Hayaku : maximiser le pouvoir d'expression du concepteur et l'efficacité de rendu de scènes graphiques interactives

Benjamin Tissoires, Jean-Luc Vinot, Stéphane Conversy

Hayaku : Maximiser le pouvoir d'expression du concepteur et l'efficacité de rendu de scènes graphiques interactives

Benjamin Tissoires[1,2,3]
tissoire @ cena.fr

[1] DSNA DTI R&D
7, avenue Edouard Belin
31055, Toulouse,
France

Jean-Luc Vinot[1,3]
vinot @ cena.fr

[2] ENAC
7, avenue Edouard Belin
31055, Toulouse,
France

Stéphane Conversy[2,3]
stephane.conversy @ enac.fr

[3] Université de Toulouse, IRIT
118 Route de Narbonne,
31062 Toulouse CEDEX 9,
France

RESUME

Les concepteurs d'outils de développement d'interfaces hautement interactives doivent souvent faire un compromis entre efficacité du moteur de rendu et puissance d'expression offerte au designer graphique. Afin d'éliminer ce compromis, nous proposons une méthode instrumentée de conception d'applications hautement interactives basée sur l'emploi d'éditeurs graphiques, d'un ensemble de langages de description conceptuelle, graphique et comportementale, et d'une toolkit, Hayaku, qui encapsule un compilateur graphique. Hayaku permet de maximiser les performances de l'application développée, ce qui d'une part autorise un rendu plus fiable grâce à des techniques de rendu appropriées, et d'autre part autorise le designer a employer la palette entière des outils d'édition mis à sa disposition. La véracité du rendu, la finesse de contrôle, la richesse d'expression, et l'efficacité de mise en oeuvre des outils permettent au concepteur d'explorer efficacement de nouvelles représentations graphiques dynamiques facilement contrôlables depuis l'application finale.

MOTS CLES : Architecture logicielle, Boîtes à outils d'IHM, Compilation graphique.

ABSTRACT

Designing a general toolkit for interactive software implies to make a choice between the speed of the renderer and the power of expression of the user. We propose an instrumented method to concieve interactive software based on the use of graphical editors, conceptual languages and a toolkit, Hayaku, that encapsulates a graphical compiler. Hayaku allows to maximize the graphical performances of the application, thus authorizing to have a better graphical renderer, and authorizing the user to use all of his editor tools. The truth of the graphics, the control easyness, the power of expression and the efficiency of the use of the designer's tools allows the conceptor to efficiently explore new dynamic graphical representations that are easy to control.

CATEGORIES AND SUBJECT DESCRIPTORS: H5.2 [Information Interfaces]: User Interfaces; D2.11 [Software Engineering]: Software Architectures.

GENERAL TERMS: Design.

KEYWORDS: GUI architectures, GUI toolkits, Graphical compiler.

INTRODUCTION

Les concepteurs d'outils de développement d'interfaces hautement interactives doivent souvent faire un compromis entre efficacité du moteur de rendu et puissance d'expression offerte au designer graphique. Ainsi, les boîtes à outils WIMP classiques (GTK, Qt, Swing) sont performantes en terme de rendu visuel, mais elles ne permettent pas au designer graphique de modifier finement le visuel des éléments graphiques ou encore de modifier leur comportement. Intuikit[3], introduit un certain degré d'expressivité pour le designer graphique qui peut alors créer plus librement les composants de la scène visuelle. Cependant, la toolkit repose essentiellement sur des changements d'états et impose un ensemble limité de comportements lors des transitions entre ces états. MaggLite[4] utilise des graphes de scènes combinés à des flots de données gérant les entrées. Si ces deux dernières toolkits apportent un plus grand pouvoir d'expression, l'application finale est pénalisée par l'efficacité d'un moteur de rendu graphique peu optimisable.

Afin d'éliminer le compromis entre richesse de description et efficacité du rendu, nous proposons une méthode instrumentée de conception d'applications hautement interactives basée sur l'emploi d'éditeurs graphiques, d'un

ensemble de langages de description conceptuelle, graphique et comportementale, et d'une toolkit, Hayaku, qui encapsule un compilateur graphique[6]. A l'instar des outils Intuikit ou MaggLite, cette méthode permet en premier lieu de renforcer l'implication du designer graphique dans le processus de développement de l'application interactive. De plus, grâce à l'utilisation du compilateur graphique, Hayaku permet de maximiser les performances de l'application développée, ce qui d'une part autorise un rendu plus fiable grâce à des techniques appropriées, et d'autre part autorise le designer a employer pleinement ses outils d'édition. Après la description des différents éléments de Hayaku, nous présentons la réalisation d'une application interactive réalisée avec cette méthode.

DESCRIPTION DE HAYAKU

Hayaku repose sur le principe de la compilation de scènes graphiques[6] qui consiste à réutiliser les concepts de compilation dans le domaine du graphisme. Ainsi, les personnes en charge de concevoir et de programmer l'application interactive s'abstraient du moteur de rendu, de même qu'un programmeur actuel ne travaille plus en assembleur. Pour intégrer ce compilateur dans une boîte à outil graphique de haut niveau, il faut intégrer plusieurs éléments (voir Figure 1).

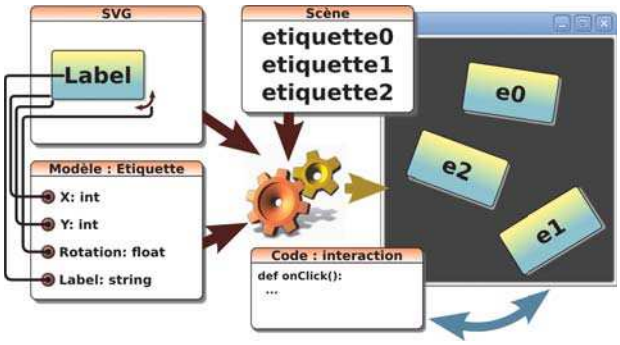

Figure 1 **:** Le schéma de principe de la toolkit.

Le langage de description des éléments graphiques. Afin de bénéficier des connaissances et outils des designers graphiques, nous nous sommes basés sur le langage vectoriel graphique SVG (Scalable Vector Graphics), qui dispose d'un ensemble de primitives graphiques complet, et qui est reconnu et manipulable par ces outils. Ceci garanti que le designer conservera son pouvoir d'expression.

Le langage d'abstraction de la scène. Le langage d'abstraction de la scène, ou langage *conceptuel*, permet de décrire une interface entre la partie graphique de l'application, et sa partie comportementale (équivalent au COG de [1]). La description de ce langage utilise la syntaxe JSON (JavaScript Object Notation) [1], un langage minimaliste spécialisé dans la description d'entités.

[1] IETF RFC 4627

La scène. La construction de la scène utilise le langage d'abstraction, en instanciant les différents objets la composant. Le langage utilisé pour cette tâche utilise aussi la syntaxe JSON.

La jonction de ces précédents langages. Les *connections* (les traits qui relient le modèle au SVG sur la Figure 1) sont la "glue" qui permet au compilateur de comprendre comment relier les diffférentes variables ou objets aux éléments graphiques.

Génération de Code

Une fois que tous ces éléments sont fournis, le compilateur graphique génère le code final correspondant. Nous rappelons ici les différentes étapes[6]. Tout d'abord, le compilateur convertit le graphisme exprimé en SVG en un fichier C contenant des appels OpenGL. Cette étape se traduit par une suite de transformations entre plusieurs langages intermédiaires. Le fichier C est ensuite compilé par un compilateur C. En plus du code C de rendu graphique, le compilateur graphique génère une description du flot de données[7]. Celle-ci permet de répercuter les modifications du modèle lors de l'exécution de l'application vers le code C généré. En effet, les paramètres des appels OpenGL qui sont susceptibles d'être modifiés au cours de l'exécution du programme dépendent du flot de données. Ainsi, si la couleur d'un objet est susceptible d'être modifiée, les paramètres du *glColor* associé pointent vers les variables référentes aux couleurs. Une modification du contenu de ces variables se répercute directement sur le rendu visuel. Ce flot de données est écrit dans un fichier C qui est à son tour compilé par gcc. Puisque ces deux fichiers sont compilés par un compilateur C, le code sera exécuté en mode "natif", sans interprétation, ce qui maximisera les performances de l'application.

L'exécutable

L'application en elle-même consiste en un petit exécutable comprenant une boucle de rendu et d'attente des entrées, et qui charge dynamiquement les deux fichiers de rendu et de flot de données. La gestion du moteur d'exécution a été revue depuis [6]. La précédente version de l'exécutable devait gérer des listes de pointeurs de fonctions. Cette gestion imposait un risque de surcoût important si trop de code mort était introduit. C'est pourquoi un compilateur "juste-à-temps" avait été implémenté. La nouvelle version du run-time s'est considérablement amoindri afin d'augmenter les performances de rendu. La gestion de l'exécutable ne consiste plus qu'à appeler des fonctions générées en C par la partie statique du compilateur graphique. Générer le code mort étant maintenant moins coûteux en termes de performances, le compilateur "juste-à-temps", et donc le surcoût introduit, a pu être supprimé.

La phase de compilation de la scène graphique est transparente pour l'utilisateur. Lorsque l'utilisateur lance l'application, le compilateur graphique vérifie en premier lieu s'il y a eu des modifications concernant les éléments

graphiques depuis la dernière compilation en comparant les dates de modifiaction des différents fichiers. Si modifications il y a eu, le compilateur ne recompile que les fichiers modifiés. Une fois ces fichiers recompilés, l'exécutable les charge et peut lancer la boucle de rendu et d'attente des entrées.

Écriture du Code du Comportement

Pour rendre l'application interactive, il reste à fournir le comportement graphique à la couche présentation. En concevant Hayaku, nous n'avons pas souhaité contraindre les utilisateurs avec un langage particulier. Pour cela, nous utilisons un langage existant, à savoir Python, mais nous exportons aussi une interface vers l'extérieur à l'aide du bus logiciel Ivy[2]. De ce fait, le comportement peut être exprimé dans n'importe quel langage supporté par Ivy. Pour modifier le rendu visuel de l'application, le designer utilise directement les variables déclarées dans le modèle. Il peut donc gérer comme il le souhaite sa scène. Pour gérer les évènements, il peut s'abonner aux modifications de n'importe quelle variable.

UN TEST D'USAGE

Nous avons expérimenté la réalisation d'une application test par un designer graphique utilisant Hayaku afin d'illustrer la méthode proposée.

Figure 2 **:** 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 redimensionnable (cf Figure 2). 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 saisie. Enfin, le clavier intègre une fonction "expanding keyboard"[5] pour les affichages de faible résolution.

Le design graphique du clavier test (type $2D^1/_2$) utilise une description uniquement vectorielle des composants, pour garantir une meilleure qualité au redimensionnement et met en oeuvre 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.

Les Étapes de la Réalisation

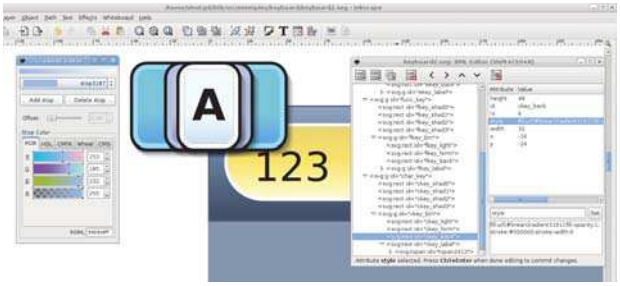

Figure 3 **:** Le fichier SVG de description des éléments graphiques du clavier, réalisé avec Inkscape.

Le travail d'élaboration graphique a été réalisé avec Inkscape (cf Figure 3). Dans un premier fichier SVG, le designer a créé tout d'abord une touche du clavier par couches graphiques superposées. 8 calques sont définis dont un calque *lumière* activé lors de l'appui de touche (événement curseur *Press*) et un groupe de calques pour réaliser l'ombre portée. Ces calques sont nommés et groupés en un unique composant SVG. Pour tester la composition globale, les touches sont ensuite dupliquées, organisées et modifiées pour générer une maquette de principe du clavier. 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 etl'arrière plan sont sauvegardés dans un nouveau fichier SVG. Pour faciliter l'utilisation des composants touches et leurs transformations géométriques, 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. Une super classe *Key* a été définie pour prendre en charge les propriétés et méthodes communes aux touches. 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, relative à d'autres paramètres (semblable aux contraintes de Garnet [7]) ou calculée par méthodes. 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.

La description JSON de la scène (ensemble des composants graphiques de l'interface) a été jugé "fastidieuse" 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 section _main_ de création de l'application, a été réalisé en s'inspirant de la structure d'un autre fichier exemple. 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, ou la fonction "fisheye" (expanding keyboard) des touches...

RÉSULTATS ET DISCUSSION

Le premier point remarquable de la toolkit, du point de vue de l'utilisateur, est la fiabilité du rendu. La toolkit utilisant un compilateur graphique, le code de rendu final ne souffre pas de compromis entre rapidité et puissance d'expression. 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, ce qui lui permet de contrôler finement le comportement graphique de ses objets d'interface. 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. L'utilisation du compilateur graphique permet en outre d'obtenir une vitesse de rendu élevée (60 images par secondes sur une carte graphique d'entrée de gamme et 125 i.p.s. sur une carte graphique haut de gamme de 2008 pour les 325 objets graphiques de la scène, sans compter les textes dont le rendu est entièrement vectoriel). Enfin, le développement de la partie gérant le comportement de l'application est facilité puisque le compilateur graphique ne recompile que si il y a eu des changements dans la partie graphique.

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. Enfin, le code final n'est toujours pas exportable dans une autre application. Le flot de données est en effet dissocié du code du rendu généré et contient l'ensemble monolithique de toutes les données.

Afin de résoudre les problèmes précédents, nous pensons orienter nos travaux dans les directions suivantes. Tout d'abord, du point de vue utilisateur, un éditeur pour les fichiers JSON qui garantirait la cohérence des fichiers semble absolument nécessaire. Du point de vue du moteur de rendu, une granularité plus fine du contrôle de la compilation graphique (certaines parties du code généré n'ont pas besoin d'être recompilées, comme les polices de caractères par exemple) est nécessaire. Cela permettra de pallier au problème de la gestion mémoire des compilations intermédiaires. Enfin, la version actuelle du compilateur graphique ne peut pas créer des objets dynamiquement. Une solution serait d'instancier dynamiquement des objets déjà compilés.

BIBLIOGRAPHIE

1. Blanch, R., Beaudouin-Lafon, M., Conversy, S., Jestin, Y., Baudel, T., and Zhao, Y. P. Concevoir des applications graphiques interactives distribuées avec indigo. *Revue d'Interaction Homme-Machine*, 7(2):247–274, 2006.

2. Buisson, M., Bustico, A., Chatty, S., Colin, F.-R., Jestin, Y., Maury, S., Mertz, C., and Truillet, P. Ivy: un bus logiciel au service du développement de prototypes de systèmes interactifs. In *IHM '02: Proceedings of the 14th French-speaking conference on Human-computer interaction*, pages 223–226. ACM, 2002.

3. Chatty, S., Sire, S., Vinot, J.-L., Lecoanet, P., Lemort, A., and Mertz, C. 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.

4. Huot, S., Dragicevic, P., and Dumas, C. Flexibilité et modularité pour la conception d'interactions: le modèle d'architecture logicielle des graphes combinés. In *IHM '06:Proceedings of the 18th French-speaking conference on Human-computer interaction*, pages 43–50. ACM, 2006.

5. Raynal, M., Vinot, J.-L., and Truillet, P. Fisheye keyboard: Whole keyboard displayed on small device. In *Proceedings of the poster session of the 20th ACM UIST Symposium (UIST '07)*, Oct 2007.

6. Tissoires, B., and Conversy, S. Graphic Rendering Considered as a Compilation Chain. In *Design Specification and Verification of Interactive Systems (DSV-IS)*, number 5136 in LNCS, pages 267–280. Springer, 2008.

7. Vander Zanden, B. T., Halterman, R., Myers, B. A., McDaniel, R., Miller, R., Szekely, P., Giuse, D. A., and Kosbie, D. Lessons learned about one-way, dataflow constraints in the garnet and amulet graphical toolkits. *ACM Trans. Program. Lang. Syst.*, 23(6):776–796, 2001.