Langage de programmation

Georges Dupéron

17 mai 2010

Table des matières

Première partie

Objectif

1 Sujet : Etude d'un paradigme de programmation : les langages graphiques à Dataflow

Il y a environ 20 ans, le langage et le système d'exploitation FORTH avaient été mis au point, avec pour but de créer un environnement totalement personnalisé pour chaque utilisateur. La particularité de FORTH était qu'il ne possédait pas de mots-clés, ou instructions figées, et que chaque utilisateur était en mesure de définir lui-même ses propres primitives, voire redéfinir ses primitives... à l'infini.

L'échec de FORTH est venu, entre autres de la nécessité d'échanger des programmes entre utilisateurs, et des conflits dus à l'homonymie (même nom, fonction différente) et à la synonymie (même fonction, noms différents). Les fonctions n'étaient pas toujours documentées, ce qui fait qu'un même programmeur ne pouvait pas faire exécuter, à un intervalle de temps relativement court, 2 fois le même programme...

Pour autant, le côté adaptatif et souple de FORTH aurait été largement plébiscité s'il n'y avait eu cette difficulté. Une manière de contourner ce genre de conflit, dû à une représentation symbolique textuelle trop fortement contrainte par la syntaxe, est de se pencher vers une programmation qui privilégie les flux de données sur les actions à réaliser, et vers des bases graphiques plutôt que textuelles, laissant donc à chaque utilisateur la liberté de définir ses actions, et préservant en revanche les flux.

L'idée de ce projet est de proposer une première architecture de compilateur ou d'interpréteur de premier niveau pour illustrer ce paradigme, et tenter d'en évaluer les propriétés. Ce projet nécessite un excellent niveau en programmation et un goût prononcé pour l'écriture de compilateurs.

2 ...

Le but de ce projet sera donc dans un premier temps de définir un langage de programmation visuel utilisant le paradigme du dataflow[1], et n'ayant pas de primitives fixes. Dans un second temps, nous implémenterons un EDI[2] permettant de créer des programmes dans ce langage, et de les interpréter.

[1] `http://en.wikipedia.org/wiki/Dataflow`
[2] Environnement de Développement Intégré

Deuxième partie
Définition du langage

3 Étude de l'existant

3.1 FORTH

– macro-expansion

3.2 Dataflow

Le paradigme du dataflow (flux de données) est connu des concepteurs de langages de programmation depuis longtemps. Il a été utilisé avec succès dans certains domaines, principalement des domaines intéressant les non-programmeurs.

3.2.1 Graphisme

Le logiciel Quartz Composer sous MacOS permet la création d'images vectorielles animées et interactives sous MacOS de manière graphique : On applique des filtres graphiques, représentés par des boîtes, aux résultats d'autres filtres, en connectant les boîtes entre elles. Dans World Machine, ces filtres sont des actions physiques (érosion, soulèvement) et permettent de générer des cartes de hauteur (heightmaps), qui sont utilisées pour modéliser des terrains en 3D.

3.2.2 Musique

L'interface de certains logiciels de musique s'inspirent de l'architecture des synthétiseurs modulaires[1] (ces grosses boîtes avec pleins de prises jack qu'on relie avec des câbles : chaque prise jack est une entrée ou une sortie d'un module, les câbles sont les connexions). Un des premiers synthétiseurs virtuels, Max/MSP, utilisait cette analogie. D'autres logiciels similaires lui ont succédé : PureData (qui est aussi un langage de programmation généraliste), Alsa Modular Synth, ...

3.2.3 Mesures scientifiques

LabView permet aux scientifiques de procéder à des traitements sur les signaux et les données acquises depuis un ordinateur, grâce à un langage de programmation graphique utilisant le paradigme du dataflow.

3.2.4 Traitement de signaux

Force est de constater que tous les exemples cités ci-dessus sont des cas particuliers de traitement de signal (image, son, signaux provenant d'appareils de mesure). Le paradigme du dataflow devrait être tout aussi efficace pour construire des programmes conventionnels : les signaux d'entrée sont les évènements

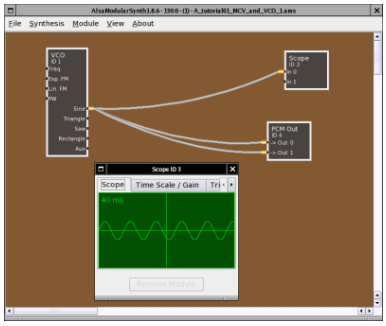

FIG. 1: Alsa Modular Synth[2]

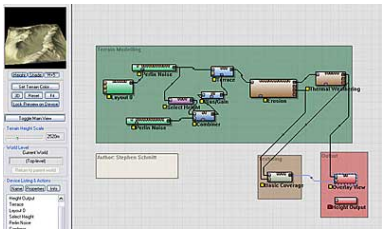

FIG. 2: World Machine[3]

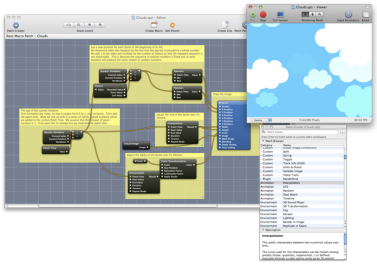

FIG. 3: Quartz Composer[4]

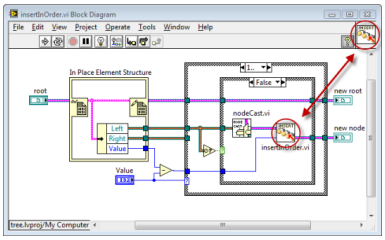

FIG. 4: LabView[5]

FIG. 5: Captues d'écrans de quelques langages de programmation utilisant le paradigme du dataflow.

provoqués par l'utilisateur (clic de souris, appui sur le clavier), ceux de sortie sont les retours (écran, haut-parleurs, ...). Cependant, les langages graphiques n'ont eu que peu de succès auprès de la communauté des programmeurs, et les raisons de ce rejet méritent d'être étudiées[3].

Récemment, un chercheur a mis en place un sondage auprès des programmeurs qui devrait à terme permettre de savoir quels langages correspondent le mieux à quelles affirmations, selon les programmeurs[6]. Ces affirmations sont du type «Ce langage est facile à utiliser» ou «Ce langage à une bonne communauté». Les affirmations qui obtiennent les moins bons scores pour un langage donné indiquent en général les défauts de ce langage. J'ai donc contacté l'auteur de ce sondage pour lui demander d'ajouter des langages de programmation visuels pendant que le sondage est encore ouvert, afin de pouvoir étudier par la suite ces résultats.

[3]Peut-être que c'est simplement que les langages textuels, c'est pour les programmeurs avec du poil sur le torse, et les langages graphiques sont pour les mauviettes...

3.3 Langages spécifiques à un domaine

Il est fréquent que des programmeurs créent des «langages spécifiques à un domaine» (DSL, Domain Specific Language) pour répondre au besoin d'exprimer facilement des instructions et notions très fortement liées à un certain domaine, sans être encombrés par la syntaxe rigide, le vocabulaire limité et le manque d'expressivité de leur langage habituel. Les fichiers de configuration sont en général écrits dans un DSL par exemple. Mais les DSL ont de grosses limitations : bien qu'ils soient adaptés à l'expression d'idées dans ce domaine, ces langages ne possèdent pas l'expressivité nécessaire à des opérations plus complexes : souvent, ces DSL ne sont complets au sens de turing et n'ont pas accès aux bibliothèques de fonctions des autres langages.

C'est pour ces raisons que les DSL sont de plus en plus laissés un peu de côté : La configuration de GRUB 2 est toujours dans un DSL, mais elle est générée dynamiquement par des scripts shell[7], la configuration du gestionnaire de fenêtres awesome est écrite en Lua[8], beaucoup de logiciels utilisent python comme langage de script, plutôt que de créer un langage de script spécifique à l'application.

D'un autre côté, le problème des langages de programmation généralistes est qu'ils permettent d'augmenter le vocabulaire disponible, par l'ajout de fonctions et la création de variables, mais ils ne permettent pas de modifier la grammaire utilisé (la structure du code). En réalité, de nombreux langages permettent cela par le biais de macros, mais cette approche est assez limitée : l'écriture de macros est en général fastidieuse – beaucoup plus difficile que l'écriture de fonctions – et l'utilisation de macros est néanmoins soumise à un grand nombre de contraintes syntaxiques du langage hôte (parenthèses, accolades, ...). De plus le système de macros de certains langages, comme le C, n'a pas la puissance d'un vrai langage : pas de récursivité des macros, pas de structures conditionnelles à l'intérieur d'une macro.

Il y a donc des avantages dans les deux approches, mais aucune ne permet de satisfaire complètement les besoins de facilité d'expression et de puissance du langage. Nous proposons une solution alternative : permettre l'écriture d'un programme sous n'importe quelle forme visuelle (grammaire libre), dans un langage graphique orienté dataflow. Nous imposons quelques contraintes, afin de garder une certaine cohérence :

- Les sous-programmes sont représentés par des éléments graphiques (blocs) ;
- Les paramètres d'entrées de ces sous-programmes sont des entités visuelles qu'on peut connecter entre elles (ports) ;
- Les valeurs de ces paramètres sont fournies à travers des liens qui relient les ports ;

Ces règles laissent toutefois une très grande liberté, les blocs peuvent être rectangulaires, ronds ou de n'importe quelle forme. Ils pourraient par exemple afficher les calculs mathémathiques qu'ils contiennent non pas sous l'apparence d'instructions mais d'une formule mathémathique. LabView utilise cette technique.

Il est même possible que certains blocs aient une apparence invisible. Par exemple les opérations de conversion de types (cast) pourraient être masquées,

et ne s'afficher que sur le lien entre la source du transtypage et sa destination.

Avec ces règles, les ports n'ont pas besoin d'être affichés en permanence. Ils peuvent par exemple être dans une boîte de dialogue de configuration dui apparaît lorsqu'on double-clique sur le bloc correspondant (World Machine utilise cette approche, et laisse aussi les ports visibles en mode «réduit»).

Les liens peuvent être annotés, regrouppés en bus, etc. Comme on peut l'imaginer, avec un minimum d'effort de cohérence, ces libertés sont suceptibles de rendre les programmes plus lisibles (non-affichage des éléments qui ne sont pas importants à la comprhéension du programme) et plus faciles à écrire (les instructions (blocs) spécialisées ont une interface d'utilisation adaptée).

3.4 Environnement de développement intégré

> Clearly good design is as important for visual languages as for textual ones. Furthermore, the effectiveness of a visual language indeed any precise language for specifying structures, depends to a large extent on the quality of the editor, browser, interpreter, debugger and other tools supplied by the language implementation
>
> Christopher C. Risley and Trevor J. Smedley [9]

Les environnements de développement intégré jouent un rôle crucial pour les langages de programmation visuels. En effet, sans la possibilité de construire un programme par une séquence (obscure) de caractères, la convivialité du langage est directement liée aux outils que propose l'EDI. Ces EDI n'ont en général que peu en commun avec les ceux destinés aux langages textuels, et se présentent plus comme des logiciels de dessin vectoriel que comme des éditeurs de texte.

Certaines questions sont récurrentes dans la conception d'EDI pour des langages visuels. Une des plus importante est la gestion de l'espace à l'écran : les primitives graphiques sont décorées de bordures, et d'autres éléments visuels, qui occupent beaucoup d'espace à l'écran, réduisant ainsi la quantité d'informations affichées. Ce problème est plus communément connu sous le nom de «Deutsch Limit»[10] :

> Well, this is all fine and well, but the problem with visual programming languages is that you can't have more than 50 visual primitives on the screen at the same time. How are you going to write an operating system?

Cependant, dans le cas d'un affichage textuel, le cerveau humain n'est pas capable d'ingurgiter en temps réel toute l'information à l'écran[4]. L'affichage sous forme graphique permet au contraire de clarifier et de nettoyer les informations à l'écran, on peut donc supposer que le cerveau est capable d'analyser plus rapidement du «code» graphique que son équivalent textuel (un dessin vaut mille mots, dit-on).

[4]Sur mon écran, on peut loger plus de 60 lignes de texte verticalement, et ce sur 2 colonnes. Allez donc analyser 120 lignes de C d'un seul coup.

De plus les programmeurs passent beaucoup de temps à chercher tel ou tel morceau de code ou fonction. L'EDI Code Bubbles[11] apporte une solution à ce problème pour les langages textuels, en fournissant une interface dans laquelle le programmeur manipule un gand nombre de fragments de code assez court, et a la possibilité d'en ouvrir facilement de nouveaux de manière contextuelle (ouvrir la définition de cette fonction, ouvrir la documentation de celle-là). Nous avons repris ce principe pour les langages visuels : Dans la définition d'un bloc, il est possible d'ouvrir sur place ou à côté les définitions des sous-blocs.

Une autre solution consiste à utiliser une interface «fish-eye» : les objets au centre de l'écran ont une taille normale, et sont de plus en plus petitsà mesure qu'ils approchent du bord. Une variante est l'interface «zoom», utilisée notamment dans certains jeux vidéos[12] pour permettre au joueur de focaliser son attention sur les ennemis les plus proches, ou bien avoir une vue d'ensemble, et dans le logiciel de prises de notes Project Cecily[13].

Plusieurs recherches ont été menées concernant l'utilisation de l'espace à l'écran par les langages visuels, une bibliographie référencant plusieurs papiers sur ce sujet est disponible à [14].

3.5 Manque d'expressivité des langages de programmation existants

- Design patterns
- Domain Specific Language

3.6 Réutilisabilité

- qqch en commun avec prog par composants.
- unix pipes
- navigateur web unix pipes
- Mais cette approche est très limitée, car le seul type de données qu'on peut échanger, c'est un flux d'octets augmenté d'un marqueur de fin (EOF). Pas de communication hors-bande, pas de structuration des données. -> pbs d'encapsulation (au sens des protocoles de communication), d'échappement (abstraction / niveaux méta, dark tower of meta levels)
- Héritage partiel (cf. tiddlywiki)

3.7 Intérêt des langages visuels

- Control flow graphs.
- Taxonomie : étudier l'utilité des différentes catégories.
- pas de conflits de nommage (SCID Source Code In Database).
- Une infinité de DSL à disposition. Par ex. on peut faire en sorte que dans un outil de conception d'interfaces utilisateur, les composants graphiques (fenêtre, bouton, etc.) ne soient pas de simples images, mais *soient* les fonctions ou objets correspondants.

FIG. 6: Aziz faces the Dark Tower of Meta-levels

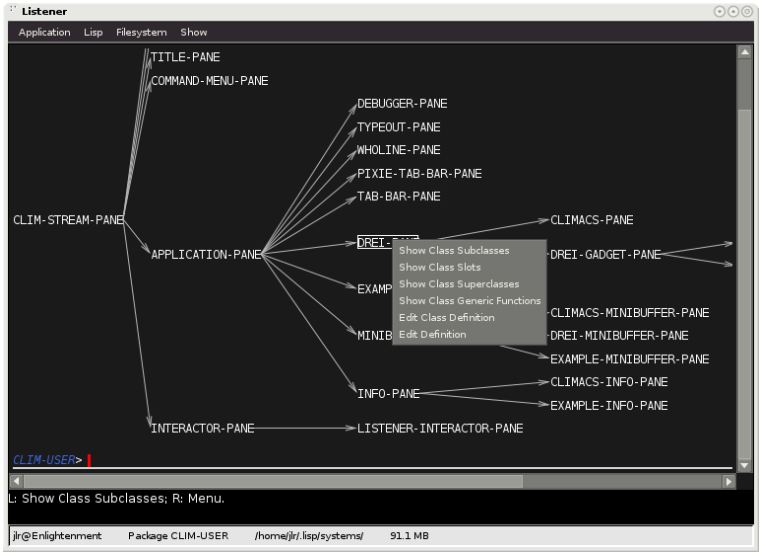

FIG. 7: Graphe d'héritage des classes en Lisp avec McCLIM

3.8 divers...

Pourquoi dataflow peu utilisé ? (chest hair ?, http ://therighttool.hammerprinciple.com/, contacter l'auteur pour qu'il ajoute des VPL)

Des recherches ont montré que dans le cadre des langages de programmation

visuels, l'éditeur jouait un rôle aussi important que le langage lui-même.

Graphes avec noeuds étiquetés et arcs éiquetés à chaque extrémité (et sur l'arc lui-même) => graphes généralisés. Le programme choisit son affichage.

Preuve de complétude de Turing : un graphe = des noeuds connectés par des arcs, un noeud = l'arc NULL ou qqch du genre. // $\lambda(\lambda(\dots))$.

L'interface utilisateur devient alors la même chose (fenêtres = blocs avec un affichage particulier).

Preuves

4 Formalisation du langage

Dans cette section, nous allons essayer de trouver quelle est la nature, l'essence d'un programme, de manière à

4.1 Programme en dataflow

Examinons un programme simple exprimé dans le paradigme du dataflow :

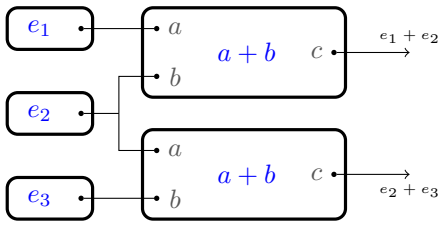

FIG. 8: Un programme simple en dataflow.

Ce programme a trois entrées (e_1, e_2, e_3), et deux sorties ($e_1 + e_2$ et $e_2 + e_3$).

4.2 Caractéristiques du programme

La question à se poser maintenant est :

Qu'est-ce qui définit ce programme ?

Ou, comme le disait un des collègues de Joe Armstrong [15] :

> « A program is a black box. It has inputs and it has outputs. And there is a functional relationship between the inputs and the outputs. What are the inputs to your problem ? What are the outputs to your problem ? What is the functional relationship between the two ? »

Ce programme est constitué
– d'entrées,
– de sorties,

- d'un graphe étiqueté exprimant la relation entre les sorties et les entrées grâce à des fonctions (+) qu'on suppose déjà définies,
- d'une sémantique d'évaluation, qui donne une signification au graphe,
- d'une représentation graphique,
- Et pour l'exécuter pour de bon, il faut une machine réelle vers laquelle on peut compiler le programme (ou un interpréteur fonctionnant sur cette machine).

4.3 Généralisation

Si on généralise ce résultat, on peut dire qu'un programme est défini par une structure de données abstraite (on n'a pas besoin de connaître la représentation en mémoire de cette structure), qui peut contenir des entrées, des sorties, un arbre (ou graphe) syntaxique, etc. Cette structure peut être représentée sous forme d'un programme dont l'entrée est la désignation d'une partie de la structure que l'on souhite accéder, et dont la sortie est la partie en question.

La représentation graphique ou textuelle du programme peut être assurée par un autre programme.

Sa sémantique d'évaluation peut être définie par une machine abstraite, dont les (méta-)entrées sont le programme, ainsi que des entrées pour le programme, et dont les (méta-)sorties sont les sorties que le programme devrait avoir. Tiens ? Entrées, sorties, une relation fonctionnelle (les sorties sont celles du programme pour ses entrées), ... Eh oui, notre machine abstraite, c'est-à-dire notre sémantique d'évaluation est bel et bien un programme elle aussi.

De même, la machine réelle vers laquelle on éspère pouvoir compiler le programme, peut être modélisée par un programme. Pendant que nous y sommes, rien ne justifie la présence de la machine réelle, car la machine abstraite pourrait très bien être la même que celle qui exécute le programme. C'est le cas par exemple si notre langage est le code machine d'un certain processeur : L'octet 0x12345678 a pour signification «diviser l'accumulateur par 2», c'est une définition de la sémantique du langage, et à la fois une définition de la machine qui exécute le programme.

La machine abstraite n'a donc pas besoin d'être si abstraite que ça, et pourrait être n'importe quelle machine, et il pourrait même y avoir plusieurs machines qui spécifient la sémantique du langage (de manière redondante, pour avoir le choix, et pour que ces définitions se vérifient mutuellement).

Et, pour continuer sur cette voie, il peut aussi y avoir plusieurs représentations syntaxiques (textuelle, avec ou sans coloration, graphique, ...).

On a donc les équations suivantes dans le cas simple (une seule machine, une seule représentation) :

$$\text{Programme} = \left\{ \begin{array}{ll} & \text{Type de données abstrait (ADT)} \\ + & \text{Machine abstraite (sémantique)} \\ + & \text{Représentation} \end{array} \right\} \quad (1)$$

$$\text{Programme} = \text{Programme} + \text{Programme} + \text{Programme} \quad (2)$$

$$\text{Programme} = 3 \times \text{Programme} \quad (3)$$

4.4 Base

Nous voilà bien avancés... Un programme est un programme. C'est donc une définition récursive. Et toute définition récursive doit avoir une base, et des règles pour générer de nouveaux éléments. Nous venons de définir les règles, cherchons les bases possibles :

- Machine de Turing
- Lambda-calcul
- Langage mathémathique

Le lambda-calcul et la machine de Turing sont équivalents[5,6], par contre le langage mathémathique permet d'exprimer des fonctions non-calculables, des ensembles infinis, et tout un tas de choses obscures. Comme nos machines physiques actuelles sont une version bâtarde des machines de Turing (qui n'ont pas de limite sur la quantité de mémoire disponible, contrairement aux notres), il semble sage de laisser de côté le langage mathémathique (pour l'instant, lorsque le langage aura gagné en maturité, peut-être qu'il sera temps de l'ajouter).

L'équivalence λ-calcul vs. Turing nous laisse le choix pour l'implémentation de notre première machine à partir de laquelle les autres seront définies, directement ou indirectement. Explorons donc la suite du problème avant de prendre une décision. À terme, le meilleur sera probablement d'implémenter les deux, comme base, et de les définir mutuellement l'une à partir de l'autre, pour avoir une vérification.

Troisième partie
Implémentation

Pour l'implémentation, nous nous limiterons à un sous-ensemble purement fonctionnel du langage défini dans les sections précédentes.

[5] `http://en.wikipedia.org/wiki/Lambda_calculus#Computable_functions_and_lambda_calculus`

[6] Bien qu'ils ne semblent pas être totalement équivalents[17] :

> However, [Lambda Calculus] is not a model of computation for we cannot calculate an upper bound on resource consumption of its reduction steps without resorting to another model of computation, such as [Turing Machines] (according to Yuri Gurevich).

5 Spécification fonctionnelle

Quatrième partie
Annexes

A Notes

A.1 notes

- gruntnetwork.com
- La thèse sur la programmation par l'exemple
- vidéo alan kay

A.2 Notes pour la suite...

Nous allons prendre un programme en dataflow, et le déconstruire le plus possible, afin de voir quelles sont les «primitives» sémantiques nécessaires à notre langage. Bien que FORTH n'ait pas à proprement parler de primitives, il a lui aussi une sémantique (chaque mot est expansé en la suite de mots le définissant, jusqu'à arriver au code machine).

Il faut définir un bloc eager-evaluation, qui prend en paramètre un graphe, et

- Soit le réécrit (compilation)
- Soit l'évalue (interprétation)

Dans le cas où on compile, on aura une "instruction" call-bloc

call-bloc prend en paramètre des fonctions permettant de calculer ses paramètres, ainsi que les paramètres eux-mêmes.

- En évaluation paresseuse, on n'évalue les paramètres que s'ils sont nécessaires.
- En évaluation «eager», on évalue les paramètres au début de l'appel du bloc, et on stocke leur valeur pour une future utilisation (ou non).
- Pour une macro, on stocke juste les paramètres eux-mêmes (avec leur fonction d'évaluation, s'il y en a une).

Références

[1] Jacques Bon. Petite histoire du synthétiseur. `http://cafcom.free.fr/ams/ams1.html`, section intitulée «Débuts de l'informatique musicale».

[2] Jacques Bon. Alsa modular synth. `http://cafcom.free.fr/ams/ams2.html`.

[3] Stephen Schmitt. Fonctionnalités de world machine, `http://www.world-machine.com/features.html`.

[4] Capture d'écran sur la page wikipédia «Quartz Composer», http://en.wikipedia.org/wiki/Quartz_Composer.

[5] Creating recursive VIs. (Logiciel de mesure LabView) http://zone.ni.com/devzone/cda/tut/p/id/9387.

[6] The right tool. http://therighttool.hammerprinciple.com/ est un sondage cherchant à déterminer quelles affirmations correspondent le mieux à quels langages de programmation.

[7] grub.cfg : le fichier de menu de grub 2. http://grub.enbug.org/grub.cfg.fr.

[8] Awesome. http://awesome.naquadah.org.

[9] Christopher C. Risley and Trevor J. Smedley. Visualization of compile time errors in a java compatible visual language. In *VL*, pages 22–29, 1998. [7]

[10] Deutsch limit. http://en.wikipedia.org/wiki/Deutsch_Limite.

[11] Code bubbles - rethinking the user interface paradigm of integrated development environments. http://www.cs.brown.edu/people/acb/codebubbles_site.htm.

[12] Mutant storm reloaded. Utilise une interface «zoom» http://pompomgames.com/mutantstormreloaded.htm.

[13] Project cecily. issu de [20] http://www.osmosoft.com/cecily/.

[14] Effective use of screen real estate. (bibliographie, partie de [18]) http ://web.engr.oregonstate.edu/ burnett/vpl.html#V6C2.

[15] Citation d'un ami de Joe Armstrong. Cité dans [16], page 217. Version électronique : http://is.gd/caFhh.

[16] Peter Seibel. *Coders at Work : Reflections on the Craft of Programming.* APress, 2009. http://www.codersatwork.com/.

[17] Koray Can. 10ecommentaire sur http://is.gd/cb7Ok.

[18] Visual programming languages research papers. (bibliographie) http://web.engr.oregonstate.edu/~burnett/vpl.html.

[19] Visual programming languages bibliography : A branch of the visual language research bibliography. (bibliographie) http://www-ist.massey.ac.nz/plyons/776/vpl%20papers.html.

[20] Tiddlywiki. http://tiddlywiki.com/.

[7] Ce papier n'est pas en accès libre. Par conséquent, je ne peux pas vérifier sont contenu. Protestons contre les papiers payants.