object_mouseMotion))
    objects = []
    for o in scene:
        if o.getClass() in ['Object_0','Object_1','Object_2','Object_3']:
            addObject(Demo_Object, o, objects, connections)
    stantum.setObjects(objects)
    return connections

def onScene(scene):
    print "onScene callback"
    Connects = []
    Connects = parseScene(None,scene,Connects)
    gc.connect(Connects)

## main
if __name__ == "__main__":
    import sys
    import graphical_compiler.gc as gc
    global stantum
    ivybus = getparams()
    stantum = Stantum(ivybus)
    gc.load("stantum_model.json", "stantum_modelToSVG.json", "stantum.json", onScene,
                                width = 640, height = 480, backColor = (100/255.0,100/255.0,100/255.0))
    glvm.wait()
```

A.6 Traces d'exécution

Enfin, voici les traces d'exécution du programme. Nos commentaires son indiqués entre crochets (<>).

Listing A.7 – Traces d'exécution.

```
tissoire@enday:~/projets/hayaku_stat/exemples/stantum# ../../glvm-bin stantum.py
< initialisation du bus logiciel Ivy >
Broadcasting on network 127.0.0.1, port 2010
Ivy will broadcast on 127:2010
GLVM.STANTUM doing IvyMainLoop
GLVM.STANTUM: An Ivy application was connected
GLVM.STANTUM: currents Ivy application are [['fr.cena.glvm.renderer']]

< vérification des modifications depuis le dernier lancement >
need to recompile the model: True
need to recompile the modelToSVG: True
need to recompile the scene: True

< construction de l'arbre SVG à partir de la scène >
tosvg

< compilation d'outils communs >
gcc -g -fPIC -c -W BUILD/common.c -o BUILD/common.o
gcc -g -fPIC -c -W BUILD/GradientShader_gl.c -o BUILD/GradientShader_gl.o
gcc -g -fPIC -c -W BUILD/ShaderEllipse_gl.c -o BUILD/ShaderEllipse_gl.o
gcc -g -fPIC -c -W BUILD/ShaderQuads_gl.c -o BUILD/ShaderQuads_gl.o

< transformations de l'arbre source en code C >
tosvgmin
togl

< compilation des polices de caractères vectorielles >
gcc -g -fPIC -c -W BUILD/DejaVuSans_gl.c -o BUILD/DejaVuSans_gl.o

< compilation du flot de données gérant les dépendances >
```

```
gcc -fPIC -c BUILD/Stantum_df.c -o BUILD/Stantum_df.o
gcc -shared -o BUILD/Stantum_df.so BUILD/Stantum_df.o

< compilation des fonctions de dessin OpenGL >
gcc -g -fPIC -c -W  BUILD/Stantum_gl.c -o BUILD/Stantum_gl.o

< production d'une bibliothèque dynamique >
gcc -g -shared -o BUILD/Stantum_gl.so \
    BUILD/common.o BUILD/GradientShader_gl.o \
    BUILD/ShaderEllipse_gl.o BUILD/ShaderQuads_gl.o \
    BUILD/DejaVuSans_gl.o BUILD/Stantum_gl.o

< import de la bibliothèque précédemment produite dans l'affichage >
setting up glvm
Status: Using GLEW 1.5.4
nombre de samples : 4

< appel des fonctions analysant le graphe de scène pour la partie interaction >
onScene callback
```

Annexe B

Détails de l'application clavier logiciel réalisée avec Hayaku

Figure B.1 – L'application test clavier en mode "expanding keyboard".

Dans cette annexe, nous fournissons l'ensemble des fichiers permettant de faire tourner notre clavier logiciel dont le visuel est présenté Figure B.1

B.1 Les classes graphiques

Tout d'abord, le designer d'interaction produit les classes graphiques de la Figure B.2. Chacun des groupes est nommé grâce au champ *id* de SVG.

Listing B.8 – le source du fichier keyboard.svg.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- Created with Inkscape (http://www.inkscape.org/) -->
<svg
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:cc="http://creativecommons.org/ns#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:svg="http://www.w3.org/2000/svg"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:sodipodi="http://sodipodi.sourceforge.net/DTD/sodipodi-0.dtd"
    xmlns:inkscape="http://www.inkscape.org/namespaces/inkscape"
    width="210mm" height="297mm" id="svg2"
```

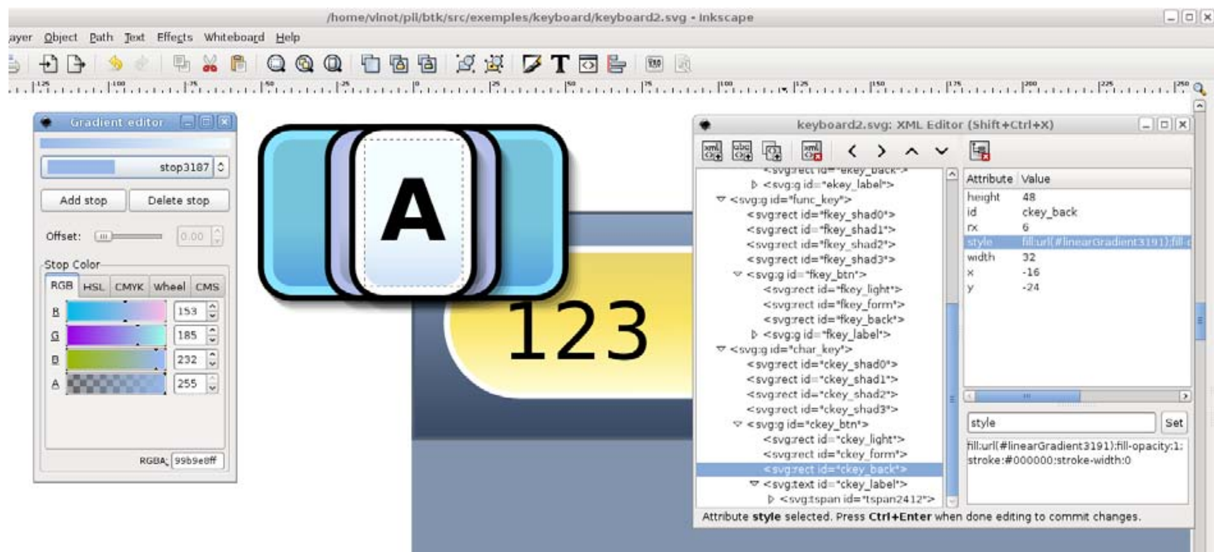

FIGURE B.2 – Le fichier SVG de description des éléments graphiques du clavier, réalisé avec Inkscape.

```
    sodipodi:version="0.32"
    inkscape:version="0.46"
    sodipodi:docname="keyboard2.svg"
    inkscape:output_extension="org.inkscape.output.svg.inkscape">
  <defs id="defs4">
    <linearGradient id="linearGradient3228">
      <stop style="stop-color:#2776e8;stop-opacity:1;"
          offset="0" id="stop3230" />
      <stop style="stop-color:#77c5db;stop-opacity:1;"
          offset="1" id="stop3232" />
    </linearGradient>
    <linearGradient
        id="linearGradient3204">
      <stop id="stop3206"
          offset="0" style="stop-color:#475a74;stop-opacity:1;" />
      <stop id="stop3208"
          offset="1" style="stop-color:#bcc4e8;stop-opacity:1;" />
    </linearGradient>
    <linearGradient id="linearGradient3277">
      <stop style="stop-color:#000000;stop-opacity:1;"
          offset="0" id="stop3279" />
      <stop style="stop-color:#000000;stop-opacity:0.20168068;"
          offset="1" id="stop3281" />
    </linearGradient>
    <linearGradient id="linearGradient3211">
      <stop style="stop-color:#3b4d68;stop-opacity:1;"
          offset="0" id="stop3213" />
      <stop style="stop-color:#8fa5c3;stop-opacity:1;"
          offset="1" id="stop3215" />
    </linearGradient>
    <linearGradient id="linearGradient3201">
      <stop style="stop-color:#ffffda;stop-opacity:1"
          offset="0" id="stop3203" />
      <stop id="stop3209" offset="0.51047271"
          style="stop-color:#fbe565;stop-opacity:1" />
      <stop style="stop-color:#fbe361;stop-opacity:0.83193278;"
          offset="1" id="stop3205" />
    </linearGradient>
    <linearGradient id="linearGradient3185">
      <stop style="stop-color:#99b9e8;stop-opacity:1;"
          offset="0" id="stop3187" />
      <stop style="stop-color:#f5ffff;stop-opacity:1;"
          offset="1" id="stop3189" />
    </linearGradient>
    <inkscape:perspective
        sodipodi:type="inkscape:persp3d"
        inkscape:vp_x="0 : 526.18109 : 1"
        inkscape:vp_y="0 : 1000 : 0"
        inkscape:vp_z="744.09448 : 526.18109 : 1"
        inkscape:persp3d-origin="372.04724 : 350.78739 : 1"
        id="perspective10" />
    <linearGradient inkscape:collect="always"
        xlink:href="#linearGradient3185" id="linearGradient3191"
```

```
         x1="20" y1="52" x2="20" y2="4"
         gradientUnits="userSpaceOnUse" />
      <linearGradient inkscape:collect="always"
         xlink:href="#linearGradient3204" id="linearGradient3199"
         x1="28" y1="52" x2="28" y2="4"
         gradientUnits="userSpaceOnUse" />
      <linearGradient inkscape:collect="always"
         xlink:href="#linearGradient3201" id="linearGradient3207"
         x1="255.18394" y1="59.5" x2="255.04308" y2="10.5"
         gradientUnits="userSpaceOnUse" />
      <linearGradient inkscape:collect="always"
         xlink:href="#linearGradient3211" id="linearGradient3217"
         x1="287.81833" y1="70.5" x2="287.87897" y2="-0.5"
         gradientUnits="userSpaceOnUse" />
      <linearGradient inkscape:collect="always"
         xlink:href="#linearGradient3228" id="linearGradient3234"
         gradientUnits="userSpaceOnUse"
         x1="28" y1="52" x2="28" y2="4" />
   </defs>
   <sodipodi:namedview
      id="base"
      pagecolor="#ffffff"
      bordercolor="#666666"
      borderopacity="1.0"
      inkscape:pageopacity="0.0"
      inkscape:pageshadow="2"
      inkscape:zoom="1.9560803"
      inkscape:cx="22.111433"
      inkscape:cy="1049.2862"
      inkscape:document-units="px"
      inkscape:current-layer="svg2"
      showgrid="false"
      inkscape:window-width="1016"
      inkscape:window-height="719"
      inkscape:window-x="833"
      inkscape:window-y="203" />
   <metadata id="metadata7">
      <dc:format>image/svg+xml</dc:format>
      <dc:type rdf:resource="http://purl.org/dc/dcmitype/StillImage" />
      <rdf:RDF>
         <cc:Work rdf:about="">
            <dc:format>image/svg+xml</dc:format>
            <dc:type rdf:resource="http://purl.org/dc/dcmitype/StillImage" />
         </cc:Work>
      </rdf:RDF>
      <g inkscape:label="Calque␣1"
         inkscape:groupmode="layer"
         id="layer1">
         <rect ry="8" rx="8" y="1" x="1"
            height="58" width="242"
            id="skey_shadow"
            style="opacity:0.2;fill:#000000;stroke:#000000;stroke-width:0" />
         <cc:Work rdf:about="" />
         <g id="key_normal"
            x="0" y="0" />
      </g>
   </metadata>
   <g id="keyb_back"
      x="0" y="0">
      <rect ry="0" rx="0" y="70" x="0"
         height="340" width="640"
         id="keyb_form"
         style="fill:#8294ad;stroke:#8294ad;stroke-width:1;stroke-miterlimit:4;
stroke-dasharray:none;stroke-opacity:1" />
   </g>
   <gid="keyb_display"
      x="0" y="0">
      <rect y="0" x="0" height="74" width="640"
         id="display_back"
         style="fill:url(#linearGradient3217);fill-opacity:1;stroke:#525d6d;stroke-width:1;
stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1" />
      <rect style="fill:url(#linearGradient3207);fill-opacity:1;stroke:#ffffff;stroke-width:2;
stroke-miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
         id="display_screen"
         width="500" height="48"
         x="11" y="12" rx="24" ry="24" />
      <text xml:space="preserve"
         style="font-size:26px;font-style:normal;font-weight:normal;text-align:left;fill:#000000;
fill-opacity:1;stroke:none;stroke-width:1px;stroke-linecap:butt;stroke-linejoin:miter;
stroke-opacity:1;font-family:Bitstream␣Vera␣Sans"
         x="29" y="48"
         id="display_txt"><tspan
            sodipodi:role="line"
            id="tspan2475" x="29" y="46">123</tspan></text>
   </g>
   <g y="0" x="0" id="enter_key">
      <rect rx="10" y="-23" x="-47"
         height="56" width="102"
         id="ekey_shad0"
         style="fill:#000000;fill-opacity:0.05;stroke:#000000;stroke-width:0" />
      <rect style="fill:#000000;fill-opacity:0.1;stroke:#000000;stroke-width:0"
         id="ekey_shad1"
```

```
        width="102" height="56" x="-48" y="-24" rx="10" />
    <rect rx="10" y="-25" x="-49" height="56" width="102"
        id="ekey_shad2"
        style="fill:#000000;fill-opacity:0.15;stroke:#000000;stroke-width:0" />
    <rect style="fill:#000000;fill-opacity:0.2;stroke:#000000;stroke-width:0"
        id="ekey_shad3"
        width="102" height="56" x="-50" y="-26" rx="10" />
    <g y="0" x="0" id="ekey_btn">
      <rect rx="14" y="-32" x="-54"  height="64" width="108"
          id="ekey_light"
          style="fill:#ffff5b;fill-opacity:0;stroke:#000000;stroke-width:0" />
      <rect rx="10" y="-28" x="-50" height="56" width="100"
          id="ekey_form"
          style="fill:#8ae4ff;fill-opacity:1;stroke:#000000;stroke-width:2" />
      <rect rx="6" y="-24" x="-46" height="48" width="92"
          id="ekey_back"
          style="fill:url(#linearGradient3234);fill-opacity:1;stroke:#000000;stroke-width:0" />
      <g id="ekey_label"
          x="0" y="0">
        <text id="ekey_txt"
            y="9" x="0"
            style="font-size:28px;font-style:italic;font-weight:bold;line-height:125%;
fill:#ffffff;fill-opacity:1;stroke:none;stroke-width:1px;stroke-linecap:butt;stroke-linejoin:miter;
stroke-opacity:1;font-family:Bitstream␣Vera␣Sans"
            xml:space="preserve"
            sodipodi:linespacing="125%"><tspan y="6" x="0" id="tspan3226" sodipodi:role="line"
              style="text-align:center;line-height:125%;writing-mode:lr-tb;text-anchor:middle">
              GO</tspan></text>
      </g>
    </g>
  </g>
  <g id="func_key" x="0" y="0">
    <rect style="fill:#000000;fill-opacity:0.05;stroke:#000000;stroke-width:0"
        id="fkey_shad0"
        width="58" height="56" x="-25" y="-23" rx="10" />
    <rect rx="10" y="-24" x="-26" height="56" width="58"
        id="fkey_shad1"
        style="fill:#000000;fill-opacity:0.1;stroke:#000000;stroke-width:0" />
    <rect style="fill:#000000;fill-opacity:0.15;stroke:#000000;stroke-width:0"
        id="fkey_shad2" width="58" height="56" x="-27" y="-25" rx="10" />
    <rect rx="10" y="-26" x="-28" height="56" width="58"
        id="fkey_shad3"
        style="fill:#000000;fill-opacity:0.2;stroke:#000000;stroke-width:0" />
    <g id="fkey_btn" x="0" y="0">
      <rect style="fill:#ffff5b;fill-opacity:0;stroke:#000000;stroke-width:0"
          id="fkey_light"
          width="64" height="64" x="-32" y="-32" rx="14" />
      <rect style="fill:#b4beeb;fill-opacity:1;stroke:#000000;stroke-width:2"
          id="fkey_form" width="56" height="56" x="-28" y="-28" rx="10" />
      <rect style="fill:url(#linearGradient3199);fill-opacity:1;stroke:#000000;stroke-width:0"
          id="fkey_back" width="48" height="48" x="-24" y="-24" rx="6" />
      <g id="fkey_label" x="0" y="0">
        <text sodipodi:linespacing="125%"
            style="font-size:18px;font-style:normal;font-weight:bold;line-height:125%;
fill:#ffffff;fill-opacity:1;stroke:none;stroke-width:1px;stroke-linecap:butt;stroke-linejoin:miter;
stroke-opacity:1;font-family:Bitstream␣Vera␣Sans"
            x="0" y="5"
            id="fkey_txt"
            xml:space="preserve"><tspan style="text-align:center;line-height:125%;writing-mode:lr-tb;
text-anchor:middle"
              sodipodi:role="line" id="tspan2433" x="0" y="5">Cap</tspan></text>
      </g>
    </g>
  </g>
  <g id="char_key" x="0" y="0">
    <rect style="fill:#000000;fill-opacity:0.05;stroke:#000000;stroke-width:0"
        id="ckey_shad0" width="42" height="56" x="-17" y="-23" rx="10" />
    <rect rx="10" y="-24" x="-18"
        height="56" width="42"
        id="ckey_shad1" style="fill:#000000;fill-opacity:0.1;stroke:#000000;stroke-width:0" />
    <rect style="fill:#000000;fill-opacity:0.1;stroke:#000000;stroke-width:0"
        id="ckey_shad2" width="42" height="56" x="-19" y="-25" rx="10" />
    <rect style="fill:#000000;fill-opacity:0.2;stroke:#000000;stroke-width:0"
        id="ckey_shad3" width="42" height="56" x="-20" y="-26" rx="10" />
    <g id="ckey_btn" x="0" y="0">
      <rect rx="14" y="-32" x="-24" height="64" width="48"
          id="ckey_light"
          style="fill:#ffff5b;fill-opacity:0;stroke:#000000;stroke-width:0" />
      <rect style="fill:#ffffff;stroke:#000000;stroke-width:2"
          id="ckey_form" width="40" height="56" x="-20" y="-28" rx="10" />
      <rect style="fill:url(#linearGradient3191);fill-opacity:1;stroke:#000000;stroke-width:0"
          id="ckey_back" width="32" height="48" x="-16" y="-24" rx="6" />
      <text id="ckey_label"
          y="9" x="0"
          style="font-size:28px;font-style:normal;font-weight:bold;line-height:125%;fill:#000000;
fill-opacity:1;stroke:none;stroke-width:1px;stroke-linecap:butt;stroke-linejoin:miter;
stroke-opacity:1;font-family:Bitstream␣Vera␣Sans"
          xml:space="preserve"
          sodipodi:linespacing="125%"><tspan y="10" x="0" id="tspan2412"
            sodipodi:role="line" style="text-align:center;line-height:125%;writing-mode:lr-tb;
text-anchor:middle;fill:#000000;fill-opacity:1">A</tspan></text>
    </g>
```

```
  </g>
</svg>
```

B.2 Le modèle conceptuel

Le modèle conceptuel est présenté dans le listing B.9.

Listing B.9 – Implémentation du modèle du clavier à l'aide de la syntaxe JSON (fichier keyb_model.json).

```
{
    "model": "KEYBOARD",
    "classes": [
    {
        "name":"Object",
        "extends":null,
        "attributes": {
            "ID":"key",
            "ParentID":"int",
            "X0":"vint",
            "Y0":"vint",
            "PRIORITY":"vfloat" }
    },{
        "name":"Keyboard",
        "extends":"Object",
        "attributes": {
            "VISIBLE":"vbool",
            "content":"items[]",
            "SCALE":"vfloat" }
    },{
        "name":"Key",
        "extends":"Object",
        "attributes": {
            "NAME":"string",
            "ENABLE":"vbool",
            "VISIBLE":"vbool",
            "HIGHLIGHTED":"vbool",
            "SELECTED":"vbool",
            "Picked_Key":"vint",
            "BTN_X0":"vint",
            "BTN_Y0":"vint",
            "BACK_COLOR0_R":"vfloat",
            "BACK_COLOR0_G":"vfloat",
            "BACK_COLOR0_B":"vfloat",
            "BACK_COLOR1_R":"vfloat",
            "BACK_COLOR1_G":"vfloat",
            "BACK_COLOR1_B":"vfloat",
            "WIDTH":"int",
            "BACK_WIDTH":"self.WIDTH-8",
            "SHAD0_WIDTH":"self.WIDTH+2",
            "SHAD1_WIDTH":"self.WIDTH+2",
            "SHAD2_WIDTH":"self.WIDTH+2",
            "SHAD3_WIDTH":"self.WIDTH+2",
            "LIGHT_WIDTH":"self.WIDTH+8",
            "HEIGHT":"int",
            "BACK_HEIGHT":"self.HEIGHT-8",
            "SHAD0_HEIGHT":"self.HEIGHT",
            "SHAD1_HEIGHT":"self.HEIGHT",
            "SHAD2_HEIGHT":"self.HEIGHT",
            "SHAD3_HEIGHT":"self.HEIGHT",
            "LIGHT_HEIGHT":"self.HEIGHT+8" }
    },{
        "name":"CharKey",
        "extends":"Key",
        "attributes": {
            "LABEL":"vstring",
            "LABEL_SCALE":"vfloat",
            "ICON":"string",
            "VALUE":"string",
            "SCALE":"vfloat",
            "FORM_X0":"(self.WIDTH-40)/-2",
            "BACK_X0":"(self.WIDTH-40)/-2",
            "LIGHT_X0":"(self.WIDTH-40)/-2",
            "LIGHT_OPACITY":"vfloat",
            "SHAD_X0":"(self.WIDTH-40)/-2"}
    },{
        "name":"FuncKey",
        "extends":"Key",
        "attributes": {
            "LABEL":"vstring",
            "LABEL_COLOR_R":"vfloat",
            "LABEL_COLOR_G":"vfloat",
            "LABEL_COLOR_B":"vfloat",
```

```
            "ICON":"vstring",
            "VALUE":"string",
            "VALUES":"string",
            "SCALE":"vfloat",
            "FORM_X0":"(self.WIDTH-56)/-2",
            "BACK_X0":"(self.WIDTH-56)/-2",
            "LIGHT_X0":"(self.WIDTH-56)/-2",
            "LIGHT_OPACITY":"vfloat",
            "SHAD_X0":"(self.WIDTH-56)/-2",
            "LABEL_Y0":"(self.HEIGHT-56)/-2",
            "TOGGLE":"bool" }
    },{
        "name":"EnterKey",
        "extends":"Key",
        "attributes": {
            "LABEL":"vstring",
            "LABEL_SCALE":"vfloat",
            "ICON":"vstring",
            "SCALE":"vfloat",
            "FORM_X0":"(self.WIDTH-100)/-2",
            "BACK_X0":"(self.WIDTH-100)/-2",
            "LIGHT_X0":"(self.WIDTH-100)/-2",
            "LIGHT_OPACITY":"vfloat",
            "SHAD_X0":"(self.WIDTH-100)/-2",
            "VALUE":"string" }
    },{
        "name":"KbDisplay",
        "extends":"Object",
        "attributes": {
            "VISIBLE":"vbool",
          "NAME":"string",
          "TEXT":"vstring",
          "TXTWIDTH":"int" }
    },{
        "name":"KbBack",
        "extends":"Object",
        "attributes": {
            "VISIBLE":"vbool" }
    }]
}
```

B.3 Les connections entre le modèle et les classes graphiques

Voici le listing contenant les connexions entre les classes graphiques et les attributs du modèle.

Listing B.10 – Fichier de connexion des items du clavier à l'aide de la syntaxe JSON (fichier keyb_modelToSVG.json).

```
{
    "model":"KEYBOARD",
    "objects": [
    {
        "className":"Keyboard",
        "connections": {
            "X0":"transform.tx",
            "Y0":"transform.ty",
            "SCALE":"transform.scale"}
    },{
        "className":"CharKey",
        "file":"keyboard2.svg",
        "graphicalItems": [
        {
            "name":"char_key",
            "connections":
            {
                "X0":"char_key.transform.tx",
                "Y0":"char_key.transform.ty",
                "PRIORITY":"char_key.transform.priority",
            "SCALE":"char_key.transform.scale",
                "WIDTH":"ckey_form.width",
                "HEIGHT":"ckey_form.height",
            "FORM_X0":"ckey_form.transform.tx",
                "BTN_X0":"ckey_btn.transform.tx",
                "BTN_Y0":"ckey_btn.transform.ty",
            "FORM_X0":"ckey_form.transform.tx",
            "BACK_X0":"ckey_back.transform.tx",
                "BACK_WIDTH":"ckey_back.width",
                "BACK_HEIGHT":"ckey_back.height",
```

```
                "BACK_COLOR0_R":"ckey_back.style.fill.stop.0.color.0",
                "BACK_COLOR0_G":"ckey_back.style.fill.stop.0.color.1",
                "BACK_COLOR0_B":"ckey_back.style.fill.stop.0.color.2",
                "BACK_COLOR1_R":"ckey_back.style.fill.stop.1.color.0",
                "BACK_COLOR1_G":"ckey_back.style.fill.stop.1.color.1",
                "BACK_COLOR1_B":"ckey_back.style.fill.stop.1.color.2",
            "LIGHT_X0":"ckey_light.transform.tx",
                "LIGHT_WIDTH":"ckey_light.width",
                "LIGHT_HEIGHT":"ckey_light.height",
              "LIGHT_OPACITY":"ckey_light.style.fillopacity",
                "HIGHLIGHTED":"ckey_light.actif",
            "SHAD_X0":["ckey_shad0.transform.tx","ckey_shad1.transform.tx",
                       "ckey_shad2.transform.tx","ckey_shad3.transform.tx"],
                "SHAD0_WIDTH":"ckey_shad0.width",
              "SHAD1_WIDTH":"ckey_shad1.width",
              "SHAD2_WIDTH":"ckey_shad2.width",
              "SHAD3_WIDTH":"ckey_shad3.width",
                "SHAD0_HEIGHT":"ckey_shad0.height",
                "SHAD1_HEIGHT":"ckey_shad1.height",
                "SHAD2_HEIGHT":"ckey_shad2.height",
                "SHAD3_HEIGHT":"ckey_shad3.height",
                "LABEL":"ckey_label",
            "LABEL_SCALE":"ckey_label.transform.scale"
            },
            "picking": {"Picked_Key":"char_key"}
        }]
    },{
        "className":"FuncKey",
        "file":"keyboard2.svg",
        "graphicalItems": [
        {
            "name":"func_key",
            "connections":
            {
                "X0":"func_key.transform.tx",
                "Y0":"func_key.transform.ty",
                "PRIORITY":"func_key.transform.priority",
                "SCALE":"func_key.transform.scale",
                "WIDTH":"fkey_form.width",
                "HEIGHT":"fkey_form.height",
                "FORM_X0":"fkey_form.transform.tx",
                "BTN_X0":"fkey_btn.transform.tx",
                "BTN_Y0":"fkey_btn.transform.ty",
                "FORM_X0":"fkey_form.transform.tx",
                "BACK_X0":"fkey_back.transform.tx",
                "BACK_WIDTH":"fkey_back.width",
                "BACK_HEIGHT":"fkey_back.height",
                "BACK_COLOR0_R":"fkey_back.style.fill.stop.0.color.0",
                "BACK_COLOR0_G":"fkey_back.style.fill.stop.0.color.1",
                "BACK_COLOR0_B":"fkey_back.style.fill.stop.0.color.2",
                "BACK_COLOR1_R":"fkey_back.style.fill.stop.1.color.0",
                "BACK_COLOR1_G":"fkey_back.style.fill.stop.1.color.1",
                "BACK_COLOR1_B":"fkey_back.style.fill.stop.1.color.2",
                "LIGHT_X0":"fkey_light.transform.tx",
                "LIGHT_WIDTH":"fkey_light.width",
                "LIGHT_HEIGHT":"fkey_light.height",
                "LIGHT_OPACITY":"fkey_light.style.fillopacity",
                "HIGHLIGHTED":"fkey_light.actif",
                "SHAD_X0":["fkey_shad0.transform.tx","fkey_shad1.transform.tx",
                           "fkey_shad2.transform.tx","fkey_shad3.transform.tx"],
                "SHAD0_WIDTH":"fkey_shad0.width",
                "SHAD1_WIDTH":"fkey_shad1.width",
                "SHAD2_WIDTH":"fkey_shad2.width",
                "SHAD3_WIDTH":"fkey_shad3.width",
                "SHAD0_HEIGHT":"fkey_shad0.height",
                "SHAD1_HEIGHT":"fkey_shad1.height",
                "SHAD2_HEIGHT":"fkey_shad2.height",
                "SHAD3_HEIGHT":"fkey_shad3.height",
                "LABEL":"fkey_label",
                "LABEL_COLOR_R":"fkey_txt.style.fill.0",
                "LABEL_COLOR_G":"fkey_txt.style.fill.1",
                "LABEL_COLOR_B":"fkey_txt.style.fill.2"
            },
            "picking": {"Picked_Key":"func_key"}
        }]
    },{
        "className":"EnterKey",
        "file":"keyboard2.svg",
        "graphicalItems": [
        {
            "name":"enter_key",
            "connections":
            {
                "X0":"enter_key.transform.tx",
                "Y0":"enter_key.transform.ty",
                "PRIORITY":"enter_key.transform.priority",
                "SCALE":"enter_key.transform.scale",
                "WIDTH":"ekey_form.width",
                "HEIGHT":"ekey_form.height",
                "FORM_X0":"ekey_form.transform.tx",
                "BTN_X0":"ekey_btn.transform.tx",
                "BTN_Y0":"ekey_btn.transform.ty",
```

```
                "FORM_X0":"ekey_form.transform.tx",
                "BACK_X0":"ekey_back.transform.tx",
                "BACK_WIDTH":"ekey_back.width",
                "BACK_HEIGHT":"ekey_back.height",
                "BACK_COLOR0_R":"ekey_back.style.fill.stop.0.color.0",
                "BACK_COLOR0_G":"ekey_back.style.fill.stop.0.color.1",
                "BACK_COLOR0_B":"ekey_back.style.fill.stop.0.color.2",
                "BACK_COLOR1_R":"ekey_back.style.fill.stop.1.color.0",
                "BACK_COLOR1_G":"ekey_back.style.fill.stop.1.color.1",
                "BACK_COLOR1_B":"ekey_back.style.fill.stop.1.color.2",
                "LIGHT_X0":"ekey_light.transform.tx",
                "LIGHT_WIDTH":"ekey_light.width",
                "LIGHT_HEIGHT":"ekey_light.height",
                "LIGHT_OPACITY":"ekey_light.style.fillopacity",
                "HIGHLIGHTED":"ekey_light.actif",
                "SHAD_X0":["ekey_shad0.transform.tx","ekey_shad1.transform.tx","ekey_shad2.transform.tx",
                           "ekey_shad3.transform.tx"],
                "SHAD0_WIDTH":"ekey_shad0.width",
                "SHAD1_WIDTH":"ekey_shad1.width",
                "SHAD2_WIDTH":"ekey_shad2.width",
                "SHAD3_WIDTH":"ekey_shad3.width",
                "SHAD0_HEIGHT":"ekey_shad0.height",
                "SHAD1_HEIGHT":"ekey_shad1.height",
                "SHAD2_HEIGHT":"ekey_shad2.height",
                "SHAD3_HEIGHT":"ekey_shad3.height",
                "LABEL":"ekey_label",
                "LABEL_SCALE":"ekey_label.transform.scale" },
            "picking": {"Picked_Key":"enter_key"}
        }]
    },{
        "className":"KbDisplay",
        "file":"keyboard2.svg",
        "graphicalItems": [{
            "name":"keyb_display",
            "connections":{
                "X0":"keyb_display.transform.tx",
                "Y0":"keyb_display.transform.ty",
                "TEXT":"keyb_display.display_txt"}
        }]
    },{
        "className":"KbBack",
        "file":"keyboard2.svg",
        "graphicalItems": [{
            "name":"keyb_back",
            "connections":{
                "X0":"keyb_back.transform.tx",
                "Y0":"keyb_back.transform.ty",
                "PRIORITY":"keyb_back.transform.priority"}
        }]
    }]
}
```

B.4 La scène

Enfin, l'instanciation des différents objets utilisés dans l'application interactive finale est réalisée par le fichier de scène généré par le code suivant.

Listing B.11 – Le générateur du fichier de *scène* des items du clavier (fichier smileys.json).

```
__author__="vinot"
__date__ ="$07 avril 2009 17:05:34$"

def generate(ktable,spaceunit):
    ## header de scene
    s = """{
    "name":"KEYB",
    "model":"KEYBOARD",
    "content": ["""
    ## variables
    sep = ""
    numline = 0
    index = 1
    x = 0
    y = 0
    s += sep+"""
    {    "type":"Keyboard",
         "attributes": {
             "ID":"keyboard",
             "ParentID":0,
             "VISIBLE":true,
             "X0":100,
             "Y0":100,
             "SCALE":0.5,
```

```
            "content":[ """
    ## afficheur
    s += sep+"""
    {   "type":"KbBack",
        "attributes": {
            "ID":%s,
            "ParentID":0,
            "VISIBLE":true,
            "X0":%s,
            "Y0":%s }
    },"""%(index, x, y)
    index += 1
    s += sep+"""
    {   "type":"KbDisplay",
        "attributes": {
            "ID":%s,
            "ParentID":0,
            "NAME":"Display",
            "VISIBLE":true,
            "X0":%s,
            "Y0":%s,
            "TXTWIDTH":365,
          "TEXT":"" }
    },"""%(index, x, y)
    ## construction des lignes
    y += 90
    for line in ktable:
        x = 24
        # construction des lignes
        for classname,name,value,width,height in line:
            if classname == "move":
                if value == "abs":
                    x = width
                    y = height
                else:
                    x += width
                    y += height
            else:
                index += 1
                classname += "Key"
                if classname == "CharKey":
                    if not width:
                        width = 40
                    if not height:
                        height = 56
                    icon = ""
                    keyname = "Key_" + name
                    valCap,valMin,valNum = value
                    s += sep+"""
    {   "type":"%s",
        "attributes": {
            "ID":%d,
            "ParentID":0,
            "NAME":"%s",
            "ENABLE":true,
            "VISIBLE":true,
            "HIGHLIGHTED":false,
            "SELECTED":false,
            "X0":%d,
            "Y0":%d,
            "WIDTH":%d,
            "HEIGHT":%d,
            "LIGHT_OPACITY":0.0,
            "LABEL":"%s",
            "ICON":"",
            "VALUE":["%s","%s","%s"],
            "Picked_Key":-1 }
    }"""%(classname, index, keyname, x+(width/2), y, width, height, valCap, valCap, valMin, valNum)

                elif classname == "FuncKey":
                    labCap,labMin,labNum,toggle = value
                    if not width:
                        width = 56
                    if not height:
                        height = 56
                    toggle = str(toggle).lower()
                    keyname = "Key_" + name
                    icon = name
                    s += sep+"""
    {
        "type":"%s",
        "attributes": {
            "ID":%d,
            "ParentID":0,
            "NAME":"%s",
            "ENABLE":true,
            "VISIBLE":true,
            "HIGHLIGHTED":false,
            "SELECTED":false,
            "X0":%d,
            "Y0":%d,
            "WIDTH":%d,
```

```
                "HEIGHT":%d,
                "LIGHT_OPACITY":0.0,
                "LABEL":"%s",
                "ICON":"%s",
                "VALUE":"%s",
                "VALUES":["%s","%s","%s"],
                "Picked_Key":-1,
                "TOGGLE":%s }
    }"""%(classname, index, keyname, x+(width/2), y, width, height, labCap, icon, name,
          labCap, labMin, labNum, toggle)

                elif classname == "EnterKey":
                    label,toggle = value
                    if not width:
                        width = 100
                    if not height:
                        height = 56
                    icon = name
                    keyname = "Key_" + name
                    s += sep+"""
    {   "type":"%s",
        "attributes": {
                "ID":%d,
                "ParentID":0,
                "NAME":"%s",
                "ENABLE":true,
                "VISIBLE":true,
                "HIGHLIGHTED":false,
                "SELECTED":false,
                "X0":%d,
                "Y0":%d,
                "WIDTH":%d,
                "HEIGHT":%d,
                "LIGHT_OPACITY":0.0,
                "LABEL":"%s",
                "ICON":"%s",
                "VALUE":"%s",
                "Picked_Key":-1 }
    }"""%(classname, index, keyname, x+(width/2), y, width, height, label, icon, name)

                x += width
                x += spaceunit
                sep=','
        y += 75
    s.rstrip(',')
    s+= """ ]
        }
    }]
}"""
    return s

if __name__ == "__main__":
    import sys
    line0 = [("move",,"abs",538,40),("Func","back",["Back","Back","Back",False],76,48),
             ("move","",None,0,12)]
    line1 = [("Char","arobas",["@","a","C"],40,56),("Char","A",["A","a","["],40,56),
             ("Char","Z",["Z","z","("],40,56),("Char","E",["E","e",")"],40,56),
             ("Char","R",["R","r","]"],40,56),("Char","T",["T","t","7"],40,56),
             ("Char","Y",["Y","y","8"],40,56),("Char","U",["U","u","9"],40,56),
             ("Char","I",["I","i","MC"],40,56),("Char","O",["O","o","M+"],40,56),
             ("Char","P",["P","p","M-"],40,56),("Char","dollar",["$","u","MR"],40,56)]
    line2 = [("Func","shift",["Min","Cap"," ",True],60,56),("move","",None,10,0),
             ("Char","Q",["Q","q","<"],40,56),("Char","S",["S","s",">"],40,56),
             ("Char","D",["D","d","-"],40,56),("Char","F",["F","f","+"],40,56),
             ("Char","G",["G","g","4"],40,56),("Char","H",["H","h","5"],40,56),
             ("Char","J",["J","j","6"],40,56),("Char","K",["K","k","*"],40,56),
             ("Char","L",["L","l","/"],40,56),("Char","M",["M","m","C"],40,56)]
    line3 = [("Char","ampersand",["&",",","sin"],40,56),("Char","antislash",["\\","c","cos"],40,56),
             ("Char","W",["W","w","tan"],40,56),("Char","X",["X","x","log"],40,56),
             ("Char","C",["C","c","."],40,56),("Char","V",["V","v","0"],40,56),
             ("Char","B",["B","b","1"],40,56),("Char","N",["N","n","2"],40,56),
             ("Char","question",["?","e","3"],40,56),("Char","dot",[".","e","1/X"],40,56),
             ("Char","slash",["/"," ","X2"],40,56),("Char","exclamation",["!"," ","%"],40,56)]
    line4 = [("move","",None,0,10),("Func","nummode",["123","123","Abc",True],70,48),
             ("move","",None,8,0),("Func","expand",["Exp","Exp","Exp",True],70,48),
             ("move","",None,15,0),("Char","space",[" "," "," "],270,48),("move","",None,15,0),
             ("Enter","enter",["GO",False],110,48)]
    keyboard = [line0,line1,line2,line3,line4]
    filecontent = generate(keyboard,10)
    scenefile = open("keyboard.json", 'w', -1)
    scenefile.write(filecontent)
    scenefile.close()
```

B.5 Le lanceur

Voici le code complet de l'application Python gérant les interactions.

Le lancement du compilateur graphique est effectué par l'appel à *gc.load* dans le *main* du fichier (à la fin du listing).

Listing B.12 – Le code python permettant de contrôler le clavier logiciel.

```
#! /usr/bin/python
# -*- coding: utf-8 -*-

## Projet KEYBOARD
## mainfile keyboard2.py

__author__ = "vinot"
__date__ = "$17 avril 2009 13:28:32$"

import sys,os
sys.path.append(os.path.abspath(os.path.join(os.getcwd(),os.pardir,os.pardir)))

import re

from graphical_compiler import *
from graphical_compiler import keyboard as gckeyboard
from graphical_compiler.utils import graphic_tools
from math import sqrt,sin,cos,tan

## keyboard initial size
kwidth = 640
kheight = 410

## expanding key settings
expanding_factor = 0.8
expanding_limit = 120
automatic_expand = 0.7

## nummode special keys
specialkeys = ('.','0','1','2','3','4','5','6','7','8','9')
specialNumChars = {u"\xd7":"*",u"\xf7":"/",u"X\xb2":"**2"}

## accents trema et circonflexe
trema_validchars = {'e':u"\xeb",'i':u"\xef",'u':u"\xfc"}
circonflexe_validchars = {'a':u"\xe2",'e':u"\xea",'i':u"\xee",'o':u"\xf4",'u':u"\xfb"}

## codes speciaux clavier
gck_validUChars = (u"\xe0",u"\xe7",u"\xe8",u"\xe9")
postscriptChars = {'LEFTPAREN':"(",'RIGHTPAREN':")",'PLUS':"+",'MINUS':"-",
                   'ASTERISK':"*",'SLASH':"/",'WORLD_89':u"\xf9",'WORLD_73':u"\xe9",
                   'WORLD_72':u"\xe8",'WORLD_64':u"\xe0",
                   'WORLD_71':u"\xe7",'UNKWNON_37':"%"}

## keyboard class
## clavier "KEYBOARD"
class keyboard(object):
    def __init__(self, id, width, height):
        self.GraphicObject = getObject(id)
        self.scale = self.GraphicObject.SCALE
        self.x0 = self.GraphicObject.X0
        self.y0 = self.GraphicObject.Y0
        self.width = float(width)
        if width < 1 :
            raise Exception,"keyboard width < 1"
        self.height = float(height)
        if height < 1 :
            raise Exception,"keyboard height < 1"
        self.allkeys = {}
        self.charkeys = {}
        self.funckeys = {}
        self.enterkey = None
        self.shiftmode = False
        self.nummode = False
        self.result = ""
        self.display = False
        self.expanding = False
        self.loaded_accent = ""

    def scaleSizeCallback(self, id):
        global automatic_expand
        newwidth = window().width.eval()
        newheight = window().height.eval()
        newscale = min(newwidth/self.width,newheight/self.height)
        self.scale.set(newscale)
        newX = (newwidth - (self.width*newscale))/2
        newY = (newheight - (self.height*newscale))/2
        self.x0.set(newX)
        self.y0.set(newY)
        try:
```

```
            expandkey = self.funckeys['expand']
            if newscale < automatic_expand and not self.expanding:
                expandkey.switchActivation(id,'auto')
                self.switchExpanding()
            elif newscale > automatic_expand and self.expanding and expandkey.activmode == 'auto':
                expandkey.switchActivation(id,'auto')
                self.switchExpanding()
        except:
            pass


    def switchExpanding(self):
        if self.expanding:
            self.expanding = False
            for num, s in self.allkeys.items():
                s.scale.set(1)
        else:
            self.expanding = True

    def expandingKeyboard(self, id):
        if self.expanding:
            xc = mouse().x.eval()
            yc = mouse().y.eval()
            kb_x0 = self.x0.eval()
            kb_y0 = self.y0.eval()
            ks = self.scale.eval()
            for num, s in self.allkeys.items():
                s.expanding(xc,yc,kb_x0,kb_y0,ks)

    def addKey(self, keyclass, o, connections):
        s = keyclass(self, o.ID)
        connections.append((s.GraphicObject.Picked_Key, s.picking_callback))
        connections.append((mouse().buttons[0], s.mouseDownCallback))
        self.allkeys[o.ID] = s
        if o.getClass() == "CharKey":
            self.charkeys[o.ID] = s
        elif o.getClass() == "FuncKey":
            name = s.value
            self.funckeys[name] = s
        elif o.getClass() == "EnterKey":
            self.enterkey = s

    def addDisplay(self, id, connections):
        s = KeybDisplay(self, id)
        self.display = s

    def addBack(self, id, connections):
        s = KeybBack(self, id)

    def changeKeys(self, mode):
        valindex = 1
        if mode == 'shift' and self.shiftmode:
            valindex = 0

        elif mode == 'num':
            shiftkey = self.funckeys['shift']
            shiftkey.setEnabled(not self.nummode)
            if self.nummode:
                valindex = 2
            else:
                if self.shiftmode:
                    valindex = 0
                else:
                    valindex = 1

        print "changeKeys␣mode:",mode

        for num, s in self.charkeys.items():
            s.updateLabel(valindex)

        for num, s in self.funckeys.items():
            s.updateLabel(valindex)

        glvm.postRedisplay()

    def keyDownCallback(self,id):
        global gck_validUChars
        key = glvm.evalVar(id)
        keyvalue = gckeyboard().getKey(key)
        if len(keyvalue) < 2 or keyvalue in gck_validUChars:
            self.display.addChar(keyvalue)
        elif postscriptChars.has_key(keyvalue):
            charvalue = postscriptChars[keyvalue]
            self.display.addChar(charvalue)
        elif keyvalue == "BACKSPACE":
            self.display.popChar()
        elif keyvalue == "SPACE":
            self.display.addChar("␣")
        elif keyvalue == "RETURN":
            s = self.enterkey
            s.computeKey(s.index)
        else:
```

```
            print "keyvalue:",keyvalue


## key class
## super classe des touches du clavier KEYBOARD
class key(object):
    def __init__(self, parent, id):
        self.parent = parent
        self.GraphicObject = getObject(id)
        self.enabled_key = self.GraphicObject.ENABLE
        self.picked_key = False
        self.selected_key = False
        self.specialkey = False
        self.state = False
        self.mouseDown = False
        self.x0 = self.GraphicObject.X0
        self.y0 = self.GraphicObject.Y0
        self.width = self.GraphicObject.WIDTH
        self.scale = self.GraphicObject.SCALE
        self.label = self.GraphicObject.LABEL
        self.lighting = self.GraphicObject.LIGHT_OPACITY
        self.priority = self.GraphicObject.PRIORITY
        self.highlighted = self.GraphicObject.HIGHLIGHTED
        self.priority.set(0)
        self.setBackColor()

    def picking_callback(self, id):
        value_picking = glvm.evalVar(id)
        self.picked_key = value_picking > -1
        self.setBackColor()

    def expanding(self, cx, cy, kb_x0, kb_y0, ks):
        global expanding_factor, expanding_limit
        priority = 0
        if self.enabled_key:
            x0 = (self.x0.eval()*ks) + kb_x0
            y0 = (self.y0.eval()*ks) + kb_y0
            dx = cx - x0
            dy = cy - y0
            res = sqrt((dx*dx)+(dy*dy))
            max_dist = expanding_limit * ks
            newscale = 1
            if res <= max_dist:
                newscale = 1 + (((max_dist - res)/max_dist)*expanding_factor)
                priority = -(newscale)
            self.scale.set(newscale)
        self.priority.set(priority)
        print priority
        #print self, newscale, priority

    def setBackColor(self):
        keyboard = self.parent
        highlighted = 0
        colors = self.colors['back']
        if self.specialkey:
            colors = self.colors['special']
        if not self.enabled_key:
            colors = self.colors['disabled']
        elif self.selected_key:
            colors = self.colors['select']
            highlighted = 1
        else:
            if self.picked_key:
                colors = self.colors['hilite']

        self.highlighted.set(highlighted)
        r0,g0,b0 = colors[0]
        r1,g1,b1 = colors[1]

        self.GraphicObject.BACK_COLOR0_R.set(r0/255.0)
        self.GraphicObject.BACK_COLOR0_G.set(g0/255.0)
        self.GraphicObject.BACK_COLOR0_B.set(b0/255.0)
        self.GraphicObject.BACK_COLOR1_R.set(r1/255.0)
        self.GraphicObject.BACK_COLOR1_G.set(g1/255.0)
        self.GraphicObject.BACK_COLOR1_B.set(b1/255.0)

        glvm.postRedisplay()

## CharKey class (classe derivee de key)
## element graphique : touches standards (caracteres) du clavier KEYBOARD
## Representation SVG char_key (keyboard3.svg)
class CharKey(key):
    colors = {'back':((147,178,245),(245,255,255)),
              'hilite':((147,178,245),(255,255,189)),
              'select':((130,197,245),(255,255,127)),
              'disabled':((153,185,232),(200,200,200)),
              'special':((95,150,232),(224,248,255))}

    def __init__(self, parent, id):
        self.GraphicObject = getObject(id)
        self.values = self.GraphicObject.VALUE
        self.labelscale = self.GraphicObject.LABEL_SCALE
```

```
        self.colors = CharKey.colors
        super(CharKey, self).__init__(parent,id)

    def mouseDownCallback(self, id):
        self.mouseDown = glvm.evalVar(mouse().buttons[0].id) > 0
        if self.mouseDown and self.picked_key:
            self.lighting.set(0.4)
            self.GraphicObject.BTN_X0.set(3)
            self.GraphicObject.BTN_Y0.set(4)
            self.selected_key = True
            self.computeKey(id)
        else:
            self.lighting.set(0.0)
            self.GraphicObject.BTN_X0.set(0)
            self.GraphicObject.BTN_Y0.set(0)
            self.selected_key = False
        self.setBackColor()

    def computeKey(self, id):
        keyboard = self.parent
        keyvalue = self.label.eval()
        loaded_accent = keyboard.loaded_accent
        if keyboard.nummode:
            if keyvalue == "CE":
                keyboard.display.erase()
            else:
                if specialNumChars.has_key(keyvalue):
                    keyvalue = specialNumChars[keyvalue]
                keyboard.display.addChar(keyvalue)
        else:
            if loaded_accent:
                if loaded_accent == "trema" and trema_validchars.has_key(keyvalue):
                    keyvalue = trema_validchars[keyvalue]
                elif loaded_accent == "circonflexe" and circonflexe_validchars.has_key(keyvalue):
                    keyvalue = circonflexe_validchars[keyvalue]
                keyboard.display.popChar()
                keyboard.loaded_accent = ""
            elif keyvalue == u"\xa8":
                keyboard.loaded_accent = "trema"
            elif keyvalue == u"\x5e":
                keyboard.loaded_accent = "circonflexe"
            if keyvalue:
                keyboard.display.addChar(keyvalue)

    def updateLabel(self,valindex):
        global specialkeys
        newlabel = self.values[valindex]
        self.label.set(newlabel)
        if newlabel in specialkeys and self.parent.nummode:
            self.specialkey = True
        else:
            self.specialkey = False
        numchar = len(newlabel)
        txtwidth = graphic_tools.getTxtWidth(newlabel)
        maxwidth = self.width - 14
        if txtwidth > maxwidth or numchar > 1:
            if maxwidth/txtwidth > 0.7 or numchar < 3:
                self.labelscale.set(0.8)
            else:
                self.labelscale.set(0.6)
        else:
            self.labelscale.set(1)

## FuncKey class (classe derivee de key)
## element graphique : touches de fonction du clavier KEYBOARD
## Representation SVG func_key (keyboard3.svg)
class FuncKey(key):
    colors = {'back':((71,90,116),(176,182,209)),
              'hilite':((71,90,116),(156,233,255)),
              'select':((71,90,116),(191,251,255)),
              'disabled':((139,148,158),(194,197,209))}

    def __init__(self, parent, id):
        self.GraphicObject = getObject(id)
        self.toggle = self.GraphicObject.TOGGLE
        self.value = self.GraphicObject.VALUE
        self.values = self.GraphicObject.VALUES
        self.colors = FuncKey.colors
        self.actived = False
        self.activmode = 'auto'
        super(FuncKey, self).__init__(parent,id)

    def mouseDownCallback(self, id):
        self.mouseDown = glvm.evalVar(mouse().buttons[0].id) > 0
        if self.mouseDown and self.picked_key and self.enabled_key:
            self.lighting.set(0.4)
            if self.toggle:
                self.switchActivation(id,'force')
            self.GraphicObject.BTN_X0.set(3)
            self.GraphicObject.BTN_Y0.set(4)
            self.selected_key = True
            self.computeKey(id)
```

```
        else:
            self.lighting.set(0.0)
            if self.toggle and self.actived:
                self.lighting.set(0.3)
            self.GraphicObject.BTN_X0.set(0)
            self.GraphicObject.BTN_Y0.set(0)
            self.selected_key = False
        self.setBackColor()

    def switchActivation(self,id,mode):
        self.actived = not self.actived
        self.activmode = mode
        self.setLabelColor()

    def setLabelColor(self):
        labelcolors = ((255,255,255),(255,240,0),(200,200,200))
        r,g,b = labelcolors[0]
        if not self.enabled_key:
            r,g,b = labelcolors[2]
        elif self.actived:
            r,g,b = labelcolors[1]
        self.GraphicObject.LABEL_COLOR_R.set(r/255.0)
        self.GraphicObject.LABEL_COLOR_G.set(g/255.0)
        self.GraphicObject.LABEL_COLOR_B.set(b/255.0)

    def computeKey(self, id):
        keyboard = self.parent
        if self.value == "shift":
            keyboard.shiftmode = not keyboard.shiftmode
            keyboard.changeKeys('shift')
        elif self.value == "nummode":
            keyboard.nummode = not keyboard.nummode
            keyboard.changeKeys('num')
        elif self.value == "back" and keyboard.display:
            keyboard.display.popChar()
        elif self.value == "expand":
            keyboard.switchExpanding()

    def updateLabel(self,valindex):
        newlabel = self.values[valindex]
        self.label.set(newlabel)

    def setEnabled(self,state):
        self.enabled_key = state
        self.setLabelColor()
        self.setBackColor()

## EnterKey class (classe derivee de key)
## element graphique : touches de type Enter du clavier KEYBOARD
## Representation SVG enter_key (keyboard3.svg)
class EnterKey(key):
    colors = {'back':((39,118,232),(119,197,219)),'hilite':((39,118,232),(156,233,255)),
              'select':((39,118,232),(119,255,219)),'disabled':((39,118,232),(200,200,200))}

    def __init__(self, parent, id):
        self.GraphicObject = getObject(id)
        self.value = self.GraphicObject.VALUE
        self.colors = EnterKey.colors
        self.index = id
        super(EnterKey,self).__init__(parent,id)

    def mouseDownCallback(self, id):
        self.mouseDown = glvm.evalVar(mouse().buttons[0].id) > 0
        if self.mouseDown and self.picked_key and self.enabled_key:
            self.lighting.set(0.4)
            self.GraphicObject.BTN_X0.set(2)
            self.GraphicObject.BTN_Y0.set(2)
            self.selected_key = True
            self.computeKey(id)
        else:
            self.lighting.set(0.0)
            self.GraphicObject.BTN_X0.set(0)
            self.GraphicObject.BTN_Y0.set(0)
            self.selected_key = False
        self.setBackColor()

    def computeKey(self, id):
        keyboard = self.parent
        if keyboard.display:
            kbvalue = keyboard.display.getvalue()
            if keyboard.nummode:
                resultat = ""
                try:
                    resultat = eval(kbvalue)
                    print "resultat:",resultat
                except Exception,e:
                    print "calcul non valide:",e
                keyboard.display.setvalue(resultat)
            else:
                print "result:",kbvalue
```

```
## KbBack class
## element graphique : fond de clavier KEYBOARD
## Representation SVG keyb_back (keyboard3.svg)
class KeybBack(object):
    def __init__(self, parent, id):
        self.parent = parent
        self.GraphicObject = getObject(id)
        self.x0 = self.GraphicObject.X0
        self.y0 = self.GraphicObject.Y0


## KbDisplay class
## element graphique : afficheur de clavier KEYBOARD
## Representation SVG keyb_back (keyboard3.svg)
class KeybDisplay(object):
    def __init__(self, parent, id):
        self.parent = parent
        self.GraphicObject = getObject(id)
        self.x0 = self.GraphicObject.X0
        self.y0 = self.GraphicObject.Y0
        self.txt = self.GraphicObject.TEXT
        self.maxwidth = self.GraphicObject.TXTWIDTH
        self.value = ""

    def setvalue(self,newvalue):
        self.value = str(newvalue)
        self.setDisplay()

    def getvalue(self):
        return self.value

    def setDisplay(self):
        v = self.value
        while graphic_tools.getTxtWidth(v) > self.maxwidth:
            v = v[1:]
        self.txt.set(v)
        glvm.postRedisplay()

    def addChar(self,char):
        self.value += char
        self.setDisplay()

    def erase(self):
        self.value = ""
        self.setDisplay()

    def popChar(self):
        v = self.value
        self.value = v[:-1]
        self.setDisplay()


## fonction initKeyboard
## creation du clavier Keyboard
def initKeyboard(id, connections):
    global kwidth, kheight
    keyb = keyboard(id,kwidth,kheight)
    connections.append((window().width, keyb.scaleSizeCallback))
    connections.append((window().height, keyb.scaleSizeCallback))
    connections.append((mouse().x,keyb.expandingKeyboard))
    connections.append((mouse().y,keyb.expandingKeyboard))
    connections.append((gckeyboard().keyDown,keyb.keyDownCallback))
    return keyb

## fonction parseScene
## Construction de la scene KEYBOARD
def parseScene(parent,scene,connections):
    for o in scene:
        if o.getClass() == 'CharKey':
            parent.addKey(CharKey, o, connections)
        elif o.getClass() == 'FuncKey':
            parent.addKey(FuncKey, o, connections)
        elif o.getClass() == 'EnterKey':
            parent.addKey(EnterKey, o, connections)
        elif o.getClass() == 'KbDisplay':
            parent.addDisplay(o.ID, connections)
        elif o.getClass() == 'KbBack':
            parent.addBack(o.ID, connections)
        elif o.getClass() == 'Keyboard':
            keyboard = initKeyboard(o.ID, connections)
            if o.content:
                parseScene(keyboard,o.content,connections)
                keyboard.changeKeys(None)
                keyboard.scaleSizeCallback(o.ID)

    return connections

def onScene(scene):
    print "onScene callback"
    Connects = []
    Connects = parseScene(None,scene,Connects)
    gc.connect(Connects)
```

```
## main
if __name__ == "__main__":
    import sys
    import time
    sceneFileName = "keyboard.json"
    begin_time = time.time()
    if len(sys.argv) > 1:
        sceneFileName = sys.argv[1]
    import graphical_compiler.gc as gc
    gc.load("keyb_model.json", "keyb_modelToSVG.json", sceneFileName, onScene,
                               width = 640, height = 410, backColor = (130/255.0,148/255.0,173/255.0))
    print "compile␣time", time.time() - begin_time
    glvm.wait()
```

Annexe C

Calculs du gain de l'approximation du rendu des lignes dans FromDaDy

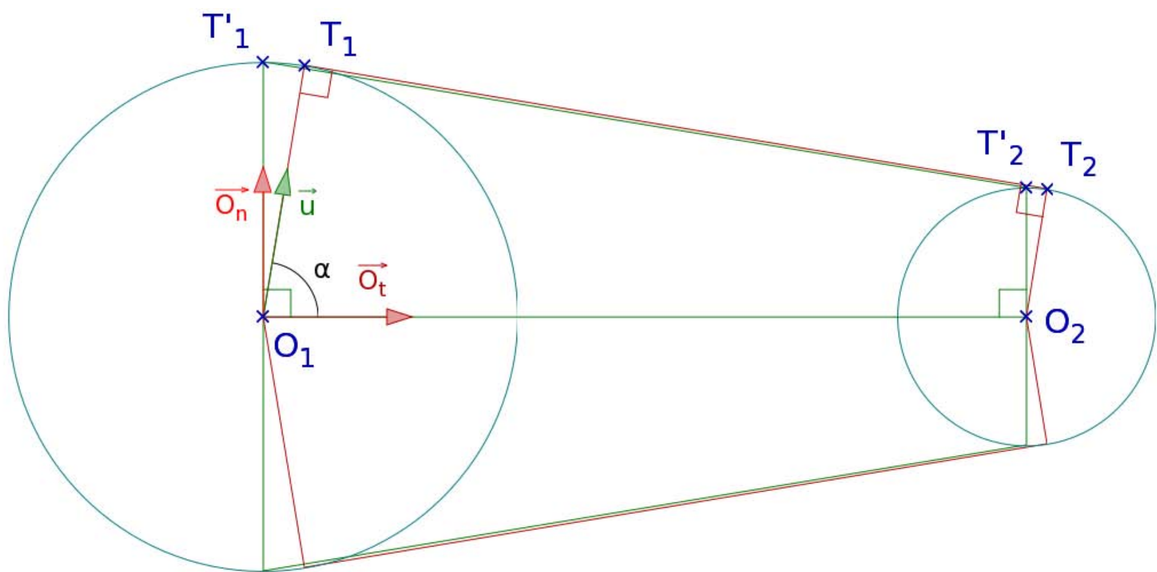

Figure C.1 – Les points idéaux sont en T1 et T2, les points approchés en T1' et T2'.

Nous pouvons exprimer $\vec{O_1T_1}$ et $\vec{O_2T_2}$ à partir du vecteur $\vec{u}$.

$$\begin{cases} \vec{O_1T_1} = t_1\vec{O_t} + n_1\vec{O_n} = R_1\vec{u} \\ \vec{O_2T_2} = t_2\vec{O_t} + n_2\vec{O_n} = R_2\vec{u} \end{cases} \Rightarrow \vec{T_1T_2} = \vec{O_1O_2} + (R_2 - R_1)\vec{u}$$

Or $\vec{T_1T_2} \cdot \vec{O_1T_1} = 0$,

$$\begin{aligned}
&\Rightarrow R_1\vec{O_1O_2} \cdot \vec{u} + R_1(R_2 - R_1) = 0 \\
&\Rightarrow \bar{O_1O_2}\vec{O_t} \cdot \vec{u} = R_1 - R_2 \\
&\Rightarrow \bar{O_1O_2}\cos\alpha = R_1 - R_2 \\
&\Rightarrow \boxed{\cos\alpha = \frac{R_1 - R_2}{\bar{O_1O_2}} = \frac{t_1}{R_1} = \frac{t_2}{R_2}}
\end{aligned}$$

D'autre part $\sin\alpha = \sqrt{1 - \cos^2\alpha}$, Donc

$$\boxed{\sin\alpha = \frac{\sqrt{\bar{O_1O_2}^2 - (R_1 - R_2)^2}}{\bar{O_1O_2}}}$$

Nous pouvons donc calculer les coordonnées de T_1 et T_2 :

$$\boxed{\begin{cases} \vec{O_1T_1} = R_1\frac{(R_1-R_2)}{\bar{O_1O_2}}\vec{O_t} \pm R_1\frac{\sqrt{\bar{O_1O_2}^2-(R_1-R_2)^2}}{\bar{O_1O_2}}\vec{O_n} \\ \vec{O_2T_2} = R_2\frac{(R_1-R_2)}{\bar{O_1O_2}}\vec{O_t} \pm R_2\frac{\sqrt{\bar{O_1O_2}^2-(R_1-R_2)^2}}{\bar{O_1O_2}}\vec{O_n} \end{cases}} \tag{C.1}$$

D'un autre côté, les calculs de T'_1 et T'_2 sont plus simples :

$$\boxed{\begin{cases} \vec{O_1T'_1} = \pm R_1\vec{O_n} \\ \vec{O_2T'_2} = \pm R_2\vec{O_n} \end{cases}} \tag{C.2}$$

Ceci montre donc que notre approximation est moins coûteuse : il y a 10 opérations supplémentaires par point dans la formule C.1. La formule utilisée (C.2) est plus simple et ne contient pas non plus de racines carrées.

Bibliographie

[Ahlberg 96] Christopher Ahlberg. Spotfire : an information exploration environment. SIGMOD Rec., vol. 25, pages 25–29, December 1996.

[Aho 86] Alfred V. Aho, Ravi Sethi & Jeffrey D. Ullman. Compilers : principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Appert 09] Caroline Appert, Stéphane Huot, Pierre Dragicevic & Michel Beaudouin-Lafon. FlowStates : prototypage d'applications interactives avec des flots de données et des machines à états. In Proceedings of the 21st International Conference on Association Francophone d'Interaction Homme-Machine, IHM '09, pages 119–128, New York, NY, USA, 2009. ACM.

[Arvind 86] Arvind & D E Culler. Dataflow architectures. In In Annual Reviews in Computer Science, pages 225–253, 1986.

[Aycock 03] John Aycock. A brief history of just-in-time. ACM Comput. Surv., vol. 35, no. 2, pages 97–113, 2003.

[Barboni 07] Eric Barboni, Stéphane Conversy, David Navarre & Philippe Palanque. Model-Based Engineering of Widgets, User Applications and Servers Compliant with ARINC 661 Specification. In Gavin Doherty & Ann Blandford, editeurs, Interactive Systems. Design, Specification, and Verification, volume 4323 of Lecture Notes in Computer Science, pages 25–38. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-69554-7_3.

[Baudel 04] Thomas Baudel. Browsing through an information visualization design space. In CHI '04 extended abstracts on Human factors in computing systems, CHI '04, pages 765–766, New York, NY, USA, 2004. ACM.

[Becker 87] Richard A. Becker & William S. Cleveland. Brushing scatterplots. Technometrics, vol. 29, pages 127–142, May 1987.

[Bederson 00] Benjamin B. Bederson, Jon Meyer & Lance Good. Jazz : an extensible zoomable user interface graphics toolkit in Java. In UIST '00 : Proceedings of the 13th annual ACM symposium on User interface software and technology, pages 171–180, New York, NY, USA, 2000. ACM.

[Bederson 04] Benjamin B. Bederson, J. Grosjean & Jon Meyer. Toolkit design for interactive structured graphics. Software Engineering, IEEE Transactions on, vol. 30, no. 8, pages 535–546, aug. 2004.

[Bertin 83] Jacques Bertin. Semiology of graphics. University of Wisconsin Press, 1983.

[Bhattacharyya 96] S S Bhattacharyya, P K Murthy & Edward Ashford Lee. Software Synthesis from Dataflow Graphs, 1996.

[Blanch 05] Renaud Blanch, Michel Beaudouin-Lafon, Stéphane Conversy, Yannick Jestin, Thomas Baudel & Yun Peng Zhao. INDIGO : une architecture pour la conception d'applications graphiques interactives distribuées. In IHM 2005 : Proceedings of the 17th international conference on Francophone sur l'Interaction Homme-Machine, pages 139–146, New York, NY, USA, 2005. ACM.

[Burke 93] Michael Burke & Linda Torczon. Interprocedural optimization : eliminating unnecessary recompilation. ACM Trans. Program. Lang. Syst., vol. 15, no. 3, pages 367–399, 1993.

[Card 99] Stuart K. Card, Jock D. Mackinlay & Ben Shneiderman, editeurs. Readings in information visualization : using vision to think. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[Chatty 04] Stéphane Chatty, Stéphane Sire, Jean-Luc Vinot, Patrick Lecoanet, Alexandre Lemort & Christophe Mertz. Revisiting visual interface programming : creating GUI tools for designers and programmers. In UIST '04 : Proceedings of the 17th annual ACM symposium on User interface software and technology, pages 267–276. ACM, 2004.

[Christensen 95] Jon Christensen, Joe Marks & Stuart Shieber. An empirical study of algorithms for point-feature label placement. ACM Trans. Graph., vol. 14, pages 203–232, July 1995.

[Cohen 93] Michael Cohen, Claude Puech, Francois Sillion, Paul Haeberli & Mark Segal. Texture Mapping as a Fundamental Drawing Primitive, 1993.

[Conversy 08] Stéphane Conversy, Eric Barboni, David Navarre & Philippe Palanque. Improving Modularity of Interactive Software with the MDPC Architecture. In Jan Gulliksen, Morton Harning, Philippe Palanque, Gerrit van der Veer & Janet Wesson, editeurs, Engineering Interactive Systems, volume 4940 of Lecture Notes in Computer Science, pages 321–338. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-92698-6_20.

[Davis 82] A L Davis & R M Keller. Data Flow Program Graphs. Computer, no. 15, pages 26–41, 1982.

[Dennis 74] Jack B. Dennis. First version of a data flow procedure language. In Programming Symposium, Proceedings Colloque sur la Programmation, pages 362–376, London, UK, 1974. Springer-Verlag.

[Dennis 75] Jack B. Dennis & David P. Misunas. A preliminary architecture for a basic data-flow processor. In Proceedings of the 2nd annual symposium on Computer architecture, ISCA '75, pages 126–132, New York, NY, USA, 1975. ACM.

[Derthick 97] Mark Derthick, John Kolojejchick & Steven F. Roth. An interactive visual query environment for exploring data. In Proceedings of the 10th annual ACM symposium on User interface software and technology, UIST '97, pages 189–198, New York, NY, USA, 1997. ACM.

[Deutsch 84] L. Peter Deutsch & Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In POPL '84 : Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 297–302, New York, NY, USA, 1984. ACM.

[Dragicevic 04a] Pierre Dragicevic. Un modèle d'interaction en entrée pour des systèmes interactifs multi-dispositifs hautement configurables. PhD thesis, Université de Nantes, 2004.

[Dragicevic 04b] Pierre Dragicevic & Jean-Daniel Fekete. The Input Configurator toolkit : towards high input adaptability in interactive applications. In AVI '04 : Proceedings of the working conference on Advanced visual interfaces, pages 244–247, New York, NY, USA, 2004. ACM.

[Dragicevic 05] Pierre Dragicevic, Stéphane Chatty, David Thevenin & Jean-Luc Vinot. Artistic resizing : a technique for rich scale-sensitive vector graphics. In UIST '05 : Proceedings of the 18th annual ACM symposium on User interface software and technology, pages 201–210, New York, NY, USA, 2005. ACM.

[Dragicevic 06] Pierre Dragicevic, Stéphane Chatty, David Thevenin & Jean-Luc Vinot. Artistic resizing : a technique for rich scale-sensitive vector graphics. In SIGGRAPH '06 : ACM SIGGRAPH 2006 Sketches, page 6, New York, NY, USA, 2006. ACM.

[Draves 96] Scott Draves. Compiler generation for interactive graphics using intermediate code. In Olivier Danvy, Robert Glück & Peter Thiemann, editeurs, Partial Evaluation, volume 1110 of Lecture Notes in Computer Science, pages 95–114. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61580-6_6.

[Elmqvist 08] Niklas Elmqvist, Pierre Dragicevic & Jean-Daniel Fekete. Rolling the Dice : Multidimensional Visual Exploration using Scatterplot Matrix Navigation. IEEE Transactions on Visualization and Computer Graphics (Proc. InfoVis 2008), vol. 14, no. 6, pages 1141–1148, 2008.

[Esteban 95] Olivier Esteban, Stéphane Chatty & Philippe Palanque. Whizz'ed : A Visual Environment For Building Highly Interactive Software. In Proceedings of the Interact'95 conference, pages 121–126. Chapman & Hall, 1995.

[Fekete 04] Jean-Daniel Fekete. The InfoVis Toolkit. In Proceedings of the 10th IEEE Symposium on Information Visualization (InfoVis 04), pages 167–174, Austin, TX, October 2004. IEEE Press.

[Gourmel 10] Olivier Gourmel, Anthony Pajot, Mathias Paulin, Loïc Barthe & Pierre Poulin. Fitted BVH for Fast Raytracing of Metaballs. Computer Graphics Forum, vol. 29, no. 2, pages 281–288, May 2010.

[Green 89] T. R. G. Green. Cognitive dimensions of notations. In Proceedings of HCI'89, pages 443–460. Cambridge University Press, 1989.

[Hanrahan 90] Pat Hanrahan & Paul Haeberli. Direct WYSIWYG painting and texturing on 3D shapes. SIGGRAPH Comput. Graph., vol. 24, pages 215–223, September 1990.

[Harris 05] Mark Harris. Mapping computational concepts to GPUs. In ACM SIGGRAPH 2005 Courses, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[Hartmann 08] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang & Scott R. Klemmer. Design as exploration : creating interface alternatives through parallel authoring and runtime tuning. In Proceedings of the 21st annual ACM symposium on User interface software and technology, UIST '08, pages 91–100, New York, NY, USA, 2008. ACM.

[Heer 05] Jeffrey Heer, Stuart K. Card & James A. Landay. prefuse : a toolkit for interactive information visualization. In CHI '05 : Proceedings of the SIGCHI conference on Human factors in computing systems, pages 421–430, New York, NY, USA, 2005. ACM.

[Heer 10] Jeffrey Heer & Michael Bostock. Declarative Language Design for Interactive Visualization. IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis), 2010.

[Hirsch 82] S. Hirsch. An algorithm for automatic name placement around point data. The American Cartographer, vol. 9, no. 1, pages 5–17, 1982.

[Hölzle 94] Urs Hölzle & David Ungar. A third-generation SELF implementation : reconciling responsiveness with performance. In OOPSLA '94 : Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications, pages 229–243, New York, NY, USA, 1994. ACM.

[Huot 04] Stéphane Huot, Cédric Dumas, Pierre Dragicevic, Jean-Daniel Fekete & Gérard Hégron. The MaggLite post-WIMP toolkit : draw it, connect it and run it. In UIST '04 : Proceedings of the 17th annual ACM symposium on User interface software and technology, pages 257–266, New York, NY, USA, 2004. ACM.

[Hurter 09] Christophe Hurter, Benjamin Tissoires & Stéphane Conversy. FromDaDy : Spreading Aircraft Trajectories Across Views to Support Iterative Queries. IEEE Transactions on Visualization and Computer Graphics, vol. 15, no. 6, pages 1017–1024, 2009.

[Hurter 10] Christophe Hurter. Caractérisation de Visualisations et Exploration Interactive de Grandes Quantités de Données Multidimensionnelles. PhD thesis, Université de Toulouse, July 2010.

[Hutchins 85] Edwin L. Hutchins, James D. Hollan & Donald A. Norman. Direct manipulation interfaces. Hum.-Comput. Interact., vol. 1, pages 311–338, December 1985.

[Ingalls 97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace & Alan Kay. Back to the future : the story of Squeak, a practical Smalltalk written in itself. SIGPLAN Not., vol. 32, pages 318–326, October 1997.

[Ishizaki 03] Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu & Toshio Nakatani. Effectiveness of cross-platform optimizations for a java just-in-time compiler. In OOPSLA '03 : Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, pages 187–204, New York, NY, USA, 2003. ACM Press.

[Johnston 04] Wesley M. Johnston, J. R. Paul Hanna, Richard & J. Millar. Advances in dataflow programming languages. ACM Comput. Surv, vol. 36, pages 1–34, 2004.

[Kahn 74] G Kahn. The Semantics of a Simple Language for Parallel Programming. In Proc. IFIP Congress, Information Processing, 1974.

[Ko 04] A.J. Ko, B.A. Myers & Htet Htet Aung. Six Learning Barriers in End-User Programming Systems. In Visual Languages and Human Centric Computing, 2004 IEEE Symposium on, pages 199–206, sept. 2004.

[Kosinski 73] Paul R. Kosinski. A data flow language for operating systems programming. In Proceeding of ACM SIGPLAN - SIGOPS interface meeting on Programming languages - operating systems, pages 89–94, New York, NY, USA, 1973. ACM.

[Krall 98] Andreas Krall. Efficient JavaVM Just-in-Time Compilation. In IEEE Computer Society Press, editeur, Conference on Parallel Architectures and Compilation Techniques (PACT '98), pages 205–212, October 12-18 1998.

[Krasner 88] Glenn E. Krasner & Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. J. Object Oriented Program., vol. 1, pages 26–49, August 1988.

[Lattner 04] Chris Lattner & Vikram Adve. LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation. In CGO '04 : Proceedings of the international symposium on Code generation and

optimization, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[Lee 87] Edward Ashford Lee & David. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. IEEE Transactions on Computers, vol. 36, pages 24–35, 1987.

[Letondal 10] Catherine Letondal, Stéphane Chatty, Greg Phillips, Fabien André & Stéphane Conversy. Usability requirements for interaction-oriented development tools. In Proceedings of the PPIG 2010 Workshop on the Psychology of Programming., 2010.

[Limbourg 05] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon & Víctor López-Jaquero. USIXML : A Language Supporting Multi-path Development of User Interfaces. In Rémi Bastide, Philippe Palanque & Jörg Roth, editeurs, Engineering Human Computer Interaction and Interactive Systems, volume 3425 of Lecture Notes in Computer Science, pages 134–135. Springer Berlin / Heidelberg, 2005. 10.1007/11431879_12.

[Liu 10] Fuchang Liu, Takahiro Harada, Youngeun Lee & Young J. Kim. Real-time collision culling of a million bodies on graphics processing units. ACM Trans. Graph., vol. 29, pages 154 :1–154 :8, December 2010.

[Luboschik 08] M. Luboschik, H. Schumann & H. Cords. Particle-based labeling : Fast point-feature labeling without obscuring other visual features. Visualization and Computer Graphics, IEEE Transactions on, vol. 14, no. 6, pages 1237–1244, nov.-dec. 2008.

[Merlin 08] Bruno Merlin, Christophe Hurter & Raïlane Benhacene. A solution to interface evolution issues : the multi-layer interface. In CHI '08 extended abstracts on Human factors in computing systems, CHI EA '08, pages 2715–2720, New York, NY, USA, 2008. ACM.

[Mertz 00] Christophe Mertz, Stéphane Chatty & Jean-Luc Vinot. The influence of design techniques on user interfaces : the DigiStrips experiment for air traffic control. In In Proceedings of HCI Aero IFIP 13.5, 2000.

[Muller 07] M.J Muller. Participatory design : The third space in HCI. In Andrew Sears & Julie A. Jacko, editeurs, The Human Computer Interaction Handbook 2nd Edition, pages 1061–1081. CRC Press, 2007.

[Myers 90] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, David S. Kosbie, Edward Pervin, Andrew Mickish, Brad Vander Zanden & Philippe Marchal. Garnet : Comprehensive Support for Graphical, Highly Interactive User Interfaces. Computer, vol. 23, pages 71–85, November 1990.

[Myers 92] Brad A. Myers & Mary Beth Rosson. Survey on user interface programming. In Proc. of CHI, CHI '92, pages 195–202, New York, 1992. ACM.

[Myers 94] Brad A. Myers. Challenges of HCI design and implementation. Interactions, vol. 1, no. 1, pages 73–83, 1994.

[Myers 00] Brad A. Myers. Usability Issues in Programming Languages. Rapport technique, School of Computer Science, Carnegie Mellon University, 2000. Part of the Natural Programming Project.

[Myers 08] Brad Myers, Sun Young Park, Yoko Nakano, Greg Mueller & Andrew Ko. How Designers Design and Program Interactive Behaviors. In Proc. of IEEE VL/HCC '08, 2008.

[Nilsson 02] Henrik Nilsson, Antony Courtney & John Peterson. Functional Reactive Programming, Continued. In Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02), pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[Peercy 00] Mark S. Peercy, Marc Olano, John Airey & P. Jeffrey Ungar. Interactive multi-pass programmable shading. In SIGGRAPH '00 : Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pages 425–432, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[Preece 94] Jenny Preece, Yvonne Rogers, Helen Sharp, David Benyon, Simon Holland & Tom Carey. Human-computer interaction. Addison-Wesley Longman Ltd., Essex, UK, UK, 1994.

[Raynal 07] Mathieu Raynal, Jean-Luc Vinot & Philippe Truillet. FishEye keyboard : Whole Keyboard Displayed on Small Device. In Proceedings of the poster session of the 20th ACM UIST Symposium (UIST '07), Oct 2007.

[Rekimoto 97] Jun Rekimoto. Pick-and-drop : a direct manipulation technique for multiple computer environments. In Proceedings of the 10th annual ACM symposium on User interface software and technology, UIST '97, pages 31–39, New York, NY, USA, 1997. ACM.

[Schmucker 87] K. J. Schmucker. Human-computer interaction. chapitre MacApp : An application framework, pages 591–594. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[Shneiderman 83] B. Shneiderman. Direct Manipulation : A Step Beyond Programming Languages. Computer, vol. 16, pages 57–69, August 1983.

[Snyder 03] Carolyn Snyder. Paper prototyping : The fast and easy way to design and refine user interfaces (the morgan kaufmann series in interactive technologies). Morgan Kaufmann, April 2003.

[Steele 93] Jr. Guy L. Steele & Richard P. Gabriel. The evolution of Lisp. SIGPLAN Not., vol. 28, pages 231–270, March 1993.

[Stuerzlinger 06] Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips & Nicolas Roussel. User interface faç ;ades : towards fully adaptable user interfaces. In Proceedings of the 19th annual ACM symposium on User interface software and technology, UIST '06, pages 309–318, New York, NY, USA, 2006. ACM.

[Sutherland 63] Ivan Edward Sutherland. Sketchpad : A man-machine graphical communication system. In Proceedings of AFIPS Spring Joint Computer Conference, pages 329–346, 1963.

[Tabart 07] Gilles Tabart. Méthodes et outils pour la conception et la vérification du rendu des IHM. In IHM '07 : Proceedings of the 19th International Conference of the Association Francophone d'Interaction Homme-Machine, pages 261–264, New York, NY, USA, 2007. ACM.

[Tissoires 08] Benjamin Tissoires & Stéphane Conversy. Graphic Rendering Considered as a Compilation Chain. In Design Specification and Verification of Interactive Systems (DSV-IS), numéro 5136 in LNCS, pages 267–280. Springer, 2008.

[Tissoires 09] Benjamin Tissoires, Jean-Luc Vinot & Stéphane Conversy. Hayaku : maximiser le pouvoir d'expression du concepteur et l'efficacité de rendu de scènes graphiques interactives. In IHM '09 : Proceedings of the 21st International Conference on Association Francophone d'Interaction Homme-Machine, pages 325–328, New York, NY, USA, 2009. ACM.

[Tissoires 11] Benjamin Tissoires & Stéphane Conversy. Hayaku : Designing and Optimizing Finely Tuned and Portable Interactive Graphics with a Graphical Compiler. In Proceedings of the 3nd ACM SIGCHI symposium on Engineering interactive computing systems, EICS '11, New York, NY, USA, 2011. ACM.

[Uims 92] Uims. A metamodel for the runtime architecture of an interactive system : the UIMS tool developers workshop. SIGCHI Bull., vol. 24, pages 32–37, January 1992.

[Vander Zanden 94] Bradley T. Vander Zanden, Brad A. Myers, Dario A. Giuse & Pedro Szekely. Integrating pointer variables into one-way constraint models. ACM Trans. Comput.-Hum. Interact., vol. 1, no. 2, pages 161–213, 1994.

[Vander Zanden 01a] Bradley T. Vander Zanden & Richard Halterman. Using model dataflow graphs to reduce the storage requirements of constraints. ACM Trans. Comput.-Hum. Interact., vol. 8, no. 3, pages 223–265, 2001.

[Vander Zanden 01b] Bradley T. Vander Zanden, Richard Halterman, Brad A. Myers, Rich McDaniel, Rob Miller, Pedro Szekely, Dario A. Giuse & David Kosbie. Lessons learned about one-way, dataflow constraints in the

Garnet and Amulet graphical toolkits. ACM Trans. Program. Lang. Syst., vol. 23, no. 6, pages 776–796, 2001.

[Wilkinson 99] Leland Wilkinson. The grammar of graphics. Springer-Verlag New York, Inc., New York, NY, USA, 1999.

Résumé

Concevoir une interface hautement interactive nécessite de faire un compromis entre efficacité du moteur de rendu et puissance d'expression offerte au designer graphique. Ainsi, les boîtes à outils WIMP classiques sont performantes en terme de rendu visuel, mais ne permettent pas de modifier finement le visuel des éléments graphiques ou de modifier leur comportement. Pour améliorer le processus de développement des applications graphiques interactives et leur réutilisabilité, il faut séparer la description de la partie interactive de son implémentation et de ses optimisations. Cette thèse présente différents moyens de séparer la description des optimisations. Tout d'abord, elle présente Hayaku, une boîte à outils reposant sur un compilateur graphique. Hayaku permet au designer d'interaction de produire lui-même l'application interactive finale, favorisant ainsi les itérations dans le développement. Nous nous sommes ensuite intéressés à la simplification de la description pour l'utilisateur final lors de la production de visualisations de grandes quantité de données avec le logiciel FromDaDy. Déporter la tâche de création d'images sur le processeur graphique permet de présenter un modèle simple à l'utilisateur. Enfin, nous avons simplifié la description d'un algorithme lent (le placement d'étiquettes sur une image) en utilisant la mémoire graphique au lieu de calculs linéaires sur le processeur central. Ceci permet de supprimer les optimisations complexes et améliore significativement les temps de calculs.

Mots-clés: Interaction homme-machine, optimisation, graphismes, flot de données.

Abstract

Designing a highly interactive application implies to make a compromise between the efficiency of the graphical renderer and the expressive power offered to the graphical designer. Thus, classical WIMP toolkits are powerful in terms of rendering, but do not allow the user to finely tune the final rendering of the graphics, or to change their behavior. In order to enhance the process of designing graphical applications and the ability to reuse them, the description of the interactive part has to be separated from the implementation and the optimizations. This thesis presents different way to separate description from optimizations. First, it presents Hayaku, a graphical toolkit that relies on a graphical compiler. Hayaku allows the interaction designer to produce himself the final interactive application. It helps an iterative development cycle. Then, we have worked on the simplification of the description for the final user when she has to create visualizations. We introduced the software FromDaDy to support this creation of large amount of data. Moving the image creation task on the GPU allows us to show a simpler model to the final user. Finally, we have simplified the description of a low algorithm (point based labeling placement algorithm) by using the GPU memory instead of the CPU. This allows us to suppress early complex optimizations and enhance the performances.

Keywords: Computer-human interaction, optimizations, graphics, data-flow.