Thèse

de

Doctorat en Informatique

École Doctorale SPI

UMR 9189 - CRIStAL

présentée et soutenue le 18/12/2019

par

Thibault Raffaillac

Améliorer les langages et bibliothèques logicielles pour programmer l'interaction

---

devant un jury composé de

Présidente	**Mme. Laurence Duchien**	Professeure, Université de Lille
Rapporteurs	**Mme. Sophie Dupuy-Chessa**	Professeure, Université Grenoble Alpes
	M. Éric Lecolinet	Maître de conférences (HDR), Télécom ParisTech
Examinateurs	**M. Stéphane Conversy**	Professeur, École Nationale de l'Aviation Civile
	M. Emmanuel Pietriga	Directeur de recherche, Inria Saclay
Directeur de thèse	**M. Stéphane Huot**	Directeur de recherche, Inria Lille

À la mémoire de Janet,
à sa bienveillance,
et son amour des lettres

Remerciements

Le document que vous avez sous les yeux est l'aboutissement de quatre années de recherches. Quatre années qui ont mis mes nerfs à rude épreuve, alternant entre apprentissages pénibles, périodes de profonde mélancolie, et moments d'épiphanie. Maintenant que le résultat est là, j'espère qu'il vous satisfera. La programmation de systèmes interactifs graphiques (interfaces de bureau, applications pour smartphone, jeux vidéo) est un domaine qui m'a toujours été particulièrement opaque, et c'est cette frustration qui m'a poussé à creuser jusqu'à une thèse. Ainsi j'ai voulu rendre ce manuscrit aussi utile que possible, en essayant de clarifier tout ce que j'avais appris sur le domaine, et en donnant parfois mon avis sur ce qu'on pourrait y apporter.

Ce travail n'aurait certainement pas abouti sans l'aide et le soutien de nombreuses personnes, que j'aimerais prendre le temps de mentionner. Je tiens tout d'abord à remercier mon directeur de thèse, Stéphane Huot, qui a fait preuve d'une grande patience à mon égard. Quand je doutais de tout, remettais en question ses conseils, et n'en faisais qu'à ma tête, il a été un soutien inconditionnel et a gardé une grande écoute. J'ai beaucoup apprécié nos discussions, où il voyait souvent “plus loin” et apportait des remarques clés qui allaient faire avancer mon travail. Les innovations de cette thèse ont ainsi été véritablement co-construites, à force d'itérations et échanges fructueux. Je remercie Stéphane de m'avoir guidé là où je voulais aller, plutôt que là où il l'avait voulu.

Je tiens ensuite à remercier mes rapporteurs, Sophie Dupuy-Chessa et Éric Lecolinet, d'avoir pris le temps de relire mon manuscrit, et tous les membres du jury (incluant Stéphane Conversy, Laurence Duchien et Emmanuel Pietriga) d'avoir accepté d'évaluer mon travail. Je suis honoré d'avoir pu recueillir les remarques et conseils de ceux que je considère comme les meilleurs experts français sur mon sujet de thèse !

Les quatre années dans l'équipe Loki (ex-Mjolnir) ont été particulièrement plaisantes, grâce à la bienveillance de tous les membres (passés et présents). Merci à mes docteurs aînés, Jonathan, Alix, Justin, Amira et Hakim, pour leur sagesse et leurs conseils (administratifs, mais pas que). Merci à Sébastien et Izzat, entre autres comme coéquipiers lors du Hash Code 2016. Merci à Nicole, Axel, Marc, Damien, Raiza, Philippe, Grégoire, Gabriel et Pierrick, d'avoir fait du Starter *the place to be*, pour râler, manger, jouer, ou encore travailler. Merci à Julien 🤘 de m'avoir accueilli dans son clan, et pour un festival mémorable. Merci à tous les permanents de l'équipe, Thomas 🤘 , Sylvain, Mathieu, Nicolas, Ed, Marcello, Fanny, Géry, d'avoir été super accessibles, et de m'avoir tous soutenu quand vous en avez eu l'occasion. C'est peut-être insignifiant pour vous, mais chaque coup de pouce a été une immense aide à mon niveau. Et merci aux assistantes d'équipe, Karine et Julie, toujours souriantes quand j'arrivais avec les demandes les plus incongrues.

Durant les deux premières années de la thèse je partageais mon temps avec une seconde équipe, RMoD, et je tiens à en remercier tous les membres qui m'ont soutenu dans mon travail en interaction avec le domaine du Génie Logiciel. Merci à Damien de m'avoir fait découvrir ECS, et d'avoir nourri de nombreuses discussions autour de modèles à objets originaux. Merci à Clément, Vincent et Brice, de

m'avoir aidé à relâcher la pression avec les afterworks. Merci à Olivier, Marcus, Anne, Christophe et Santiago pour votre gentillesse tout au long de mon séjour dans l'équipe, c'était salutaire. Merci à Stéphane pour les critiques pertinentes sur ECS, qui m'ont incité à bétonner sa défense par rapport au modèle objet.

Pendant toute la durée de ma thèse j'avoue avoir eu un goût immodéré pour les projets secondaires, au désespoir de mon directeur de thèse. Ces projets m'ont pourtant beaucoup apporté humainement et professionnellement, et je remercie les personnes qui y ont participé. Merci en premier lieu à Yoann Dufresne, Yann Secq et Sophie Tison, qui ont formé le noyau de promotion des concours d'algorithmique à l'université, grâce auxquels on a organisé des ateliers d'entraînement hebdomadaires pendant presque trois ans, et qui m'ont soutenu ensuite pour assurer des fonctions d'enseignement. Merci à Lucien Mousin pour sa formidable plateforme de code en ligne qui nous a permis de dynamiser les TDs d'algorithmique (et les ateliers d'entraînement). Merci à Benjamin Torres également dans l'organisation du hub du concours Catalysts. Je tiens aussi à remercier les membres de l'espace bidouille, communauté hyperactive qui m'accueillait les samedis pour coder et paporter : Yoann (à nouveau), Matthieu Falce, Pierre Marijon, Pierre Pericard, Camille Lihouck et Sophie Kaleba, en particulier pour le festival Zoo Machines qui a été un sacré moment de création !

En dehors de la thèse j'ai pu compter sur un ami et coloc' hors pair, Léonard 🤘 , que je remercie pour m'avoir initié aux festivals de musique (... Boom-lay, BOOM !), fait découvrir pleins de super groupes, et pour la communauté des M's qui m'a redonné confiance à un tournant de mon travail. Merci à Pierre pour les sessions Rock, et son goût pour les noms d'équipe de blinds tests, à défaut de les gagner (Emile Louise Attaque FTW). Merci à Sandra et Amédée, Marie et Tupak, Charlotte et Guille, et tous ceux dont j'ai croisé la route durant ces quatre ans à Lille. Merci à Aurélien, Florian, Magda, Arnaud et Lela pour les voyages dans le Grand Nord (Norvège), et à Antoine Tran, Alexandre Kohen et Flora Weissgerber pour les voyages dans le Grand Sud (Paris).

À plus haut niveau je tiens à remercier tous les enseignants qui m'ont transmis leur passion de la Science tout au long des études, et m'ont souvent soutenu pour poursuivre plus loin. Je leur dois beaucoup, et espère en faire de même à mon tour un jour. Merci à Olivier Guillaumin pour les discussions très instructives sur l'avenir des langages de programmation, et pour avoir soutenu ma candidature en thèse, alors même que je démissionnais de son entreprise. Merci également à Wendy Mackay d'avoir fait le pari de me prendre en stage prédoctoral sans expérience préalable, et d'avoir soutenu ma candidature de thèse à Lille ensuite.

Enfin je ne pouvais pas remonter dans le temps sans mentionner les outils qui ont facilité ma vie durant la rédaction de ce manuscrit, et dont je suis heureux qu'ils aient existé à temps pour mon travail. Je pense en particulier à Paged.js (conversion HTML vers livre), qui m'a permis de rédiger ce manuscrit intégralement en HTML, de gérer la typographie avec CSS, et de générer les éléments dynamiques (table des matières, numéros de figures, liens des références, etc.) avec JavaScript. Merci à Zapf et Frutiger pour leurs travaux sur les polices Palatino et Avenir. Et un grand merci à Hal Elrod pour sa méthode *Miracle Morning* qui a rendu la période de rédation particulièrement agréable et productive.

Pour finir, je tiens à remercier mes parents pour avoir été l'ilot de stabilité auquel je pouvais toujours me raccrocher quand les choses n'allaient pas. Et je remercie tout spécialement Audrey, qui a illuminé mon existence depuis notre rencontre, et a fait preuve d'une grande lucidité lorsqu'il fallait démêler des situations inextricables. Avec elle j'ai appris à désirer une vie simple, avec bienveillance, randonnées, et écologie. Pour synthétiser notre épopée je terminerai sur la réplique finale de *Fight Club* : « *You met me at a very strange time in my life.* »

Résumé

L'objectif de cette thèse est d'étudier et développer des modèles de programmation de systèmes interactifs, pour favoriser le prototypage et le développement de nouvelles techniques d'interaction. Dans ce contexte, les développeurs utilisent principalement des frameworks génériques d'interfaces graphiques, qu'ils “bricolent” pour intégrer de nouvelles idées. Or ces frameworks sont peu adaptés à de tels usages, et restreignent la liberté des développeurs à dévier des standards établis. Une première étude basée sur des interviews relève les problèmes, besoins, utilitaires, et techniques de “bricolage” rencontrés dans ce contexte. Une seconde étude basée sur un questionnaire en ligne évalue la prévalence des problèmes et techniques relevées dans la première étude, ainsi que les raisons pour lesquelles les frameworks issus de la recherche sont peu utilisés pour concevoir de nouvelles techniques d'interaction. Les résultats de ces études aboutissent à la définition de trois Essentiels d'Interaction, des recommandations pratiques destinées à améliorer le support de l'interaction dans les frameworks et langages de programmation : (i) une orchestration explicite et flexible des comportements interactifs, (ii) un environnement d'interaction minimal et initialisé au démarrage de toute application, et (iii) des mécanismes et conventions standardisés et appuyés par un langage flexible. Ces recommandations sont illustrées une première fois par une extension au langage Smalltalk, qui permet d'exprimer des animations en ajoutant une durée aux appels de fonctions. L'application des Essentiels d'Interaction est illustrée une seconde fois par la création du framework Polyphony, qui se base sur le modèle Entité-Composant-Système issu du Jeu Vidéo, et l'adapte pour la création d'interfaces graphiques et de techniques d'interaction.

Abstract

The aim of this thesis is to study and develop programming models for interactive systems, to promote the prototyping and development of new interaction techniques. In this context, developers mainly use generic interaction frameworks, which they "hack" to integrate new ideas. However, these frameworks are poorly adapted to such uses, and restrict the freedom of developers to deviate from established standards. A first study based on interviews identifies the problems, needs, utilities, and "hacking" techniques encountered in this context. A second study based on an online questionnaire assesses the prevalence of the problems and techniques identified in the first study, as well as the reasons why research frameworks are rarely used to develop new interaction techniques. The results of these studies lead to the definition of three Interaction Essentials, practical recommendations to improve the support of interaction in frameworks and programming languages: (i) an explicit and flexible orchestration of interactive behaviours, (ii) a minimal and initialized interaction environment at the start of any application, and (iii) standardized mechanisms and conventions supported by a flexible language. These recommendations are illustrated initially by an extension to the Smalltalk language, which allows animations to be expressed by adding a duration to function calls. The application of the Interaction Essentials is illustrated once more with the creation of the Polyphony framework, which is based on the Entity-Component-System model originating from Video Games, and adapted for the creation of graphical user interfaces and interaction techniques.

Table des matières

Introduction

La recherche en Interaction Homme-Machine explore sans cesse de nouvelles manières d'interagir avec les systèmes numériques. Elle mêle par exemple l'usage de périphériques mobiles, l'interaction au doigt et avec le reste du corps, ou encore la navigation immersive dans des univers virtuels en 3D. Ces développements suivent et accompagnent des progrès techniques réguliers. Ils assurent ainsi le développement régulier de nouvelles formes d'interactions, l'émergence d'innovations autrefois impossibles, et le renouvellement des usages modernes.

Pourtant, les outils que nous utilisons ont peu progressé depuis l'apparition des premières interfaces graphiques. Ils sont souvent peu adaptés à la recherche de nouvelles formes d'interactions, orientant les développements vers des interfaces stéréotypées — comme la sélection de boutons, l'entrée de texte avec un clavier, ou la séparation d'applications avec des fenêtres. La recherche est active dans ce domaine, et de nombreux outils émergent régulièrement pour supporter le développement de nouvelles interactions [Led18], mais ils ne sont pas suffisamment utilisés en pratique [Mar17]. Avec le temps, il semble qu'un fossé se soit creusé entre les outils démontrant des architectures logicielles innovantes, et ceux plus complets bénéficiant de décennies de développement (les *frameworks*).

Le développement de prototypes de recherche est une activité difficile, qui requiert du temps et de l'expertise. Les chercheurs doivent souvent déployer beaucoup d'efforts pour intégrer leurs idées et prototypes dans des systèmes informatiques complexes, et par ailleurs peu flexibles, afin d'en démontrer la pertinence et la validité. Les démonstrateurs issus de la recherche peinent à égaler l'utilisabilité des interfaces existantes, auxquelles sont habitués les utilisateurs finaux. En conséquence, l'aspect souvent rudimentaire des prototypes de recherche freine leur diffusion auprès des publics ciblés, alors que ce sont souvent des contributions pratiques, vouées à être intégrées dès que possible à des interfaces modernes.

Questions de recherche

Nous sommes face à une situation complexe, avec de nombreuses causes et beaucoup de tentatives de solutions. Le premier objectif de cette thèse est donc de comprendre cette situation, et d'en exposer chacun des aspects, afin de poser des bases de discussions et d'argumentations. Au fil de cette recherche, nous nous concentrons moins sur les causes qui ont mené aux outils actuels, mais plutôt sur ce que nous pouvons y apporter aujourd'hui. Ainsi, nous tentons de répondre dans un premier temps à la question **Quels sont les besoins des chercheurs pour le prototypage de nouvelles interfaces et techniques d'interaction ?**

L'objet de cette thèse est l'étude des outils et langages pour concevoir des programmes interactifs (interagissant avec des personnes), en particulier pour la recherche. Notre objectif ici est d'étudier les outils existants et ce qu'en font les chercheurs. Nous analysons également les outils issus de la

recherche qui ont exploré des paradigmes alternatifs de programmation d'interfaces, avec des succès divers. En outre, les “outils” ne se réduisent pas uniquement à des bibliothèques logicielles, et nous considérons dans notre étude les travaux de production et de promotion de connaissances. Il s'agit alors de comprendre l'impact des différents types de contributions, pour proposer de nouveaux outils adaptés aux besoins des chercheurs. Ainsi, nous répondons dans un second temps à la question **Sous quelles formes peut-on contribuer à l'activité de prototypage en IHM ?**

Enfin, nous choisissons dans cette thèse de nous concentrer sur les bibliothèques et architectures logicielles pour le prototypage. Ces types de contributions étant très répandues en IHM, nous pouvons nous baser sur un vaste corpus de travaux existants, et ainsi analyser leurs avantages et inconvénients. Il s'agit de déterminer les besoins auxquels ils répondent, et ceux qui manquent encore de support aujourd'hui. Nous souhaitons alors proposer des travaux innovants, et en adéquation avec les technologies contemporaines. En particulier, l'influence des langages de programmation sur l'expression des programmes interactifs est explorée plus en avant dans cette thèse. Ainsi, nous répondons dans un dernier temps à la question **Comment concevoir un framework ou un langage de programmation qui satisfasse une partie des besoins des chercheurs en IHM ?**

Méthodologie

Notre sujet d'étude au sens large est la *programmation d'interactions*. Comme toute activité humaine, elle est difficile à quantifier, c'est-à-dire qu'il sera difficile d'avancer et de valider des vérités objectives (ex. *plus de 50% des chercheurs ont des difficultés avec l'outil X*). De plus, de très nombreux facteurs peuvent influencer l'activité de programmation (ex. le confort physique de l'utilisateur, son humeur du moment, ses collègues de bureau), qui ne peuvent pas tous être contrôlés. Ainsi, la complexité de la situation que nous étudions est telle que nous n'avons pas cherché à en isoler chacune des causes. Nous nous sommes basés sur des observations systématiques de chercheurs ayant programmé des interactions, et à partir d'un nombre suffisant d'observations avons cherché à formuler des hypothèses *probables*. Notre méthode s'inspire de la théorie ancrée (*grounded theory*), par laquelle nous formulons des hypothèses par induction, à partir de nos observations [Gla17]. Ces hypothèses sont limitées par la variété de nos observations, donc se généralisent à une population très réduite de personnes. En ce sens, nous ne cherchons pas des vérités absolues et des réponses certaines à nos questions de recherche, mais à argumenter en faveur de solutions probables. Ainsi ce manuscrit est structuré comme un support de discussion, afin de laisser le soin aux lecteurs de nuancer nos conclusions, et si possible de contribuer à y voir plus clair.

Nous avons donc commencé ce travail de thèse par une série d'interviews de chercheurs en IHM, à propos de leurs projets passés de développement d'interactions. Une analyse systématique de ces interviews nous permet ensuite de formuler des observations, que nous avons ensuite testées à l'aide d'un questionnaire en ligne. À partir des résultats des interviews et du questionnaire, nous identifions des besoins de haut-niveau pour le prototypage d'interactions innovantes, et proposons différentes pistes de contributions pour y répondre. Dans un second temps, nous avons rapproché la programmation d'interactions de l'usage d'un langage de programmation, et considéré le rapport étroit entre une bibliothèque d'interaction et le langage sur lequel elle s'appuie. Nous formulons alors trois *Essentiels d'Interaction*, des recommandations pratiques visant à orienter la conception des systèmes interactifs, que ce soit par un langage de programmation, ou par une bibliothèque logicielle.

Ces recommandations sont destinées à justifier les choix de contributions de ce manuscrit, ainsi que nos travaux futurs dans le domaine. Pour finir, nous avons exploré les deux types de contributions, en développant d'une part une extension du langage Smalltalk dédiée aux animations, et d'autre part un framework d'interaction dédié au prototypage d'interfaces ad hoc.

Contributions

Deux contributions se dégagent principalement de cette thèse. La première est une étude approfondie de l'activité de programmation d'interactions dans un contexte de recherche, à partir des interviews et du questionnaire en ligne. Nous présentons en particulier les problèmes principaux auxquels les chercheurs que nous avons interrogés sont confrontés, ainsi que les stratégies de résolutions qu'ils ont employées. Ces observations nous ont amenés à considérer que les domaines nécessitant des investissements futurs sont le support de bas-niveau de l'interaction, et le rapprochement des technologies du Web avec les applications de bureau.

La deuxième contribution principale est l'application d'un modèle original de programmation, l'Entité-Composant-Système, à la programmation d'interfaces graphiques. Ce modèle, basé sur le principe de *Composition plutôt qu'Héritage*, facilite la manipulation explicite des comportements interactifs dans une interface, et promeut leur réutilisation. En outre, il matérialise les périphériques d'entrée/sortie en entités virtuelles, qui facilitent l'implémentation de techniques d'interaction utilisant ces périphériques. Nous avons implémenté cette contribution dans une bibliothèque logicielle, Polyphony.

Plan du manuscrit

Dans le **Chapitre 1. Programmer l'interaction**, nous détaillons le contexte et les problématiques de recherche de cette thèse, et présentons l'état actuel des connaissances sur la programmation d'interactions. Nous introduisons ensuite les interviews et le questionnaire en ligne, qui nous permettent de dépeindre une vision plus claire de l'activité de programmation dans un contexte de recherche en IHM, et de finir avec un ensemble de recommandations pour les contributions futures à la programmation d'interactions.

Dans le **Chapitre 2. Les Essentiels d'Interaction**, nous élaborons une série de trois principes pour l'ingénierie des systèmes interactifs. Ces principes ont guidé nos travaux durant ce travail de thèse, et justifient nos choix de contributions. Chaque Essentiel d'Interaction est présenté indépendamment, et discuté à l'aide d'observations et de réflexions sur l'état courant et futur des systèmes informatiques. Le dernier point est élaboré à partir d'un travail exploratoire sur l'expression des animations, avec une extension du langage Smalltalk.

Dans le **Chapitre 3. La boîte à outils Polyphony**, nous présentons notre travail sur la conception d'une bibliothèque logicielle pour créer des interfaces graphiques. Nous y détaillons le modèle Entité-Composant-Système, que nous avons adapté du domaine du Jeu Vidéo aux Interactions Homme-Machine. Nous démontrons une implémentation fonctionnelle de ce travail, et détaillons son architecture logicielle. L'ensemble du chapitre est structuré pour faciliter la reproduction de ce travail, pour quiconque voudrait l'adapter à nouveau.

Publications

Nous présentons ici la liste des publications produites durant ces quatre années de thèse. Pour les articles dont le contenu est en partie présenté dans ce manuscrit, nous indiquons entre parenthèses le chapitre ou la section correspondante.

1. **Raffaillac, T.** (2017). Language and System Support for Interaction. *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (Doctoral Consortium)*, 149–152. [Raf17]
2. **Raffaillac, T.**, Huot, S., & Ducasse, S. (2017). Turning Function Calls into Animations. *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 81–86. (section 2.4) [Raf17]
3. **Raffaillac, T.**, & Huot, S. (2018). Application du modèle Entité-Composant-Système à la programmation d'interactions. *Proceedings of the 30th Conference on L'Interaction Homme-Machine*, 42–51. [Raf18]
4. **Raffaillac, T.**, & Huot, S. (2019). Polyphony : Programming Interfaces and Interactions with the Entity-Component-System Model. 🏆 Best Paper. *Proc. ACM Hum.-Comput. Interact.*, *3*(EICS), 8:1–8:22. (chapitre 3) [Raf19]

Chapitre 1. Programmer l'interaction

Nos manières d'interagir avec les ordinateurs ont fortement évolué au cours des dernières décennies. Leurs capacités techniques ont beaucoup progressé, et continuent de s'améliorer. Ils sont capables de calculer plus vite, et réaliser des opérations plus complexes, tout en consommant moins d'énergie. Ils sont plus petits, plus légers, et sont capables aujourd'hui de tenir dans une poche ou dans le creux de la main. Ils sont plus versatiles, et multiplient les capteurs interagissant avec l'homme et leur environnement (souris, écrans tactiles, boutons physiques, microphones, accéléromètres, etc.). Enfin ils ne sont plus circonscrits à un boîtier dédié, mais sont intégrés aux objets du quotidien, dans notre environnement, voire accessibles par Internet.

Dans un même temps, les usages liés aux ordinateurs ont évolué à mesure que ceux-ci devenaient plus puissants. Alors que dès les années 50 on trouvait une machine pour plusieurs utilisateurs, à partir des années 70 se sont développés les ordinateurs à usage personnel, dès les années 90 on pouvait trouver plusieurs machines pour un utilisateur [Wei97], et de nos jours il est courant que plusieurs utilisateurs se partagent plusieurs machines. De même les contextes dans lesquels on utilise des ordinateurs se sont très largement multipliés (calcul scientifique, bureautique, production industrielle, télécommunications, divertissements, etc.), et ont contribué à faire évoluer les interfaces entre humains et ordinateurs.

1.1 Contexte

Aujourd'hui, alors que les contextes d'utilisation des ordinateurs continuent de changer, nous avons toujours besoin de développer de nouveaux types d'interfaces, et d'améliorer nos manières d'interagir avec des ordinateurs. Ainsi, le contexte de ce travail de thèse est la **recherche et le développement de nouvelles interactions avec les ordinateurs**, afin d'étudier et de proposer des outils adéquats pour cette activité. Mais alors, *qu'est-ce que l'interaction ?* Et qu'entend-on par *programmer l'interaction* ?

1.1.1 Caractérisation de l'interaction

La définition de l'interaction est depuis longtemps un sujet de discussions dans la communauté du domaine Interaction Homme-Machine (IHM). Cette définition est importante, car elle doit idéalement englober toutes les productions humaines liées à l'interaction entre des personnes et des machines — passées, présentes, et prospectives. Elle délimite le domaine IHM, en caractérisant de qu'il *est* et ce qu'il *n'est pas*. Nous n'avons pas l'ambition d'en fournir une définition ici, cependant il nous faut la caractériser afin de clarifier l'objet de cette thèse — la *programmation* d'interactions. Ainsi le Dictionnaire de l'Académie Française définit l'interaction par :

INTERACTION n. f. XIX^e siècle.

PHYS. Action réciproque de deux ou plusieurs corps. *La gravitation est un phénomène d'interaction entre deux corps*. *Interaction électromagnétique*. *Interaction forte*, action attractive entre particules, qui assure la cohésion des noyaux atomiques. *Interaction faible*, qui se manifeste par des forces d'attraction ou de répulsion entre particules, responsables en particulier de la radioactivité bêta. • Par ext. Influence qu'exercent les uns sur les autres des phénomènes, des faits, des objets, des personnes. *L'interaction des faits économiques et politiques*.

Cette définition, issue de la physique, s'applique par extension à l'Interaction Homme-Machine, et nous pouvons la résumer par "influence réciproque". Or elle ne précise en aucune mesure quels types d'influences l'homme et la machine peuvent avoir l'un sur l'autre. Elle est donc insuffisante pour caractériser les travaux en IHM et leurs besoins.

Dans leur article *What is Interaction?* paru en 2017, Hornbæk et al. s'attachent à présenter les principaux "courants de pensée" qui donnent des définitions de l'interaction [Hor17]. Ils en tirent la tentative de définition suivante : « *Following Bunge, interaction concerns two entities that* determine *each other's behavior over time. In HCI, the entities are computers (ranging from input devices to systems) and humans (ranging from end effectors to tool users). Their mutual determination can be of many types, including statistical, mechanical, and structural. But their causal relationship is teleologically determined: Users, with their goals and pursuits, are the ultimate metric of interaction* ».

Cette définition abandonne la notion d'action ou d'influence, au profit de celle de détermination — ou d'observation pour acquérir une compréhension. En ce sens elle nous semble compléter la définition donnée par l'Académie Française, plutôt que la remplacer. À partir de ces définitions, nous pouvons préciser en quoi consistent les recherches de "nouvelles interactions" avec les ordinateurs, et ainsi raffiner le sujet d'étude de cette thèse. Ces recherches se caractérisent par :

- de nouvelles manières pour les Hommes d'agir sur les Machines — ou dans la terminologie Informatique, de leur transmettre de l'information ;
- de nouvelles manières pour les Machines de transmettre de l'information aux Hommes ;
- de nouvelles manières de lier réception et émission d'informations (chez l'Homme ou la Machine), afin d'aider l'autre à interpréter un comportement cohérent à partir de ces échanges.

Les deux premiers points réfèrent principalement au développement de nouveaux *périphériques d'interaction*, ainsi que les traitements qui leur sont liés (ex. fonctions de transfert pour les périphériques de pointage). Les périphériques d'interaction sont des dispositifs (physiques ou virtuels), qui captent des actions (physiques ou virtuelles), et les traduisent pour qu'elles puissent être perçues par le receveur. Par exemple, la souris est un dispositif physique, qui capte les mouvements de la main, et les transmet à l'ordinateur sous forme de déplacements relatifs. Le clavier de smartphone est un dispositif virtuel, qui capte les appuis sur écran tactile, et les transmet à l'ordinateur sous forme d'entrée de texte. Enfin, l'écran est un périphérique physique, qui capte des matrices de couleurs (les pixels), et les transmet à l'utilisateur sous forme de points lumineux.

Le dernier point — la cohérence entre réception et émission — fait référence aux *techniques d'interaction*. Ce sont généralement des programmes informatiques, qui expriment des réactions complexes à partir de règles simples "*si je reçois A, alors j'émets B*". Les techniques d'interaction sont idéalement *cohérentes*, c'est-à-dire que les utilisateurs peuvent anticiper leur fonctionnement avant

d'avoir interagi, et comprendre ce qui se passe pendant l'interaction. Par exemple si on clique sur un menu déroulant, on s'attend à ce qu'une liste d'options s'affiche, et que l'option visuellement mise en valeur soit celle sélectionnée. Lorsque l'intention de l'utilisateur ne correspond pas très bien à ce que propose la technique, on parle de "Gulf of Execution", et lorsque la technique ne communique pas très bien son état courant, on parle de "Gulf of Evaluation" [Nor88]. Notons enfin qu'on parle majoritairement du comportement de la Machine vue par l'Homme, et rarement l'inverse, ce qu'Hornbæk et al. synthétisent par « *There is little "C" in HCI* » [Hor17].

Parce qu'elles s'expriment généralement en programmes, les techniques d'interaction sont les plus représentatives de la programmation d'interactions : ce sont des programmes qui interagissent avec des utilisateurs. Quant aux périphériques d'interaction, ce sont souvent des objets physiques plutôt que des programmes, et dans notre caractérisation ci-dessus ils ont pour rôles de transmettre plutôt qu'interagir — d'où leur qualité de "périphériques". En pratique lorsqu'ils sont l'*objet* de l'interaction plutôt que le *moyen*, nous les considérons comme des techniques d'interaction. Par exemple, nous pouvons dire que le clavier virtuel est un périphérique d'interaction, mais que l'étude de l'utilisation de gestes pour y saisir des mots est une technique d'interaction [Zha12]. Ainsi dans cette thèse nous nous concentrons principalement sur les techniques d'interaction.

Enfin, nous parlons souvent de programmation d'*interfaces* conjointement aux interactions. La distinction entre les deux termes est historique : on parlait autrefois d'Interfaces Homme-Machine, puis le sens s'est généralisé avec Interactions Homme-Machine (bien que le premier sens soit encore souvent utilisé). En effet, l'interface matérialise l'objet d'interconnexion entre humain et machine, alors que l'interaction désigne plus largement la situation autour. Dans notre cas, la programmation résulte toujours en du code concret. En conséquence, lorsque nous programmons de l'interaction, elle sera nécessairement matérialisée par une interface. Lorsqu'elle est visible sur un écran, on parle d'interface graphique (*Graphical User Interface*). Dans le sens commun, la différence entre une interface et une technique d'interaction est que la première est le siège d'un plus grand nombre d'interactions — on développe souvent une technique d'interaction *pour* une interface. Nous choisissons donc de nous intéresser à cette brique de base, qu'il nous faut à présent décrire plus précisément, pour comprendre l'activité de prototypage qui nous intéresse.

1.1.2 Caractérisation des techniques d'interaction

Tucker [Tuc04] définit les techniques d'interaction ainsi : « *An interaction technique is the fusion of input and output, consisting of all software and hardware elements, that provides a way for the user to accomplish a task* ». Cette définition est une caractérisation pragmatique et liée à la technologie, qui délimite bien l'objet initial de notre étude. Il existe de très nombreux travaux pouvant être qualifiés de techniques d'interaction, qui ont donné lieu à de nombreuses classifications dans la littérature scientifique [Jai07, Jan13, Bai16]. De plus, il existe des contextes de développement distincts pour chaque technique d'interaction, avec des modalités d'interaction distinctes (ex. applications de bureau avec clavier/souris/écran, applications mobiles avec doigt/écran, Réalité Virtuelle avec manette/casque). Pour donner une vue d'ensemble des techniques d'interaction sans entrer dans les détails des différents contextes, nous les associons par les *types de tâches* qu'elles permettent d'accomplir, qui sont représentés partout. Ainsi nous pouvons distinguer :

- les techniques de menus (voir figure 1)
- les techniques de pointage (voir figure 2)
- les techniques de navigation (voir figure 3)
- les techniques de sélection (voir figure 4)
- les techniques d'apprentissage/*feedforward* (voir figure 5)
- les techniques d'édition (voir figure 6)

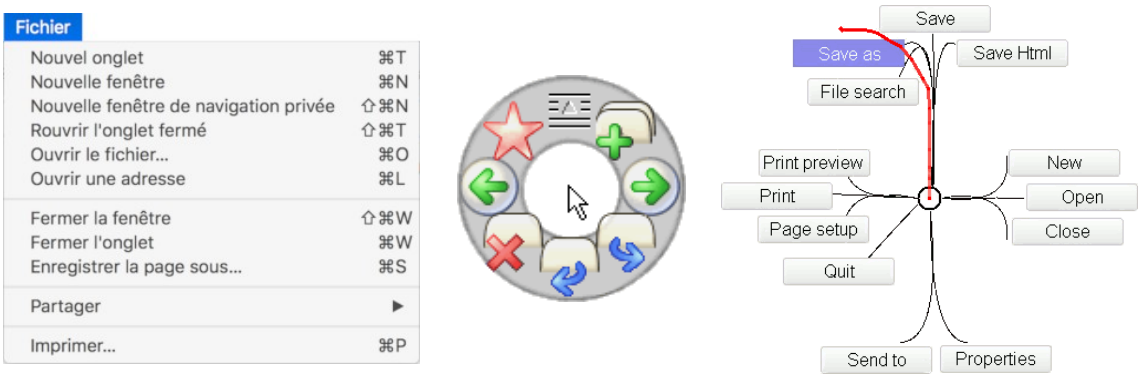


Figure 1 : Illustation de trois techniques de menus, de gauche à droite : menu déroulant de macOS, menu circulaire, et *Flower Menu* [Bai08]

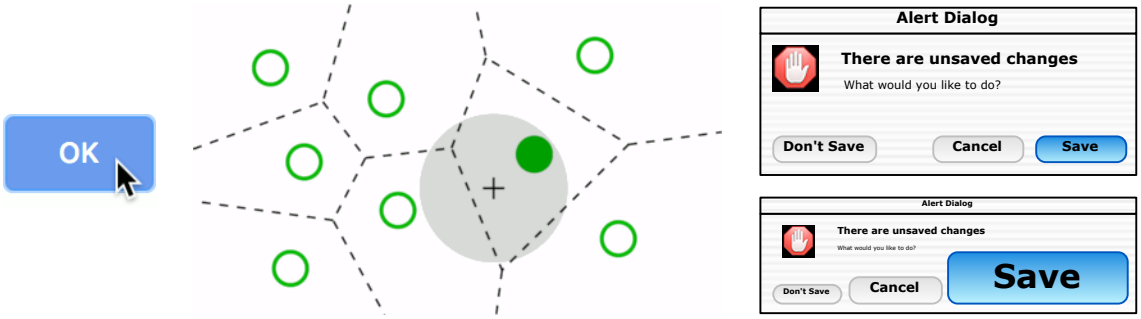


Figure 2 : Illustration de trois techniques de pointage, de gauche à droite : pointage par survol, *Bubble Cursor* [Gro05], et *Semantic Pointing* [Bla04]

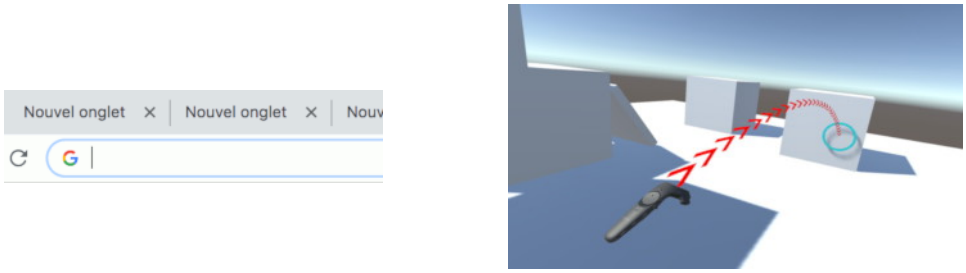


Figure 3 : Illustration de deux techniques de navigation, de gauche à droite : onglets de navigateur Web, et téléportation d'avatar en Réalité Virtuelle dans Unity

Figure 4 : Illustration de trois techniques de sélection, de gauche à droite : sélection de texte, sélection par lasso de Photoshop, et sélection d'éléments par *relaxation de requête* [Hee08]

Figure 5 : Illustration de trois techniques d'apprentissage/*feedforward*, de gauche à droite : infobulle lors du survol du curseur sur un élément interactif, *ExposeHK* [Mal13], et *Octopocus* [Bau08]

Figure 6 : Illustration de trois techniques d'édition, de gauche à droite : écriture de texte au clavier, dessin au pinceau dans Gimp, et *Toolglass* [Bie93]

Enfin, lorsqu'on parle de techniques d'interaction aujourd'hui il faut mentionner l'influence majeure du paradigme WIMP (*Windows, Icons, Menus, Pointer*). Ce paradigme, développé dès 1973 avec le système Xerox Alto, et popularisé par le Macintosh en 1984, a posé les bases de nos manières d'interagir avec les ordinateurs. Il définit un environnement virtuel inspiré du bureau de travail, et à l'origine destiné aux employés de bureau. Celui-ci est constitué de *fenêtres* déplaçables et redimensionnables, d'*icônes* interactives représentant les éléments à manipuler (fichiers, programmes), de *menus* permettant d'énumérer, sélectionner et exécuter des commandes, ainsi qu'un *curseur* contrôlé par la souris, qui permet d'interagir directement avec l'environnement virtuel. Grâce à sa simplicité d'apprentissage et d'utilisation, ainsi que son adaptabilité à de nombreux contextes, le paradigme

WIMP s'est très largement imposé sur les systèmes interactifs modernes, devenant une norme *de facto*. La plupart des techniques d'interaction s'*intègrent* donc avec ce paradigme, en proposant des artefacts qui puissent cohabiter au sein de systèmes WIMP.

Le succès du paradigme WIMP a permis l'uniformisation de l'expérience d'utilisation sur les différents systèmes (Windows, MacOS, Linux, etc.). En revanche il a aussi été suivi d'une cristalisation des techniques d'interaction autour de concepts bien établis, donnant lieu à des interfaces graphiques stéréotypées, et à des innovations en apparence moins ambitieuses qu'autrefois. En réaction à ce phénomène, des techniques d'interaction dites “post-WIMP” ont été développées [Dam97], avec pour ambitions de redéfinir plus profondément nos manières d'interagir avec les ordinateurs. Ces travaux ont en outre mené à reconsidérer les manières de *programmer* des techniques d'interaction, et sont le point d'origine du sujet de cette thèse [Huo06].

1.1.3 Programmation d'interactions

Nous en venons au développement de nouvelles interaction, et plus particulièrement à leur programmation. Dans un système informatique, la majorité des réactions aux actions de l'utilisateur sont exprimées par des programmes. Certaines réactions sont spécifiées mécaniquement (la sensation du “clic” de souris), et d'autres électriquement (l'allumage d'une led lorsque l'ordinateur est sous tension), mais une fois conçues il est très difficile de les modifier. Ce sont donc principalement par des programmes que les chercheurs et concepteurs développent de nouvelles interactions avec les systèmes informatiques. Nous parlerons aussi de programmation d'IHMs, ou lorsqu'elles sont spécifiques aux interfaces graphiques, de programmation d'interfaces.

Un programme est une séquence d'instructions, qui envoyées une par une au processeur central (CPU), permettent de le contrôler en lui faisant exécuter des tâches simples (lire/écrire vers la mémoire, faire des opérations sur des nombres, communiquer avec des périphériques branchés). Chaque programme s'exprime dans un *langage de programmation*, qui définit une syntaxe par laquelle on spécifie les différentes instructions à exécuter. Il existe de très nombreux langages de programmation, qui se distinguent par les manières de raisonner avec, ainsi que leurs contextes d'utilisation. Cependant, nous pouvons citer les langages principalement utilisés aujourd'hui pour exprimer des techniques d'interaction : C/C++, Java, Python, C#, JavaScript, et Objective-C.

La conception d'une nouvelle technique d'interaction implique de décrire au préalable son fonctionnement, voire de le modéliser à l'aide d'outils mathématiques. Nous nous intéressons uniquement à la programmation, c'est-à-dire une fois que la technique est modélisée, son expression en code. La programmation d'une nouvelle technique d'interaction met en œuvre un certain nombre d'étapes :

- On crée des éléments “visibles” (le plus souvent à l'écran), qui suggèrent par leur présence qu'une interaction avec eux ou l'environnement est possible.
- On énumère les actions sur les éléments visibles auxquelles réagir, et pour chacune on définit un sous-programme à exécuter lors de son déclenchement.
- Après exécution du sous-programme d'une action, on met à jour les éléments visibles et on en crée éventuellement de nouveaux.

Pour réaliser ces étapes, on utilise un langage, ainsi qu'une bibliothèque logicielle permettant de détecter les actions des utilisateurs et d'enregistrer des programmes à exécuter sur actions. Ce type de bibliothèques fournit des services liés à la gestion des entrées et des sorties avec les périphériques d'interaction, qui les rendent indispensables pour programmer des interactions. On y distingue communément les *frameworks* et les *boîtes à outils*.

1.1.4 Frameworks et boîtes à outils

Les “frameworks” et “boîtes à outils” sont des termes relativement flous, utilisés pour désigner des bibliothèques logicielles respectivement de grandes et petites tailles. Pour simplifier dans ce manuscrit, nous qualifions de boîte à outils ce qui n'est pas un framework, et nous tâchons de clarifier ce qu'est un framework. De façon générale, un framework est une bibliothèque logicielle qui facilite la création d'interfaces, en les déclarant comme des structures de données plutôt qu'avec du code brut. Elle fournit des éléments réutilisables permettant d'assembler ou de composer avec un minimum d'efforts. Dans le sens commun, un framework est une bibliothèque qui :

- ne peut pas s'utiliser conjointement à d'autres frameworks ou versions d'elle-même
- conserve le contrôle de l'exécution lorsque la machine ne fait rien (“ne rend pas la main”)
- fournit un cadre de développement (*cadriciel*) plutôt qu'une simple collection d'outils ou de fonctions
- nécessite d'adapter les autres bibliothèques à son fonctionnement (par des mécanismes d'extensions) plutôt que l'inverse

On associe parfois aux frameworks l'apport de concepts et *styles* de programmation (ex. *widgets*, *listeners*, signaux/slots dans Qt), mais ces caractéristiques ne sont pas systématiquement présentes. De même, les bibliothèques qualifiées de frameworks sont souvent des projets d'envergure, dotés de fonctionnalités nombreuses et variées, mais sans qu'on puisse s'appuyer sur ces observations pour les caractériser. Une seule caractéristique semble lier les points énumérés ci-dessus, c'est l'*Inversion de Contrôle*, qui est généralement reconnue comme distinctive des frameworks [Fow05, Joh88]. Avec l'Inversion de Contrôle, une application confie son code au framework pour exécution, plutôt que de l'exécuter elle-même. On l'illustre par le principe d'Hollywood [Swe85] : « *Don’t call us, we’ll call you* ». Le contrôle est ainsi inversé, puisque c'est le framework qui se charge d'organiser les différentes fonctions sous sa responsabilité.

Cependant, aujourd'hui plusieurs formes d'Inversion de Contrôle existent, qui compliquent la classification des différentes bibliothèques. Ainsi, dans le cas d'une bibliothèque comme JavaFX, il s'agit de fournir des objets dont les méthodes sont exécutées par le framework (les *widgets*), et aussi de fournir des blocs de code à exécuter lors du déclenchement d'évènements (les *callbacks*). La bibliothèque répond aux points énoncés plus haut, on peut parler de framework. Dans le cas d'une bibliothèque comme SDL [Lan98], on peut aussi fournir des *callbacks* pour réagir aux évènements d'entrée, cependant la bibliothèque ne conserve pas le contrôle de l'exécution lorsque la machine ne fait rien. En effet, on doit appeler la fonction `SDL_PollEvent()` régulièrement, qui exécute les *callbacks* enregistrés puis rend immédiatement le contrôle. Dans ces conditions, ici on ne parle pas de framework. Nous pouvons donc préciser notre définition d'un framework, qui est **une bibliothèque logicielle qui s'approprie de façon permanente le contrôle d'exécution de l'application, et permet l'exécution de code tiers en recevant des fonctions et méthodes de *callback***.

Lorsque les chercheurs prototypent de nouvelles techniques d'interaction, ils se basent nécessairement sur une ou plusieurs bibliothèques logicielles, qui leur permettent d'accéder aux périphériques d'interaction. Le prototypage en IHM est donc étroitement lié aux frameworks et boîtes à outils. On leur associe différents types de travaux facilitant la programmation d'interaction :

- la création de nouvelles boîtes à outils d'interaction, qui peuvent à terme évoluer en *frameworks*
- l'invention d'outils de modélisation à haut-niveau des techniques d'interaction, qui sont ensuite éventuellement *transpilés* vers des langages de plus bas-niveau
- le développement de nouvelles architectures logicielles pour les systèmes interactifs, qui peuvent en faciliter le développement et l'utilisation
- le développement ou la modification de langages de programmation, qui facilitent l'utilisation de frameworks existants ou nouveaux

Dans ce contexte, les chercheurs ont à leur disposition de nombreux outils pour mener à bien l'exploration et le développement de nouvelles techniques d'interaction. Pourquoi alors cette situation n'est-elle pas satisfaisante ?

1.2 Problématique

Aujourd'hui, on peut considérer que l'interaction avec les systèmes informatiques a atteint un point de stabilité. Les environnements de bureau (Windows, macOS, Linux) partagent de nombreux points communs liés au modèle WIMP (fenêtres déplaçables et redimensionnables, icônes sur le bureau, menus déroulants en haut, utilisation de la souris). De même, l'interaction au doigt sur smartphones et tablettes s'est uniformisée entre systèmes (utilisation des gestes au doigt, icônes d'applications sur des "écrans" virtuels, exécution d'une seule application à la fois). Les interfaces se sont "standardisées" [Bea00, Poo16], facilitant l'accès à la technologie pour les utilisateurs, et leur permettant de réutiliser les mêmes acquis entre les différents systèmes. En outre, la programmation des interfaces a été rendue plus rapide, en les exprimant par des assemblages de contrôles standards (ex. boutons, cases à cocher, menus, onglets), et grâce à une uniformisation des architectures logicielles autour du modèle objet [Kra88, Cou87, Con08].

Cependant, cette standardisation a donné lieu à des interfaces stéréotypées, qui se distinguent peu les unes des autres, et dont les évolutions (aussi bien fonctionnelles que cosmétiques) sont faibles dans le temps. En conséquence, il est difficile d'adapter les interfaces actuelles à de nouveaux usages, et elles manquent de flexibilité pour introduire de nouveaux modes d'interaction. La cristallisation de la programmation d'interfaces a généré un écosystème complexe, duquel il est difficile de s'extraire aujourd'hui. Plusieurs freins s'opposent au travail de prototypage des chercheurs : la multiplicité *horizontale* des outils avec lesquels travailler, la multiplicité *verticale* des couches dans les systèmes interactifs, et l'extensibilité limitée des bibliothèques.

1.2.1 Choix des outils de programmation

Le premier frein est l'état des outils de base des systèmes, pour la programmation d'entrées/sorties avec les utilisateurs. Les systèmes d'exploitation, qui gèrent ces entrées/sorties, fournissent des interfaces de programmation (ou API, pour *Application Programming Interface*) complexes et/ou de très bas niveau, c'est-à-dire qui nécessitent de s'occuper de nombreux détails. Par exemple, dans Linux au

moins deux APIs majeures sont disponibles pour dessiner à l'écran, framebuffer et X11. Le premier ne fournit aucune primitive de dessin (ligne, polygone, image), et donne uniquement accès à une matrice de pixels. Le second fournit une API de 239 fonctions, décrites dans un document de 476 pages [Get02].

Face à cette complexité sous-jacente, les programmeurs doivent recourir à des bibliothèques intermédiaires, qui s'interfacent avec les APIs de bas niveau, tout en fournissant des APIs de plus haut niveau permettant d'exprimer des comportements plus complexes avec moins de code. Or il en existe un *très grand nombre*. En effet de multiples contextes nécessitent la programmation d'interactions, chacun avec des particularités qui nécessitent des outils spécifiques : interfaces de bureau, visualisation interactive de données, jeux vidéos, interfaces d'environnements critiques, smartphones, ou encore interfaces sur microcontrôleurs. Chaque contexte bénéficie de bibliothèques dédiées, cependant les contextes partagent souvent des besoins (ex. la gestion de la souris entre interfaces de bureau et visualisations interactives). Ainsi il est courant qu'une bibliothèque soit utilisée hors de son contexte d'origine, et certaines bibliothèques "généralistes" mettent en avant leur utilisabilité dans de nombreux contextes. C'est le cas par exemple de Qt [Qt19], qui cite sur sa page principale les contextes *Embedded Devices*, *Application UIs and Software*, *Internet of Things*, *Mobile*, *Automotive*, *Automation*, *Set-top-box & DTV*, et *Medical*.

Les programmeurs sont donc confrontés entre le choix de bibliothèques spécialisées, aptes à répondre en détails à un contexte d'utilisation mais potentiellement limitées en dehors, et des bibliothèques généralistes, aptes à répondre à de nombreux contextes, mais potentiellement de façon superficielle. En outre, lorsque plusieurs bibliothèques sont utilisées conjointement, un problème supplémentaire d'interopérabilité se pose. Comme l'écrit Huot [Huo13], « *As discussed before, we now have many different tools that address specific problems, but they are difficult or impossible to use together. Different toolkits are based on different abstractions and models, requiring to deal with several programing paradigms, abstractions, data structures and even sometimes programming languages within the same prototype* ». En pratique il faut consacrer beaucoup de temps pour chercher et comparer de telles bibliothèques. Lorsque le choix se porte sur l'une d'elles, on doit investir du temps et des efforts conséquents dans l'apprentissage de son utilisation, sans garantie qu'elle répondra à tous les besoins. C'est particulièrement délicat dans le cas du développement de nouvelles interactions, qui touchent fréquemment aux limites des bibliothèques d'interaction.

1.2.2 Structure des APIs en couches

Alors que le premier frein à la programmation de nouvelles techniques d'interaction fait état d'une multiplicité *horizontale* des bibliothèques disponibles, le deuxième frein concerne une multiplicité *verticale*. La gestion des entrées et sorties avec les utilisateurs, dans un système interactif, est généralement organisée en couches successives (voir figure 7), formant une hiérarchie de dépendances de pair à pair.

Au plus bas niveau, les pilotes de périphériques (*drivers* en anglais) communiquent avec les puces connectées à la carte mère qui gèrent les prototoles de communication externes (USB, audio, HDMI), et transmettent ces informations au système d'exploitation. Ce dernier fait correspondre les informations reçues en des périphériques *virtuels* (comme le curseur à l'écran), qui permettent d'utiliser uniformément différents types de périphériques (souris, doigt, *eye-tracking*). Ensuite, certains langages

de programmation (rares) interceptent les informations d'entrée/sortie du système d'exploitation, pour les mettre à disposition des programmeurs. Plus haut, le nombre de couches est variable et peut représenter des graphes de dépendances complexes. Cependant, on distingue généralement l'existence de bibliothèques de bas niveau, qui se spécialisent chacune dans un type de périphérique physique/virtuel (ex. OpenGL pour l'affichage, TUIO pour le multitouch, OpenAL pour l'audio). Enfin, à haut niveau se trouvent les frameworks d'applications, qui combinent et donnent accès à tous les périphériques.

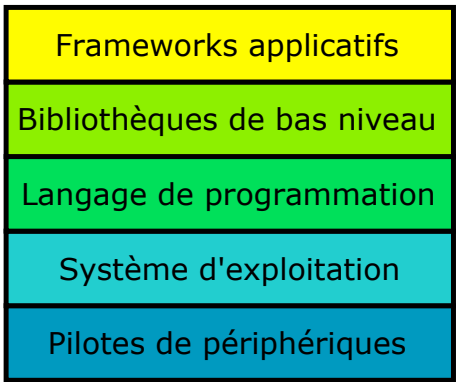


Figure 7 : Structure en couches des différents types de bibliothèques logicielles permettant de gérer l'interaction avec les utilisateurs.

Lors d'une action de l'utilisateur sur la machine, les données générées par les capteurs des périphériques transitent par chacune des couches de cette hiérarchie. Chacune reçoit des informations d'une couche plus basse, l'interprète éventuellement en actions plus complexes (ex. un clic de souris est un appui suivi d'un relâchement), et le met à disposition des couches supérieures. Le même phénomène se produit pour les actions de la machine vers l'utilisateur, même si nous parlons préférentiellement des *entrées*. Avec cette structure, les données de plus haut niveau (qui ont transité par le plus de couches) sont aussi les plus abstraites. Les programmeurs interrogent une seule couche, le plus haut possible, et n'interagissent normalement pas avec les autres couches.

Cependant, cette hiérarchie en couches rend la programmation d'interactions complexe. En effet, chaque transmission d'une couche à l'autre peut voir des informations être ignorées/perdues (ex. la résolution du capteur de la souris), converties dans d'autres unités (ex. un déplacement en millimètres de la souris vers un déplacement en pixels), voire corrompues lors de la traduction. Il arrive aussi qu'une couche fusionne des informations, et perde trace de leur cause. C'est le cas par exemple lorsqu'un bouton est déclenché au niveau du framework applicatif, il est parfois impossible de savoir si c'est un clic de souris ou un raccourci clavier qui l'a déclenché.

Du point de vue du programmeur cherchant à prototyper une nouvelle technique d'interaction, il faut déjà choisir entre toutes les couches disponibles, où intercepter les données d'interaction. Chaque couche dispose d'une API qu'il faut apprendre au préalable, ainsi qu'un modèle de programmation potentiellement différent des autres (*callbacks*, évènements, *listeners*). Lorsqu'une donnée est manquante, il faut s'adresser à une couche plus basse, apprendre l'utilisation de son API, et risquer d'avoir de multiples interlocuteurs. Dans ce cas, on s'expose à des problèmes de cohérence entre couches : rien ne garantit qu'une information propagée par une couche le soit aussi par une plus

haute. Par exemple, si une couche basse signale un appui de bouton de souris suivi d'un relâchement du même bouton, on ne peut pas garantir qu'une couche supérieure interprétera un clic de souris. Le programmeur doit donc considérer que les informations interceptées à plusieurs niveaux peuvent ne pas être cohérentes, et les comparer à chaque fois pour "rétablir" leur cohérence. Dans l'ensemble cette organisation en couches tend à augmenter la tâche du programmeur, tant dans le temps passé à appréhender la topologie des couches, que dans les efforts pour assurer la cohérence des données interceptées.

1.2.3 Extensibilité des bibliothèques

Le troisième frein au prototypage de nouvelles interactions est la faible extensibilité des bibliothèques logicielles, et plus précisément des frameworks applicatifs. Tout d'abord, dans le domaine de l'IHM, les interfaces graphiques sont majoritairement codées en utilisant le modèle *objet*. Dans ce modèle, tous les éléments du programme (entiers, chaînes de caractères, boutons, fenêtres, etc.) sont des objets. Un objet est une collection de variables et de fonctions (les méthodes), qui ne fait que réagir à son environnement par des appels de méthodes, et gère son état en manipulant ses propres variables. Dans les frameworks, tous les éléments graphiques interactifs (boutons, labels, *sliders*, etc.) sont représentés par des objets, qu'on appelle alors des *widgets*. Ainsi toute interface est un assemblage de widgets, qui communiquent entre eux et avec leur environnement par des appels de méthodes.

Pour un chercheur, la création d'une nouvelle forme d'interaction revient normalement à créer de nouveaux widgets. Cependant, ces objets sont intrinsèquement complexes. Ils doivent gérer de nombreuses interactions avec leur environnement (être affichables, redimensionnables, réagir au clavier, être compatibles pour l'accessibilité aux handicaps, etc.), qui doivent nécessairement être implémentées lors de la création d'un nouveau widget. On a tendance à qualifier ces frameworks qui se basent sur des objets lourds et peu flexibles, de "monolithiques" [Bed04]. Face à cette situation, l'ajout d'une nouvelle fonctionnalité est une opération difficile, qui nécessite de comprendre (au moins en partie) les comportements gérés par les widgets, les dynamiques entre objets dans le programme, et les structures de données utilisées pour stocker les widgets.

En pratique, en plus de leur complexité structurelle, les frameworks sont souvent peu extensibles, principalement car c'est un besoin secondaire pour leur base d'utilisateurs. Les APIs proposées pour intégrer des extensions sont souvent des fonctions internes, qui n'ont pas nécessairement été conçues pour être exposées, ni beaucoup testées dans des cas d'utilisation avancées. Des bugs peuvent subsister pour de tels usages avancés, et recevoir peu d'attention étant donné le faible intérêt pour d'autres utilisateurs potentiels. La documentation de telles portions d'APIs est souvent insuffisante, d'autant que pour des fonctions internes l'API est plus souvent sujette à évoluer, donc il faut régulièrement corriger leur documentation. Le faible nombre d'utilisateurs avec des besoins analogues étant faible, l'aide en ligne est principalement limitée, et on y marche "hors des sentiers battus". Enfin, il arrive que les frameworks intègrent des présomptions dans leur code, qui compromettent sérieusement la flexibilité et le remplacement de certaines fonctionnalités. C'est le cas par exemple pour le changement de la fonction de transfert du pointeur, qui fait correspondre les déplacements de

la souris avec les déplacement du curseur à l'écran [Cas11]. Cette modification est impossible dans un framework comme Qt, et nécessite de faire appel au plus bas niveau, mettant en défaut la flexibilité du framework qui devrait pouvoir exprimer ce type de changements.

Pour finir, dans cette section nous avons souligné la multiplicité *horizontale* des bibliothèques à choisir, la multiplicité *verticale* des couches interdépendantes dans les systèmes interactifs, et les difficultés d'extension des frameworks logiciels. Maintenant que nous avons présenté l'écosystème de la programmation d'interactions et les problématiques liées au prototypage de nouvelles techniques d'interaction, il convient de s'intéresser aux besoins des *chercheurs* eux-mêmes.

1.3 État des connaissances sur le prototypage d'interactions

Dans le cadre de cette thèse, il est important de faire un état des connaissances sur le prototypage de techniques d'interaction dans un contexte de recherche. En effet, les outils de programmation d'interfaces et d'interaction ont été principalement conçus pour des utilisateurs souhaitant réutiliser des modalités d'interaction standard et éprouvées (ex. boutons, onglets, raccourcis clavier), et acceptant des interfaces stéréotypées. Leurs besoins sont bien connus, et ont contribué à façonner les outils actuels.

Pour un contexte de recherche et de prototypage de nouvelles techniques d'interaction, les besoins sont moins bien connus, ce qui amène à mettre en question la pertinence des outils de création classiques pour ces usages. De plus, la notion de “prototypage” est floue, elle s'applique dans de nombreux domaines, et n'est pas cantonnée à l'informatique. Nous cherchons donc ici à clarifier l'**activité de programmation de prototypes fonctionnels démontrant des techniques d'interaction avancées**. L'état des connaissances présenté ici doit nous permettre de décrire cette activité aujourd'hui, afin de former une base à compléter par les travaux de cette thèse. Nous avons analysé les travaux existants selon un ensemble de questions :

- Qu'est-ce qu'ils nous apprennent concrètement sur l'activité de programmation de techniques d'interaction avancées ?
- À quels contextes s'intéressent-ils ?
- Est-ce qu'ils soulèvent d'autres problématiques que celles que nous avons énoncées ?
- Quelles solutions proposent-ils de réaliser ?

Cet état de l'art est divisé en trois parties. Dans la première partie, nous présentons une vue d'ensemble des types de travaux ayant pour but d'améliorer l'utilisabilité des outils de programmation. Ces travaux sont pour la plupart présentés pour des contextes hors recherche, cependant le fait qu'ils soient souvent testés et validés par des chercheurs nous incite à les mentionner ici. Dans la deuxième partie, nous présentons le domaine de la Programmation Opportuniste [Bra09], qui a eu pour objet une activité de programmation similaire à la nôtre. Dans la troisième partie, nous décrivons les travaux mentionnant ou visant spécifiquement le contexte de la recherche en IHM.

1.3.1 Utilisabilité des bibliothèques logicielles

Le domaine d'étude de l'utilisabilité des outils de programmation est très vaste. La norme ISO 9241-11 [ISO18] la définit comme « *le degré selon lequel un produit peut être utilisé par des utilisateurs identifiés pour atteindre des buts définis, avec efficacité, efficience et satisfaction dans un contexte d'utilisation spécifié* ». L'utilisabilité des outils influence directement la productivité des programmeurs, et a motivé des recherches actives dans ce domaine. Nous n'avons pas pour ambition de toutes les répertorier ici, mais l'étude des facteurs affectant l'utilisabilité peut nous aider à mieux comprendre l'activité de programmation à l'origine de ces facteurs.

1.3.1.1 Qualité des APIs

Une API (*Application Programming Interface*) est l'ensemble des fonctions et types de données proposées par une bibliothèque logicielle pour ses utilisateurs (programmeurs). Elle est l'interface entre les fonctionnalités de la bibliothèque et les programmeurs. La conception d'une API implique le choix des fonctions qui sont exposées (certaines peuvent rester internes à la bibliothèque), leurs noms, ainsi que l'ordre dans lequel elles vont être utilisées. C'est le résultat d'un compromis entre puissance d'utilisation (contrôler chaque aspect de la bibliothèque), et simplicité (utiliser moins de fonctions). L'API est le deuxième artefact auquel sont confrontés les utilisateurs d'une bibliothèque (après la documentation). Elle conditionne la quantité de code qu'ils vont écrire, et le temps qu'ils vont passer à comprendre sa logique. C'est donc un élément majeur de l'*expérience utilisateur* de toute bibliothèque logicielle.

Les contributions caractéristiques liées aux API ont pour objets de :

- déterminer ce qu'est une *bonne* API [Blo06, McL98]
- déterminer ce qu'est une *mauvaise* API [Pic13, Gri12, Zib11, Hen09]
- mesurer le succès ou l'utilisabilité d'une API [Bor05]
- faciliter leur apprentissage par des utilisateurs [Dua12, Rob11, Ko11, Rob09]
- favoriser leur utilisation "correcte", c'est-à-dire comme l'ont anticipé leurs concepteurs [Sai15]
- permettre l'évolution d'APIs existantes pour accompagner celle des bibliothèques [Sty08]

Par les connaissances qu'ils nous donnent sur l'utilisation des APIs, ces travaux soulignent que l'activité de programmation repose sur la formation d'un modèle mental de la bibliothèque utilisée, et qu'on observe principalement les symptômes de son inconsistance [Ko11, Gri12, Dua12]. Dans cette optique, l'apprentissage est une part importante des efforts consacrés à l'utilisation d'une bibliothèque [Rob11], et les concepteurs d'APIs reconnaissent que leurs choix de conception ont des conséquences importantes qu'il faut anticiper. Enfin, les recommandations faites par ces travaux nous révèlent des hypothèses sur les programmeurs, qui sans être démontrées explicitement, peuvent être inférées à partir des observations. Ainsi les programmeurs seraient réfractaires au changement [Blo06], consacrent peu d'efforts à la lecture de documentations [Dua12], ne font jamais ce qu'on attend d'eux [Blo06], et supportent mal l'ambiguïté dans une documentation [Bor05].

Les problématiques soulevées ici sont principalement la complexité des APIs, et le temps passé à les utiliser. De plus, les travaux que nous avons étudiés décrivent peu les contextes dans lesquels ils s'appliquent, se concentrant principalement sur l'expérience de programmation de leurs participants, et décrivant des tâches génériques pouvant s'appliquer dans plusieurs contextes. Ces travaux s'inscrivent principalement dans un cadre industriel, pour lequel l'optimisation d'une API doit avoir un impact mesurable [Bor05], qui puisse justifier en retour ce qu'on y investit.

À la lumière de l'importance des choix de conception, les solutions proposées sont souvent d'étudier soigneusement des scénarios d'utilisation avant de concevoir une API [Sty08]. Certains auteurs proposent aussi de fournir des exemples et une documentation cohérente pour faciliter la construction d'un modèle mental consistant [Rob11, Ko11], et de concevoir de nouveaux outils pour l'exploration et l'utilisation d'une API [Dua12].

1.3.1.2 Programmation par les utilisateurs finaux

La Programmation par les Utilisateurs Finaux (*End-User Programming*) désigne les outils permettant à des personnes qui ne se considèrent pas comme des développeurs professionnels, de programmer. Ces outils se substituent généralement aux langages de programmation, en laissant les utilisateurs décrire des comportements et des structures de données, sans nécessiter la maîtrise d'un langage. Les tableurs comme Excel sont un exemple de programmation par les utilisateurs finaux : la manipulation de formules dans les cellules d'un tableur permet, comme avec un langage, de spécifier des comportements complexes de transformation de données.

On trouve de la programmation par les utilisateurs finaux dans :

- les tableurs (ex. Excel, Google Docs, Open/Libre Office)
- la programmation visuelle, avec des paradigmes par flux de données [App09, Dra01], machines à états [App06, Bux90], ou réseaux de Pétri [Nav01, Bea00]
- la programmation par démonstration (ou par l'exemple), dans laquelle le système observe les actions de l'utilisateur pour les reproduire en les traduisant en un programme [Mye86, Cyp93, Lie01]
- les langages de scripts, dans une moindre mesure, lorsqu'ils sont destinés à des utilisateurs avec une faible expérience de la programmation (ex. Arduino, Processing, PHP)

Ce type de programmation est particulièrement utile dans les contextes où se rencontrent des utilisateurs qui ne sont pas des développeurs de formation (ex. domotique, production audiovisuelle, arts, administration). C'est aussi le cas dans le domaine IHM, où cohabitent des personnes issues des domaines Informatique, Design, et Sciences Sociales. En outre, c'est un moyen d'améliorer l'adaptabilité des systèmes interactifs, en permettant aux utilisateurs de ces systèmes de les paramétrer et les contrôler au plus près de leurs besoins.

Tout comme les travaux sur l'utilisabilité des APIs, on remarque ici que l'apprentissage et l'expertise jouent un rôle crucial, puisque les travaux discutés dans cette section ont pour but de les réduire. La programmation visuelle se dégage des autres domaines, car elle a été utilisée extensivement pour programmer des techniques d'interaction. En effet, la plupart des techniques d'interaction se représentent bien avec des graphes, où les arêtes sont les actions que doivent réaliser les utilisateurs. Le nombre d'actions à prendre en compte étant souvent réduit, et les tâches

principalement séquentielles (à cause de la difficulté humaine à gérer des tâches simultanées), il est envisageable de représenter ces graphes visuellement. On remarque enfin que beaucoup de travaux de programmation par les utilisateurs finaux sont liés à une construction *incrémentale* des programmes. Cette méthode de programmation consiste à maintenir un programme fonctionnel (mais incomplet) à tout instant, et d'ajouter au fur et à mesure les fonctionnalités manquantes, plutôt que d'obtenir un programme fonctionnel uniquement à la fin du développement. La construction incrémentale est particulièrement adaptée à la programmation des interactions humaines, lorsque celles-ci sont segmentées en tâches (elles-mêmes subdivisées en actions).

Pour ce qui est des problématiques soulevées ici, la programmation par les utilisateurs finaux répond principalement au “ticket d'entrée” élevé de la programmation d'interfaces et d'interactions. Comme nous l'avons évoqué au début de cette thèse, les systèmes informatiques sont des écosystèmes très complexes, qui dissuadent les novices de contribuer à proposer de nouvelles formes d'interaction. Les plateformes proposées ici cachent donc une partie de cette complexité, améliorant l'accessibilité à la programmation d'interactions. En outre, elle répondent à la difficulté pour les programmeurs novices d'*abstraire* le fonctionnement d'une technique d'interaction, quand les actions à réaliser semblent très concrètes. Un périphérique d'interaction pourra par exemple être représenté avec un objet manipulable directement, sur lequel le programmeur pourra attacher des comportements [Dra01]. Enfin, une autre problématique soulevée par ces travaux est qu'il est difficile avec des frameworks “classiques” de contrôler qu'un programme (ou une portion d'un programme) se comporte comme on le souhaite. C'est particulièrement important dans des contextes de sûreté de fonctionnement, pour lesquels il est essentiel d'avoir une vue complète et inambigue du programme. Pour ce problème, des représentations plus riches (telles qu'en programmation visuelle) permettront d'inclure plus de détails, pour décrire exhaustivement l'état du programme.

Les contextes auxquels s'adresse la programmation par les utilisateurs finaux sont très variés et spécifiques, chacun avec des besoins précis :

- Les tableurs sont conçus pour l'automatisation des tâches de bureautique.
- La programmation visuelle est utilisée pour la sûreté de fonctionnement, mais aussi dans les arts, et l'enseignement.
- La programmation par démonstration est utilisée en domotique, qui se prête bien à la définition de règles simples (ex. *Si quelqu'un entre dans la pièce, allumer la lumière*), mais aussi en robotique, et en enseignement.
- Les langages de scripts “accessibles” sont utilisés dans tous les domaines où de nombreux programmeurs sont novices ou autodidactes (ex. Web, informatique embarquée, arts).

Pour finir, la majorité des solutions proposées dans les travaux étudiés ici sont des plateformes complètes et autonomes. Elles contrôlent ainsi en grande partie l'expérience utilisateur, et épargnent aux utilisateurs l'interaction avec d'autres outils. Cette autonomie leur permet de réduire les connaissances nécessaires avant de commencer à programmer, au prix d'une liberté plus limitée pour concevoir des techniques avancées. En outre, dans le cas de l'utilisation de programmation visuelle, une question beaucoup débattue est celle du choix par rapport à un modèle de programmation textuelle [Con13]. Historiquement on les distinguait clairement, par l'utilisation d'un éditeur de texte d'un côté, et d'un éditeur graphique de l'autre. Or aujourd'hui les deux mondes tendent à se rapprocher, avec des éditeurs capables de gérer aussi bien du texte que des formes graphiques. La

question est alors de savoir *comment* combiner le meilleur des deux mondes [Con14]. Un exemple notable est l'exploration d'une interface pour apprendre la programmation par Bret Victor, qui enrichit l'interactivité du texte, et présente des vues graphiques sur le code [Vic12]. Des pistes de solutions sont donc possibles et à explorer, qui intègreraient des éléments de programmation visuelle avec des langages textuels.

1.3.1.3 Environnements de développement intégrés

Un Environnement de Développement Intégré (IDE, pour *Integrated Development Environment*) est un ensemble d'outils, généralement fournis dans un même programme, qui forment un environnement complet de programmation dans le but d'optimiser la productivité des utilisateurs. Parmi les IDE connus, on peut citer Eclipse, Visual Studio, Xcode, ou encore Qt Creator. Ils contiennent toujours un éditeur de texte, et un compilateur/interpréteur pour chaque langage de programmation supporté.

En plus de ces fonctionnalités basiques, chaque IDE se distingue par des fonctionnalités avancées, qui peuvent offrir des gains de temps significatifs à leurs utilisateurs :

- la coloration syntaxique, qui utilise un codage par couleur pour distinguer les différents éléments syntaxiques du programme (mots clés du langage, noms de variables, noms de fonctions, constantes)
- la complétion automatique de code, qui permet sur pression d'une touche d'insérer un nom de fonction ou un bloc de code sans avoir à le taper entièrement au clavier
- la navigation dans les fichiers et classes d'un même *projet* (ou entre projets), afin de passer rapidement des uns aux autres
- le pliage de code, qui permet de cacher des blocs de code source (commentaires, corps de classe/fonction) afin de se concentrer sur le reste
- l'intégration d'outils de tests, de débogage, ou de gestion des versions, accessibles à l'aide de raccourcis clavier, et dans des interfaces directement intégrées à l'IDE
- l'export d'un projet dans une archive prête à installer et utiliser, afin de faciliter sa diffusion

Les environnements de développement sont un domaine de recherche actif, en particulier pour améliorer la contextualité et la pertinence des suggestions de code [Oma12, Moo10, Bru09], la visualisation du code et des dépendances entre blocs de code [Con12, Ase16], et un accès à la documentation intégré aux outils de développement [One12, Dua11, Zho09].

Les connaissances que ces travaux nous apportent sur l'activité de programmation avec un IDE sont d'abord que les programmeurs *font usage* des fonctionnalités de facilitation de code, qui justifient l'intérêt et le succès d'outils comme Eclipse, ou Sublime Text. Ensuite, l'intégration des différents outils dans une même interface nous rappelle que l'activité de programmation (y compris de techniques d'interaction) met en œuvre de nombreux outils : éditeur de texte, compilateur/interpréteur, sauvegarde et synchronisation du code en ligne, tests unitaires, débogage, et génération d'un installeur pour déploiement. Lorsque chaque outil nécessite un apprentissage distinct, des approches plus accessibles comme celles présentées dans la partie précédente sont envisageables. Une autre connaissance à tirer de l'utilisation des IDEs est la coexistence de vues détaillées (le code de l'application) avec des vues d'ensemble (ex. la liste des fichiers du projet, ou des diagrammes

UML [Dur18]). L'activité de programmation requiert donc différents niveaux d'inspection, selon qu'on souhaite raisonner sur l'ensemble du projet ou se concentrer sur une portion précise. Enfin, le succès des environnements de développement souligne l'importance d'une utilisation extensive du spectre visuel humain, avec l'exemple courant de la coloration syntaxique, et des travaux explorant des différenciations plus riches [Ase16, Con14].

Un problème majeur auquel tentent de répondre les IDEs est en effet la trop grande quantité d'information accessible aux programmeurs. Elle inclut la documentation, les tutoriels et forums d'entraide en ligne, les APIs, mais aussi la surcharge visuelle due aux modules de l'interface se partageant l'écran. Cette remarque appuie le problème de complexité que nous avons soulevé au début de cette thèse. Les travaux présentés ici cherchent moins à réduire cette quantité d'information, qu'à la *hiérarchiser*, c'est-à-dire mettre en valeur les éléments plus importants. Ils aident ainsi les programmeurs à composer avec la complexité, lorsque celle-ci semble inévitable.

Pour finir, comme pour les travaux sur la qualité des APIs, les contextes de programmation visés ici sont peu définis, les auteurs s'adressant souvent à des programmeurs avec au moins une certaine expérience. Beaucoup des travaux que nous avons étudiés plaident en faveur de contributions intégrées aux IDEs existants, plutôt que comme des outils indépendants [Moo10, Sty09]. Ainsi que l'écrit Omar à propos de ces deux articles [Oma12], « *Empirical evidence presented in these studies, however, suggests that directly integrating these kinds of tools into the editor is particularly effective* ». Une autre caractéristique utile des outils contribuant à l'activité de programmation est l'observation et l'adaptation au contexte d'utilisation. Bruch et al. l'illustrent par un outil de complétion de code qui trie les suggestions par pertinence dans le contexte courant d'utilisation estimé dynamiquement [Bru09].

1.3.1.4 Connaissances issues de ces travaux

La plupart des travaux que nous avons étudié ici ont en commun un rapport étroit avec la *complexité*. Ils s'intègrent dans un écosystème complexe, qu'ils cherchent à rendre plus accessible pour les programmeurs. Dans le cas des travaux sur la qualité des APIs, la complexité est celle des interfaces avec les bibliothèques logicielles, qui reflète leur puissance et leur expressivité. Dans le cas des outils de programmation par les utilisateurs finaux, ce sont les langages de programmation qui sont en cause. Enfin pour les IDEs, la complexité réside essentiellement dans les sources de documentation, et l'accumulation visuelle d'informations. Dans tous les cas, les travaux que nous avons étudiés cherchent peu à réduire la complexité, mais plutôt à la rendre acceptable. Comme l'écrit Niedoba [Nie19], « *Simplification is necessary. But there is a point, simplification reaches a limit. The limit is an unavoidable complexity* ». On peut ainsi distinguer la complexité de la tâche (qui dépend de celle du système informatique), de la complexité de l'outil pour réaliser la tâche. En admettant que la tâche soit aussi simple qu'elle puisse l'être, c'est l'utilisabilité de l'outil qui est en cause [Nor10].

Ainsi, beaucoup des travaux d'utilisabilité des outils de programmation que nous avons étudiés se concentrent sur l'*interface* entre la bibliothèque logicielle et ses utilisateurs — c'est-à-dire les outils et documents qui permettent aux programmeurs d'interagir avec la bibliothèque. Par ce choix, peu de travaux remettent en question la bibliothèque, son fonctionnement interne, ou sa position dans l'écosystème de bibliothèques auxquelles sont confrontés les utilisateurs. En ce sens, c'est implicitement à l'utilisateur de s'adapter, et l'interface est là pour l'y aider. Nous sommes encore loin

d'un support de la co-adaptation (adaptation de la technologie par les utilisateurs, pour leur contexte) tel que mis en évidence par Mackay [Mac00], ni même d'une adaptation à l'initiative de la machine. Des travaux comme la programmation par démonstration font de l'interface un traducteur (entre l'intention de l'utilisateur et les capacités de l'outil) qui épargne en partie cette adaptation, cependant ils sont limités en puissance d'expression et généralement peu utilisés aujourd'hui.

Ainsi, la seule observation des travaux sur l'utilisabilité des outils de programmation peut nous laisser penser que les tâches de programmation sont préétablies, immuables, et doivent être correctement enseignées. Nous considérons au contraire qu'il convient de remettre en question la nature de certaines tâches, dans une démarche effective de simplification, ce que nous abordons dans ce travail de thèse.

1.3.2 Développement opportuniste et *mashups*

Le développement *opportuniste* [Bra08] et les applications composites (*mashups*) [Cao10, Wei10] font référence à des pratiques de développement étudiées depuis une dizaine d'années, qui se concentrent sur le développement rapide d'application, quelle que soit la qualité du résultat. On relie généralement ce type de pratiques aux communautés de “bidouilleurs”, au mouvement *Do It Yourself*, ainsi qu'à l'étape de prototypage dans un projet. En effet, depuis le développement relativement récent du Web, les outils de développement et de diffusion des “créations informatiques” ont gagné en accessibilité, attirant de nombreux développeurs non-professionnels. L'accroissement du nombre de développeurs aidant, des communautés se sont formées autour de la création informatique (ex. Arduino, Processing, Raspberry Pi), se distinguant des autres communautés par l'inclusion de membres amateurs. Ces développeurs ont des usages particuliers, distincts des communautés plus professionnelles, et ont donc motivé des travaux de recherche afin de les comprendre, et de mieux supporter leurs usages.

Parmi ces recherches, le domaine de la programmation opportuniste s'est développé autour des pratiques de développements ponctuels, marqués par des projets de durées courtes, qui ne s'inscrivent pas dans des projets plus ambitieux. L'édition de novembre 2008 (volume 25, numéro 6) d'IEEE Software en dresse une vue d'ensemble, à laquelle les lecteurs intéressés peuvent se référer [IEE08]. Brandt [Bra08] définit la Programmation Opportuniste ainsi : « *It is an activity where non-trivial software systems are constructed with little to no upfront planning about implementation details, and ease and speed of development are prioritized over code robustness and maintainability* ». Les mashups sont une pratique connexe à la programmation opportuniste, qui s'apparente à la réutilisation et la combinaison d'artefacts qui n'ont pas nécessairement été conçus pour être réutilisés. Yu et al. [Yu08] les définissent dans le contexte du Web par : « *Web mashups are Web applications generated by combining content, presentation, or application functionality from disparate Web sources* ».

La programmation de nouvelles techniques d'interaction dans un contexte de recherche se rapproche de la programmation opportuniste, en ce que les chercheurs se concentrent sur le résultat, souvent au détriment de sa robustesse. Les travaux de ce domaine nous aident donc à clarifier en quoi les activités de programmation en IHM sont “opportunistes”. D'après Brandt [Bra09], elles consistent en :

- l'apprentissage ponctuel de nouvelles compétences et méthodes
- la clarification et l'extension de connaissances existantes
- la réobtention de connaissances jugées inutiles à retenir

Ce type de pratiques répond au manque d'interopérabilité horizontale que nous avons évoquée en problématique. Les concepteurs de mashups sont confrontés à des artefacts aux fonctionnalités complémentaires, mais qui n'ont pas nécessairement été conçus pour cohabiter. Certains ne sont d'ailleurs pas non plus conçus pour être réutilisés. Ces problèmes peuvent être causés par l'insuffisance de documentation, ou l'inaccessibilité au code source, qui les transforment en "boîtes noires" pour les programmeurs [Obr08]. Enfin, comme le suggère la définition de Brandt ci-dessus, la programmation opportuniste est liée à des besoins peu connus en amont, voire changeants. Ce problème limite la robustesse des prototypes, en ce que leur architecture est sujette à changer en cours de projet.

Les contextes liés à la programmation opportuniste et aux mashups sont peu détaillés par rapport aux domaines d'utilisation des artefacts créés. Cependant, tous les articles que nous avons étudiés sont liés au moins en partie aux technologies du Web. Ce medium a aujourd'hui un rôle central dans la gestion des connaissances, pour leur mise à disposition, leur navigation, et leur référencement. Les solutions proposées s'intègrent principalement avec le Web, et promeuvent son intégration plus étroite avec les outils de développement [Bra09]. Nous rejoignons et appuyons cette recommandation dans les discussions à l'issue de ce chapitre.

1.3.3 Études des besoins de programmation en IHM

La compréhension des besoins des développeurs de systèmes interactifs est un problème étudié depuis longtemps dans la littérature scientifique. Pourtant, c'est aussi l'un de ceux qui sont les moins bien compris aujourd'hui. En effet, à mesure que les technologies évoluent, de nouveaux usages émergent ou deviennent possibles, pour lesquels il faut développer de nouveaux outils [Nor10]. En retour, l'évolution des usages crée une tension (des besoins) sur les technologies, qui les pousse à évoluer, dans un cheminement des *découvertes* à la *maturité* illustré par le modèle BRETAM [Gai91]. Le cycle d'interdépendances entre les évolutions technologiques et d'usages, est quant à lui illustré par le concept de *Designeering Interaction* proposé par Huot [Huo13]. Ainsi, aussi bien les systèmes interactifs que leurs utilisateurs sont des cibles mouvantes, sur lesquels il faut régulièrement réévaluer les connaissances.

1.3.3.1 Études des besoins des designers

Parmi les catégories de développeurs, les *designers* ont souvent été au cœur des études de besoins. On leur associe l'expertise de la conception d'interfaces graphiques, mais aussi une moindre maîtrise de la programmation, qui les rend plus exigeants sur ce point. Ils sont donc souvent considérés comme des utilisateurs extrêmes, car leurs difficultés à programmer sont exacerbées par rapport aux programmeurs de systèmes interactifs. Certains travaux se focalisent donc sur ces utilisateurs, avec l'hypothèse que les contributions qu'ils en tirent puissent se généraliser à d'autres contextes.

Ainsi en 2008, Myers et al. ont observé dans une étude de 259 designers, que 86% considèrent que les comportements sont plus difficiles à prototyper que l'apparence, et que 78% du temps ils doivent collaborer avec des développeurs pour concevoir les comportements interactifs [Mye08]. Ils suggèrent le développement d'outils permettant d'explorer des comportements plus complexes qu'en les sélectionnant dans de simples menus, et d'en tester plusieurs simultanément. Ils appellent aussi à une meilleure intégration des outils de code aux outils de dessin.

En 2009, Grigoreanu et al. ont observé que les besoins des designers sont peu connus, et qu'on connaît mal leurs difficultés avec des outils tels que Adobe Dreamweaver, Adobe Flash, et Microsoft Expression Blend [Gri09]. À partir d'une étude complète incluant des discussions sur des listes de diffusion et forums d'utilisateurs, ainsi que 10 interviews de designers, ils extraient 20 besoins régulièrement exprimés par les designers, et classés par importances perçues. Deux besoins se dégagent des autres : (i) représenter comment les données, évènements, et autres ressources circulent dans l'application, et (ii) s'assurer que l'interactivité (*feel*) de l'application est conforme à l'intention de l'auteur. L'étude est notable pour le pragmatisme des besoins formulés. Tous peuvent directement être utilisés pour suggérer de nouveaux outils, et les auteurs incluent même une discussion sur la conversion des différents besoins en outils.

Dans le cadre d'un projet industriel de surveillance maritime, Letondal et al. ont étudié les besoins des designers liés à l'utilisation d'architectures dirigées par les modèles [Let14]. Ils cherchent ainsi à adapter et améliorer les modèles qu'ils utilisent, et leur processus d'ingénierie d'applications interactives en général. À l'aide d'interviews en situation et de conception participative, ils concluent sur l'utilité des modèles comme support d'échanges entre designers et ingénieurs, bien qu'ils ne soient pas utilisés pour générer des interfaces. Ils suggèrent aussi un meilleur partage des représentations, afin d'améliorer la collaboration entre équipes.

Parmi ces travaux, on remarque l'importance donnée à la transparence des applications. Ce point rejoint le “Gulf of Evaluation” de Norman évoqué plus haut [Nor88]. Le manque d'informations sur l'état interne des applications force les designers à le deviner à partir de ce qu'ils observent. Enfin, tous les travaux suggèrent le développement de nouveaux outils pour faciliter l'activité de programmation par les designers. En ce sens, ils rejoignent les travaux sur l'utilisabilité des bibliothèques logicielles.

1.3.3.2 Besoins en outils de programmation

Il est bien connu que la programmation d'interactions avec des utilisateurs humains est différente de celle d'algorithmes [Bea08]. L'informatique étant née avec le calcul scientifique, les premiers langages de programmation ont été conçus pour le calcul. Ainsi, les concepts de base des langages utilisés aujourd'hui sont issus du calcul (fonctions, expressions arithmétiques, structures de données). Ces langages relèguent souvent la programmation d'interactions à un rang secondaire, et la rendent naturellement difficile [Mye94] Des initiatives de recherche ont donc visé les langages et outils de programmation, afin d'y améliorer le support de l'interaction.

Parmi ceux-ci, en 2010 Letondal et al. se sont concentrés sur les besoins d'utilisabilité des outils de programmation [Let10]. Ils reconnaissent que les nombreux outils, langages, patrons d'architecture et modèles proposés pour supporter l'interaction n'ont pas eu d'influences significatives. Ils dressent donc une liste de 12 exigences pour ces travaux, construite à partir de l'étude d'une cinquantaine de

travaux précédents. Ces exigences sont formulées comme des discussions de haut niveau sur les objectifs qu'ont partagé des travaux notables. Elles on pour but d'orienter les travaux futurs pour mieux correspondre aux besoins des programmeurs d'interactions.

Dans le contexte de la conception du framework *djnn*, Chatty et Conversy ont cherché à caractériser les langages futurs pour les concepteurs de systèmes interactifs [Cha14]. À partir de trois thèmes de recherche en ingénierie des systèmes, ils formulent six directions de recherche, pour orienter l'évolution des langages adaptés à l'interaction : étendre les fonctionnalités, unifier les concepts, formaliser les concepts, étendre les concepts des langages (leur sémantique), étendre les notations des langages (leur représentation), et consolider les résultats passés. Ces travaux ont accompagné l'évolution de djnn, et mené à la création du langage de programmation Smala [Mag18]. Parmi ces directions de recherche, l'extension des fonctionnalités se rapporte aux contributions par les outils que beaucoup d'autres travaux ont proposés. L'étude des concepts de programmation, et en particulier leurs liens aux langages de programmation, est une démarche originale, qui s'est très bien illustrée par la *réification* du concept de binding [Cha07] en des opérateurs du langage Smala ($\rightarrow$ et $\Rightarrow$). Nous fournissons une autre illustration de cette démarche dans cette thèse, pour les animations. Enfin la consolidation des résultats existants est une démarche intéressante, qui mérite d'être appuyée. L'écosystème de l'informatique évoluant rapidement, des travaux de fond peuvent avoir du mal à s'imposer, lorsqu'ils nécessitent du temps pour arriver à maturité. Ce phénomène peut expliquer la relative inertie des architectures des interfaces, qui ont peu évolué dans les dernières décennies.

1.3.3.3 Limites et opportunités de recherche en besoins de programmation

La plupart des travaux étudiant les besoins en programmation d'IHMs se basent sur des populations externes aux équipes de recherche. Ce sont des designers, des ingénieurs, ou encore des utilisateurs finaux. Rares sont ceux qui observent spécifiquement des chercheurs concevant de nouvelles formes d'interaction. Pourtant cette catégorie de programmeurs lutte aussi face à la complexité des outils de programmation. De plus ce sont des utilisateurs extrêmes, comme les designers, puisqu'ils utilisent les frameworks d'interaction au-delà des cas d'utilisation prévus. Le contexte de la recherche se distingue des autres contextes d'utilisation par les points suivants :

- des temps d'itération courts, le temps d'évaluer si un projet a des chances d'aboutir
- une population formée à l'Informatique, de bon niveau en programmation
- la combinaison fréquente de multiples bibliothèques logicielles, pas forcément conçues pour cohabiter
- une utilisation “avancée” des frameworks hors du cadre de l'utilisation prévue et recommandée

Le dernier point est caractéristique de la démarche de recherche, et est un point important qu'il nous faut encore clarifier dans ce chapitre. Il s'agit des **besoins des chercheurs, dans le cadre spécifique de la programmation de nouvelles techniques d'interaction**. De nombreux travaux présentés ici ont cherché à comprendre l'activité de prototypage dans son ensemble, dans l'enchaînement de ses étapes, et là où on rencontre des difficultés. Or peu d'études se penchent à plus bas niveau sur les problèmes spécifiques rencontrés par les utilisateurs de frameworks d'interaction. La conséquence est qu'on peut aujourd'hui proposer des outils pour *accompagner* le développement de prototypes de recherche, mais pas remettre en question les outils déjà utilisés, faute de savoir

comment ils devraient être améliorés. Il en découle une accumulation d'outils, qui contribuent malgré eux à alimenter la complexité de l'activité de programmation en IHM. L'objet des études qui suivent est donc de combler ce manque.

De nombreux travaux précédents ont reconnu la difficulté de programmer l'interaction, et ont proposé des directions de recherche pour améliorer la situation. Or ces propositions sont souvent de haut niveau, et il est difficile de les capitaliser en évolutions pragmatiques des outils de développement. Par exemple, dans les six pistes de recherche appuyées par Chatty et Conversy, les "concepts" évoqués sont difficiles à lier à des exemples de réalisations possibles. Ils s'adressent principalement à un public de chercheurs en IHM, qui peuvent comprendre la portée de ces concepts dans le cadre de la programmation d'interactions. Ils leur permettent aussi d'expliquer et de justifier leurs travaux en cours (voir section 2.1.5.1). Ce problème fait écho aux travaux sur l'utilisabilité des bibliothèques logicielles, qui contribuent beaucoup par l'intermédiaire de nouveaux outils, et recommandent souvent ce type de contributions. Nous en sommes venus à nous demander s'il existe **d'autres manières de contribuer à la programmation de techniques d'interaction**, que par de nouveaux outils de développement. Cette question alimente principalement les discussions à la fin de ce chapitre, ainsi que la conclusion de ce manuscrit.

Dans les sections qui suivent, nous présentons deux études qui ont eu pour but d'acquérir une compréhension plus fine des besoins pratiques des programmeurs, pour le développement de prototypes de recherche en IHM.

1.4 Interviews de chercheurs du domaine

La première étude durant ce travail de thèse a consisté à observer et interroger des chercheurs sur la programmation de techniques d'interaction. En premier lieu, nous avions besoin d'un aperçu de ce en quoi consiste le développement de techniques d'interaction dans un contexte de recherche, pour identifier des pistes de points à améliorer. En second lieu, il nous fallait énumérer les problèmes que les chercheurs rencontrent le plus, qui nous aideraient à expliquer pourquoi l'implémentation de techniques d'interaction est si difficile. En effet, peu de travaux on étudié en particulier la population des chercheurs en IHM. De plus, alors que beaucoup se sont concentrés sur l'activité de prototypage en général, nous visons spécifiquement les frameworks utilisés par les chercheurs. Ce type d'étude est inédit à notre connaissance, et nous espérons ainsi contribuer à l'amélioration des frameworks dans des contextes de recherche. Ainsi, nous avons d'abord choisi de conduire des *interviews* de chercheurs en IHM. Nous détaillons d'abord le protocole d'étude des interviews, puis en présentons les résultats, et enfin les analysons.

1.4.1 Protocole de l'étude

Cette étude se base sur les principes de la théorie ancrée (*grounded theory*) [Gla17], qui consiste à collecter des données phénoménologiques, sans a priori, et y chercher des motifs récurrents et du "sens". Nous avons donc conduit des interviews de concepteurs de techniques d'interaction, en cherchant à comprendre l'activité de conception dans son ensemble.

1.4.1.1 Sélection des participants

Notre étude se place dans le contexte du prototypage de nouvelles techniques d'interaction. Nous avons donc cherché des participants développant ou ayant développé de nouvelles techniques d'interaction. Comme critère de sélection nous demandions aux éventuels participants s'ils s'étaient sentis limités par leurs outils de programmation. Ce critère n'était pas strictement nécessaire mais favorisait la sélection des candidats, car nous nous attendions à ce qu'ils aient des besoins à énoncer. Nous avons recruté des chercheurs parmi les membres de notre équipe de recherche, et en priorité ceux avec le plus d'expérience. Cette proximité a pu créer un “biais de proximité”, où les participants exagéreraient certains problèmes pour nous satisfaire. Or la majorité des participants aux interviews étant des experts internationaux en IHM, ils sont familiers de ce type de problèmes, et nous considérons qu'ils ont su rester objectifs pour ne pas biaiser les résultats de l'étude. Au total, 9 interviews ont été réalisées (6 chercheurs, un ingénieur, un doctorant, et un étudiant de Master), pour une expérience moyenne en programmation de 14 ans.

1.4.1.2 Plan des interviews

Durant chaque interview, nous passions en revue 2~3 projets du participant. Pour faciliter le choix, nous proposions a priori une sélection de projets, choisis parmi ceux référencés sur la page personnelle de chaque participant. Nous demandions en particulier des projets pour lesquels ils avaient eu le sentiment de “hacker”, ou avaient été limités par leurs outils. Le plan d'interview était conçu comme un guide dans le cycle de conception de chaque projet et les différentes activités mises en oeuvre, en se concentrant sur les problèmes rencontrés liés à l'utilisation de bibliothèques d'interaction. Ce plan accompagnait une analyse exploratoire, destinée à abstraire les problèmes du cadre d'une bibliothèque d'interaction en particulier. Néanmoins, nous avons aussi cherché à comprendre comment les programmeurs perçoivent leur activité, dans le cadre de la conception de nouvelles techniques d'interaction. Le plan prévoyait donc aussi de demander à chaque participant de définir les notions de *hacking* (ou bidouillage) et *bas niveau* dans ses projets (voir le plan en annexe).

Les interviews étaient semi-directives [Wen01], pour laisser à l'interrogateur le soin d'approfondir un aspect pertinent si nécessaire. Le plan abordait les points suivants : choix du framework, nature et nombre des réécritures de code, nature du prototype initial, ambitions initiales et réalisation effective, perception du *hacking* et du bas niveau, perception de la propreté du code, ressources pour l'apprentissage du framework, et partage du code pour une communauté. Ce plan était conçu comme un guide dans les différentes “étapes” d'un projet, en commençant par le choix des outils, puis les premiers prototypes, et en finissant par la diffusion du travail. Il supportait ainsi une étude exploratoire de l'activité de programmation, et plus particulièrement l'utilisation de frameworks dans un contexte de recherche. Nous avons utilisé la méthode des incidents critiques (*Critical Incident Technique*) [Che98] pour aider les participants à se remémorer chaque projet ainsi que leurs principales difficultés.

De plus, nous avions un nombre d'hypothèses à tester, issues de l'expérience personnelle, pour lesquelles nous avons pris soin d'éviter d'encourager des réponses positives ou négatives de la part des participants :

- que les ambitions initiales sont souvent revues à la baisse à cause de limites des frameworks
- que les participants considèreraient leur code plus sale s'ils pensaient avoir fait du *hacking*
- que les participants préfèreraient une meilleure documentation à une meilleure API
- que les participants préfèreraient partager leur code en tant que projet indépendant plutôt que l'intégrer à un projet existant

En pratique, les participants nous ont souvent spontanément partagé les *techniques* (ou *stratégies*) qu'ils avaient utilisées pour surmonter chacun des problèmes. Bien que le plan n'était pas initialement conçu pour recueillir ces techniques, nous les avons trouvées intéressantes, et avons considéré que leur connaissance pourrait contribuer à l'amélioration des outils de programmation. Nous les avons donc étudiées et classifiées durant ce travail.

1.4.1.3 Déroulement de l'étude

Pour chaque entretien, nous avons demandé au participant d'utiliser son ordinateur de travail (portable ou de bureau), afin d'aider à remémorer le projet, et montrer du code lorsqu'approprié. Les interviews se sont déroulées dans les bureaux des participants lorsqu'ils étaient inoccupés, autrement nous nous déplacions dans une pièce calme pour réduire les interférences à l'enregistrement audio. Chaque entretien était mené par un seul intervenant, et un seul participant.

Tous les entretiens ont été conduits en français (langue maternelle de 8 participants), puis transcrits et analysés dans cette langue, afin de ne pas introduire de biais lié à une éventuelle traduction intermédiaire en anglais. De plus, l'usage de langage familier nous a permis d'observer les *nuances* dans les propos des participants, et d'identifier avec quelles importances les différents problèmes étaient perçus.

1.4.1.4 Collecte des données

Tous les entretiens ont été enregistrés, à l'aide d'une application de microphone sur smartphone. Les interviews ont duré plus d'une heure en moyenne, pour un total de 9,6h de fichiers audio. Nous les avons ensuite intégralement transcrites, pour faciliter l'analyse, et pour en assurer la transparence en citant précisément l'origine de chaque observation. Ces transcriptions ont dû être effectuées manuellement, faute d'outils de transcription adéquats. En effet les voix enregistrées manquaient de clarté et étaient parasitées par des bruits de fond, certains participants parlaient vite, et leurs phrases étaient parfois fragmentaires. Des notes prises sur papier durant chaque entretien ont facilité le travail de transcription, pour lever les ambiguïtés lorsque l'audio était difficile à interpréter.

1.4.2 Analyse des données

Nous avons concentré notre analyse sur l'utilisation d'outils de programmation pour prototyper de nouvelles techniques d'interaction. Notre but était de comprendre de façon générale *comment* les participants utilisaient les outils à leur disposition. Nous cherchions en particulier à identifier les difficultés qu'ils avaient éprouvées, et les aspects limitants dans leurs outils. Bien que les interviews soient orientées vers les problèmes rencontrés, en pratique les participants nous ont souvent décrit les

moyens mis en oeuvre pour les résoudre. Ce faisant, ils détaillaient leurs “astuces”, ainsi que les outils qu'ils avaient trouvés utiles pour les mettre en oeuvre. Il nous a semblé que ces moyens pourraient être une connaissance utile à la communauté IHM.

Nous avons donc divisé les analyses en quatre types d'observations : problèmes, besoins, utilitaires, et stratégies. À ces types s'ajoutent trois thèmes que nous cherchions à clarifier avec l'aide des participants : le bas niveau, le *hacking*, et la propreté du code. L'analyse des données s'est faite dans un premier temps en extrayant des observations de chaque type depuis les transcriptions, et dans un deuxième temps en les synthétisant.

1.4.2.1 Extraction des observations

Durant cette première étape, nous avons parcouru les transcriptions de chaque interview, et extrait des observations de chaque type, dans sept fichiers. Chaque observation est précédée d'un numéro de participant, et d'un numéro de ligne dans la transcription correspondante — par exemple P2:L1. Le participant 1 (pilote) n'est pas inclus dans cette étude, le fichier audio n'ayant pas été correctement enregistré, et perdu ensuite.

Les **problèmes** ont été sélectionnés en relevant ce que les participants ont décrit qu'ils ne pouvaient pas faire, ou les limites perçues. Par exemple, P10:L80 décrit « *Le mode d'interaction fonctionne, sauf que je n'avais pas le contrôle sur la vitesse de défilement* ». Les problèmes importants ayant souvent été formulés plusieurs fois avec des termes différents, nous avons ignoré les occurences multiples.

Les **besoins** ont été principalement sélectionnés à partir des requêtes explicites des participants, par exemple P4:L206 « *il y aurait un machin où le système sait pour chaque méthode quel est le pourcentage d'utilisation par les développeurs en général* ». Nous les avons également interprétés à partir de ce que les participants cherchaient à obtenir, avant d'être confrontés à des problèmes. Par exemple, P3:L126 « *Et il y a des fréquences que je galérais à faire. Des fréquences lentes c'est difficile, parce que [...]* » a donné lieu au besoin “*Contrôler un compteur hardware en fréquence*”. Bien que l'observation ci-dessus puisse aussi être interprétée comme “*Paramétrer un compteur hardware plus lent*”, nous avons toujours choisi le besoin le plus général pour faciliter la synthèse ensuite.

Les **utilitaires** sont les ressources et bibliothèques utilisées par les participants *en compléments* des bibliothèques d'interaction, pour le prototypage de nouvelles techniques d'interaction. Nous avons relevé des bibliothèques, comme P2:L34 « *API d'accessibilité pour intercepter les activations de commandes* ». Nous avons aussi relevé des patrons de conception, comme P6:L47 « *Et après on a aussi utilisé décorateur, il n'était pas là, c'est moi qui l'ai ajouté [...]* ». Plus loin dans cette analyse, nous classifions les différents types de ressources utiles énumérées dans cette étude.

Les **stratégies** sont les actions prises après chaque problème ou limitation énoncés. Comme pour les utilitaires, ce type d'observation est sujet à interprétation. Nous avons relevé de façon générale les actions mettant en oeuvre les frameworks, qui allaient à l'encontre des pratiques recommandées et documentées. Plus loin nous identifions des motifs récurrents et les classifions.

Les points sur le **bas niveau**, le **hacking**, et la **propreté du code**, ont été relevés à partir des réponses aux questions dédiées. Par la nature semi-directive des interviews, ces questions étaient secondaires et n'ont pas systématiquement été posées. Elles nous servent principalement à clarifier les différents aspects de l'activité de conception de techniques d'interaction.

1.4.2.2 Synthèse des observations

À l'issue de la phase d'extraction des observations, nous sommes passés de 2519 lignes de transcriptions dans 9 fichiers, à 276 lignes d'observations dans 7 fichiers. À ce niveau les observations sont factuelles, il a fallu les regrouper en catégories, pour pouvoir les synthétiser ensuite. Nous avons donc cherché des motifs récurrents entre les observations, et avons classifié les quatre types d'observations dans des tableaux inspirés de [Dua12], en se concentrant sur l'importance relative entre les items plutôt que leur prévalence dans le groupe de chercheurs étudiés. Dans chaque tableau, le nombre de participants ayant été concernés par chaque point est indiqué, ainsi que le nombre total d'observations s'y rapportant. Chaque point est illustré par une des observations qu'elle synthétise. Enfin il faut rappeler que le nombre d'observations n'est pas le nombre de fois que chaque type d'observation a été mentionné, mais bien le nombre d'observations *uniques*, d'où les nombres faibles en apparence.

Pour les **problèmes** (tableau 1), nous avons formé un premier niveau de catégories à partir des différentes étapes d'ingénierie d'une bibliothèque logicielle : définition des fonctionnalités, API, documentation, évolutivité, correction de bugs, et architecture interne. Dans ces catégories, nous avons classé les observations en 25 sous-catégories, formant les *types de problèmes* rencontrés par les personnes que nous avons interrogées.

Pour les **besoins**, après l'extraction des observations il nous est apparu que les problèmes en étaient fortement corrélés. Par exemple, au problème "*P2 : La tooltip est un objet statique au framework, qui ne peut être instanciée qu'une fois, et ne peut donc pas être utilisée simultanément en plusieurs endroits*" correspond le besoin "*P2 : Afficher un texte à côté de plusieurs boutons, dans le style des tooltips*". Comme nous avons relevé plus de problèmes que de besoins (principalement grâce à la méthode des incidents critiques), nous avons choisi de nous concentrer sur les premiers.

Pour les **utilitaires** (tableau 2), nous avons classé les observations en fonction de ce que permettent les outils de programmation, en nous aidant du premier niveau de catégories des types de problèmes. Étant donné que seules 28 observations ont été recueillies, cette classification n'a pas pour but d'être exhaustive, mais d'alimenter les suggestions d'outils pour aider au prototypage de techniques d'interaction.

Pour les **stratégies** (tableau 3), nous avons formulé des catégories initiales à partir de verbes d'action (ex. intercepter, reproduire, détourner, ou combiner). Nous les avons ensuite raffinées et regroupées en trois groupes (Obtention de données, Interaction avec d'autres bibliothèques/outils, et Utilisation opportuniste), afin de faciliter leur lecture dans le tableau.

Problèmes	participants	observations
Fonctionnalités		
Comportement insuffisamment déterministe/spécifié — *P3 : Parfois le framework laisse passer des press sans release, parfois des release sans press*	4	6
Absence d'une fonctionnalité utile au prototypage — *P7 : diff ne supporte pas les déplacements*	4	5
Mauvais choix d'ingénierie compliquant le travail — *P3 : L'horloge sur Arduino est gérée en software, la rendant sensible aux ralentissements user*	4	5
Inadéquation entre offre et demande — *P6 : Les trackpads Apple donnent un déplacement relatif plutôt qu'absolu*	3	3
API		
API incohérente/illogique — *P7 : Un appui long sur une flèche clavier se traduit par une séquence d'évènements, plutôt qu'un évènement avec durée*	3	4
API insuffisamment contrôlable — *P10 : Impossibilité de contrôler la vitesse de scroll lors d'un drag*	3	6
Fonctions/données cachées/interdites au programmeur — *P2 : Certains évènements système ne sont pas observables par les applications (ex: coins actifs)*	5	6
API fournissant trop de données — *P4 : Les structures de données des APIs Windows sont "un peu barbares", il est difficile de savoir comment les utiliser*	1	1
API trop complexe à l'usage — *P4 : Le GridBagLayout est trop complexe pour être utilisable*	1	2
Documentation		
Manque de contexte et d'exemples — *P5 : Manque de documentation claire pour créer un évènement custom pour Qt*	3	6
Documentation fragmentée/parcellaire — *P4 : L'information est "parcellaire", on la récupère au fur et à mesure depuis différentes sources*	2	2
Outils de documentation insuffisants/passifs — *P4 : A partir du nom d'une méthode, "ok je fais comment"*	2	3
Comportement significatif non documenté — *P9 : Vicon ne documente pas tous ses formats de données utiles*	3	4
Recherche demandant trop d'investissement — *P5 : Étendre les signaux/slots à des évènements custom demande beaucoup de travail*	3	5
Limites et inconsistances		
Problème de changement d'échelle — *P2 : La tooltip est un objet statique au framework, qui ne peut être instanciée qu'une fois, et ne peut donc pas être utilisée simultanément en plusieurs endroits*	4	4
Limitation artificielle (codée) du framework — *P2 : Le dessin d'un bouton est clampé aux bornes du bouton*	2	2
Comportement inconsistant dans le temps (versions) — *P8 : Abandon par Apple de la rétro-compatibilité avec l'API Carbon*	2	4
Fonctionnalité inconsistante entre Systèmes — *P7 : L'utilisation du scrolling (molette) sous Java est difficile entre Windows et Mac*	2	2
bugs et ralentissements		
Fonction documentée donnant un mauvais résultat — *P4 : Sous Windows, la densité de pixels donnée par les fonctions système est fausse*	2	2
Erreur/plantage non déterministe — *P5 : Les callbacks des sockets réseau de Qt ne fonctionnent pas tout le temps*	1	1
Comportement inutilement/anormalement lent — *P5 : L'utilisation d'un thread dédié pour lire un port série fait lagger Qt*	4	5
Instrumentation coûteuse en performances — *P5 : Pour des applications interactives lourdes, le debugger et Valgrind nécessitent trop d'efforts et de ressources machine*	1	1
Aspects internes		
Interférence non anticipée d'un comportement interne — *P3 : Les timers préconfigurés par Arduino interfèrent avec les timers utilisateurs*	2	3
Manque de stockage des traces d'exécution — *P2 : Un déclenchement de commande ne conserve pas l'historique des activations qui ont conduit à son occurence*	1	1
Complexité interne compliquant l'introspection — *P2 : Deux éléments activant la même commande le font parfois avec des indirections*	1	1

Tableau 1 : Différents types de problèmes observés durant les interviews

Utilitaires	participants	observations
Outil permettant d'accéder à des données privées — *P2 : API d'accessibilité pour savoir quelle commande est exécutée*	3	3
Classe ou protocole extensible du framework — *P9 : Le protocole OSC, pour ajouter des évènements custom à un flux existant*	6	8
Comportement implicite utile — *P2 : Animation automatique entre des états dans CALayer*	3	3
Outil de déboggage/visualisation — *P3 : printf pour débugr son protocole de communication basé sur du texte*	1	1
Patron de conception/ingénierie — *P8 : Les conventions de nommage, pour économiser du temps à converger sur ses propres règles*	5	6
Documentation active ou bien conçue — *P4 : Complétion de noms de méthodes pour aider à découvrir une API*	3	4

Tableau 2 : Différents types d'utilitaires relevés durant les interviews

Stratégies	participants	observations
Obtention de données		
Instrumentation/Intrusion dans un objet/application existant — *P2 : Injection de code pour instrumenter les exécutions de commandes et déterminer leurs déclencheurs*	5	5
Accès à des données/fonctions privées ou non-exposées — *P3 : Écriture de code assembleur pour accéder à des modes alternatifs des timers sur Arduino*	5	5
Reconstitution de donnée à partir d'états antérieurs/bruts — *P4 : Décodage du descripteur HID à la main*	5	7
Récupération/combinaison d'information de différentes sources — *P4 : Lecture des différentes APIs systèmes pour trouver celle qui renvoie les informations bas-niveau dont on a besoin*	1	1
Interaction avec d'autres bibliothèques/outils		
Réimplémentation fonctionnelle d'un widget/mécanisme existant — *P5 : Dessiner ses propres widgets pour pouvoir faire des formes "bizarres"*	7	9
Inhibition/remplacement d'un comportement existant — *P6 : Filtrage des évènements système pour bloquer et déplacer le curseur manuellement*	3	3
Augmentation/modification d'un mécanisme existant — *P7 : Modification du widget de progress bar pour rafraîchir plus souvent*	4	4
Superposition d'un overlay à une application existante — *P2 : Utilisation d'un overlay Canvas sur le home screen d'une tablette Android pour intercepter les appuis tactiles*	3	3
Extension à l'aide des outils dédiés du framework — *P5 : Création d'évènements custom pour Qt, avec de nouvelles métadonnées*	1	1
Rétro-ingénierie (extraction de connaissances) de boîte noire — *P8 : Détermination de la MTU du réseau par diminution adaptative et statistiques*	5	8
Modification de l'environnement d'un outil pour l'altérer — *P7 : Modifier l'horloge de son PC pour continuer à utiliser le framework*	4	4
Utilisation opportuniste		
Reproduction factice d'application existante — *P5 : Reproduction d'une application Google Maps factice*	2	3
Duplication pour contourner une limite en nombres — *P9 : Brancher 4 souris sur 4 ordinateurs et envoyer les évènements à un programme principal, pour gérer 4 curseurs dans une application Swing*	2	3
Détournement d'un outil/paramètre hors de son cadre prévu — *P2 : Ajout d'espaces dans les noms de menus pour pouvoir tous les dérouler sans qu'ils se recouvrent*	4	5
Introduction d'un nouveau paradigme d'API — *P3 : Développement d'un parseur+protocole générique de communication basé texte (pour débugr) flexible, avec interrogation de "qui est branché à ce port" et commandes extensibles*	3	3
Utilisation complémentaire d'un outil/source de bas niveau — *P4 : Récupération des informations brutes sur l'écran dans la base de registres*	4	5
Combinaison d'outils pour réaliser une tâche plus complexe — *P4 : Combinaison de plusieurs gestionnaires de placement en widgets imbriqués pour réaliser un placement impossible avec chacun*	2	2

Tableau 3 : Différents types de stratégies observées durant les interviews

1.4.3 Résultats de l'étude

À présent, il s'agit de discuter et interpréter ces résultats et observations afin d'en tirer les causes et implications possibles. Nous abordons séparément les problèmes, utilitaires, et stratégies, puis nous nous attachons à clarifier le bas-niveau, le *hacking*, et son lien à la propreté du code.

1.4.3.1 Causes des problèmes rencontrés lors de l'utilisation d'un framework

Dans la classification du tableau 1, nous remarquons que de nombreux problèmes peuvent s'expliquer par une mauvaise compréhension réciproque entre les utilisateurs et les concepteurs de frameworks. Tout d'abord, les problèmes *Comportement insuffisamment déterministe/spécifié*, *API insuffisamment contrôlable*, et *Fonctionnalité inconsistante entre Systèmes* montrent que les participants ont utilisé des fonctionnalités dans des conditions qui n'avaient pas été anticipées (par les concepteurs de frameworks), et probablement pas testées. De plus, avec les problèmes *Mauvais choix d'ingénierie compliquant le travail* et *Problème de changement d'échelle*, les participants ont utilisé plus de ressources du framework que prévu, ce qui n'avait pas été anticipé. Il y a donc un décalage entre les besoins auxquels répondent les frameworks, et les besoins des participants. On ne peut néanmoins pas en conclure que ce décalage concerne la *perception* des besoins par les concepteurs de frameworks, car on peut penser qu'ils ne les prennent simplement pas en compte.

Ensuite, les problèmes *Recherche demandant trop d'investissement*, *Outils de documentation insuffisants/passifs*, et *API trop complexe à l'usage* nous montrent que certains participants ne sont pas prêts à consacrer autant de temps et d'efforts qu'il le faudrait, pour utiliser un framework comme ses développeurs l'ont prévu. Il y a ainsi un décalage entre les hypothèses des développeurs sur les utilisateurs (ainsi que leur expérience d'utilisation), et la réalité.

Enfin, par les problèmes *Inadéquation entre offre et demande* et *API incohérente/illogique* nous remarquons un décalage entre ce que les participants ont voulu faire, et ce que les bibliothèques pouvaient faire. Dans ce cas, il ne s'agit pas seulement de considérer le nombre de fonctionnalités de chaque bibliothèque, car les besoins des participants ont pu être influencés par ce qu'ils pensaient pouvoir faire avec. Il s'agit aussi d'un manque de cohérence dans l'interface des frameworks, qui trompe les utilisateurs dans ce qu'ils veulent y faire. De plus, nous considérons que le problème *Manque de contexte et d'exemples* montre que le modèle conceptuel du framework n'est pas suffisamment explicité, pour une utilisation avancée dans notre contexte d'étude. Il est possible que les développeurs considèrent leur documentation plus claire qu'elle ne l'est réellement, faute d'avoir interrogé les utilisateurs concernés. Ce point se rapporte ainsi au "Gulf of Execution" de Norman [Nor88].

1.4.3.2 Artefacts contribuant à l'activité de prototypage

Parmi les outils de programmation ayant été classifiés comme utilitaires (tableau 2), nous remarquons que la plupart se rapportent à des conventions et des principes unifiants, que les utilisateurs peuvent s'approprier. Par exemple, dans le cas *Classe ou protocole extensible du framework*, les participants bénéficient d'une interface spécifiée explicitement. Une telle interface est utile pour les développeurs du framework, qui savent précisément quels types de sous-classes ou données vont s'y

attacher, les cas imprévus étant exclus. Elle est aussi utile pour les utilisateurs du framework, qui y découvrent comment communiquer avec le framework, de façon complète et non ambiguë. Dans le cas *Comportement implicitement utile*, ce sont des conventions du framework, définies explicitement ou non, mais qui sont normalement appliquées de façon cohérente dans tout le framework. Par exemple, pour *P2 : Animation automatique entre des états dans CALayer*, le participant a pu considérer que toute animation souhaitée pourrait s'exprimer par une simple assignation de variable, plutôt que d'avoir à vérifier au cas par cas comment animer chaque propriété.

De telles conventions sont essentielles dans un framework. Elles forment les éléments du *langage* avec lequel les utilisateurs expriment des interfaces et des interactions. Lorsqu'elles sont suffisamment bien spécifiées, il est possible de les extrapoler dans des situations inédites, en sachant comment elles vont s'y appliquer. Elles permettent donc aux chercheurs de s'approprier le framework, voire de le détourner pour y exprimer de nouveaux usages. Nous illustrons ce point en section 2.4, avec la convention de nommage des propriétés des widgets.

1.4.3.3 Interprétations des types de stratégies observées

Dans le tableau 3, de nombreuses stratégies s'apparentent à la pratique du développement de *mashups* évoquée dans l'état de l'art. En particulier, *Détournement d'un outil/paramètre hors de son cadre prévu*, *Combinaison d'outils pour réaliser une tâche plus complexe*, *Superposition d'un overlay à une application existante*, et *Modification de l'environnement d'un outil pour l'altérer*. Les participants abstraient ainsi le fonctionnement des différents outils et fonctions, pour les combiner et les détourner. Ce faisant, ils peuvent éventuellement construire un modèle mental approximatif simplifié d'un composant, afin de faciliter son appropriation - cf. stratégie *Rétro-ingénierie (extraction de connaissances) de boîte noire*. Par exemple, pour *P8 : Détermination de la MTU du réseau par diminution adaptative et statistiques*, le participant a pu réduire un comportement complexe (la taille maximale des paquets pouvant transiter par Internet varie en fonction des routes prises), à une donnée simple à mémoriser (une valeur maximale de cette taille).

Par ailleurs, nous observons que les participants ont souvent pris l'opportunité de réutiliser leurs propres connaissances, en particulier avec les stratégies *Introduction d'un nouveau paradigme d'API*, *Duplication pour contourner une limite en nombres*, et *Réimplémentation fonctionnelle d'un widget/mécanisme existant*. Ils travaillent ainsi en terrain familier, souvent au prix d'une entorse aux pratiques recommandées du framework. Avec cette observation, nous considérons que les frameworks devraient faciliter leur réappropriation par les programmeurs, plutôt que de chercher à contraindre une manière unique de programmer. Nous rejoignons ici l'Interaction Instrumentale de Beaudouin-Lafon [Bea00], et en particulier les principes de *réification*, *polymorphisme*, et *réutilisation*, qui sont très pertinents dans notre contexte [Bea00].

Pour finir, nous remarquons qu'avec les stratégies *Inhibition/remplacement d'un comportement existant*, *Réimplémentation fonctionnelle d'un widget/mécanisme existant*, et *Reproduction factice d'application existante*, les participants ont amené le système dans un état contrôlé avant de développer dessus. Il s'agit dans ce cas de *réinitialiser* une partie du framework, pour éliminer toute interférence de comportements existants. Dans le cas de la réimplémentation d'un widget, on part ainsi d'une base

neutre, qui ne s'affiche pas et ne réagit pas aux évènements d'entrée. Il nous semble donc important de minimiser les comportements existants, lors de la conception de nouveaux widgets, ce que nous illustrons avec le mécanisme de composition utilisé pour Polyphony, dans le chapitre 3.

1.4.3.4 Clarification du bas-niveau

Durant les interviews, nous avons demandé aux participants de définir ce qu'ils considéraient être *de bas-niveau* dans leur projet — dans leur code ou dans les outils qu'ils utilisaient. En effet, le qualificatif “de bas niveau” revient souvent dans le domaine du développement d'applications avancées, et revêt parfois une connotation négative, synonyme de “source de difficultés”. Sa clarification nous permettra donc d'apporter un nouveau point de vue sur les difficultés rencontrées par les développeurs de nouvelles techniques d'interaction, mais aussi pourquoi ils choisissent de se tourner vers le bas niveau durant leurs projets.

Le bas niveau ne possède pas de définition dans le dictionnaire de l'académie française (en dénote l'absence de trait d'union entre les deux mots). Cependant on l'utilise comme locution adverbiale (travailler *à bas niveau*), locution adjective (une bibliothèque *de bas niveau*), et locution nominale (caractériser *le bas niveau*). Le bas niveau s'emploie rarement seul et est utilisé en Informatique pour caractériser des types d'artefacts logiciels. On parle souvent de *langage de bas niveau*, de *protocole de communication de bas niveau*, ou de *bibliothèque de bas niveau*.

On observe que le bas niveau est quasi systématiquement associé à une organisation en couches, dans laquelle des couches de haut niveau *dépendent* des couches de bas niveau. Effectivement, les langages de programmation sont organisés en couches séparées par des phases de compilation (traduction) — par exemple Python vers C, puis C vers Assembleur, et Assembleur vers Code machine. Le bas niveau désigne ici les langages **plus proches de la machine**. Par extension, on dira qu'un langage est de plus bas niveau qu'un autre s'il permet d'exprimer des opérations plus proches de ce que permet le Code machine, même si les deux langages n'appartiennent pas aux couches d'une même chaîne de compilation (ex. C++ est de plus haut niveau que C mais compile directement vers Assembleur).

Dans le cas des protocoles de communication, les couches sont séparées par des relations d'inclusion des données de haut niveau dans les canaux de communication de bas niveau. On parle généralement du modèle OSI simplifié en 5 couches — transmission physique (câbles), liaison (MAC), réseau (IP), transport (TCP et UDP), et application (HTTP et FTP). Ici, le bas niveau désigne les protocoles plus fondamentaux, qui héritent peu des caractéristiques d'autres protocoles. Ils sont donc généralement **moins fiables**, car ils accumulent moins de fonctionnalités de robustesse aux erreurs et contrôle de réception. Mais ils ont une **meilleure latence**, car celle-ci s'accumule et augmente à chaque nouvelle couche de communication.

Enfin dans le cas des bibliothèques logicielles, les couches sont séparées par des relations de dépendance. Ces relations comprennent l'inclusion et la compilation évoquées pour les protocoles et les langages, donc les caractéristiques énumérées ci-dessus s'appliquent toujours ici. Les bibliothèques échangent des données entre couches, et peuvent éventuellement les transformer. Dans le cas des données d'entrée qui nous intéressent pour les techniques d'interaction, les bibliothèques de bas niveau manipulent des données **plus brutes** puisque plus proches du matériel, moins transformées.

Ces bibliothèques sont aussi **plus rapides**, puisqu'elles exécutent moins de code issu d'autres bibliothèques. Cependant elles sont aussi **moins puissantes**, c'est-à-dire qu'à taille de code équivalente elles réalisent des comportements moins complexes.

Durant l'analyse des interviews nous avons relevé 25 observations liées au bas niveau, dont certaines apportent de nouvelles caractéristiques. Le bas niveau permet ainsi d'accéder à **plus de données**, car elles ont été moins bloquées ou ignorées entre couches — *P10 : Une information plus nombreuse, plus précise, non modifiée en aval*. Les fonctions et comportements réalisés sont **moins biaisés/stéréotypés** — *P8 : « Quand tu veux faire des trucs qui sortent un peu des sentiers battus »*. Les bibliothèques de bas niveau requièrent **plus d'efforts** de développement — *P5 : Pas de widget ou librairie disponible pour réaliser un besoin précis, il faut réimplémenter soi-même*. Elles sont aussi parfois **moins documentées** dans le système d'exploitation, pour pallier à des failles de sécurité ou forcer l'utilisation des couches hautes — *P5 : Fonctions cachées du système, réservées à ses développeurs, pas destinées à des développeurs extérieurs*. Elles sont plus sujettes aux **limitations techniques** étant donné que les bibliothèques de bas niveau ciblent souvent des matériels et contextes d'utilisation spécifiques — *P8 : Lié à des limitations techniques*. Enfin, elles sont **plus sensibles** à une mauvaise utilisation, par un *effet papillon* des dépendances entre bibliothèques — *P7 : Conséquences plus graves si on fait n'importe quoi*.

À partir des points mis en évidence en gras, nous concluons que les programmeurs se tournent vers des bibliothèques de bas niveau pour :

- accéder à des données peu filtrées, peu transformées, et rapidement après leur production
- utiliser des fonctions cachées du système, moins contraintes vers un style de résultats, et plus rapides

Cependant, ils sont confrontés à :

- un code plus lourd, et lié plus étroitement à un matériel ou contexte d'utilisation
- un investissement plus important en temps et en énergie pour apprendre et développer avec le bas niveau
- une moindre fiabilité aux imprévus, et des conséquences plus graves des mauvaises utilisations

1.4.3.5 Hacking et propreté du code

Durant les interviews nous avons demandé aux participants à quel point ils considéraient que leur code était propre, sur une échelle de 1 (très sale) à 5 (très propre), et d'expliquer pourquoi. Nous leur avons aussi demandé d'indiquer s'ils considéraient que leur travail était du *hacking*, puis de définir cette notion dans leur travail. L'action de "hacker" un système est parfois employée dans le contexte du prototypage de systèmes interactifs. Elle est souvent associée à une utilisation avancée, à l'encontre des pratiques établies, cependant le terme a été utilisé dans de nombreux contextes, et est aujourd'hui vague. Tout comme le bas niveau, la clarification du hacking nous permettra de mieux comprendre les besoins des programmeurs d'interactions, ainsi que les problèmes auxquels ils sont confrontés. De plus, nous avons voulu évaluer la perception du code sale, et son lien à l'activité de hacking.

En tant qu'anglicisme plutôt récent, le *hacking* ne possède pas de définition dans le dictionnaire de l'académie française. Nous commençons par la définition donnée sur l'encyclopédie collaborative Wikipédia [Wik19] : « *Le* hacking *peut s'apparenter au piratage informatique. Dans ce cas, c'est une pratique visant à un échange "discret" d'informations illégales ou personnelles. Cette pratique, établie par les hackers, apparaît avec les premiers ordinateurs domestiques. Le hacking peut se définir également comme un ensemble de techniques permettant d'exploiter les failles et vulnérabilités d'un élément ou d'un groupe d'éléments matériels ou humains* ». Cette définition montre que le hacking se comprend en premier lieu comme du piratage informatique, or ce n'est pas notre contexte d'étude, bien que la fin de la définition tente de généraliser l'usage du mot. Elle apporte néanmoins les notions d'**interaction avec un système existant** et de **transgression**. Le dictionnaire anglais Merriam-Webster nous donne plus d'informations sur le mot *hack* :

***hack** noun*

1 : a tool for rough cutting or chopping : an implement for hacking
2 : nick, notch
3 : a short dry cough
4 : a rough or irregular cutting stroke : a hacking blow
5 : restriction to quarters as punishment for naval officers — usually used in the phrase under hack
6a : a usually creatively improvised solution to a computer hardware or programming problem or limitation
6b : an act or instance of gaining or attempting to gain illegal access to a computer or computer system
6c : a clever tip or technique for doing or improving something

Ce sont les définitions 6a et 6c qui nous intéressent et éclairent la pratique du hacking. Elles soulignent les notions de **créativité**, d'**improvisation**, de **résolution de problème**, et d'**astuces**.

Ensuite, durant l'analyse des interviews nous avons relevé 12 observations sur l'activité de hacking. Ainsi, elle peut impliquer une **mise à profit de ressources** sans se borner à un outil en particulier — *P7 : Utilisation des ressources existantes autant que possible*. Elle consiste parfois en un **changement d'environnement** ou de contexte — *P9 : Utiliser une librairie pour quelque chose pour lequel elle n'est pas faite*. Enfin, c'est une activité **difficile** — *P6 : Être bloqué*.

À l'aide de ces caractéristiques de l'activité de hacking, nous pouvons en donner la définition :

Dans le contexte de la programmation de techniques d'interaction, le hacking *est la résolution opportuniste et créative d'un problème lié à un système existant, par improvisation avec toute ressource disponible, et mettant en œuvre des astuces déviant du cadre des pratiques permises.*

En pratique lors des interviews, une partie des participants nous ont confié ne pas se considérer comme faisant du *hacking* — alors que les descriptions de leurs activités correspondaient beaucoup à la définition que nous avons donnée. Cela peut s'expliquer par le sens moins riche en français, synonyme de piratage informatique. Il est aussi envisageable que certains participants considèrent leur processus de création comme *systématique*, plutôt qu'improviste et opportuniste . En ce qui concerne la propreté du code perçue par les participants, sur 10 projets nous avons relevé une moyenne de 3,55 ainsi qu'une médianne de 3,75. Les participants considèrent donc leur code plutôt

propre en pratique. Faute de plus d'observations, et avec le faible intérêt des participants, nous ne pouvons donc pas conclure sur les liens entre hacking et propreté du code, et avons choisi de ne pas poursuivre cette voie.

1.4.3.6 Interprétation des résultats

À l'issue de l'analyse des interviews, nous avons obtenu deux classifications principales, des *problèmes* rencontrés par les développeurs de nouvelles techniques d'interaction, et des *stratégies* employées pour les résoudre. En outre, nous avons clarifié les notions de bas niveau et de hacking, qui constituent une partie importante du contexte de travail de ces développeurs. Pour revenir à la question "**Comment supporter au mieux le développement de nouvelles techniques d'interaction ?**", le type de contribution de ce chapitre de thèse s'oriente vers la mise en évidence de problèmes majeurs à résoudre, ainsi que d'opportunités d'innovations dans les outils de développement.

Cependant à ce niveau, nous ne sommes pas en mesure d'évaluer à quels points les différents problèmes et stratégies sont importants les uns par rapport aux autres. Les nombres d'observations relevées dans les tableau 1 et tableau 3 représentent un petit échantillon de personnes, sélectionnées près de notre équipe de recherche. Ils ne sont donc pas représentatifs de l'ensemble des chercheurs en IHM. De plus, les catégories ont été déduites à partir d'interviews qui n'avaient pas pour objectifs initiaux de formuler des classifications. Nous n'avons pas conçu le plan d'interviews pour explorer systématiquement tous les problèmes et toutes les stratégies des participants. Il a donc été nécessaire de complémenter cette première étude pour compléter et valider nos classifications.

1.5 Questionnaire en ligne

Les interviews de chercheurs en IHM nous ont permis d'observer le manque de compréhension entre les concepteurs de frameworks et leurs utilisateurs, dans le contexte de la recherche. Pourtant les frameworks (Qt, Android, JavaFX, Cocoa, HTML, etc.) sont bien établis dans les pratiques de programmation. Comme nous avons observé dans l'état de l'art, peu de travaux se sont attachés à *améliorer* les frameworks d'interaction, faute de connaissances sur leurs limites. Nous cherchons donc ici à compléter ces connaissances, pour contribuer à l'adaptation des outils existants au contexte de la recherche. Nous espérons ainsi contribuer à réduire de façon générale la difficulté de programmation de nouvelles techniques d'interaction. Les interviews nous ont permis d'avoir un premier ensemble d'observations, qui nous ont permis de dégager des tendances à confirmer. Nous formulons donc les questions de recherche suivantes :

- **Q1 — Quels sont les critères de choix principaux des bibliothèques logicielles utilisées pour programmer des applications interactives en contexte de recherche ?**
- **Q2 — Quels sont les problèmes limitant le plus le travail des chercheurs avec l'utilisation de ces bibliothèques ?**
- **Q3 — Quelles stratégies de développement sont principalement utilisées pour contourner les problèmes rencontrés dans ce contexte ?**

1.5.1 Protocole de l'étude

Pour répondre à ces questions, nous avons opté pour un questionnaire en ligne et en anglais, qui nous permet de recueillir plus de réponses qu'avec des participants locaux, et de réduire le biais d'un domaine de recherche local.

1.5.1.1 Plan du questionnaire

Le questionnaire était conçu pour évaluer différentes réponses possibles à nos trois questions de recherche. Des questions préliminaires recueillaient l'environnement dans lequel les participants programmaient de l'interaction (profession, type d'institution, nombre de collaborateurs), leur expertise dans la programmation d'applications interactives, et la proportion de leur temps de travail passé à programmer. Ces questions étaient destinées à cerner le contexte de la recherche et du développement de systèmes interactifs, pour orienter à terme les propositions de travaux futurs. Ensuite, trois séries de questions évaluaient la pertinence de différents critères pour chacune des questions de recherche : importance des critères de choix des bibliothèques d'interaction (Q1), sévérité des problèmes rencontrés (Q2), et fréquence des types de solutions apportées (Q3). Les critères pour le choix des bibliothèques ont été formulés initialement par notre expérience du domaine, puis raffinés par des questionnaires pilotes. Les sélections des problèmes et solutions ont été formulées à partir des classifications des problèmes et stratégies issues des interviews, et raffinées après les questionnaires pilotes.

Le questionnaire tel que présenté en ligne est inclus en annexe. Nous avons opté pour des questions à choix multiples, afin de limiter tout biais lié à l'interprétation des résultats, et pour gérer éventuellement un plus grand nombre de réponses. L'évaluation de la pertinence des critères se fait avec des échelles de Likert à 5 niveaux, avec pour les problèmes un niveau supplémentaire permettant de distinguer les problèmes jamais rencontrés des problèmes déjà rencontrés mais de sévérité nulle. Pour chaque critère de choix de bibliothèques (Q1) et chaque type de solution (Q3), nous avons inclus un champ de texte libre pour que les participants puissent éventuellement détailler et préciser leurs réponses. Ces champs nous ont aussi permis de réorganiser les différentes catégories à l'issue des questionnaires pilotes. Enfin, avec les critères de choix des bibliothèques (Q1), nous avons demandé aux participants de renseigner les frameworks ou boîtes à outils qu'ils utilisaient le plus, pour pouvoir en faire une première estimation.

Notre hypothèse principale était que les participants utilisent systématiquement des frameworks majeurs plutôt que des boîtes à outils issues de la recherche. Nous avons donc pris soin d'adopter des formulations neutres (“*interaction libraries*”, “*frameworks/toolkits*”) pour ne pas induire les participants à répondre avec un type de bibliothèque donné, et qu'ils considèrent tout type de bibliothèque logicielle permettant de programmer des applications interactives. Concernant les critères de choix des bibliothèques (Q1), nous voulions également évaluer la prévalence de critères subjectifs (ex. réputation, expérience personnelle) par rapport à des critères objectifs plus directement actionnables (ex. performance, qualité de la documentation). L'*objectivité* des différents critères étant sujette à interprétation, nous avons simplement mélangé les critères que nous considérions objectifs, et subjectifs.

Un premier questionnaire pilote a été soumis à quatre participants. Les réponses nous ont permis d'inclure de nouveaux critères de choix des bibliothèques (Q1). Nous avons aussi retiré des problèmes trop spécifiques (Q2) afin de limiter le nombre de critères à évaluer, ainsi que le temps passé à répondre. Enfin, pour les types de solutions aux problèmes (Q3), nous avons retiré toutes les mentions à des exemples factuels, que les participants citaient fréquemment dans les détails des réponses, et qui risquaient de biaiser les résultats en fonction de l'expérience des participants. À cause de la modification des critères de choix des bibliothèques, les réponses au questionnaire pilote ont été exclues des résultats.

Nous avons utilisé le logiciel *open source* en ligne Framaforms [Fra19] pour créer et publier le questionnaire. À cause de la présence de champs permettant de détailler les réponses (Q1 et Q3), Framaforms ne nous permettait pas de mélanger la présentation des différents critères aux participants. Pour ces questions nous avons donc choisi un ordre logique entre les critères (ex. *Tool reputation* → *Developer reputation* → *User community* → *Documentation quality*), afin de réduire l'effort de mémoire à transitionner entre chacun. Nous discutons plus loin de l'influence probable de cette limitation sur les résultats.

1.5.1.2 Sélection des participants

Les participants de ce questionnaire devaient idéalement avoir prototypé de nouvelles formes d'interaction (techniques d'interaction, interfaces, artefacts interactifs) dans un contexte de recherche. Comme il est difficile de caractériser précisément ce qui qualifie des personnes à avoir une expérience suffisante dans ce domaine, nous avons recruté des participants exclusivement au sein de la communauté IHM. Le questionnaire a été diffusé une première fois par l'intermédiaire des listes de diffusion *chi-Announcements@acm.org* et *annonces@afihm.org*, afin d'atteindre un maximum de participants en réduisant le nombre de mails reçus en doublon. Ces premières listes ont ensuite été complétées par des diffusions auprès des anciennes équipes de nos collègues.

L'introduction du questionnaire indiquait clairement le contexte de la recherche, afin de sélectionner spécifiquement des chercheurs : « *The goal of this survey is to better understand the process of programming new interactive artefacts for research and innovation purposes, and the software libraries used to support this process. We are interested in projects where you reached the limits of libraries (e.g. undocumented needs, accessing private functionalities, interfacing with existing applications, combining with other libraries), for the prototyping and implementation of innovative interactive artefacts (e.g. non-standard interaction techniques, data visualisations, UI toolkits, interactive applications)* ».

Nous avons recueilli 32 réponses au total, sur une période de 2 mois. 25 des participants ont comme activité principale chercheur, et 27 travaillent pour une université, école ou institution publique. Deux tiers des participants se disent de niveau *au moins* avancé. Les profils des participants sont donc très pertinents pour le contexte de notre étude. Néanmoins, nous ne sommes pas en mesure d'affirmer s'ils sont représentatifs des proportions générales d'utilisateurs qui prototypent des applications interactives dans un contexte de recherche.

1.5.2 Analyse et interprétation des résultats

Le but du questionnaire était de comprendre les choix de bibliothèques et leurs usages pour prototyper de nouvelles techniques d'interaction, puis de formuler des recommandations pour les concepteurs d'outils ainsi que les chercheurs. Nous avons donc concentré notre analyse sur les aspects des bibliothèques qui importent le plus pour les chercheurs lors de la conception de nouvelles interactions, ainsi qu'à l'opposé les aspects qui importent moins.

Nous pouvons d'abord relever des questions préliminaires que deux tiers des participants disposent de moins de 40% de leur temps pour programmer, et que les participants collaborent en moyenne avec deux autres personnes. En somme, les participants disposent de peu de temps et de moyens pour mener à bien leurs projets, ce qui n'est pas étonnant si on considère que la majorité sont des chercheurs, dont il est généralement admis que le développement informatique est une fraction de leur activité. Cette observation justifie l'idée que les améliorations soient principalement à chercher du côté des outils de programmation, plutôt que des méthodes de programmation des utilisateurs.

1.5.2.1 Bibliothèques d'interaction utilisées

Immédiatement après les questions préliminaires du questionnaire, nous avons demandé aux participants quels frameworks ou boîtes à outils ils utilisaient le plus. Plusieurs réponses pouvaient être données, séparées par des virgules. Les résultats sont présentés en figure 8. Nous y avons représenté en abscisse les noms des bibliothèques citées dans les réponses, et en ordonnée les nombres de participants ayant cité chacune des bibliothèques. À citations égales, les bibliothèques sont classées par ordre alphabétique. Nous avons exclu des résultats les réponses qui n'étaient clairement pas des bibliothèques d'interaction (“Atom” et “Eclipse”).

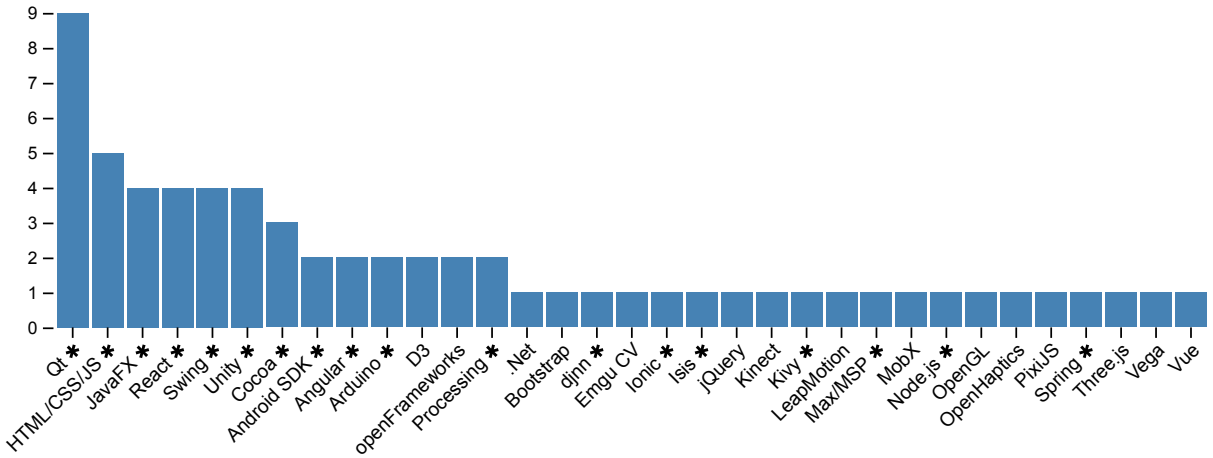


Figure 8 : Liste des bibliothèques citées dans les réponses, classées par nombre de participants. Les frameworks sont indiqués par une étoile.

Nous observons d'abord la prédominance nette du framework Qt, utilisé par plus du quart des participants. Les frameworks Swing et JavaFX (le successeur officiel de Swing) totalisent à eux deux le quart des participants, aucun n'ayant cité les deux simultanément. Ensuite, nous pouvons observer que les frameworks Web (basés sur les technologies HTML, CSS, et JavaScript) sont très largement représentés. Ils représentent 14 des bibliothèques citées, et 37,5% des participants en ont cité au moins une. Enfin, un grand nombre de bibliothèques ont été citées une seule fois. La majorité d'entre elles sont apparues il y a moins de 10 ans, en particulier parmi les outils Web. Cette observation soutient le caractère *opportuniste* de la recherche et du prototypage de techniques d'interaction, en ce qu'elle s'appuie pour une part non négligeable sur des technologies récentes et émergentes.

Nous avons indiqué par une étoile (✱) sur la figure 8 les bibliothèques que nous considérons comme des frameworks, d'après la définition donnée en section 1.1.4. Ainsi, 48 des citations cumulées des participants sont des frameworks, tandis que 17 n'en sont pas. On remarque de plus que les bibliothèques les plus utilisées (sur la gauche de la figure) sont principalement des frameworks. Enfin, parmi les 33 bibliothèques citées, 4 sont issues de la recherche académique (D3 [Bos11], djnn [Cha16], Max/MSP, et Vega [Sat14]), pour un total de 7,6% citations. Cette observation peut être modérée par le fait que D3 et Max/MSP sont très utilisés dans des communautés spécifiques (Visualisation et Musique Assistée par Ordinateur), qui n'ont pas été spécifiquement ciblées par nos listes de diffusion. De plus, djnn et Vega sont des bibliothèques relativement récentes, ce qui peut expliquer leurs faible proportions dans les citations. Néanmoins, le faible nombre de boîtes à outils issues de la recherche est préoccupant pour ce qui est de faire le lien entre les besoins des chercheurs et leurs outils, puisque ce sont principalement des développeurs hors du contexte de la recherche qui doivent répondre à ce contexte. De plus, cela signifie que les bibliothèques faites *par* des chercheurs *pour* des chercheurs ne rencontrent pas suffisamment de succès. Les critères de choix peuvent expliquer ce phénomène.

1.5.2.2 Critères de choix des bibliothèques d'interaction

Les résultats d'évaluation de l'importance des critères de choix des bibliothèques sont résumés en figure 9. Nous les avons classés en fonction du nombre de votants ayant répondu *Very important* ou *Absolutely essential*, pour distinguer les critères plus importants que la moyenne. Pour chacun, nous avons affiché un intervalle de confiance à 95% obtenu par la méthode du bootstrapping [Efr79], qui correspond à la probabilité qu'un ré-échantillonnage avec remise issu des données donne un nombre de votants compris dans l'intervalle. Lorsque la borne supérieure de cet intervalle est inférieure à la moitié des votants, nous considérons qu'il est **très probable** que plus de la moitié des chercheurs en IHM issus de la communauté représentée par notre échantillon considèrent le critère comme au moins *Very important*.

Trois critères se dégagent : *Documentation quality* (24 votants), *Quality of the API* (23 votants), et *User community* (22 votants). Ils nous suggèrent que le facteur principal influençant le choix d'une bibliothèque est sa facilité d'appentissage et d'utilisation par les programmeurs. Le critère *Compatibility with other tools/libraries* se dégage également par le nombre important de votes *Absolutely essential*, et nous suggère que les frameworks sont rarement utilisés seuls dans le contexte de la recherche en IHM.

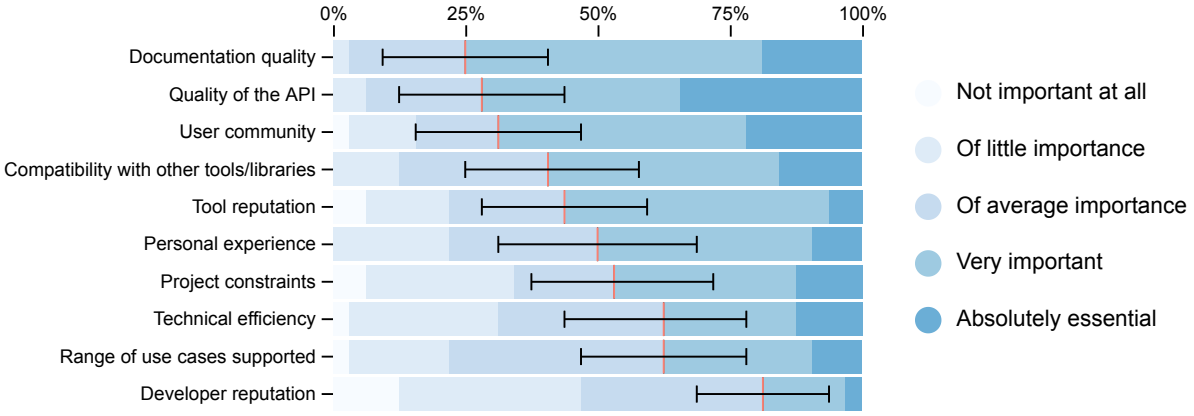


Figure 9 : Évaluation des critères de choix des bibliothèques par les participants, classés en fonction du nombre de réponses au minimum *Very important*, avec les intervalles de confiance à 95% pour ces nombres.

Enfin, pour évaluer la possibilité pour les concepteurs de frameworks d'influencer ces critères, nous les avons répartis en trois groupes :

- mesurables (*Documentation quality*, *Range of use cases*, *Quality of the API*, *Technical efficiency*)
- mixtes (*User community*, *Compatibility with other tools/libraries*)
- subjectifs (*Tool reputation*, *Developer reputation*, *Project constraints*, *Personal experience*)

Les critères mesurables sont ceux pour lesquels il existe des outils permettant d'évaluer leur qualité, et des méthodes pour l'améliorer. Pour les critères mixtes, certaines métriques et méthodes existent mais ne suffisent que partiellement. Par exemple, pour *Compatibility with other tools/libraries*, on peut mesurer le nombre de *plugins* disponibles pour un framework, mais il est difficile d'évaluer leur utilisabilité (absence de bugs, fonctionnalités implémentées, etc.). Pour les critères subjectifs, il n'existe pas ou peu de mesures objectives, et les concepteurs de frameworks n'ont pas de contrôle direct pour améliorer leur évaluation sur ces critères.

Les critères mesurables ont en moyenne 55% des votes au dessus de *Of average importance*. Les critères mixtes sont à 64%. Les critères subjectifs sont à 43%. Les participants de notre étude ont donc eu tendance à favoriser des critères objectifs, sans toutefois qu'une différence significative puisse être inférée pour l'ensemble de la population. Ces résultats nous suggèrent aussi que les deux critères mixtes *User community* et *Compatibility with other tools/libraries* gagneraient à bénéficier d'outils de mesures et de méthodes d'amélioration dans la littérature.

1.5.2.3 Problèmes liés à l'utilisation des bibliothèques d'interaction

Les résultats de criticité des différents problèmes liés à l'utilisation des bibliothèques d'interaction sont résumés en figure 10. Ils sont classés en fonction du nombre de votants ayant répondu au moins *Met, medium trouble*, seuil à partir duquel nous considérons que les problèmes ont accaparé une part suffisante de temps/énergie des participants. En dessous de chaque type de problème, des barres plus fines représentent les mêmes résultats mais en incluant uniquement les participants se disant *Advanced* ou *Expert* (20 des 32 participants).

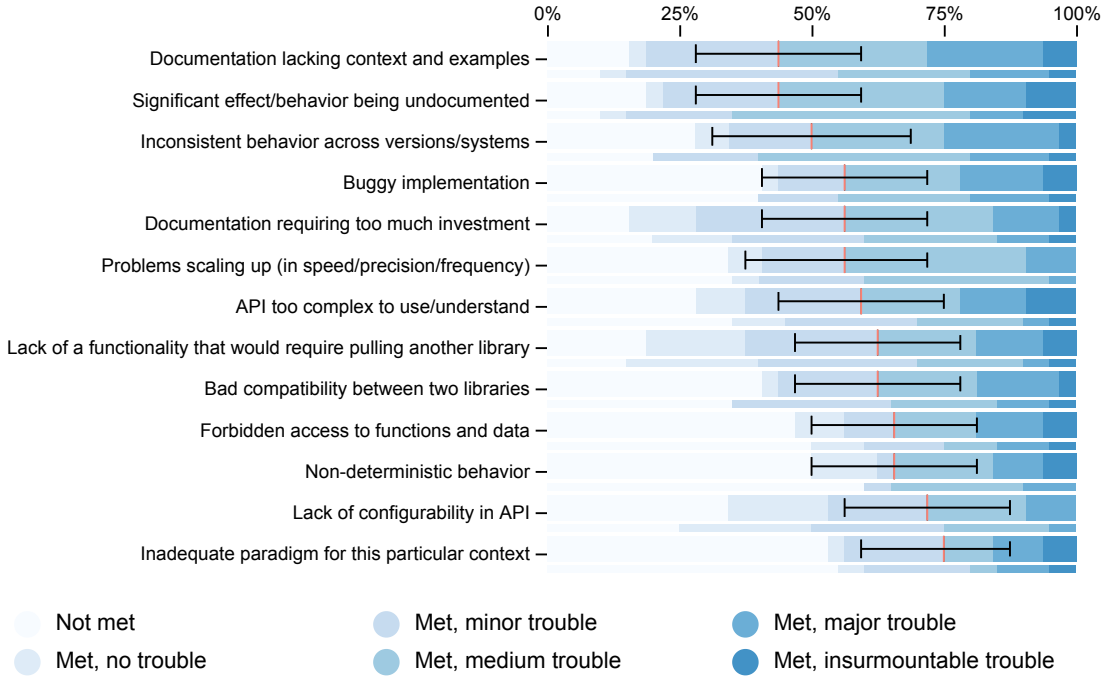


Figure 10 : Évaluation des problèmes liés à l'utilisation des bibliothèques d'interaction, classés en fonction du nombre de réponses au minimum *Met, medium trouble*, avec les intervalles de confiance à 95% pour ces nombres, et des barres secondaires représentant les mêmes résultats pour les participants d'expertise au minimum *Advanced*.

Parmi les problèmes les plus importants, ceux liés à la **documentation** occupent les 1e, 2e et 5e positions, ce qui est cohérent avec les observations sur les critères de choix des bibliothèques. Nous n'observons pas de fortes variations entre les résultats complets et ceux n'incluant que les participants avancés et experts. Ce peut être dû au faible nombre de participants, qui ne nous permet pas de diviser la population en deux et d'observer des différences significatives. Cependant, nous observons que de nombreux participants ont répondu *Not met*, y compris parmi les participants avancés.

Nous avons donc représenté en figure 11 les résultats de criticité des différents problèmes pour les participants n'ayant pas répondu *Not met*. Nous considérons que c'est une estimation plus fiable de la criticité des problèmes pour la population des chercheurs en IHM. En effet, le fait de rencontrer un type de problème particulier dépend beaucoup des projets sur lesquels les participants ont travaillé. Si un groupe de participants d'une même équipe a répondu à notre questionnaire, ils peuvent ainsi biaiser l'évaluation en figure 10, en faveur des problèmes qu'ils ont rencontrés en groupe. En représentant la criticité uniquement pour les personnes ayant rencontré les différents problèmes, nous limitons ce biais, au prix d'un plus faible nombre de réponses par problème. Nous avons donc écarté 33% des données, ce qui explique l'élargissement des intervalles de confiance à 95%, représentés sur la figure.

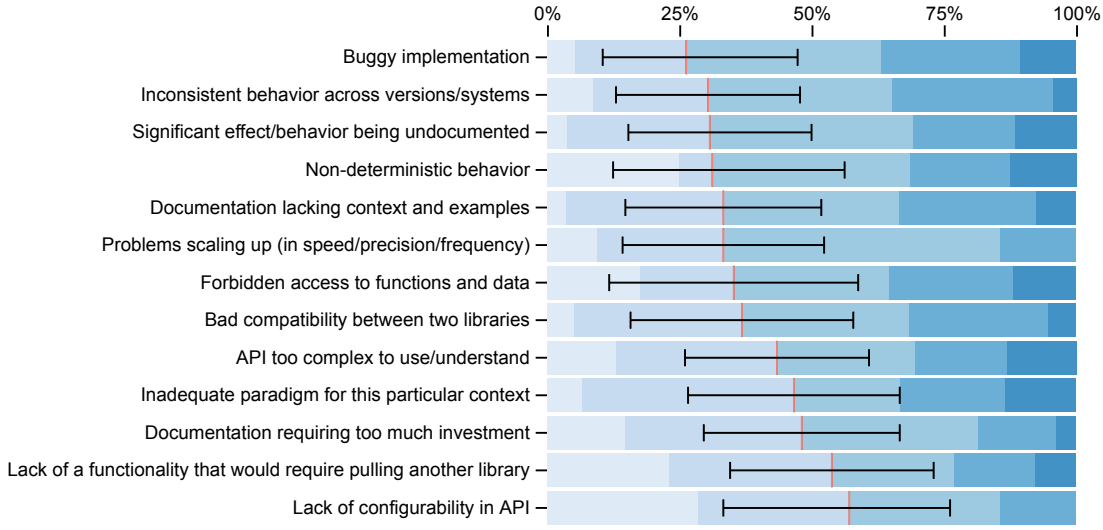


Figure 11 : Évaluation de la criticité des problèmes par les participants les ayant déjà rencontrés, avec les intervalles de confiance à 95%.

Deux problèmes se dégagent par rapport à la figure précédente, *Buggy implementation* et *Non-deterministic behavior*. Avec le problème *Significant effect/behavior being undocumented*, ils nous suggèrent que la caractéristique la plus importante d'un framework est sa **fiabilité**. C'est-à-dire qu'un framework devrait toujours *documenter ce qu'il fait et faire ce qu'il documente*. Ce point ainsi que le précédent nous ramènent à l'importance de la documentation soulevée par de nombreux travaux de l'état de l'art. Ils nous incitent à considérer les travaux de documentation comme des contributions importantes, en ce qu'ils pourraient avoir un impact majeur de réduction des problèmes rencontrés.

Un certain nombre de problèmes secondaires ont eu une criticité majeure pour plus d'un quart des participants les ayant rencontrés. Ainsi, *Inconsistent behavior across versions/systems* et *API too complex to use/understand* nous amènent à considérer la **cohérence** comme une caractéristique secondaire des frameworks. *Forbidden access to functions and data* nous font considérer la **transparence** comme une autre caractéristique importante. Enfin, *Bad compatibility between two libraries* et *Inadequate paradigm for this particular context* nous amènent à ajouter l'**adaptabilité**. Alors que nous pouvons ramener la cohérence et la transparence à des travaux de documentation, l'adaptabilité est un concept qui nous semble intéressant à approfondir. Il pourrait être associé à la flexibilité et au caractère dynamique des frameworks et langages de programmation. En pratique nous avons exploré ces points dans les prototypes présentés dans les chapitres 2 et 3.

1.5.2.4 Stratégies de prototypage de techniques d'interaction

Dans la dernière partie du questionnaire, nous demandions aux participants d'évaluer la prévalence de différentes stratégies de programmation dans leur travail. Les résultats sont présentés en figure 12. Une stratégie, *Reimplementing an existing widget/mechanism [...]*, se distingue nettement des autres. Dans un framework, cette pratique de programmation est rendue possible par les mécanismes d'**extension** et de **réutilisation**. Ensuite, pour les 2e et 3e stratégies, *Using accessible raw data to reconstruct/reinterpret a state[...]* et *Using an external mechanism to obtain and process data that is not exposed by an application*, c'est l'accès à des données cachées qui est important, donc la **transparence** des

frameworks. Enfin, parmi les stratégies suivantes, les 4e (*Aggregating multiple sources of interaction data [...]*), 5e (*Using a visual overlay [...]*) et 7e (*Introducing a different programming model [...]*) se rapportent à l'extensibilité des frameworks, et la 6e (*Reverse-engineering a closed tool or library*) appuie leur transparence.

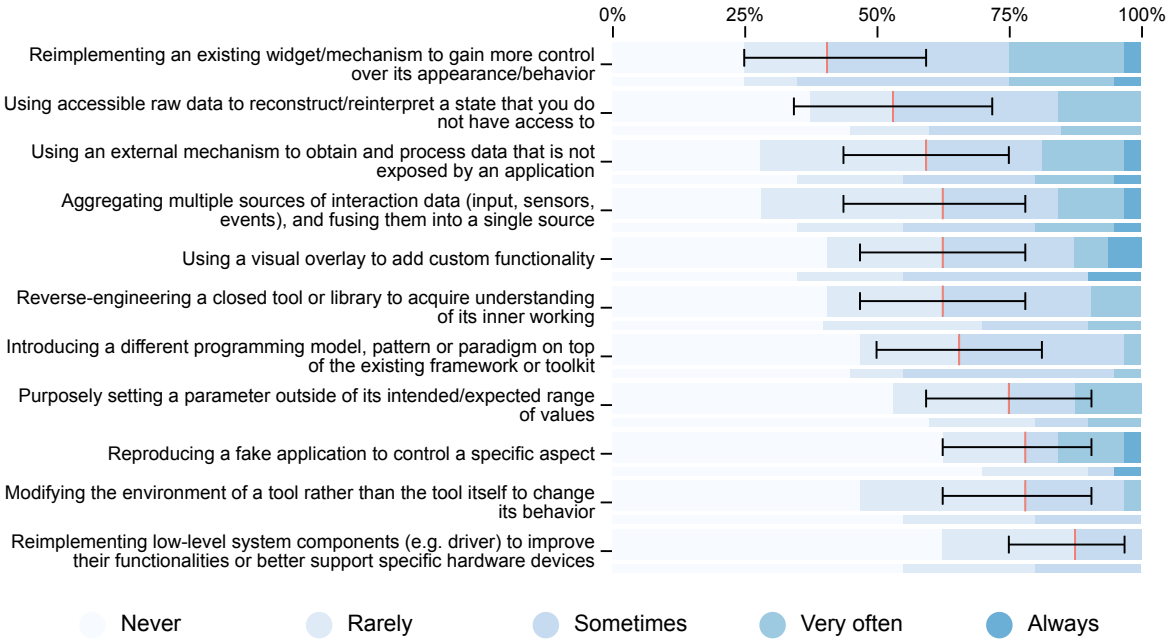


Figure 12 : Évaluation de la fréquence d'utilisation de différentes stratégies de programmation, classées en fonction du nombre de réponses au minimum *Sometimes*, avec les intervalles de confiance à 95% pour ces nombres, et des barres secondaires représentant les mêmes résultats pour les participants avancés.

Nous remarquons que beaucoup de participants ont répondu ne jamais avoir utilisé chacune des stratégies. Comme pour les problèmes évalués dans la partie précédente, il est possible qu'un biais lié à des sous-groupes de participants ait favorisé certaines stratégies au détriment des autres. Si deux participants ayant travaillé sur les mêmes projets ont répondu à notre questionnaire, leurs évaluations de la prévalence de chaque stratégie pourraient ainsi être corrélées. Cependant ici nous ne pouvons pas éliminer ce biais en ignorant les réponses *Never*, car la prévalence est une mesure plutôt objective (donc pouvant être similaire entre deux participants d'une même équipe), alors que la criticité était principalement subjective (donc propre à chaque participant).

1.6 Discussions et implications

Maintenant que nous avons analysé les résultats des interviews et des questionnaires, il convient de répondre à la question : *Que pouvons-nous faire ?* Nous discutons donc ici des solutions qui peuvent être apportées aux problèmes évoqués dans les deux études, et quels travaux futurs nous semblent les plus pertinents.

1.6.1 Adéquation des frameworks avec la recherche en IHM

Les frameworks d'interaction les plus répandus sont-ils réellement adéquats pour prototyper de nouvelles techniques d'interaction en IHM ? Cette question mérite d'être soulevée, car les interviews nous ont révélé que les frameworks d'interaction posaient de nombreux problèmes, et les questionnaires ont confirmé que ces problèmes étaient rencontrés par une part non négligeable de développeurs. Nous avons relevé trois contradictions entre la *nature* des frameworks et celle de la recherche.

Tout d'abord, le support d'usages avancés implique un plus grand risque de bugs. En effet, de tels usages impliquent d'écrire du code plus complexe, qui fasse appel à des concepts et des technologies moins maîtrisés ou peu robustes, voire liés au bas niveau (et toutes les difficultés que cela apporte). Ici, les bugs concernent le code écrit par les chercheurs, et se rapportent à des comportements non souhaitables, qui incluent ainsi les problèmes de manque de précision ou de latence, bien qu'ils ne compromettent pas forcément l'exécution du programme. Du côté des frameworks, le support de fonctionnalités avancées (comme la possibilité de créer de nouveaux types de widgets) revient souvent à écrire *plus de code*, soit parce qu'il faut proposer de nouvelles fonctions, soit pour exposer une interface à des fonctions internes. Or de nombreux auteurs évaluent le ratio de bugs par lignes de code, défendant implicitement l'hypothèse que *de plus gros projets contiennent plus de bugs* [May12, McC04]. Dès lors, nous considérons que le support de fonctionnalités avancées contribue, au moins indirectement, à un plus grand risque de bugs. Cette hypothèse met les frameworks en équilibre entre deux objectifs contradictoires : supporter de nouveaux usages liés à la recherche, et diminuer leur nombre de bugs.

Ensuite, le faible nombre d'utilisateurs ne favorise pas l'exhaustivité de la documentation. Pour le framework Qt le plus cité dans les réponses au questionnaire, le domaine de la recherche n'est aucunement mentionné sur la page principale de Qt [Qt19], ni les fonctionnalités d'extension, de réutilisation, et de transparence relevées dans les questionnaires. Ces absences sont un symptôme de la faible part d'utilisations de Qt à des fins de développement de projets de recherche, et de l'intérêt relativement moindre que porte le développeur à ce contexte. Ainsi il est probable que les développeurs consacrent moins d'énergie à fournir une documentation adaptée aux besoins des développeurs menant des projets de recherche. Nous en observons principalement les conséquences, à savoir une documentation insuffisante vis-à-vis des capacités d'extension, de réutilisation, d'adaptabilité et de transparence du framework.

Enfin, la robustesse d'un framework tend à limiter sa flexibilité aux extensions. À mesure que de nouveaux usages se développent, les besoins en données évoluent. C'est ce que nous observons avec la *transparence* des données soulevée par les stratégies du questionnaire. Or la mise à disposition de données autrefois cachées présente toujours un risque de sécurité. Par exemple, autoriser l'interception d'évènements du clavier au niveau du système permet également d'intercepter les mots de passe. C'est un jeu du chat et de la souris, dans lequel toute ouverture de données est souvent contrebalancée par une mesure restrictive.

Dans ces conditions, nous arguons que l'utilisation d'un framework sera *par nature* entravée dans un contexte de recherche. Sans être rédhibitoires, les contraintes évoquées n'ont pas de raison d'évoluer favorablement à l'avenir. Les frameworks ne sont donc pas inadéquats, mais ne nous

semblent pas les solutions les plus prometteuses. De meilleures solutions seraient plutôt :

- des projets moins complexes, réduisant le nombre de bugs
- dédiées à la recherche, et documentées pour
- dotées d'une grande flexibilité, plutôt que la robustesse

1.6.2 Faciliter le développement d'applications *ad hoc*

Face aux difficultés rencontrées avec les frameworks d'interaction, une solution serait de faciliter le développement d'applications *ad hoc* (ou *from scratch*), c'est-à-dire sans l'utilisation de patrons de conception réutilisables. Il s'agirait alors d'utiliser directement des bibliothèques de bas-niveau, pour les entrées (SDL, TUIO [Kal05, libpointing [Cas11], etc.) et les sorties (OpenGL, Cairo, etc.).

Une telle approche aurait l'avantage d'offrir une plus grande liberté d'adhérer aux usages contemporains, en termes d'apparence des éléments interactifs et de leur interactivité. Elle favorise l'évolution des architectures d'interaction et la remise en question des fonctionnalités à inclure, n'étant pas sujette à des contraintes historiques (manque de mémoire, vitesse du rendu graphique sur CPU). De plus, la quantité de travail à produire est très inférieure à celle d'un framework, car il n'est pas nécessaire d'inclure des fonctionnalités en dehors de celles dont on a besoin, ni de mécanismes d'extension, ou encore de documenter pour d'autres développeurs. Enfin, du fait de l'ampleur réduite de tels projets, les outils de gestion de projets (CMake, Gradle, Maven) peuvent être épargnés, contribuant à réduire les efforts d'apprentissage et de programmation.

Cependant, la conception d'une application interactive *ad hoc*, même partielle, est un projet complexe, dont les efforts augmentent avec la taille du projet. L'utilisation de méthodes de conception opportunistes adaptées aux petits projets peut se heurter à un mur de complexité, si son ambition n'a pas été correctement planifiée en amont. De plus, le code produit ne sera probablement pas réutilisable pour de futurs projets, ni portable vers d'autres systèmes. Enfin, une telle démarche requiert beaucoup de connaissances ou d'expérience pour être avantageuse en temps par rapport à l'utilisation d'un framework éprouvé.

En pratique lors des interviews, trois participants nous ont explicitement indiqué s'être affranchis des éléments réutilisables d'un framework : P2:L114 « *C'est rapide, c'est super rapide. Ça dépend comment tu le fais, mais ce que j'ai fait c'est que j'ai fait une application dans laquelle je dessine des rectangles, il charge une image, dessine des rectangles, [...]* », P5:L88 « *Beaucoup de fois oui, je redessine mes propres widgets, parce que pareil il me faut des widgets avec des formes assez... bizarres, ou... particulières* », P7:L149 « *Mais moi j'aurais fait tout connement des boutons. Je n'aurais pas cherché un code, je sais comment faire ça, à la main, from scratch* ». En outre, P4 nous a décrit ses difficultés lorsque son projet *ad hoc* a commencé à prendre des proportions excessives : P4:L46 « *Là ça c'est comme ça, du coup mon application a commencé à grossir, grossir, grossir. Et après tu te rends compte qu'il te manque des trucs, donc je modifiais, et ça passait par les mêmes tuyaux, mais des fois il y avait des hacks parce que j'avais mis en place un truc qu'il prévoyait pas* ».

Que pourrions-nous faire ? Selon notre interprétation des résultats du questionnaire, deux problèmes limitent principalement le développement d'applications ad hoc. Tout d'abord, il y a un **manque de documentation sur la programmation d'interfaces ad hoc**. En effet, les références principales sur le sujet sont des architectures d'interaction (comme MVC, MVP, et PAC), qui sont conçues pour être robustes, c'est-à-dire qui fonctionnent de façon prévue quels que soient les évènements d'entrée. Il

existe peu d'architectures ou de documentations dédiées à mettre *rapidement* en œuvre une application interactive. À notre connaissance, seule ImGui [Mur05] répond à ce besoin en s'affranchissant de la construction d'un arbre de scène, mais elle est populaire uniquement dans le domaine du Jeu Vidéo. Le second problème est un **manque de documentation et d'exemples sur l'utilisation des périphériques à bas niveau**. Aujourd'hui, il est difficile d'utiliser des APIs de bas niveau directement, sans l'utilisation d'un framework. Ceux-ci fournissent souvent leurs propres APIs internes — permettant par exemple d'initialiser une fenêtre d'affichage pour OpenGL. Autrement, il faut se confronter à la complexité des fonctions du système, que nous avons détaillée dans la problématique de ce chapitre. D'ailleurs, les trois participants cités ayant programmé des applications ad hoc l'ont tous fait *dans le cadre* d'un framework. Des bibliothèques existent pour fournir un accès direct et simple aux périphériques d'entrée/sortie, cependant ils suffisent rarement seuls et ne cohabitent pas toujours très bien à plusieurs. Par exemple, lors de l'utilisation de libpointing pour la souris [Cas11], nous n'avons pas été en mesure de capturer les entrées du clavier avec SDL. La multiplication de bibliothèques d'accès au bas niveau ne nous semble donc pas la solution la plus pérenne, car elles contribuent à la structure des APIs en couches, que nous cherchons par ailleurs à éviter. Il nous semblerait préférable de *documenter* l'utilisation des APIs de bas niveau des différents systèmes d'exploitation, et de fournir des exemples de code rapides à mettre en œuvre.

1.6.3 Rapprocher le Web des applications de bureau

Au cours de nos recherches, nous avons observé que les technologies Web sont de plus en plus utilisées dans le domaine des applications de bureau. Il peut s'agir par exemple de réaliser un prototype d'application de bureau en Web, et de l'utiliser pour effectuer une série d'expériences contrôlées. Ce glissement des usages mérite d'être souligné. Lorsqu'Internet s'est diffusé initialement auprès des particuliers, les technologies HTML puis CSS servaient à concevoir des *sites Web*. Un site Web est une page à défilement vertical, de largeur fixe et de hauteur variable, dont le contenu (principalement des blocs de texte et des images) s'arrange automatiquement de haut en bas et de gauche à droite. Ce fonctionnement est inspiré des logiciels de traitement de texte, mais sans la séparation en pages. Les sites Web sont accessibles par l'intermédiaire d'un navigateur, qui convertit les pages HTML/CSS en des interfaces visibles et interactives. D'un autre côté, les *applications de bureau* sont des fenêtres de largeur et hauteur variables, dont le contenu est librement déterminé par chaque application. Ce sont des programmes exécutables qui ne dépendent pas d'un navigateur, mais s'intègrent directement dans le système de fenêtrage du système d'exploitation. Chaque application gère donc une ou plusieurs fenêtres, dans lesquelles elle peut dessiner et capturer les évènements d'entrée.

Historiquement, les sites Web et applications de bureau étaient considérés comme des domaines distincts, avec des frameworks distincts. Cependant, l'émergence des *applications Web* (webmails, systèmes de gestion de contenu, wikis, jeux en ligne, etc.) a rapproché les deux mondes. Du côté Web, des widgets sont apparus pour reproduire une partie de l'expérience utilisateur de bureau — comme les boutons, onglets, ou menus. Du côté bureau, les frameworks ont réutilisé des technologies à la base du Web pour bénéficier de leur robustesse héritée de nombreuses évolutions — comme JavaScript dans QML, ou CSS dans JavaFX. Aujourd'hui, alors que les technologies du Web permettent de plus en plus de s'aventurer sur le terrain de la bureautique, nous arguons que des travaux futurs devraient appuyer cette tendance.

En effet, le mouvement initié par le *World Wide Web Consortium* (W3C) est une tentative de standardisation de la structure interne d'un framework. Au delà de l'utilisation d'une architecture comme MVC, il s'agit de définir les classes disponibles (*Document Object Model* — DOM), les attributs de style (CSS), les instructions de dessin (`canvas` et SVG), et beaucoup d'autres. Ces normes sont ainsi implémentées par plusieurs navigateurs concurrents, qui aident à détecter les ambiguïtés des spécifications à cause des détails de comportements différents. Ce fut le cas aussi grâce aux tests Acid1, Acid2 et Acid3 [Hic99], qui ont contribué à uniformiser les comportements observables des navigateurs. En interne, la spécification DOM permet d'accéder à une partie des sructures privées des navigateurs — certaines n'étant pas standardisées et spécifiques à chaque navigateur. On dispose ainsi d'une certaine *transparence*, critère essentiel aux stratégies de programmation classées dans les résultats du questionnaire. En outre, les technologies du Web sont conçues pour être portables entre les systèmes d'exploitations, et c'est le rôle des navigateurs de gérer les éventuelles différences. Enfin, les standards du W3C sont en constante évolution, poussés par les concepteurs de navigateurs et toutes les personnes volontaires. Ils peuvent ainsi corriger les éventuelles “erreurs de jeunesse”, et s'adapter aux usages contemporains. C'est le cas par exemple avec les balises HTML dépréciées à chaque nouvelle version.

Les technologies Web souffrent cependant de quelques inconvénients majeurs. Tout d'abord, il est difficile pour les navigateurs d'atteindre une vitesse d'exécution et empreinte mémoire comparables à celles de frameworks de bureau. Ceci est dû en partie à l'utilisation du langage dynamique JavaScript, et aussi à l'*immense* complexité des spécifications nécessaires pour implémenter un navigateur Web moderne, qui compliquent les efforts d'optimisation. Le nombre et la complexité des standards est lié à la pression importante de nombreux acteurs pour ajouter diverses fonctionnalités [W3C19]. Cette complexité est encore accentuée par la permissivité des navigateurs, qui devaient historiquement interpréter le code du mieux qu'ils le pouvaient, malgré d'éventuelles erreurs, afin de faciliter la création de sites Internet par les novices. La complexité des technologies Web tend néanmoins à s'auto-réguler grâce aux mécanismes de dépréciation, et les standards n'obtenant pas un succès d'usage sont à terme abandonnés ou remplacés. Enfin, tout comme les frameworks de bureau, la durée de vie d'une extension accédant à des APIs internes est limitée dans le temps, et ceci même lorsque les APIs sont standardisées. En effet, avec l'apparition de mécanismes d'extension trop souples, se développent de nouveaux *malwares* les utilisant pour porter atteinte aux utilisateurs. Des sécurités supplémentaires apparaissent alors, qui invalident à terme les extensions précédemment développées. C'est le cas par exemple avec la technologie Ajax, qui permettait d'émettre des requêtes HTTP depuis le code JavaScript, mais qui s'est vue contrainte ensuite par la technologie *Cross-Origin Resource Sharing*.

Que pourrions-nous faire ? L'utilisation des technologies du Web pour construire des applications de bureau, et prototyper des techniques d'interaction liées à ce type d'applications, nécessite des *ponts* entre les deux domaines. Le framework Electron, par exemple, permet de distribuer une application Web comme application de bureau, sans l'interface d'un navigateur [Git13]. Cependant, l'API pour accéder aux services du système d'exploitation est en partie spécifique à Electron, et n'est pas standardisée en tant que technologie du W3C. Des standardisations des widgets en balises HTML ont été tentées par le passé, certaines ayant été adoptées (`button`, `progress`, ou `textarea`), d'autres ayant été abandonnées (`menu`, `command`). Avec le développement récent des Web Components, il est possible de créer de nouvelles balises HTML pour prototyper des extensions à HTML. Il nous semble

donc opportun et crucial de **participer aux efforts de standardisation en cours du W3C**, pour faciliter la création d'applications de bureau avec les technologies Web. Pour informer et orienter ces évolutions, il convient aussi de **démontrer l'implémentation de techniques d'interaction en Web** (voir par exemple [Roy19]).

Dans ce travail de thèse, comme nous le verrons plus tard, nous avons utilisé les technologies du Web pour leur flexibilité, ce qui nous a permis d'explorer des syntaxes de code innovantes. La contribution aux groupes de travail du W3C relève des travaux futurs.

1.6.4 Liens avec ce travail de thèse

Les deux études présentées dans ce chapitre nous ont permis d'observer la faible utilisation des boîtes à outils d'IHM en pratique, donc l'intérêt limité d'une telle contribution dans ce travail de thèse. L'important est la *connaissance* qu'on en retire. Pour que nos travaux contribuent aux connaissances sur la programmation d'interactions, nous les avons toujours attachés à des projets ou communautés existants, afin de contribuer à leur développement. Nous nous sommes orientés en particulier vers le développement d'interfaces ad hoc, et l'utilisation des technologies Web en lien direct avec le bas niveau (sans navigateur).

Chapitre 2. Les Essentiels d'Interaction

Comme nous l'avons observé, les chercheurs utilisent principalement des frameworks d'interaction éprouvés pour développer de nouvelles techniques d'interaction. Cependant ces frameworks sont excessivement complexes, et restreignent la liberté des chercheurs à explorer de nouvelles idées et dévier des standards établis. Le problème principal est que leurs développeurs ne prennent pas suffisamment en compte les besoins des chercheurs, car ce ne sont pas leurs priorités. Ces besoins étant exprimés par une communauté relativement petite d'utilisateurs, les développeurs de frameworks seraient peu enclins à des changements d'architecture majeurs (tels que suggérés en section 1.5.2). Nous proposons donc de contribuer par des travaux qui reconnaissent la prépondérance des frameworks, et s'intègrent de façon complémentaire dans cet écosystème. En lien avec le manque de support de l'interaction à bas niveau (évoqué en section 1.6.2), et inspirés par les travaux sur Amulet [Mye97], Smalltalk [Kra88], et Smala [Mag18], nous avons considéré la possibilité d'étendre les langages de programmation, autant que les frameworks. Par nos travaux, nous avons aussi cherché à étudier le manque de diffusion des outils issus de la recherche, et y proposer des réponses. Enfin, nous avons choisi de nous orienter vers des contributions logicielles (plutôt que des documents comme discuté en section 1.6). En effet il nous a semblé essentiel, dans le cadre d'une thèse de doctorat, de bénéficier d'une expérience pratique dans les outils de programmation d'interactions, avant de contribuer ultérieurement à des efforts de standardisation de ces outils.

Ainsi, partant du problème que l'expression et l'implémentation de l'interaction est souvent rendue complexe avec les outils actuels, notamment à cause d'une multitude de niveaux d'abstraction, nous proposons et défendons la thèse selon laquelle **il faut rendre plus accessible la programmation de l'interaction à bas niveau, à l'aide de concepts applicables aux langages de programmation et aux architectures de frameworks**. Nous illustrons cette thèse avec une extension au langage Smalltalk (en section 2.4), ainsi qu'un nouveau framework dédié au développement d'interfaces ad hoc (au chapitre 3).

Nous commençons par formuler des *Essentiels d'Interaction*, recommandations pratiques visant à orienter notre démarche d'accessibilité. Ces recommandations ont été définies à partir de mentions régulières dans l'état de l'art, d'observations récurrentes dans les frameworks d'interaction majeurs, ainsi que nos études préliminaires Les Essentiels d'Interaction sont :

- *Une orchestration explicite et flexible des comportements interactifs*, où les règles de déclenchement des blocs de code sont clairement mises en valeur, et permettent de donner l'initiative de leur exécution directement à l'environnement
- *Un environnement d'interaction minimal et initialisé au démarrage de toute application*, pour réduire les efforts initiaux des développeurs lors de l'accès aux entrées et sorties utilisateurs.
- *Des mécanismes et conventions standardisés, ainsi qu'un langage à la syntaxe flexible*, pour assurer la compatibilité et l'extensibilité des bibliothèques logicielles pour y exprimer de l'interaction

La première section de ce chapitre est consacrée à un état de l'art des recommandations d'améliorations des frameworks et langages de programmation. Les deux sections suivantes sont dédiées aux deux premiers Essentiels d'Interaction. Dans la quatrième section, nous illustrons notre démarche de simplification avec l'intégration des animations dans un langage de programmation, et en déduisons un troisième Essentiel d'Interaction.

2.1 État de l'art des recommandations pour langages et frameworks d'interaction

La syntaxe et la sémantique de la programmation d'interactions ont été relativement peu étudiées dans les domaines des Interactions Homme-Machine et du Génie Logiciel. Elles sont dépendantes de l'expressivité donnée par les langages de programmation, or ceux-ci sont souvent considérés comme immuables, et l'interaction y est exprimée par extension. Par exemple, pour connecter un *signal* à un *slot* dans Qt, on écrirait `QObject::connect(&objA, &ClassA::fctA, &objB, &ClassB::fctB)`. Cette syntaxe est largement basée sur celle du langage C++ : la création d'une connection se fait par un appel de fonction `QObject::connect`, on fait référence aux objets par leurs pointeurs `&objA` et `&objB`, et on fait référence aux signaux et slots sur ces objets par des pointeurs de fonctions `&ClassA::fctA` et `&ClassB::fctB`. À l'opposé, pour exprimer un binding (concept relativement similaire) dans Smala [Mag18], on écrirait `objA -> objB`. Ici, le binding est *intégré* à la syntaxe du langage, ce qui offre aux utilisateurs de Smala plus de facilité pour exprimer des applications interactives, et résulte en du code plus court.

Par définition, la *syntaxe* est la manière dont des mots s'assemblent pour former des phrases. Dans un langage de programmation, ces mots sont des lexèmes (ou *tokens*), qui sont généralement de cinq types : identifiants (noms de variables et de fonctions), mots-clés du langage, ponctuation (opérateurs et parenthèses), litéraux (constantes numériques et chaînes de caractères), et commentaires. La syntaxe d'un langage s'accompagne de règles de *grammaire*, qui permettent de vérifier qu'une séquence de lexèmes est un programme valide ou non (sans définir ce qu'il produira).

La *sémantique* d'un langage est le sens donné à un programme (séquences de lexèmes), c'est-à-dire l'effet qu'il produira à l'exécution. Par exemple, un identifiant suivi d'une parenthèse ouvrante et une parenthèse fermante (`f()`) forme un appel de fonction, qui correspondra à l'exécution en une instruction de branchement du flux d'exécution, vers le code de la fonction.

Dans le cas d'un framework de programmation, la syntaxe et la sémantique sont toujours définies dans les limites permises par le langage de base. Il sera donc impossible d'ajouter des règles de grammaire qui violent les règles du langage (par exemple autoriser une parenthèse ouvrante sans parenthèse fermante). De plus, la sémantique d'un framework sera généralement définie par *extension* de celle du langage, car il est souvent impossible d'inhiber un comportement de base du langage (comme par exemple intercepter tous les appels de fonctions du programme pour en changer les paramètres).

L'étude et le prototypage de nouvelles syntaxes et sémantiques pour la programmation d'interactions est une activité difficile, qui revient soit à modifier un langage existant (et accepter ses contraintes), soit à créer un nouveau langage. Ceci explique le faible nombre de travaux ayant opéré à

l'interface entre frameworks et langages. Nous présentons ici des travaux significatifs ayant proposé ou suggéré de nouvelles formes de programmation de l'interaction. Ils se distinguent des boîtes à outils proposées en IHM par la remise en question de concepts à la base des langages de programmation, qui nécessite d'opérer aux deux niveaux (langage et framework).

2.1.1 Les trois services du noyau sémantique indispensables à l'IHM

Fekete a proposé trois services du noyau sémantique, qu'il qualifie d'indispensables à l'IHM, pour la conférence IHM [Fek96], puis lors des journées du Groupe de Recherche Programmation du CNRS [Fek96]. Ces points s'adressent au “noyau sémantique” des systèmes interactifs, qu'on peut assimiler aux langages de programmation. Ils désignent des fonctionnalités difficiles à implémenter sans un support explicite du noyau sémantique, et détaillent chacun des mécanismes communs pour les implémenter. Les services présentés par Fekete sont :

la notification
: Ce service se rapporte au besoin de pouvoir observer tous types d'évènements dans le système (écriture de variable, modification à un fichier, etc.). Le problème est que certains types d'évènements ne sont pas observables, et que l'enregistrement d'observateurs (*callbacks*) n'est pas un mécanisme cohérent dans tout le système, ce qui invite à un support unifié au niveau du langage de programmation.

la prévention des erreurs
: Ce service revient à savoir si une fonction va déclencher une erreur avant même de l'exécuter (ex. Est-ce je peux lire le contenu d'un fichier comme une image ?), voire d'énumérer les fonctions valides dans un contexte particulier. En l'absence de ce type de fonctionnalité, les systèmes interactifs doivent stocker l'information de validité en amont (ex. l'extension d'un fichier avant sa lecture par une application), ce qui représente une redondance et un risque de correspondances erronnées.

l'annulation
: Ce service revient à sauvegarder l'état de l'application à des moments donnés, et à pouvoir y revenir à tout moment, afin d'implémenter les commandes *undo* et *redo*. Il est présent dans toutes les applications aujourd'hui, et c'est parmi les trois services de Fekete celui qui a été le plus étudié dans le domaine IHM [Nan14, Hee08]. Cependant il reste aussi très difficile à implémenter, à cause de la nature irrémédiable de nombreuses commandes dans les systèmes informatiques (assignation de variable, suppression de fichier, etc.). L'implémentation de l'annulation nécessite de matérialiser et manipuler l'état courant du système, ce qui pourrait être facilité par le langage.

Ce travail est notable pour avoir été présenté aux deux communautés de l'Interaction Homme-Machine et du Génie Logiciel. Le besoin émane donc de l'expérience d'un praticien ayant implémenté des techniques d'interaction et un outil de programmation d'applications interactives, et est adressé à une communauté qui peut y apporter des solutions. De plus, les services illustrent des besoins précis, et sont décrits techniquement aussi bien que théoriquement. Cependant, le travail n'a pas été suivi de recherches pour résoudre les problèmes évoqués — qui restent toujours d'actualité aujourd'hui. L'article ayant principalement adopté un point de vue d'IHM, il reste peu détaillé sur les pistes d'implémentation du côté des langages de programmation, ce qui a probablement été insuffisant pour

convaincre les chercheurs de la communauté GL. Néanmoins, le point correspondant à la notification globale a été sujet à des recherches actives en IHM peu après l'article de Fekete, avec en particulier la création du bus logiciel Ivy [Bui02], le protocole d'entrées *multi-touch* TUIO [Kal05], et le concept de composants activables publiquement de djnn [Cha15].

2.1.2 *Usability requirements for interaction-oriented development tools*

Les "Exigences d'usabilité pour les outils de développement orientés-interaction" [Let10] ont été étudiées par Letondal et al. à l'ENAC, dans le cadre du développement de systèmes critiques pour l'aviation civile. Dans ce contexte, les auteurs observent un fossé entre ce qu'offrent les langages de programmation communs (conçus pour la plupart pour le calcul), et les pratiques et besoins identifiés en IHM. Ils énumèrent donc les différences entre les programmes de calcul et les programmes interactifs, afin d'en dégager des recommandations pour orienter la conception d'outils de programmation. Ainsi, leur travail s'apparente à des dimensions de comparaison des outils de programmation vis-à-vis du support de l'interaction, à la manière des *Cognitive Dimensions of Notations* de Green pour les langages [Gre96]. Ils tirent de chaque dimension des recommandations de haut niveau, que nous synthétisons ici :

- minimiser la complexité des informations, la complexité des accès (qui s'apparente à réduire la fragmentation du code et des structures de données), et l'imprévisibilité
- fournir des concepts de graphisme, d'adaptation à l'environnement, de différents modèles d'interaction (machines à états, flux de données, parallélisme, réactivité), et d'applications distribuées
- supporter la production de code, et la visualisation ainsi que le débogage à l'exécution
- gérer la vie du projet (en ses différentes étapes), la réutilisation de code et de connaissances, et le développement à plusieurs

Pour justifier le faible impact de telles recommandations en dehors du domaine IHM, les auteurs reconnaissent que les interfaces sont une cible mouvante, et que les solutions proposées dans la recherche sont probablement trop éloignées des pratiques industrielles. Leur étude est remarquable pour avoir été suivie par le développement d'un modèle d'architecture par "composants interactifs" [Cha12], ainsi que la formalisation de relations de causalité avec les *bindings* de djnn/Smala [Mag18]. Ces travaux à la fois théoriques et pratiques ont abouti à des réalisations techniques concrètes, qui valident écologiquement la faisabilité des concepts produits. Cependant, le public visé par les travaux théoriques est resté lié à l'IHM plutôt qu'au Génie Logiciel, et de haut niveau plutôt que basé sur des descriptions techniques. Ce manque de diffusion auprès des communautés concernées par le type d'innovations requises, a probablement, et pour le moment du moins, limité l'adoption de leurs exigences d'utilisabilité pour adapter les outils de programmation à l'interaction.

2.1.3 Factorisons la gestion des évènements des applications interactives !

Dans leur article présenté lors de la conférence IHM'98 [Con98], Conversy et al. présentent une prise de position à partir d'une recommandation unique, très explicite et présentée en détails. Dans le cadre de leur travail, les auteurs ont développé de nombreuses applications interactives et outils de programmation d'interactions. Ils relatent l'utilisation de plusieurs types de bibliothèques logicielles,

qu'ils qualifient de *composants*, et qu'ils doivent faire cohabiter dans les applications interactives. Le problème qu'ils soulèvent dans ce contexte est que chaque composant définit des sources d'évènements, et propose son propre mécanisme de gestion des évènements, souvent partiellement incompatible avec les autres composants. Les auteurs suggèrent donc d'en factoriser la gestion, et présentent trois approches communes de propagation des évènements dans les systèmes interactifs :

- l'enregistrement d'un callback au niveau de chaque source d'évènement, à la manière du patron de conception *listener*
- la notification de chaque évènement à un intermédiaire unique (“aiguilleur”), auquel les drains d'évènements s'abonnent pour écouter (à la manière du bus de communication Ivy [Bui02])
- l'utilisation de plusieurs aiguilleurs intermédiaires, pour permettre éventuellement l'existence de plusieurs contextes d'interaction

Leur travail est notable pour avoir mis en lumière un problème éprouvé, bien identifié, et d'avoir détaillé les sources de ce problème. L'article nous donne une bonne vision du contexte de la gestion des évènements, et nous laisse imaginer des solutions potentielles à sa factorisation. Cependant, les auteurs s'adressent principalement à un public de chercheurs en IHM, qui sont susceptibles de contribuer à la création de boîtes à outils plutôt que de travaux sur les langages de programmation. De plus, faute d'un positionnement sur le type de solution à apporter, l'article ne peut pas nous montrer comment une telle solution résoudrait le problème posé. En particulier, une boîte à outils factorisant la gestion des évènements pourrait être considérée comme un nouveau “composant”, et accentuer les problèmes de cohabitation plutôt que les résoudre.

2.1.4 *The Natural Programming Project*

Le *Natural Programming Project* est en cours depuis plus de deux décennies au HCII de l'université Carnegie Mellon [Mye08]. Il vise à produire des outils de programmation “naturels”, c'est-à-dire plus proches de la manière dont les personnes pensent les algorithmes et la résolution de tâches. Le projet s'adresse aux très nombreuses personnes qui interagissent professionnellement avec des ordinateurs, voire qui se disent programmeurs, sans pour autant être des “professionnels”. L'équipe travaillant sur ce thème de recherche a produit un très grand nombre d'études et d'outils, synthétisés dans [Car19]. L'un de ces projets a été la création d'un langage de programmation et son environnement, HANDS, à partir d'études sur les raisonnements naturels liés à la résolution de problèmes de programmation [Mye04]. Le projet se concentre sur la réalisation de l'outil plutôt que pour dégager des recommandations explicites. Cependant, quatre points énumérés à l'issue des études préliminaires nous semblent être des recommandations pertinentes. Ces études visaient à observer des personnes résoudre des problèmes de programmation, sans informatique, et sans influencer le type de raisonnement à adopter. Nous traduisons les points et reproduisons leurs exemples :

- Une structure basée sur des évènements ou règles est souvent utilisée (« *When PacMan loses all his lives, it's game over.* »).
- Des opérations sont spécifiées sur des groupes d'éléments plutôt qu'en itérant sur chacun (« *Move everyone below the 5th place down by one.* »).
- Les opérations logiques booléennes sont rarement utilisées, et ne correspondent pas à la logique mathématique usuelle.
- Les comportements et actions sont spécifiées avec du texte plutôt que des dessins.

Ces points sont intéressants car ils apportent des réponses claires et orientées à des choix de conception débattus (ex. le choix d'un langage textuel ou visuel). Ils sont donc utilisables directement et clairement pour orienter le design d'outils de programmation. De plus, ils ont été appliqués en pratique pour la conception de l'environnement de programmation HANDS, dont l'utilisation a été testée expérimentalement ensuite. Cependant on peut remarquer que ce travail a dérivé vers une cible d'utilisateurs jeunes et novices (figure 13), alors que le projet ne ciblait pas uniquement les jeunes initialement. Leur travail pourrait donc être appliqué à des langages intermédiaires à avancés, qui correspondent mieux au domaine d'étude de la présente thèse.

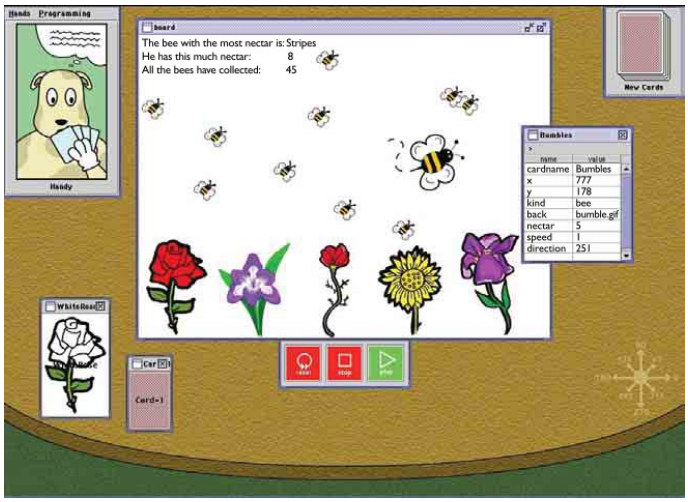


Figure 13 : Interface de l'environnement HANDS (figure extraite de [Mye04]).

2.1.5 Limites de l'état de l'art et leçons à tirer

Malgré les nombreux travaux signalant le manque d'adéquation des langages de programmation à l'expression d'interactions, et les travaux présentés offrant des pistes d'améliorations, l'état des outils de programmation a peu évolué dans le sens voulu. Nous imputons deux raisons principales à ce constat : la trop grande distance entre les besoins relevés et les améliorations techniques requises, et le manque de portée écologique des recommandations.

2.1.5.1 Des recommandations aux réalisations

Les travaux que nous avons étudiés se concentrent sur la formulation de recommandations pour l'implémentation de langages ou de frameworks dédiés à l'interaction. Malheureusement, la plupart de ces recommandations n'ont pas abouti à des réalisations tangibles. Plusieurs raisons peuvent expliquer cette observation. D'abord, l'implémentation de ces recommandations peut s'avérer beaucoup plus difficile qu'envisagé. Lorsqu'on commence par modifier un langage ou framework existant, celui-ci a souvent une extensibilité limitée, qui peut imposer des contraintes inattendues aux solutions. Si on souhaite développer un langage ou framework à partir de rien, il faudra développer de très nombreuses fonctionnalités pour obtenir un système testable en conditions réelles (ou même contrôlées). Ensuite, l'ingénierie de langages et frameworks demande des compétences spécifiques, distinctes de l'ingénierie d'applications interactives. Elle nécessitera de faire appel à des personnes

n'ayant pas nécessairement participé à la définition des recommandations. Si ces personnes n'ont pas été impliquées au début du projet, elles peuvent avoir des avis différents, qui risquent d'entraver et ralentir la poursuite du projet. Enfin, nous observons qu'une fois présentées, les recommandations liées à l'évolution des systèmes interactifs sont souvent insuffisamment valorisées par leurs auteurs. Il est probable qu'un "mur" effectif entre les idées et leurs réalisations décourage les auteurs, qui ont déjà investi beaucoup d'efforts dans l'identification de problèmes pertinents, et renoncent à investir plus d'efforts faute de perspectives claires de résultats.

À quoi servent donc de telles recommandations ? Puisque peu de langages ont répondu à des recommandations énoncées en IHM, nous devons justifier leur intérêt, en particulier pour les Essentiels d'Interaction formulés dans ce manuscrit. La fonction principale de ce type de recommandations n'est pas leur utilisation par d'autres chercheurs pour guider la conception de systèmes interactifs. En effet, ces points sont toujours discutables, particulièrement lorsqu'ils ne se ramènent pas à des hypothèses vérifiables expérimentalement. Ils peuvent éventuellement éclairer les décisions d'autres chercheurs, voire les conforter dans leurs décisions. Néanmoins nous considérons que ce sont principalement des *déclarations d'intentions*. On les formule afin de susciter des discussions, et de recueillir les avis d'autres chercheurs. En ce sens, les Essentiels d'Interaction sont une étape de notre travail et non sa finalité. Elles sont conçues pour orienter les réalisations présentées dans ce manuscrit, et n'ont pas de portée en outre. Ainsi ce sont les réalisations techniques qui véhiculeront nos véritables recommandations sur l'évolution des outils de programmation des systèmes interactifs, et ce sont elles que nous nous efforcerons de valoriser à l'avenir.

2.1.5.2 Des réalisations aux utilisateurs

Parmi les travaux que nous avons étudiés, nous avons remarqué qu'ils répondent peu à la question *Pourquoi devrait-on améliorer les outils de programmation d'interactions ?* Cette question doit justifier l'énumération de recommandations, et ensuite les travaux de modification des langages et frameworks de programmation. C'est en y répondant qu'on peut convaincre les développeurs d'outils de programmation d'adopter nos recommandations. Sans cela, il sera difficile pour eux d'imaginer les apports de telles contributions. Cependant nous avons observé que certaines justifications sont endogènes, en ce qu'elles se ramènent à la difficulté des chercheurs à implémenter leurs recommandations, plutôt qu'à des besoins externes aux auteurs. Les solutions apportées tendent alors à s'éloigner des besoins exprimés par les utilisateurs des langages et frameworks d'interaction, au risque de ne plus les influencer. Face à ce problème, il nous semble important de prévoir en amont l'*intégration écologique* des réalisations techniques à une communauté d'utilisateurs.

Cette communauté devrait être clairement identifiée, ainsi que l'a fait Fekete avec le Groupe de Travail Programmation. Il est préférable également de travailler de pair avec elle, afin de s'assurer l'aboutissement des réalisations techniques et recueillir régulièrement des retours sur leur adoption. De par nos observations des travaux ayant proposé des recommandations liées à l'interaction, nous considérons le rapprochement à une communauté d'utilisateurs comme un critère essentiel de réussite. De plus, la réussite ou l'échec d'un projet devrait pouvoir être mesurée, à l'aide de critères énoncés au préalable. Dans notre cas, nous évaluons la réussite de nos travaux à l'un de ces critères :

- l'utilisation de nos outils par des développeurs extérieurs à nos collaborateurs directs
- l'adoption d'un des concepts proposés, dans une autre bibliothèque logicielle que la nôtre
- la publication de travaux inspirés par (ou remettant en question) nos résultats

2.2 Une orchestration explicite et flexible des comportements interactifs

Dans un langage de programmation, on qualifie de *modèle d'exécution* le comportement des différents éléments du langage lors de l'exécution. Il s'accompagne d'un ensemble d'éléments de syntaxe et de règles de grammaire, et permet de prédire le comportement observable d'une application à partir de son code source. Par exemple, le langage C spécifie que les instructions (séparées par des points-virgules) s'exécutent en séquence, chacune attendant que la précédente se termine avant de commencer. Il modélise l'exécution d'un programme comme un enchaînement continu d'instructions, divisées en blocs de code, qui se suivent par des instructions de branchement telles que `goto` ou les appels de fonctions. Le modèle d'exécution est une caractéristique essentielle de chaque langage de programmation, et plusieurs modèles d'exécution peuvent cohabiter dans un même langage.

Il arrive qu'une bibliothèque logicielle utilise un modèle d'exécution différent que le langage de programmation sur lequel elle s'exécute — par exemple pour exécuter les instructions en parallèle plutôt qu'en séquence. Sans modifier le modèle d'exécution du langage, elle y superpose son propre modèle à l'aide d'appels de fonctions, qui forment l'API de cette bibliothèque. Son modèle d'exécution est alors qualifié de *modèle de programmation*. Lorsqu'on désigne le style de programmation (les manières de raisonner avec un modèle de programmation donné), on parlera plutôt de *paradigme de programmation*. Par exemple, le "modèle objet" consiste à représenter les éléments du programme par des conteneurs autonomes (les objets), possédant un ensemble de données et de fonctions (les méthodes) pour opérer sur ces données et les autres objets. Ici, le paradigme de "programmation orientée objet" consiste à organiser le fonctionnement d'un programme par des échanges entre objets, où chacun possède une responsabilité définie, et délègue éventuellement certaines tâches à d'autres objets. En programmation orientée objet, il est courant de rencontrer des méthodes courtes, de quelques lignes de code, ce qui fait ici partie du style de programmation.

Dans le contexte de la programmation en IHM, nous qualifions de *comportements interactifs* les comportements observables en réaction à des actions des utilisateurs. Dans ce domaine, on observe que de nombreux frameworks appliquent un modèle de programmation différent du modèle d'exécution du langage sur lequel ils se basent. Par exemple, Qt implémente son modèle de signaux et slots par dessus le modèle procédural du C++ [Qt19]. De même, Amulet implémente un modèle d'objets à prototypes par dessus le modèle à classes du C++ [Mye97]. Ces observations appuient l'idée que les modèles d'exécution natifs des langages actuels ne sont pas suffisamment adaptés pour exprimer les besoins des applications interactives [Bea08]. Nous cherchons donc à répondre à la question *Comment concevoir un modèle d'exécution adapté à l'interaction ?*

Dans les sous-sections qui suivent, nous présentons différentes *pratiques de programmation* courantes dans le développement d'applications interactives, et mises en pratique dans de nombreux frameworks. Notre but est de les transposer en concepts généralisables aux langages, et de discuter des formes qu'ils devraient prendre à l'avenir.

2.2.1 Les différents modèles de programmation d'interactions

De nombreux modèles de programmation ont été utilisés pour exprimer de l'interaction. Nous en dressons une liste dans cette sous-section, afin de nous positionner sur les modèles à privilégier dans nos Essentiels d'Interaction. Par souci de lisibilité, nous les séparons en deux groupes, ceux qui définissent les comportements *individuels* des instructions, et ceux qui définissent les comportements de *groupes* d'instructions. Les premiers modèles se concentrent sur les opérations d'un langage (ou framework) qui ne peuvent être sous-divisées. Ils incluent les opérations arithmétiques, les accès à la mémoire, voire des opérations de plus haut niveau comme les accès aux fichiers. Ils définissent aussi comment ces différentes opérations se succèdent entre elles (séquencement, simultanéité). Les principaux modèles d'exécution utilisés pour programmer des comportements interactifs individuels aujourd'hui sont :

- la programmation impérative (utilisée dans les langages impératifs), qui modélise les instructions comme des ordres ponctuels donnés à la machine, ordonnées en séquence et exécutées sans se chevaucher
- la programmation déclarative, dans laquelle l'application est une structure de données qui spécifie son comportement (ex. l'apparence visuelle avec HTML), mais ne décrit pas *comment* l'exécuter
- la programmation par flux de données (*dataflow*), dans laquelle les instructions mettent en permanence à jour des valeurs de sortie en fonction de valeurs d'entrée, et qui est implémentée dans les langages VHDL et Max/MSP, ou le framework ICON [Dra01]
- les machines à états et réseaux de Petri, dans lesquels le système est à tout instant dans un état parmi un ensemble fini, et évolue par transitions atomiques entre ces états, avec comme représentants des frameworks comme CPN2000 [Bea00] et ICO [Nav09]

Au niveau des *groupes* d'instructions, on désigne la définition des blocs de code d'un langage (ou framework). Il s'agit de répondre à *quand* et *comment* ils sont exécutés, et comment le comportement global du programme s'exprime par assemblage de ces blocs. Les principaux modèles d'exécution utilisés pour programmer des comportements interactifs de groupes aujourd'hui sont :

- la programmation procédurale, dans laquelle les blocs de code sont des fonctions, exécutées par des appels de fonction statiques ou dynamiques (pointeurs de fonction), et qui retournent l'exécution à la fonction appelante à la fin du bloc, avec éventuellement une valeur de retour
- la programmation fonctionnelle, qui modélise l'exécution du programme comme l'évaluation de fonctions mathématiques, en interdisant les effets de bord autorisés par les affectations de variables
- la programmation orientée objet, qui modélise les blocs de code comme des fonctions appartenant aux objets (les méthodes), et organise le flux d'exécution par des messages envoyés entre objets

- la programmation réactive, qui lie le déclenchement des blocs de code à l'activation de signaux (des évènements internes ou externes au programme), et modélise l'exécution du programme par des cascades d'activations de signaux
- la programmation parallèle, utilisable en conjonction des quatre autres, et qui permet d'exécuter plusieurs flux d'exécution simultanément, en proposant divers paradigmes de programmation permettant les accès concurrents à des ressources communes

La programmation impérative est aujourd'hui omniprésente pour programmer des interfaces graphiques et interactions. En effet, ce modèle correspond au fonctionnement séquentiel des processeurs les plus répandus, et permet de générer du code rapide. De plus, la programmation d'interfaces est largement basée sur des langages de programmation orientés objet (C++, Java, JavaScript, Swift, etc.). En pratique, ces modèles sont utilisés dans la quasi-totalité des bibliothèques d'interaction que nous avons relevées dans le questionnaire en ligne (voir section 1.5). Cependant, nous observons aussi que de nombreux frameworks supportent *plusieurs* modèles de programmation, autorisant plusieurs manières de définir une même interface. Par exemple, Qt permet de construire une interface par appels de fonctions en C++, mais définit aussi le langage dédié QML pour l'y exprimer de manière déclarative. De même, le framework React introduit un modèle *dataflow* sans se substituer au modèle impératif du JavaScript, ainsi que le langage dédié JSX. Ces approches *multi-paradigmes* nous semblent essentielles à la programmation d'interactions. Elles laissent les développeurs libres de choisir leur manière de raisonner, et de changer en fonction du type d'interface à réaliser. C'est particulièrement utile pour associer un type de besoin à un modèle adapté (ex. *dataflow* pour la propagation de contraintes, réactif pour l'enregistrement aux périphériques d'interaction, déclaratif pour la construction d'interface) Nous avons donc choisi de favoriser par la suite une bibliothèque qui inclut ou supporte l'utilisation de plusieurs modèles de programmation.

2.2.2 Les propagations d'évènements par attente active et passive

En programmation d'interactions, les types d'attente active (*polling*) et passive (*pushing*) désignent deux manières de propager une information d'entrée utilisateur dans un programme informatique. En attente active, on interroge de façon répétée une source d'entrées (souris, clavier, manette, etc.). Lorsqu'une nouvelle entrée est détectée, on exécute du code qui va se charger des traitements consécutifs à l'entrée (commande, retour visuel, etc.). Ensuite, qu'une entrée ait été détectée ou non on temporise éventuellement avant d'interroger à nouveau la source d'entrées, afin de ne pas accaparer les ressources de la machine. L'attente active s'illustre communément dans la gestion de bas niveau des dispositifs d'entrée, et la synchronisation entre processus parallèles sur des machines multi-cœurs, où il est essentiel de réagir aussi vite que le permet la machine :

```
volatile int condition;     // force les accès systématiques en mémoire
while (condition == 0) {
    wait(10);               // temporisation de 10ms
}
```

En l'absence de délai, l'attente active garantit une réaction instantanée à tout évènement, au prix d'une consommation continue du processeur, qui le rend indisponible pour d'autres tâches. Par extension, elle s'illustre aussi lorsqu'on interroge tous les évènements d'entrée survenus durant un laps de temps, afin de gérer les entrées dans une boucle à fréquence fixe :

```
while (!end) {
    while (pending_input()) {
        event = poll_input();   // demande non-bloquante
        process_input(event);   // traitement
    }
    render_scene();             // dessin
    wait(10);                   // temporisation de 10ms
}
```

Ce type de boucle et d'obtention des évènements est particulièrement utile lorsque l'application interactive doit réagir à plusieurs sources d'évènements concurrentes (affichage, souris, clavier, réseau). La fréquence fixe est généralement synchronisée avec celle de l'écran, pour minimiser le délai entre le rendu visuel de l'application, et sa perception par l'œil de l'utilisateur. Elle est utile aussi pour maîtriser le déterminisme de l'application, en particulier comme *tickrate* dans les jeux multijoueurs qui synchronisent les calculs entre toutes les machines connectées. Cependant, du fait de la temporisation, l'utilisation d'une attente active introduit ici une latence entre la survenue d'un évènement et son traitement, qui vaut en moyenne la moitié du délai de temporisation (5ms dans l'exemple ci-dessus).

En attente passive (*pushing*), c'est l'émetteur d'un évènement qui le "pousse" aux consommateurs, plutôt que ces derniers en fassent la demande. Pour cela, une fonction de rappel (*callback*) est enregistrée au préalable, qui contient le code à exécuter lorsqu'une entrée est détectée. C'est la source d'entrées qui se charge d'attendre la survenue d'un évènement, et qui exécute alors la fonction de rappel instantanément. Avec ce type de propagation d'évènements, l'exemple ci-dessus deviendrait :

```
set_on_event(e -> {
    process_event(e);
})
set_on_display(() -> {
    render_scene();
})
```

L'attente passive est utile lorsque les évènements surviennent sporadiquement — l'interaction n'est pas continue. C'est le cas du clavier et de la souris, pour lesquels il est normal n'observer des périodes de temps sans activité, pendant lesquelles des demandes répétées aux sources d'entrées seraient superflues. Ce type d'attente est peu utilisé à bas niveau, du fait du coût à l'exécution des fonctions de rappel. En revanche, on l'observe communément dans les frameworks, qui propagent les évènements d'entrée au code utilisateur à l'aide de *callbacks*. Par rapport à une boucle à fréquence fixe, l'attente passive n'introduit aucune latence dans le traitement des entrées utilisateurs, ce qui est désirable dans les domaines où cette latence est néfaste (systèmes temps réel). Cependant, alors qu'avec l'attente active c'est l'application qui reste maîtresse de l'exécution, avec les fonctions de rappel elle cède la main à l'environnement qui se charge de l'appeler à un moment indéterminé. L'attente passive introduit donc un non-déterminisme *du point de vue de l'appelé*, ce qui explique que les boucles de gestion des évènements soient encore utilisées en interne dans de nombreux frameworks.

À cause de la latence induite par l'attente active dans une boucle de fréquence fixe, on lui préfèrera toujours l'attente passive. C'est particulièrement vrai en IHM, où l'effet néfaste de la latence sur les performances des utilisateurs a été beaucoup étudié [Pav11, Ng14, Deb15]. Cependant, le non-déterminisme dû à la délégation de contrôle à l'environnement est problématique. En effet, sans connaître les traitements effectués par le framework, il est impossible de savoir si un délai sépare la survenue d'un évènement, du moment où on reçoit le contrôle. De même, dans le cas d'une application effectuant des calculs complexes (ex. jeu vidéo), il est impossible de garantir qu'on aura assez de temps de calcul chaque seconde, car les traitements du framework sont imprévisibles (ex. mise en cache du rendu graphique, ramasse-miettes). Enfin, comme nous l'avons souligné dans la problématique de cette thèse, avec l'organisation en couches des bibliothèques d'interaction, on ne peut pas garantir que le framework fait suivre tous les évènements d'entrée, et un développeur qui interagit avec plusieurs couches devra s'assurer de la cohérence des données reçues. Dans ces conditions, nous avons choisi de promouvoir une attente passive des évènements, mais où les traitements effectués par le framework sont spécifiés et observables.

2.2.3 Les traitements bloquants et asynchrones

Les modèles de traitements bloquant et asynchrone désignent deux manières de prévoir l'exécution d'un bloc de code après avoir exécuté une fonction pouvant prendre beaucoup de temps. Le modèle bloquant consiste à mettre en pause l'exécution du programme tant que la fonction n'a pas fini de s'exécuter. Une fois qu'elle se termine, le programme reprend son exécution. Ce type de traitements est très utilisé pour les entrées/sorties avec le disque dur ou le réseau. Dans le cadre de la programmation d'interactions, il est utilisé lorsqu'on attend une entrée de l'utilisateur en réponse à une requête du système (par exemple un formulaire avec bouton “Continuer”). On l'illustre par exemple en C :

```
print("Quel est votre nom ?")
nom = read_string()                 // traitement bloquant
print("Bonjour %s, quel âge avez-vous ?", nom)
age = read_number()                 // traitement bloquant
print("Vous avez %d ans.", age)
```

L'intérêt de ce modèle est qu'on peut représenter chaque “fil d'exécution” par un bloc de code continu, en particulier quand celui-ci s'étire dans le temps. En effet, les instructions `print` ci-dessus appartient à un même bloc de code séquentiel, pourtant elles s'exécuteront à des moments très différents. Le modèle bloquant synchrone permet donc très bien de modéliser une *conversation* entre l'utilisateur et la machine, dans laquelle ils communiquent par questions et réponses. En revanche, on ne peut pas attendre plusieurs questions simultanément avec ce modèle (à moins d'autoriser plusieurs fils d'exécution simultanés).

Avec les traitements asynchrones, l'exécution d'une fonction prenant du temps ne bloque plus le fil d'exécution courant. Au lieu de cela, les traitements coûteux de la fonction sont retardés, en attendant que la machine n'ait plus de calculs en cours. C'est la fonction elle-même qui détermine les traitements à effectuer instantanément et ceux qui sont mis en attente, et c'est le “système” (framework, langage,

ou système d'exploitation) qui gère les traitements en attente et choisit quand les exécuter. Lorsque du code doit être exécuté *après* l'exécution de la fonction, une fonction de rappel est souvent passée en argument, qui sera appelée après les traitements en attente. L'exemple précédent devient :

```
print("Quel est votre nom ?")
read_string_async(nom -> {
    print("Bonjour %s, quel âge avez-vous ?", nom)
    read_number_async(age -> {
        print("Vous avez %d ans.", age)
    })
})
```

L'exécution asynchrone permet de gérer plusieurs tâches simultanément avec un seul fil d'exécution. Comme chaque fonction asynchrone ne bloque pas l'exécution, on pourrait par exemple poser plusieurs questions en même temps, et réagir aux réponses dans l'ordre dans lesquelles elles sont données. Dans le cas des interfaces graphiques, la programmation asynchrone est particulièrement utilisée pour les entrées/sorties dont les résultats sont *secondaires* à l'interface, par exemple pour attendre le chargement d'images qui sont insérées dans l'interface à mesure qu'elles sont reçues. Pour modéliser des interactions directes avec l'utilisateur, le problème majeur est qu'on ne maîtrise pas le moment où la fonction de rappel s'exécute. On introduit donc du non-déterminisme, qui peut être source de bugs lorsque le résultat des différentes tâches dépend de leur ordre d'exécution. De plus, la programmation asynchrone tend à produire des fonctions de rappel en cascade, qui nuisent à la lisibilité du programme (voir l'exemple ci-dessus).

Lorsqu'on étend la conversation entre utilisateur et machine à une interface graphique, nous pouvons distinguer deux fils d'exécution principaux. Le fil d'*interface* contient les traitements réalisés en séquence à intervalles réguliers, pour rafraîchir l'interface (ex. traitement des entrées, mise à jour des contraintes de positionnement, rendu visuel). C'est un fil d'exécution réel, qui correspond généralement à un ou plusieurs processus sur la machine. Le fil d'*interaction* représente tous les traitements réalisés (en séquence ou parallèlement) en réaction aux actions de l'utilisateur (ex. afficher un retour visuel, changer de mode d'interaction, déclencher une commande). Il représente la conversation (ou les interactions) entre l'utilisateur et la machine. Pourtant il correspond rarement à un processus dédié sur la machine. En effet, la gestion de multiples processus est complexe pour un framework, car il faut gérer leurs éventuelles synchronisations. Alors que certains frameworks séparent en processus la mise à jour de l'interface et le reste des traitements (ex. le *Event Dispatching Thread* de AWT), la gestion de l'interaction est étroitement liée à la gestion de l'interface, ce qui nécessiterait de nombreuses synchronisations.

Ainsi nous avons deux opportunités à l'issue de cette discussion. La première est de représenter le fil d'interaction par un processus réel (en surmontant les défis sus-mentionnés), à la manière des graphes d'interaction de Huot [Huo06], mais dans un modèle d'exécution impératif plutôt que déclaratif. La deuxième est d'intégrer plus étroitement le fil d'interaction avec le fil d'interface, afin de ne conserver qu'un unique processus. La simplicité conceptuelle de cette approche nous a amenés à l'explorer dans cette thèse. À cause de l'existence d'un unique fil d'exécution, nous avons exclu les traitements bloquants synchrones. Pour éviter d'introduire un non-déterminisme indésirable, nous avons aussi exclu les traitements asynchrones. Nous avons donc cherché à intégrer le fil d'interaction

dans les traitements de l'interface. En pratique ce principe est illustré dans le chapitre 3, où les changements d'état et éventuelles temporisations sont *réifiées* en données globales, sur lesquelles des blocs de code insérés dans un unique fil d'exécution peuvent réagir.

2.2.4 Les callbacks et la logique répartie

Dans les langages de programmation procéduraux, toute fonction doit être appelée explicitement pour s'exécuter. L'initiative d'un appel de fonction est habituellement interne à l'application, elle permet de "sauter" d'une portion du programme à une autre, et d'y revenir ensuite. Lorsque l'initiative vient de l'extérieur (par exemple suite à un évènement d'entrée utilisateur), on utilise des fonctions de rappel (*callbacks*). Ce sont des fonctions matérialisées par des valeurs, qu'on peut passer en argument à d'autres fonctions, et sauvegarder dans des variables. Cette matérialisation prend la forme d'un pointeur de fonction (adresse de la première instruction en mémoire) dans un langage impératif comme le C, ou d'un objet contenant un pointeur de fonction (fonction *lambda*) dans un langage à objets comme Python ou Java. L'utilisation de callbacks permet le principe d'Inversion de Contrôle, dans lequel on délègue le déclenchement d'une fonction au framework (ou langage). On l'illustre avec le Principe d'Hollywood : « *Don't call us, we'll call you* » . Ils sont utilisés pour spécifier des comportements à exécuter après des évènements utilisateurs, pour lesquels l'initiative vient nécessairement de l'extérieur de l'application (par le framework, langage, ou système d'exploitation).

L'utilisation de callbacks favorise une répartition décentralisée du code. En effet, chaque *type* de traitement en réaction à l'utilisateur (déclenchement de chaque commande, propagation de chaque contrainte) est contenu dans un callback distinct. Nous parlons alors de *logique répartie* pour signifier que le comportement observable de l'application est provoqué par des séquences d'instructions fragmentées en plusieurs endroits dans le code. Du point de vue du programmeur d'application, ces fragments peuvent être répartis dans plusieurs fichiers, ou en plusieurs endroits dans un même fichier. La logique répartie se rapporte aussi au stockage des callbacks durant l'exécution de l'application. Ceux-ci sont généralement attachés aux widgets qui les déclenchent — par exemple une commande est attachée au bouton qui la déclenche. Chaque widget stocke alors les valeurs (pointeurs, ou fonctions lambdas) des callbacks qu'il déclenche, et le stockage de l'ensemble des callbacks est donc réparti entre tous les widgets.

Des interfaces réalistes peuvent contenir des centaines, voire des milliers de callbacks — au moins autant que le nombre d'éléments clickables et interactifs de l'interface. Dans ces conditions, la création, la compréhension, et la maintenance de l'interface sont rendues difficiles par la multiplicité et la fragmentation des comportements interactifs. Le problème est qualifié de "*spaghetti of callbacks*", du nom de l'article de Myers qui l'a mis en évidence [Mye91]. Il a donné lieu à de nombreux travaux visant à :

- séparer l'interface et le code pour réduire la fragmentation dans les fichiers (HTML/JavaScript, QML/Qt, FXML/JavaFX)
- regrouper le stockage des callbacks liés à une technique d'interaction en particulier (machines à états)
- fusionner les callbacks réalisant des traitements similaires (voir par exemple `ToggleGroup` dans JavaFX, ou `NSSegmentedControl` dans Cocoa)

La répartition de la logique dans une application interactive est une question de point de vue, en fonction de l'objet d'intérêt dont on évalue la cohésion. Lorsqu'on observe la logique d'un programme du point de vue de la machine, une répartition non fragmentée est équivalente à la séquentialité des instructions exécutées, c'est-à-dire la réduction des appels de fonction et branchements conditionnels. Lorsqu'on observe la logique du point de vue des widgets, elle est équivalente à regrouper toutes les actions faites sur un widget dans un seul et même objet. Enfin lorsqu'on observe la logique du point de vue d'une modalité d'interaction (ex. la souris), elle est équivalente à regrouper toutes les actions pouvant être exécutées par une modalité dans un seul et même objet. En pratique on observe un compromis entre ces trois points de vue dans les frameworks d'interaction, et le point de vue des widgets est souvent favorisé par rapport aux autres dans les architectures basées sur des widgets.

Le problème soulevé par la fragmentation de la logique dans les applications interactives, est que les interfaces s'expriment au mieux dans un point de vue, souvent au détriment des autres. Par exemple, la technique du glisser-déposer est notoirement plus difficile à implémenter lorsqu'on l'exprime par des callbacks sur des widgets [Wag95], que par des callbacks liés à la technique d'interaction [App08]. Le manque de centralisation de la logique du point de vue de la machine complique la compréhension et la visualisation du flux d'exécution du programme, ce qui rend particulièrement difficile le débogage des interfaces graphiques. Nous souhaitons donc améliorer la centralisation de la logique pour chacun des trois points de vue, et améliorer la transition entre les trois afin de ne pas favoriser l'un au détriment des autres. Dans cette optique, nous avons choisi une orchestration de l'exécution peu fragmentée, dans laquelle les actions relatives à un objet ou une technique d'interaction sont autant que possible séquentielles en code.

2.2.5 Le support d'une orchestration explicite et flexible de l'interaction

À la lumière des aspects des paradigmes d'exécution énumérés ci-dessus dans le contexte de l'interaction, nous voulons formuler un Essentiel d'Interaction avec les caractéristiques suivantes :

- inclut ou supporte l'utilisation de plusieurs modèles de programmation
- propage les évènements de façon passive (*pushing*), mais où les traitements liés à la gestion d'évènements sont spécifiés et observables
- intègre le “fil d'interface” et le “fil d'interaction” de façon cohérente dans une même procédure
- réunit les traitements similaires (par type, widget d'appartenance, ou modalité d'interaction) dans des blocs séquentiels de code

Ces caractéristiques ont en commun qu'elles ont pour but de donner plus de contrôle au programmeur cherchant à exprimer des comportements interactifs. Il peut s'agit d'exprimer clairement quelles actions de l'utilisateur peuvent déclencher un bloc de code, ou encore réunir tous les traitements liés à une même technique d'interaction dans un même bloc de code. Nous comparons ce contrôle étendu à celui d'un chef d'orchestre. L'*orchestration* se réfère à l'arrangement des différentes actions les unes par rapport aux autres. Elle se compare aisément avec les orchestres de musique. Chaque musicien doit jouer en rythme, dans la même tonalité (fréquences des notes), et avec la même intensité que les autres musiciens, pour apporter de la cohérence à l'ensemble de la composition. Le chef d'orchestre, qui recherche cette harmonie, communique collectivement et individuellement avec les musiciens, par un vocabulaire spécifique, pour unifier leur rythme, tonalité, et intensité. Nous

parlons d'orchestration pour désigner les outils à la disposition du chef d'orchestre (et dans notre contexte, des programmeurs), pour arranger, ordonner, et synchroniser des musiciens (ou des processus interactifs).

L'orchestration intervient à un niveau macroscopique, comme le souligne Berry dans le développement de Hop et HipHop [Ber14]. Il ne s'agit pas d'orchestrer des instructions atomiques entre elles, mais des processus complexes. Les besoins liés à l'orchestration, dans le contexte des applications interactives, ont été étudiés avec les modèles d'exécution "réactifs", avec des langages comme Esterel [Ber87], Lustre [Cas87], et Signal [LeG91]. Des travaux ont aussi proposé des extensions aux langages impératifs, qui facilitent leur intégration dans des écosystèmes existants. Ainsi, HipHop.js s'intègre dans le langage JavaScript pour produire des applications Web distribuées entre client et serveur [Vid18]. ReactiveC s'intègre dans le langage C à l'aide d'un préprocesseur [Bou91]. Enfin, ReactiveML s'intègre dans le langage OCaml, et a été notablement appliqué aux simulations interactives [Man09] et à la production musicale [Bau13].

Cependant, les travaux basés sur le modèle réactif présupposent que *toute* l'exécution se déroule dans le programme, et ne permettent pas de donner l'initiative à l'environnement — à l'exception d'ICon qui a étendu ce modèle aux évènements extérieurs pour l'appliquer aux entrées de dispositifs d'interaction [Dra04]. Les processus sont toujours déclenchés depuis l'application même, et c'est depuis celle-ci qu'on attend des évènements, ou qu'on synchronise des processus. Ce fonctionnement est différent de celui des frameworks d'interaction courants, dans lesquels l'application laisse la main au système, pour être appelée en retour. Le problème de l'initiative influence le modèle d'interaction qui est adopté, or comme nous l'avons présenté la conservation de l'initiative dans l'application favorise un modèle conversationnel. De plus, ils ne permettent pas d'exprimer des relations de causalité entre les processus, telles que proposées par les *bindings* de djnn [Cha12], ou la Programmation Orientée Aspect [Kic97]. Sans trop orienter le type de solutions à adopter dans notre travail, nous formulons donc le premier Essentiel d'Interaction :

Essentiel d'Interaction n°1

Orchestrer des comportements interactifs procéduraux, en exprimant leur séquencement, leur causalité, et leur réaction à l'initiative d'évènements extérieurs

2.3 L'environnement d'interaction de toute application

L'*environnement d'interaction* d'une application désigne l'ensemble des fonctions et structures de données lui permettant d'interagir avec les périphériques d'interaction (clavier, souris, écran, etc.), et avec les utilisateurs par l'intermédiaire des périphériques. Il désigne les bibliothèques logicielles qui permettent d'intercepter les évènements de la souris et du clavier, d'interroger leurs caractéristiques physiques, ou encore de dessiner à l'écran. La simplicité d'utilisation de ces outils détermine les efforts qu'investiront les programmeurs pour concevoir des techniques d'interaction. Malheureusement, on peut considérer aujourd'hui que ces outils sont généralement complexes et peu accessibles, et qu'ils entravent le travail des chercheurs en IHM. Les problèmes sont multiples et ont été mis en évidence dans le premier chapitre de ce manuscrit. D'abord, les programmeurs sont confrontés à un vaste choix de bibliothèques logicielles, et doivent consacrer du temps à les choisir correctement à partir d'informations partielles ou incomplètes. Ensuite, les bibliothèques et les systèmes interactifs en général sont arrangés en couches interdépendantes, qui posent des problèmes de cohérence d'information, et compliquent le choix des couches auxquelles s'adresser. Enfin, la plupart des bibliothèques d'interaction manquent d'extensibilité, et se prêtent difficilement à des travaux de recherche qui explorent — par définition — des solutions inédites.

Dans cette section, nous argumentons en faveur d'une définition des fonctions et structures de données dédiées à l'interaction avec les utilisateurs, qui puissent être intégrées à un langage ou framework de programmation, afin de répondre aux problèmes soulevés. Les sous-sections qui suivent sont consacrées à l'étude de trois aspects de la programmation d'entrées et sorties avec les utilisateurs, qui nous amènent à définir comme Essentiel d'Interaction **un environnement d'interaction minimal et initialisé au démarrage de toute application**.

2.3.1 Le support de l'interaction dans les langages de programmation

Le langage de programmation consiste a priori en une syntaxe ainsi que des règles de grammaire, qui permettent d'exprimer des ordres à donner à une machine. La complexité des ordres donnés dépend de la puissance d'expression du langage, et est conditionnée par les types de problèmes ciblés par le langage. La plupart des langages de programmation aujourd'hui ciblent en priorité le calcul scientifique et la communication de données (avec les périphériques de stockage, ou entre machines). En pratique cela signifie que leurs concepts de base sont très adaptés au calcul (opérations arithmétiques, fonctions mathématiques, paradigme MapReduce de traitement de données, etc.), et aux communications (lecture/écriture dans des flux de données, abstraction des descripteurs de fichiers, attente asynchrone, etc.). Outre ces concepts, l'*interaction* ne semble pas être supportée explicitement dans les langages de programmation courants.

Qu'entend-on par supporter l'interaction dans un langage de programmation ? Il s'agit de faciliter la programmation d'applications interactives à bas niveau — c'est-à-dire la création de programmes qui reçoivent des données issus de périphériques d'entrée, et émettent des données vers des périphériques de sortie, afin de réaliser une boucle de *perception-action* du point de vue de l'utilisateur. En entrée, on contrôle principalement les ordinateurs au clavier et à la souris pour les ordinateurs de bureau, et au doigt pour les smartphones. En sortie, la majorité des applications interactives utilisent un écran

matriciel (une matrice de points colorés). D'autres types de périphériques sont utilisés dans des contextes spécifiques (manettes, voix, regard, audio, haptique, etc.), cependant le support limité des périphériques de base nous incite à nous concentrer d'abord sur eux. Un support *essentiel* de la programmation d'interactions consiste donc en le triplet clavier/souris/écran.

Pour expliquer notre idée d'un meilleur support des périphériques d'interaction, il convient d'abord de décrire leur support dans les systèmes informatiques actuels. Commençons par prendre l'exemple du langage C, car c'est un exemple clair, et le second langage le plus utilisé aujourd'hui (septembre 2019) d'après l'index TIOBE [Tio19]. Le cœur du langage supporte principalement : les variables, les types, les pointeurs, les enregistrements `struct`, les tableaux, les énumérations, la portée lexicale, les fonctions, les blocs conditionnels, les boucles, les branchements `goto`, les opérations arithmétiques/binaires/logiques, les entiers, les nombres à virgule flottante, les nombres complexes, les chaînes de caractères, l'import/export de symboles externes, et un préprocesseur [ISO18]. Ensuite, tout langage possède une bibliothèque "standard", qui implémente les fonctionnalités ne nécessitant pas de support syntaxique du langage, et s'expriment simplement par des fonctions. Le langage C fournit les modules suivants : `assert.h`, `complex.h`, `ctype.h`, `errno.h`, `fenv.h`, `float.h`, `inttypes.h`, `iso646.h`, `limits.h`, `locale.h`, `math.h`, `setjmp.h`, `signal.h`, `stdalign.h`, `stdarg.h`, `stdatomic.h`, `stdbool.h`, `stddef.h`, `stdint.h`, `stdio.h`, `stdlib.h`, `stdnoreturn.h`, `string.h`, `tgmath.h`, `threads.h`, `time.h`, `uchar.h`, `wchar.h`, et `wctype.h`.

Parmi ces deux niveaux (syntaxe et bibliothèque standard), le langage C ne spécifie *aucun* support du triplet clavier/souris/écran. Seules les conventions UNIX et POSIX permettent d'intégrer un support rudimentaire avec l'utilisation de fichiers : l'entrée de texte au clavier et le retour visuel dans un terminal textuel. C'est à partir du système d'exploitation qu'on dispose d'un support suffisant des périphériques d'interaction, cependant de nombreuses alternatives existent et compliquent le choix des programmeurs : GNU Readline, OpenGL, conio.h (MS-DOS), ncurses (UNIX), Newt (Linux), X11 (UNIX), Wayland (Linux), Framebuffer (Linux), DRM (Linux), KMS (Linux), evdev (Linux), udev (Linux), libinput (Linux), GDI (Windows), Direct2D (Windows), DirectDraw (Windows), DirectWrite (Windows), QuickDraw GX (macOS), Carbon Event Manager (macOS), Quartz 2D (macOS), Metal (macOS). Ici nous n'avons énuméré que des bibliothèques *natives* des différents systèmes. En outre il existe un grand nombre de bibliothèques qui se basent sur les précédentes, fournissent des services similaires, et contribuent à compliquer la topologie des outils de programmation d'interaction dans les systèmes informatiques.

Cette situation peut s'expliquer par les changements historiques des modalités d'interaction en entrée et sortie. Bien que l'usage du clavier et de la souris soient standards depuis maintenant plus de 40 ans, leurs caractéristiques ont beaucoup évolué. Les claviers ont évolué principalement dans les dispositions des touches et les touches "spéciales" (commandes, contrôle multimédia). Les souris ont évolué dans les types de capteurs, la précision de la molette, les boutons présents, ainsi que les fonctions de transfert entre déplacement physique et position du curseur à l'écran. De plus, de nombreux périphériques ont été développés qui ont tenté de faire évoluer ces modalités d'interaction standard, comme par exemple la souris 3D, le trackpad, ou le clavier "méduse". Face à ces évolutions, les fonctionnalités d'interaction ont pu être considérées comme instables dans le temps, et dissuader les concepteurs de langages à les intégrer dans les bibliothèques standard. En outre, il est courant que certains périphériques d'interaction fonctionnent partiellement (audio muet, absence d'accélération

matérielle, touches multimédia non reconnues), sans empêcher le système d'exploitation de fonctionner correctement. Or la spécification de *toute* fonctionnalité d'un langage implique qu'elle doive fonctionner sur tous les systèmes, donc qu'elle est sous la responsabilité des concepteurs de compilateurs. Dans ces conditions, un support de l'interaction dans les langages obligerait les concepteurs de compilateurs à suivre les évolutions des périphériques d'interaction, et demanderait un travail de maintenance conséquent, ce qui peut expliquer que l'interaction n'y figure finalement pas.

Comment formuler une recommandation réaliste et réalisable dans cette situation ? Nous proposons trois pistes de solutions, sans en choisir une en particulier pour le moment. La première possibilité est d'**inclure dans les langages les fonctionnalités stables depuis plusieurs décennies** :

- pour le clavier, les évènements d'appui et relâchement de touche, avec le code de position de la touche et le caractère imprimable correspondant
- pour la souris, les évènements de déplacement physique relatif, d'appui et relâchement de bouton, et de déplacement physique de la molette
- pour l'écran, l'évènement de synchronisation avec le rafraîchissement de l'affichage, avec la matrice de couleurs à éditer

Ces fonctionnalités sont basiques, mais faciliteraient le développement de bibliothèques logicielles robustes et multi-plateformes. En effet, elles ne nécessiteraient pas d'utiliser une bibliothèque dédiée pour chaque périphérique et chaque système, et réduiraient ainsi l'apprentissage nécessaire. Ensuite, par leur inclusion dans le langage elles ne nécessiteraient pas l'installation de bibliothèque tierce, ni d'actions spécifiques pour les activer, ce qui améliorerait leur accessibilité. Enfin, nous conjecturons que leur inclusion dans un langage (aux côtés de fonctionnalités basiques telles que la manipulation de texte ou les accès aux fichiers) les forcerait à adopter une interface simple, qui puisse raisonnablement être supportée par les compilateurs tout en conservant une certaine puissance d'expression.

Une seconde possibilité est de **reconnaître la longévité de certaines bibliothèques logicielles liées à l'interaction, et de faciliter l'utilisation de leur API**. On pourrait ainsi imaginer que le compilateur initialise automatiquement ces bibliothèques, s'il détecte que leurs fonctions sont utilisées. De plus il ne serait pas nécessaire d'installer ces bibliothèques en plus d'un compilateur, pour avoir à les utiliser. Enfin, la standardisation de ces bibliothèques faciliterait le dilemme du choix lorsque des alternatives existent, et contribuerait à les renforcer ainsi que leurs communautés en ligne. Parmi ces bibliothèques, nous pouvons citer :

- OpenGL (1994), pour le rendu 3D
- FreeType (1996), pour le rendu des polices de caractères
- SDL (1998), pour la gestion des fenêtres et entrées

Enfin, une troisième possibilité est d'**intégrer le support de l'interaction dans la syntaxe des langages, plutôt qu'en utilisant des fonctions**. Ces mécanismes étendraient les langages de programmation, pour exprimer l'interaction de façon plus “native”. Un exemple d'une telle extension est le support de la souris dans le langage Processing [Rea14], qui consiste en des variables globales comme `mouseX` et `mouseY`, et des noms de fonctions comme `mousePressed()` qui sont appelées automatiquement lors d'un appui de la souris.

La troisième piste nous semble la plus intéressante, car elle reconnaît le caractère essentiel de l'interaction dans la programmation. Après tout, il nous serait impossible d'observer le fonctionnement d'un programme, ni d'agir dessus, sans un support minimal du triplet clavier/souris/écran. Nous nous sommes donc attachés à proposer des concepts qui puissent s'intégrer naturellement dans les langages de programmation, non pas en tant qu'appels de fonction mais comme syntaxes natives. Dans ce travail de thèse, nous proposons une extension du langage Smalltalk dédiée à l'animation, que nous présentons en section 2.4. Enfin dans le framework présenté au chapitre 3, nous représentons les périphériques d'interaction comme des objets globaux, que l'on peut interroger à tout moment pour observer les évènements d'entrée.

2.3.2 Des alternatives à la programmation évènementielle

La programmation évènementielle est un concept populaire en IHM, mais qui est aussi très peu défini. Elle se caractérise principalement par la manipulation d'évènements (*events*), qui peuvent représenter des actions de l'utilisateur (ex. mouvement de souris), des messages entre processus ou machines, ou tous types de changements d'état. Par exemple, pour modéliser les mouvements de la souris en programmation évènementielle, on définirait une structure `MouseMoveEvent`, contenant le déplacement relatif de la souris, ainsi que d'éventuelles informations comme le nom du périphérique ou sa précision.

```
struct MouseMoveEvent {
    int dx;
    int dy;
    String productID;
    int dpi;
    ...
}
```

La programmation évènementielle est souvent utilisée pour qualifier des systèmes qui écoutent leur environnement, cependant la *manière* de gérer les évènements varie. Lorsqu'elle est utilisée pour propager des évènements avec attente passive, on l'associe avec le patron observateur (*listener*) pour enregistrer des fonctions de rappel (*event handler*) traitant chaque type d'évènement. Avec ce mécanisme, chaque widget stocke une fonction de rappel pour chaque type d'évènement qu'il peut recevoir (clic de souris dans sa zone, appui de touche lorsqu'il est actif, etc.). Lorsque le framework détecte la survenue d'un évènement, il remplit les champs d'une structure d'évènement, trouve le widget sur lequel l'évènement s'est produit, et exécute la ou les fonctions de rappel qui y sont enregistrées pour le type d'évènement en question. Les fonctions reçoivent en argument la structure initialisée.

Lorsque la programmation évènementielle est utilisée en propagation avec attente active, on lui associe une file d'évènements (*event queue*), qui les accumule afin de les traiter périodiquement par paquets. La file est alors parcourue dans l'ordre d'arrivée des évènements, ceux-ci sont traités un par un, puis la file est vidée pour accueillir les prochains évènements. Ce type de stockage des évènements est courant dans les jeux vidéo, lorsqu'ils sont synchronisés avec un tickrate particulier — celui d'un serveur dans le cas des jeux multijoueurs.

La modélisation des évènements dans une application interactive est un compromis entre la taille des structures d'évènements, et la spécialisation des traitements, qui forme un continuum de granularité des solutions allant de *monolithique* (avec un type d'évènement complexe et unique) à *fine* (avec une grande variété de types d'évènements à la structure simple). Ce continuum est représenté en figure 14. À une extrémité, tous les évènements sont représentés par une même structure, et chaque gestionnaire doit filtrer les évènements qui ne l'intéressent pas. En contrepartie, on ne conserve qu'une seule liste de gestionnaires d'évènements, ou une seule file d'évènements. Des exemples d'application de ce principe sont le framework SDL [Lan98] ainsi que le protocole TUIO [Kal05]. À l'autre extrémité, chaque type d'évènement est représenté par sa propre structure, et un gestionnaire d'évènements n'a jamais à filtrer d'évènement qui ne l'intéresse pas. En contrepartie, il faut maintenir autant de listes de gestionnaires, de files d'évènements, et de canaux de transmission d'évènements que le nombre d'évènements possibles. Des exemples d'application de ce principe sont les frameworks AWT et Swing, ainsi que le framework GLFW [GLF02] (celui-ci ne déclare aucune structure mais passe les variables directement en arguments des fonctions de rappel).

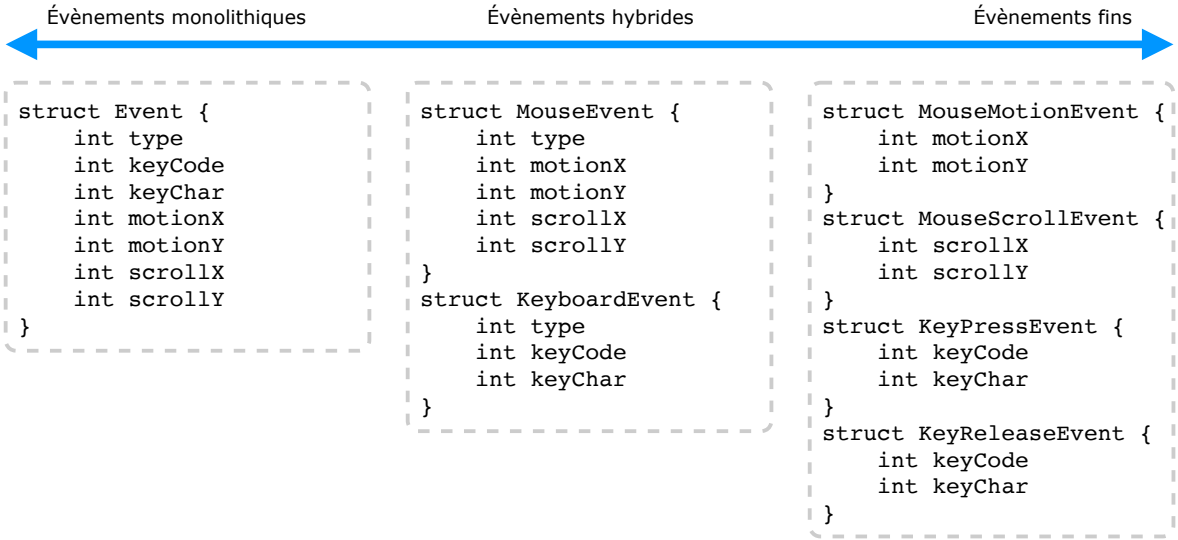
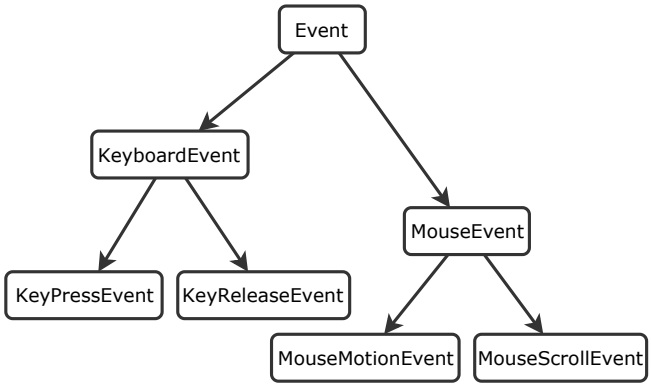


Figure 14 : Le continuum de granularité des types d'évènements, illustré par un exemple de code supportant les mouvements de souris et les appuis/relâchements de touches du clavier.

Dans le cas des évènements fins, le type de chaque évènement n'est pas stocké explicitement mais correspond au nom de la structure. Il épargne au programme l'utilisation de structures conditionnelles (`if/else` et `switch`) pour filtrer les évènements, et laisse cette responsabilité au framework et au langage. En pratique la plupart des frameworks orientés objets définissent des hiérarchies d'évènements (figure 15), qui leur permettent de proposer des gestionnaires à tous les niveaux du continuum, au prix d'un code plus complexe.

Cependant, en pratique l'utilisation d'évènements fins favorise la fragmentation de la logique discutée en section 2.2.4. En effet, les différents types de traitements sont séparés en différentes fonctions (ex. `onMouseMotion(...)`, `onMouseScroll`, `onKeyPress`, `onKeyRelease`). Ces fonctions contribuent à séparer tous les comportements correspondant aux actions possibles dans l'interface, en une multitude de fonctions individuelles qui compliquent la maintenance globale de

l'application. En outre, la programmation évènementielle abstrait les caractéristiques de chaque périphérique, en le réduisant en une *modalité* d'interaction. Bien qu'elle ait ainsi permis, par exemple, d'adapter facilement des interfaces de bureau à l'utilisation du doigt (en générant des évènements de souris), ce type d'abstraction réduit la richesse de l'interaction au doigt (précision, intensité de pression, orientation) en le réduisant à un pointeur. Enfin, les structures d'évènements communiquent par définition les données relatives à l'action observée, et omettent souvent les données relatives aux périphériques d'entrée. Elles rendent ainsi plus difficile l'adaptation de l'interaction aux types de périphériques qui génèrent des évènements.




```
class Event {}
class MouseEvent extends Event {}
class MouseMotionEvent extends MouseEvent {
    int motionX
    int motionY
}
class MouseScrollEvent extends MouseEvent {
    int scrollX
    int scrollY
}
class KeyboardEvent extends Event {
    int keyCode
    int charCode
}
class KeyPressEvent extends KeyboardEvent {}
class KeyReleaseEvent extends KeyboardEvent {}
```

Figure 15 : Exemple de hiérarchie d'évènements supportant les mouvements de souris et les appuis/relâchements de touches du clavier, et le code correspondant

Nous considérons donc comme essentiel de **propager les données d'entrée en conservant les caractéristiques des périphériques qui les ont générées**, et avons cherché des alternatives à la programmation évènementielle au cours de ce travail de thèse. Dans un système interactif, tout changement d'état s'accompagne de données décrivant ce changement (quel bouton est pressé, de combien d'unités la souris se déplace, etc.). Ces données doivent être mises à disposition du programme. Des alternatives à la programmation évènementielle consistent donc en leur stockage autre part que des structures d'évènements :

- sur l'objet recevant l'interaction (bouton cliqué, champ de texte actif)
- sur le périphérique à l'origine de l'interaction (souris, clavier)
- comme arguments aux gestionnaires d'évènements
- dans des variables globales, accessibles depuis tout gestionnaire d'évènement

Dans le chapitre 3, nous présentons une alternative dans laquelle ces données sont stockées dans des objets représentant les périphériques à l'origine de l'interaction.

2.3.3 Le rendu immédiat et différé

Pour dessiner des formes géométriques, du texte et des images à l'écran, on distingue deux méthodes de dessin adoptées par diverses bibliothèques graphiques, le mode immédiat (*immediate*) et le mode différé (*retained*). En dessin immédiat, on exécute des instructions de dessin comme on donnerait des ordres au système graphique. Celui-ci peint les instructions à l'écran immédiatement

lors de leur exécution, ou peut les ajouter à une file d'attente pour les exécuter en groupe plus tard. Les bibliothèques de dessin immédiat fournissent un “canevas” dans lequel les instructions viendront peindre, et qui sera ensuite affiché sur tout ou partie de l'écran.

En pratique les bibliothèques de dessin de bas niveau (OpenGL, Skia, Cairo) se contrôlent avec des instructions immédiates. La plupart des frameworks d'interaction fournissent aussi des modes de dessin immédiat (`canvas` pour HTML/JS, `QPainter` pour Qt, ou `Canvas` pour JavaFX). Enfin le framework ImGui est notable pour avoir étendu le mode immédiat à la définition d'interfaces graphiques, et clarifié la distinction entre les deux modes [Mur05]. Le mode immédiat est généralement perçu comme offrant le meilleur contrôle, car les instructions sont exécutées à un moment et dans un ordre déterminés. Cependant, il n'offre aucune persistance aux instructions de dessin, elles devront être renvoyées à nouveau pour le rendu de chaque nouvelle trame.

En dessin différé, on fournit une description des éléments à dessiner au système (un *modèle*), et celui-ci se charge de les afficher sans contrôle explicite du programmeur. Les descriptions incluent la couleur de fond des éléments, leur forme géométrique, ou encore du texte à afficher. Le système décide l'ordre de leur affichage, la fréquence de rafraîchissement à l'écran, ainsi que des détails visuels comme le lissage sous-pixel ou l'utilisation de flou de mouvement. Le modèle est généralement structuré au-delà d'une simple liste, en un *arbre de scène* qui ne se limite pas au rendu visuel, mais sert aussi à la distribution des évènements d'entrée et la résolution des contraintes de positionnement. L'apparence des éléments est aussi parfois spécifiée en code (de rendu immédiat), pour contrôler précisément leur apparence. Ce code est alors spécifié dans une méthode de rappel (ex. `paintComponent` avec Swing), déléguée au framework par Inversion de Contrôle.

Notre objectif dans cette partie serait de concilier le contrôle fin du rendu immédiat, avec la persistence du rendu différé. En effet, l'ordre indéterminé dans lequel le framework exécute les instructions de dessin en rendu différé limite le contrôle donné aux développeurs de techniques d'interaction, car ils doivent spécifier explicitement les relations d'ordre entre éléments dessinés (ce qui nécessite plus de code). Nous cherchons donc à **supporter un modèle de rendu différé, mais dans lequel les étapes de rendu soient explicites et non réservées à la discrétion du framework**.

2.3.4 Le support d'un environnement d'interaction minimal et initialisé

À partir des discussions développées dans cette section, nous voulons formuler un Essentiel d'Interaction avec les caractéristiques suivantes :

- rend disponible dans le langage un support simple du triplet clavier/souris/écran
- intègre des concepts liés à l'interaction dans la syntaxe même d'un langage
- propage les données des dispositifs d'entrée en conservant leurs caractéristiques
- supporte un rendu graphique différé, avec des étapes de rendu explicites (voire flexibles)

Nous arguons que ces points pourraient faciliter le développement d'applications interactives, et plus particulièrement celui de nouvelles techniques d'interaction dans un contexte de recherche. Tout d'abord, en enlevant l'étape d'initialisation des différentes bibliothèques de périphériques, nous réduisons le code d'une application (ce que font déjà la majorité des frameworks). De plus, en intégrant les périphériques au plus près du langage, nous comptons réduire l'apprentissage nécessaire à chaque bibliothèque supplémentaire. Il est important alors que cette intégration repose sur des

concepts qui la rendent plus claire que l'utilisation classique des fonctions d'une API. Nous conjecturons que la définition de concepts intégrant l'interaction dans la syntaxe du langage favoriserait leur appropriation par les développeurs, ce qui est essentiel aux pratiques de développement opportunistes. Nous illustrons ces principes avec le concept d'*animation de fonction* présenté dans la section suivante, ainsi que celui de *réification des périphériques en objets globaux* présenté dans le chapitre 3. Ainsi, notre second Essentiel d'Interaction est le suivant :

Essentiel d'Interaction n°2

Fournir un support minimal des entrées et sorties dans l'environnement de toute application, supporté par des concepts intégrés de façon cohérente avec le langage de programmation

2.4 Transformation des appels de fonctions en animations

Motivés par la prédominance des frameworks observée durant les premières interviews, nous avons entrepris la réalisation d'un projet les améliorant, sans pour autant investir de l'énergie pour en modifier un explicitement. De plus, à partir des Essentiels d'Interaction, nous souhaitions démontrer notre vision d'une syntaxe intégrée à un langage de programmation. Nous avons donc développé, à partir du concept d'*animer une fonction*, une extension du langage Smalltalk qui convertisse une transition instantanée par appel de méthode, en une transition animée dans le temps. Avec ce travail, nous nous sommes également attachés à inclure le support des traitements en *durée* dans l'environnement d'interaction des applications (c'est-à-dire les traitements qui se déroulent sur une période de temps plutôt qu'instantanément). Conformément au premier Essentiel d'Interaction, l'orchestration des animations est rendue explicite par une règle simple : toute fonction animée est exécutée à fréquence fixe pendant la durée d'animation.

Ce travail a pu être réalisé grâce à la collaboration avec une équipe de recherche en Génie Logiciel, RMoD, qui développe son propre interpréteur et environnement de développement pour Smalltalk, Pharo [Duc17]. Avec l'aide des chercheurs et ingénieurs de cette équipe, nous avons pu réaliser une extension du langage Smalltalk dédiée à l'expression des animations dans les interfaces graphiques. Ce travail a été présenté en *Late Breaking Result* lors de la conférence EICS'17 [Raf17], et démontré en direct lors de la conférence Pharo Days 2017 [Raf17]. Nous détaillons ici ce travail.

2.4.1 Introduction

Depuis le développement des ordinateurs, les animations ont été utilisées dans une gamme de plus en plus large de scénarios tels que : l'enseignement de la programmation avec des environnements visuels [Sta93, Res09, Dan11], les transitions dans les interfaces graphiques et visualisations pour redimensionner les fenêtres ou alterner entre vues sur des données [Sha07, Kle05, Dra11], ou l'animation de personnages virtuels dans les jeux vidéo par interpolation entre images-clés [Bro88, Wil97]. Les animations sont considérées comme utiles dans les interfaces utilisateur pour aider à suivre les changements [Sch07], et dans les visualisations pour construire une carte mentale des informations spatiales [Bed99]. Elles peuvent aussi donner un sens à la visualisation des données [Gon96], à la narration [Ken02], ainsi que de nombreux autres usages dans les interfaces utilisateur [Che16].

Les frameworks d'interaction ont évolué au fil des ans pour supporter une plus grande variété d'usages, en proposant des manières plus flexibles d'animer les éléments des interfaces utilisateur. Alors que les systèmes d'autrefois animaient quelques propriétés (position, couleur) avec des fonctions dédiées pour chacune, les systèmes modernes en contiennent trop pour poursuivre de cette façon. Par exemple, CSS a 44 propriétés animables [Bar18], et Core Animation a 29 propriétés animables [App06]. Pour gérer ce nombre croissant de propriétés animables, la plupart des frameworks définissent des types génériques qui peuvent être animés (comme `IntProperty`, ou `DoubleProperty`), au lieu d'avoir une fonction spécifique pour chacune. Ils améliorent ainsi la flexibilité du choix des propriétés à animer à l'exécution, et réduisent la taille de leur API. Ils indiquent aussi implicitement que *toute* propriété peut être animée, ou au moins celles qui auraient du sens pour le programmeur.

Pourtant cette flexibilité a un prix. L'animation de types définis par l'utilisateur nécessite de fournir une API avancée, qui expose les détails de bas niveau des systèmes d'animation des frameworks, en particulier les timers et threads. Il en résulte des APIs d'animation plus larges et des syntaxes lourdes en raison des indirections supplémentaires pour accéder aux types animables. Il se crée aussi une courbe d'apprentissage abrupte entre l'API de base et l'API avancée, qui est susceptible de forcer les programmeurs à s'en tenir autant que possible aux propriétés animables existantes.

Dans cette section nous introduisons un *opérateur de durée*, pour exprimer les animations par transformation d'appels de mutateurs (*setters*) en transitions animées. Nous l'illustrons avec le pseudo-code suivant :

```
object.setProperty(target) during 2s
```

Cette syntaxe s'étend très simplement aux appels de fonctions (hors méthodes d'objets), cependant dans le cadre de ce travail nous l'avons implémentée dans un langage à objets. Dans le reste de cette section, nous parlerons exclusivement de méthodes, et spécifions les traitements des fonctions lorsque ceux-ci diffèrent. Nous commençons par énumérer les attributs caractérisant une animation, ainsi que les étapes nécessaires pour la construire à partir d'un appel de méthode avec durée. Ensuite, nous

décrivons l'implémentation d'un prototype fonctionnel pour la plateforme Pharo. Pour finir, nous comparons six frameworks d'interaction modernes, et discutons des limites de notre système et l'implication de ce travail pour le reste de ce travail de thèse.

2.4.2 Caractérisation d'une animation

Dans les systèmes interactifs modernes, la transition d'un état initial à un état final est instantanée. Changer la position d'un objet à l'écran le fait disparaître de sa position actuelle, et en même temps apparaître à sa nouvelle position. Ce changement brusque peut casser notre perception d'un seul et même objet s'est déplacé, plutôt qu'un nouvel objet soit apparu à une autre position. Lorsqu'on souhaite souligner le déplacement d'un tel objet, ou maintenir la perception de son unicité, on a recours aux animations pour adoucir la transition dans le temps, afin qu'elle semble continue. La définition qu'en donnent Betrancourt et Tversky est : « *any application which generates a series of frames, so that each frame appears as an alteration of the previous one, and where the sequence of frames is determined either by the designer or the user* » [Bet00].

L'animation d'une propriété dans un framework d'interaction consiste à remplacer une transition instantanée par plusieurs modifications mineures de la même propriété, en séquence rapide. La fréquence de ces changements intermédiaires est aussi importante que possible, pour créer une illusion de continuité dans la transition principale. Elle est généralement fixée au maximum de la fréquence de rafraîchissement de l'écran, qui est le nombre de trames générées pouvant être peintes à l'écran chaque seconde. Sur la plupart des écrans, cette fréquence est de 60 Hz, et est parfois ralentie lorsque le rendu d'une trame est trop long.

Chaque modification commence par le calcul d'un temps relatif entre le début et la fin de l'animation : $t = f(now)$, où $t = 0$ au début, et $t = 1$ à la fin. Le temps t ne croît pas nécessairement uniformément dans [0, 1] : il peut accélérer au démarrage, rebondir avant l'arrêt, et même osciller autour de l'arrivée (voir [Sit19] pour une liste complète). Nous appelons f une fonction de transition (en anglais *easing function*).

Ensuite, la valeur calculée pour chaque modification est obtenue avec une fonction d'interpolation, $interpolate(start, target, t)$, qui renvoie *start* lorsque t vaut 0 et *target* lorsque t vaut 1. Cette fonction est spécifique pour chaque type de valeurs, par exemple des couleurs et des positions seraient interpolées avec des algorithmes différents.

Pour décrire le concept d'une propriété animée, nous nous sommes basés sur les 5 *aspects de haut niveau des animations* définis par Mirlacher et al. [Mir12], et que nous avons étendus. Nous définissons donc un *objet de transition animée* comme contenant :

- **receveur** — l'objet à animer (absent dans le cas d'un appel de fonction)
- **signature de mutateur** — le nom de la méthode modifiant la propriété visée, et les types de ses arguments
- **valeurs clés** — les valeurs de départ et d'arrivée pour chacun des arguments
- **durée** — en secondes
- **fonction de relaxation** — qui contrôle les effets de vitesse durant la transition

- **fonction d'interpolation** — pour chaque argument, spécifique à chaque type
- **état d'exécution** — permet de mettre en pause, de redémarrer, de faire boucler, ou d'inverser la transition animée

2.4.3 L'opérateur de durée

Notre opérateur associe une durée à un appel de fonction, créant un objet de transition animée `<appelFonction> during <durée>`. Son interprétation consiste en quatre étapes, comme illustré dans la figure 16.

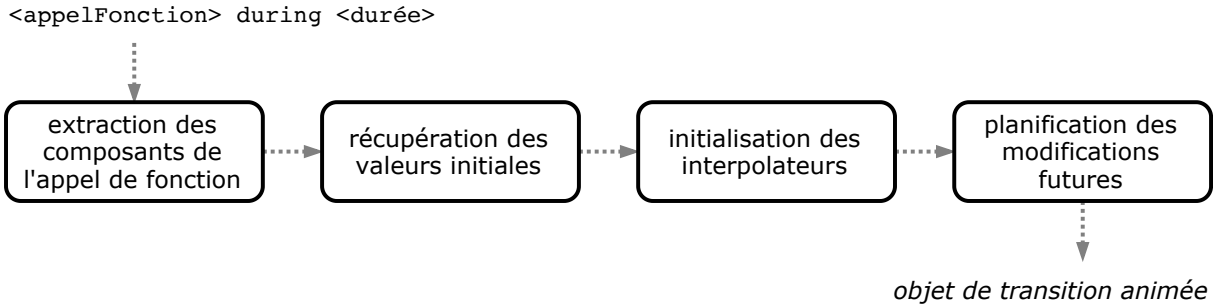


Figure 16 : Les quatre étapes pour transformer un appel de fonction en objet de transition animée

La première étape, *extraction des composants de l'appel de fonction*, récupère quatre éléments de l'appel de fonction : son receveur (si c'est une méthode d'objet), son nom, les valeurs de ses arguments, et leurs types. Elle annule également l'exécution immédiate de la fonction. Cette étape *réifie* en fait l'appel, étant donné que nous inspectons ses données, et extrayons ses caractéristiques. Elle nécessite que le langage de programmation supporte l'introspection du code, c'est-à-dire la possibilité d'examiner ses propriétés au lieu de l'exécuter (plus de détails sont donnés dans la partie Implémentation).

Pour la *récupération des valeurs initiales*, nous avons besoin d'un mécanisme pour obtenir les valeurs courantes d'une propriété ciblée par une méthode ou fonction mutateur. Heureusement, de nombreux frameworks adoptent des conventions de nommage pour les accesseurs (*getters*) et mutateurs. Qt [Qt19], par exemple, fait correspondre la plupart des mutateurs `setProperty(...)` avec des accesseurs `getProperty()`. Sur la plateforme Smalltalk, la convention est de leur donner le même nom, l'un prenant simplement un argument et l'autre non — comme `object property` et `object property: <value>`. Les conventions de nommage sont importantes : sans elles, il faudrait associer explicitement chaque mutateur à l'accesseur correspondant, ce qui nécessiterait d'éditer chaque framework et invaliderait l'intérêt de ce travail. Pour les fonctions à plusieurs arguments, on peut exiger que les accesseurs renvoient plusieurs valeurs si le langage le permet (ex. Python), ou les renvoyer dans des références passées en paramètres. Une fois que l'accesseur est obtenu à partir de son nom, il est appelé dynamiquement pour récupérer les valeurs courantes, qui seront les valeurs initiales de la transition animée. Pour les langages sans appels de fonction dynamiques (*dynamic dispatch*) comme le C, une convention de nommage intelligente (ex. `property()` ⇒ `get_property()`) permettrait une substitution par le préprocesseur. Autrement, cela nécessiterait un support explicite de la part du compilateur, que nous n'avons pas exploré dans ce travail de thèse.

La troisième étape, *initialisation des interpolateurs*, attribue une fonction d'interpolation par défaut à chaque argument, en fonction de leur type. Cette fonction pourra être remplacée ultérieurement sur l'objet de transition. Pour les entiers et nombres réels, nous utilisons la formule :

$$interpolate(start, target, t) = start \times (1 - t) + target \times t$$

Les types composites tels que les positions et les couleurs ont leurs champs interpolés séparément sous forme de nombres. Lorsque le langage de programmation supporte le polymorphisme, la formule ci-dessus peut être utilisée pour ces types composites (à condition qu'ils implémentent l'addition et la multiplication à un réel), et ainsi supporter un grand nombre de types par défaut. Néanmoins, il y a des types pour lesquels ce type d'interpolation n'aurait pas de sens, comme les tableaux ou les chaînes de caractères, par exemple en figure 17. Dans ces cas, les programmeurs doivent être en mesure de fournir leur propre fonction d'interpolation.

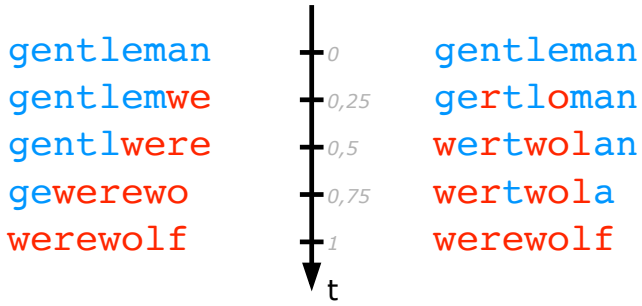


Figure 17 : Deux interpolateurs alternatifs pour les chaînes de caractères. À gauche, la nouvelle chaîne est insérée par la droite. À droite, chaque modification est faite sur des indices aléatoires.

Pour la *planification des modifications futures*, il s'agit de demander au framework ou au système d'exploitation d'exécuter le code de mise à jour de l'animation à intervalles réguliers dans le futur. En pratique l'interaction avec les utilisateurs nécessite une latence de retour la plus faible possible, c'est-à-dire le délai entre chaque modification et son résultat à l'écran. C'est important dans de nombreux contextes, tels que l'interaction avec les écrans tactiles [Deb15], les jeux en ligne [Cla06], ou encore les instruments de musique numériques pour lesquels on animerait le signal audio [Jac16]. Par conséquent, chaque mise à jour de l'animation doit être programmée aussi tard que possible, et avant toute fonction qui en dépend, comme le montre la figure 18.

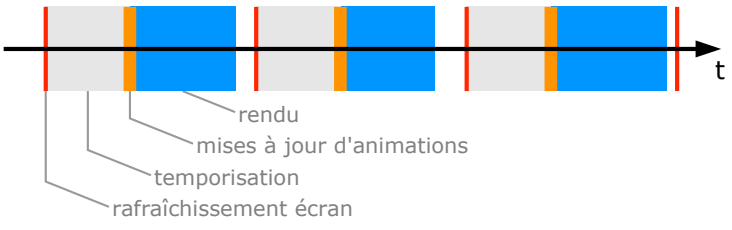


Figure 18 : Représentation simplifiée du moment où insérer les mises à jour d'animations. Le système d'exploitation reçoit un signal de l'écran lorsque la trame précédente est peinte, puis on temporise pour minimiser le temps restant entre le rendu et le prochain rafraîchissement. Le code d'animation est inséré juste avant chaque exécution du rendu.

Ici, le support idéal prendrait la forme d'un mécanisme de *callback* pour exécuter le code juste avant la phase de rendu, comme la fonction `requestAnimationFrame` du DOM en HTML [Fau16]. Pour finir, une fois la planification effectuée, l'opérateur de durée initialise l'état d'exécution pour démarrer immédiatement l'animation.

2.4.4 Implémentation

Notre prototype fonctionnel et preuve de concept de l'opérateur de durée a été réalisé avec le langage Smalltalk, et testé avec trois frameworks d'interaction. Smalltalk est un langage orienté objet, dans lequel les appels de méthode sont remplacés par des *envois de messages* : le code `object x: 10 y: 20` envoie à `object` le message `x:y:`, avec arguments `10` et `20`. Les messages sont distribués dynamiquement, c'est-à-dire que la méthode exécutée pour le message `x:y:` est sélectionnée à l'exécution. Notre prototype exprime les animations avec :

```
[object property: target] during: 2 seconds
```

Notre extension est incorporée dans le message `during:`. Les crochets autour de l'appel de mutateur créent une fermeture (*closure*), qui est un objet contenant le code. Nous pouvons ainsi inspecter le code à l'intérieur sans l'exécuter, et résolvons les quatre étapes décrites dans la partie précédente comme suit :

- L'extraction des composants du message se fait en analysant le *bytecode* de la fermeture, à l'aide des fonctionnalités d'introspection de la plateforme ;
- Pour récupérer une valeur initiale, nous enlevons les deux points de `property:` pour obtenir le nom de l'accesseur correspondant. Ce message est alors envoyé à `object`, et la valeur qu'on obtient en retour est la valeur initiale. Notons que ce mécanisme ne permet pas d'animer des fonctions avec plusieurs arguments, car un accesseur ne retourne qu'une seule valeur ;
- Notre interpolateur par défaut utilise le polymorphisme, en exécutant directement `(start * (1 - t)) + (target * t)`, qui envoie les messages `*` et `+` aux valeurs de début et de fin. Les développeurs peuvent également remplacer ce code d'interpolation pour les types incompatibles. Dans le cas contraire, le Debugger (un mécanisme standard d'erreurs en Smalltalk) est invoqué lors de la première mise à jour de l'animation ;
- Étant donné que nous n'avons pas accès aux évènements de rafraîchissement de l'écran au sein de la plateforme Pharo, les mises à jour futures sont enregistrées à l'aide du mécanisme de callbacks d'un des frameworks hôtes. Cette limitation est discutée plus en détail dans la présente section.

L'extension renvoie un objet de type `Animation`, qui contient tous les paramètres de l'animation et permet de les modifier. Il est ainsi possible de remplacer l'interpolateur par défaut, de spécifier une fonction de transition, voire même de modifier la valeur cible de l'animation :

```
animation := [object position: 100@100] during: 2 seconds.
animation interpolator: [:v :t | (v first * t) + (v last * (1 - t))].
animation easing: animation backOut.
[animation to: 0@0] during: 2 seconds.      "animation de l'animation !"
```

2.4.5 Tests préliminaires

Nous avons testé cette extension d'animation avec les trois frameworks d'interaction les plus populaires pour la plateforme Pharo : (i) Morphic [Mal95], une interface graphique initialement développée pour le langage Self, puis portée pour Smalltalk avec Squeak, et maintenant disponible dans Pharo ; (ii) Bloc [Pla15], qui vise à être le successeur de Morphic en supportant plus de périphériques d'entrée, et en utilisant le rendu vectoriel pour supporter de hautes densités de pixels ; et (iii) Roassal [Deh13], un framework de visualisation avec une grande sélection de modèles et un langage dédié pour exprimer des visualisations interactives avec peu de code.

Notre plus grand défi a été la planification des animations. Pour satisfaire l'objectif représenté en figure 18, nous devions exécuter les mises à jour des animations avant le code de rendu de chacun des frameworks, idéalement sans avoir à les modifier explicitement. Morphic (i) fournit un callback, `World defer: <closure>`, pour exécuter un bloc de code avant la prochaine étape de rendu. Nous l'avons utilisé pour planifier nos mises à jour d'animations. En effet, Morphic est tellement lié au système que tous les autres frameworks en dépendent, ce qui permet à notre extension de fonctionner pour tous. Bien que Bloc (ii) fournisse un callback équivalent, `BlUniverse defer: <closure>`, nous ne voulions pas patcher notre système d'animation pour chaque nouveau framework, et avons compté exclusivement sur Morphic. Cela soulève le problème de l'indépendance réelle aux frameworks : *notre système n'est pas réellement indépendant*, il fonctionne avec d'autres frameworks car ceux-ci dépendent aussi de Morphic. Idéalement, cela ne devrait pas être le cas, et l'insertion de code avant la phase de rendu devrait être la responsabilité du *langage de programmation*, ou de sa *librairie standard*. Cela permettrait à des bibliothèques tierces comme la nôtre d'apporter des améliorations aux frameworks, sans en dépendre explicitement. Nous discutons de ce point en conclusion de cette section.

En utilisant l'opérateur de durée avec Morphic, nous avons pu animer des changements de position, de couleur de fond, de style de bordure, et de titre d'une fenêtre (voir figure 19). Nous avons également pu modifier ces mêmes attributs sur des boutons à l'intérieur de la fenêtre. Pour permettre une interpolation par défaut des chaînes de caractères comme sur la gauche de la figure 17, nous avons implémenté + comme concaténation de deux chaînes, et * comme extraction d'une fraction de la chaîne à partir de la gauche.

Cependant, en pratique certaines propriétés ne pouvaient pas être animées avec notre système. Par exemple, le texte de remplissage à l'intérieur d'un champ de saisie n'était pas modifiable une fois affiché (ses accesseurs étant conçus pour être utilisés uniquement avant insertion du widget dans l'arbre de scène). L'animation n'avait donc aucun effet, notre système ne se souciant que d'appeler des méthodes, sans savoir si elles fonctionnent.

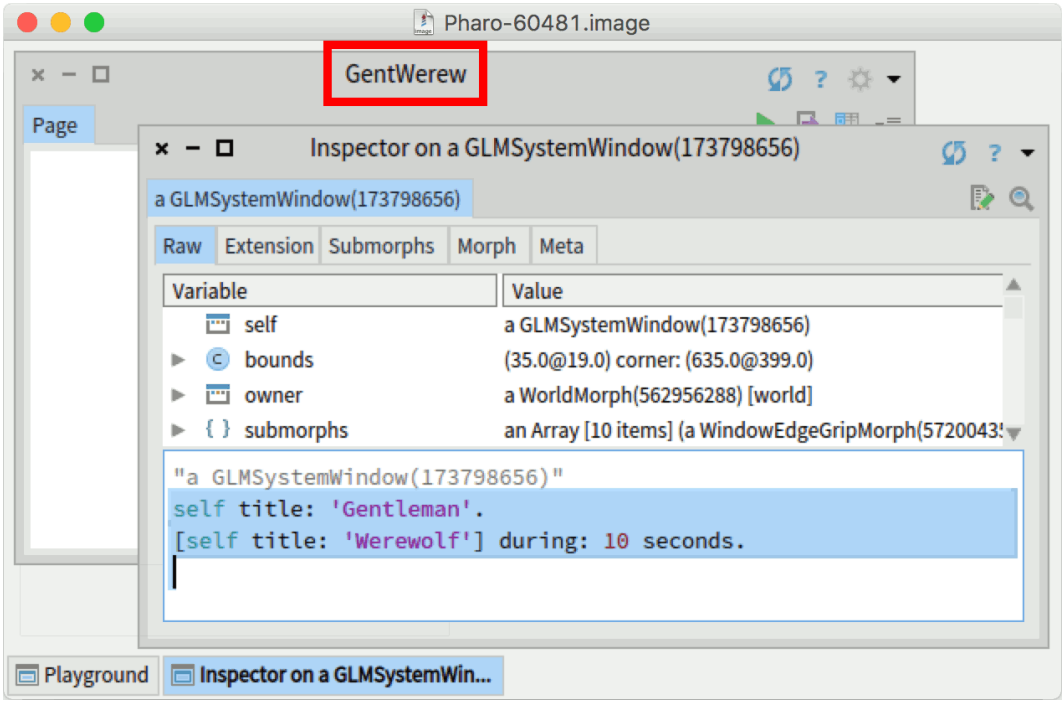


Figure 19 : Fenêtre d'introspection ouverte sur une fenêtre en arrière-plan. Le code surligné est en cours d'exécution et change le titre de la fenêtre ciblée, de `'Gentleman'` à `'Werewolf'`.

Notre système fonctionne aussi avec Bloc (ii) et Roassal (iii), bien que Roassal transgresse parfois la convention de nommage entre accesseurs et mutateurs. Pour la position, les widgets ont un mutateur `translateTo:`, et un accesseur `position`, dont les noms ne correspondent pas. Dans un tel cas, nous avons dû patcher Roassal avec un nouveau message `position:` renvoyant vers `translateTo:`. Pour la couleur, les messages `color` et `color:` correspondent, mais l'accesseur ne renvoie pas le même objet `Color` qui est passé au mutateur. Dans ce cas, nous avons dû corriger notre système pour toujours *convertir* les valeurs `start` et `target` au type renvoyé la première fois par l'accesseur. Ces problèmes montrent l'importance et l'utilité d'une convention de nommage globale au système : elle permet à notre système de fonctionner sans stocker une table de correspondance entre mutateurs et accesseurs.

Un problème plus sérieux dans Roassal était que la mise à jour d'un widget ne rafraîchissait pas automatiquement sa vue contenante pour l'affichage. Ainsi, toutes les animations étaient invisibles jusqu'à ce que nous déplacions ou redimensionnions la vue. Bien que nous considérons ce comportement comme un bug du framework, nous l'avons résolu en *rafraîchissant manuellement la vue* avec une syntaxe spécifique au framework. Cependant, avec plus de travail, nous corrigerions les widgets de Roassal, pour rafraîchir automatiquement les conteneurs dans chaque mutateur.

Ces problèmes révèlent que l'extensibilité des frameworks est influencée par l'existence et le respect de conventions communes. Lorsqu'un framework manque de cohérence vis à vis des conventions de la plateforme (nommage des méthodes, paramètres des accesseurs) et des pratiques répandues avec les autres frameworks (rafraîchissement automatique des vues), il est plus difficile à

adapter pour de nouveaux usages. La mise en évidence et le respect de ces conventions nous semblent donc essentielles pour supporter le développement de nouveaux frameworks, et ainsi améliorer la programmation de nouvelles interactions.

Pour finir, nous avons ajouté le support de l'opérateur de durée sur des groupes de messages :

```
[object position: 100@100. object color: Color blue]
    during: 2 seconds
```

Cet exemple renvoie deux objets de transition animée dans un tableau, que nous avons modifié pour transférer les messages de contrôle (`start`, `stop`, `pause`) à ses éléments. Pour implémenter correctement cette fonctionnalité, nous avons intégré dans Pharo la possibilité d'extraire tous les envois de messages d'une fermeture, en un tableau d'objets `MessageSend`. Notre système d'animation itère alors simplement sur ces envois de messages réifiés, et pour chacun initialise une animation.

L'objectif de ce travail étant d'illustrer une preuve de concept liée à l'interaction dans un langage de programmation, nous n'avons pas poursuivi l'exploration d'usages plus avancés des animations. Il aurait par exemple été envisageable d'étudier l'expression de “chorégraphies” d'animations multiples, à l'aide de relations de séquencement et simultanéité, qu'il aurait fallu intégrer de façon cohérente au langage. De plus, bien que nous suggérions l'intérêt de ce travail pour gérer les transitions de signaux non-visuels (ex. audio, haptique), nous nous sommes basés en pratique sur des *callbacks* liés à l'affichage (voir figure 18). L'extension à des signaux génériques est donc à réserver à de futurs travaux.

2.4.6 État de l'art

Pour ce travail, nous avons étudié les APIs d'animation de six frameworks d'interaction répandus. Qt [Qt19] est un framework d'interaction courant pour C++, disponible sur plateformes bureau et mobile. Core Animation [App06] est le framework standard d'Apple recommandé pour les plateformes OSX et iOS. JavaFX 8 [Ora08] est le successeur officiel de Swing pour Java, et Android [Goo08] est un autre framework Java courant pour les téléphones du même nom. Nous avons également choisi D3.js [Bos11] pour sa popularité en tant qu'outil de visualisation web, et GSAP [Gre14] pour son choix le plus flexible de propriétés animables. Nous nous sommes intéressés à la *flexibilité* et la *compacité* de chacun de ces frameworks pour exprimer les animations. Notre but était de relever leurs différentes techniques pour animer des propriétés sans avoir recours à des fonctions statiques.

La comparaison est présentée dans le tableau 4. Les dimensions considérées étaient :

- *propriétés animables* — les propriétés supportées, en soulignant les techniques utilisées pour en supporter de nouvelles ;
- *types animables* — les types supportés, avec les techniques utilisées pour en supporter d'autres;
- *exemple de syntaxe* — des extraits de code implémentant un exemple minimal d'animation de `property` à la valeur `target` pendant 2 secondes.

Framework	Propriétés animables	Types animables	Exemple de syntaxe
Opérateur de durée (Smalltalk)	toute propriété avec un mutateur et un accesseur (*qui doivent avoir le même nom*)	types supportant les opérations `*` et `+` (*les autres peuvent fournir des interpolateurs*)	`[object property: target] during: 2 seconds`

Qt 5 (C++)	toute propriété fournissant un mutateur (*l'objet propriétaire doit hériter de `QObject`*)	entiers, réels, `QLine`, `QPoint`, `QSize`, `QRect`, et `QColor` (les types additionnels doivent être supportés par `QMetaType`)	`QPropertyAnimation a(object, "property");`
			`a.setDuration(2000);`
			`a.setEndValue(target);`
			`a.start();`

Core Animation (Swift)	29 propriétés par défaut (*les objets doivent hériter de `CALayer` et les nouvelles propriétés doivent offrir un accesseur et un mutateur*)	entiers, réels, `CGRect`, `CGPoint`, `CGSize`, `CGAffineTransform`, `CATransform3D`, `CGColor`, et `CGImage` (*pas de support d'autres types animables*)	`let a = CABasicAnimation(keyPath:"property")`
			`a.toValue = target`
			`a.duration = 2.0f`
			`object.addAnimation(a, forKey:"property")`

D3.js (JavaScript	propriétés `attr` et `style` des éléments du DOM (*limité aux objets du DOM*)	nombres, couleurs (divers espaces), dates, nombres dans du texte, tableaux, dictionnaires, transformations 2D (*d'autres types sont animables en fournissant des interpolateurs*)	`object.transition()`
			`  .duration(2000)`
			`  .attr("property", target);`

JavaFX 8 (Java)	toutes les propriétés implémentant l'interface `WritableValue<T>` (*les objets ont seulement besoin de stocker ces propriétés*)	entiers, réels, `Color` (les types additionnels doivent implémenter l'interface `Interpolatable`)	`Timeline t = new Timeline();`
			`t.getKeyFrames().add(`
			`  new KeyFrame(Duration.seconds(2),`
			`  new KeyValue(object.property(), target)));`
			`t.play();`

Android Property Animation (Java)	toute propriété avec un mutateur `set<Propriete>()` (*pas de restriction sur l'objet conteneur*)	entiers, réels, couleurs (*d'autres types sont animables en fournissant des interpolateurs*)	`ObjectAnimator a = ObjectAnimator`
			`  .ofInt(object, "property", target);`
			`a.setDuration(2000);`
			`a.start();`

GSAP (JavaScript)	toute propriété exceptés un ensemble de noms réservés pour passer des options (*pas de restrictions sur les objets*)	nombres, nombres dans du texte (*d'autres types sont animables en fournissant des interpolateurs*)	`TweenLite.to(object, 2, {property: target});`

Tableau 4 : Comparaison avec les fonctionnalités d'animation de six autres frameworks

Nous observons trois types de restrictions sur les propriétés animables : la nécessité d'hériter d'une classe ou une interface en particulier (Qt 5, Core Animation, JavaFX 8), une convention de nommage pour les mutateurs des propriétés (Android), et la réservation de mots-clés spéciaux (GSAP). Pour les types, nous identifions deux groupes de frameworks : ceux avec un ensemble fixe (Qt 5, Animation de base), et ceux nécessitant une fonction d'interpolation personnalisée. Elle est souvent fournie sous la forme d'un *objet functor*, qui est un objet avec une unique méthode. De plus, D3 se distingue par l'interpolation automatique des champs d'objets inconnus, grâce à la capacité de JavaScript à les énumérer.

Quant à la syntaxe, certains frameworks ont d'autres variantes, les plus complètes étant présentées ici. Presque tous construisent des objets d'animation à partir d'une chaîne de caractères `"property"`. Pour GSAP, `TweenLite` récupère le nom de la propriété en énumérant les champs de son 3e argument. La démarche de D3 est d'intercaler `.transition()` entre le mutateur et l'objet receveur, un objet proxy interceptant les appels de fonction `style` et `attr` pour les animer automatiquement. Avec le concept d'animation de fonction, nous parvenons à une syntaxe aussi concise que celle de GSAP, avec comme unique restriction celle que le framework doive spécifier et respecter une convention de nommage. L'inconvénient majeur de notre approche est cependant que nous dépendons de fonctionnalités avancées du langage de programmation. Plutôt que de simples appels de fonctions, notre syntaxe repose sur un langage dynamique, avec de fortes capacités d'introspection et de polymorphisme.

La finalité de ce travail était de fournir un concept que les développeurs puissent s'*approprier*, afin de favoriser l'utilisation des animations lorsqu'elles sont appropriées, et de susciter des solutions originales détournant leur usage pour répondre à d'autres problèmes. Cette hypothèse est difficile à confronter, car le sentiment d'appropriation d'un concept est difficile à mesurer objectivement. Nous nous sommes donc limités à cette preuve de concept, qui constitue une brique à assembler dans un éventuel langage futur *adapté* à l'interaction.

2.4.7 Conclusion et formulation d'un Essentiel d'Interaction

En Programmation Orientée Objet, les objets encapsulent leur propre état et l'exposent avec des interfaces. S'ils possèdent les coordonnées d'une position, alors ils exposeront certainement des fonctions `getPosition()` et `setPosition(Point)`. L'animation par appels dynamiques utilise ce principe, en obtenant une valeur `start` avec la première fonction, et en envoyant de petites variations entre `start` et `target` à la seconde. Les objets n'ont pas besoin de savoir *comment* animer une position : avec une série de petites modifications nous simulons un mouvement fluide vers la valeur finale.

À partir de cette observation, nous avons exploré le concept d'animation pour *tous* les objets du système, indépendamment d'un framework. De plus, nous avons créé une syntaxe réutilisant autant que possible les éléments lexicaux existants. Le résultat est un opérateur de durée rattaché aux appels de fonction :

```
object.setProperty(target) during 2s
```

Nous avons présenté un processus et une implémentation fonctionnelle pour produire un objet de transition animée à partir de cette syntaxe. Ce travail donne lieu à deux recommandations pour les systèmes interactifs : (i) *fournir un mécanisme de notification de l'affichage global au système*, et (ii) *encourager les conventions de nommage pour les accesseurs et mutateurs*. À partir de ces points, et de notre expérience dans l'exploration de syntaxes alternatives pour exprimer de l'interaction, nous formulons comme dernier Essentiel d'Interaction d'**assurer la compatibilité et l'extensibilité des outils (langages et bibliothèques) pour la conception et le prototypage d'interactions**. Nous discutons de ce point dans les parties qui suivent.

2.4.7.1 Utilisation de la métaprogrammation

La métaprogrammation désigne l'utilisation de structures de données qui décrivent des programmes. Il peut s'agir de générer du code qui soit ensuite interprété ou compilé, de manipuler le comportement des objets avec un protocole à méta-objets (*Meta-Object Protocol*), ou encore d'utiliser les macros de certains langages (ex. préprocesseur du C). Dans notre travail, la métaprogrammation nous a permis d'intégrer l'expression des animations de façon cohérente avec le reste du langage, comme si elle faisait partie du langage de base. En pratique, il s'agissait de récupérer le *bytecode* d'un bloc de code compilé, de le parcourir à l'aide des fonctions de la plateforme, et de générer un objet `MessageSend` pour chaque envoi de message énuméré. Dans une première version de ce travail, nous exprimions les animations avec la syntaxe `object property: (target during: 2 seconds)`. Nous utilisions alors aussi la métaprogrammation, en invoquant le Debugger intégré lors de l'exécution de `during:`, pour récupérer l'appel à `property:` sur la pile d'envois de messages.

Sans l'utilisation de métaprogrammation, notre syntaxe pour exprimer les animations aurait ressemblé à `object animate: 'property' to: target during: 2 seconds`. Or cette syntaxe n'est pas cohérente avec l'appel de méthode sans animation, ce qui nous avait motivé à étendre la syntaxe du langage. Ainsi, dans le cadre de la recherche, la métaprogrammation permet de *modifier le sens des éléments de syntaxe existants*, afin de proposer de nouvelles formes d'expression cohérentes. Dans notre cas, la syntaxe `[object property: target] during: 2 seconds` aurait dû signifier l'exécution immédiate ou retardée du bloc de code, ce que nous avons changé en sa conversion vers un objet de transition animée.

Plus généralement, nos besoins en métaprogrammation ont été de *décorréler la syntaxe et la sémantique du code*, c'est-à-dire de pouvoir inspecter le code comme une structure de données (pas seulement une chaîne de caractères), et de *remplacer certaines règles sémantiques* en conservant les autres. De nombreux langages dynamiques aujourd'hui (ex. Smalltalk, JavaScript, Python, Lua) définissent un protocole à méta-objets, c'est-à-dire qu'il est possible d'intercepter et de changer le sens d'expressions comme `object.property = value`, ou `object.func()`. Cependant, dans notre cas c'est l'inspection de blocs de code qui aurait été utile, et celle-ci est encore mal supportée dans la majorité des langages.

En effet, la métaprogrammation expose souvent des structures internes au langage, qui sont documentées *par* les développeurs du langage, et *pour* leur propre usage. Elles ne sont donc pas toujours accessibles pour des utilisateurs tiers, d'autant qu'ils ne sont pas au fait des éventuels effets de bord. En outre, certains détails internes peuvent se complexifier avec les évolutions du langage, rendant l'utilisation de la métaprogrammation plus difficile. C'est le cas par exemple avec l'introduction de nouveaux types de fonctions (génératrices, asynchrones). Ces nouvelles fonctionnalités ont apporté de nouveaux types primitifs dans les langages JavaScript et Python, avec pour conséquence qu'une fonction peut être représentée en interne par plusieurs types. Pour notre travail, nous avons eu la chance de travailler avec un développeur de la machine virtuelle Sista pour Smalltalk, Clément Béra [Ber17]. Il nous a permis d'implémenter notre extension en intégration parfaite avec la plateforme, en particulier pour identifier tous les types de *bytecodes* qui traduisaient des envois de messages. Étant donné que cette opportunité est rare, il est plus probable à l'avenir que nous devions nous contenter des ressources disponibles dans le langage. Nous considérons donc qu'il

est essentiel d'**utiliser un langage de programmation à la métaprogrammation la plus spécifiée et accessible possible**. Smalltalk se distingue favorablement des autres langages sur ce point, et notre travail sur l'expression des animations aurait été plus difficile (voire impossible) avec la plupart des autres langages de programmation.

2.4.7.2 Interaction avec le bas-niveau

Durant notre utilisation de la plateforme Pharo Smalltalk, nous avons conçu d'autres prototypes liés à la programmation d'interactions. Le premier consistait en une syntaxe de causalité `[<message>] afterDo: <bloc>`. Il se basait sur les travaux réalisés pour les animations, et permettait d'exécuter un bloc de code systématiquement après qu'un envoi de message soit effectué, à la manière de la programmation par Aspects [Kic97], et des *bindings* de Djnn [Mag17]. Nous avons démontré ce prototype conjointement aux animations lors de la conférence Pharo Days 2017 [Raf17]. Le second prototype était une version préliminaire aux travaux présentés dans le chapitre suivant, et consistait en une boîte à outils rudimentaire pour dessiner des formes graphiques à l'écran. Nous l'avons présenté lors de la conférence ESUG 2016 [Raf16].

Pour l'ensemble des prototypes développés avec Pharo, nous avons souvent travaillé étroitement avec le bas-niveau. En pratique, nous souhaitions accéder aux évènements de la souris et du clavier avec une très faible latence, donc avec le minimum d'intermédiaires, et surtout pour être certains qu'aucune file d'attente n'était utilisée. De plus, nous avions besoin de dessiner des formes géométriques simples et du texte dans une fenêtre, le plus rapidement possible. Or sur cette plateforme le framework natif Morphic [Mal95] était notablement lent, au point de fonctionner à une fréquence très inférieure à celle de l'écran. Quant à la récupération des évènements d'entrée, elle se basait sur une ancienne version de SDL [Lan98], et ne gérait pas le défilement continu avec les trackpads.

Nous avons donc entrepris d'utiliser les bibliothèques OpenGL, FreeType, GLFW (alternative à SDL) et libpointing [Cas11] directement avec Pharo, car nous les avions déjà utilisées par le passé. Pour accéder à ces bibliothèques, la machine virtuelle doit faire appel à des fonctions dans des bibliothèques partagées (fichiers `.dll` sous windows, `.so` sous Linux, `.dylib` sous macOS). Il faut alors écrire des *bindings*, qui permettent à du code Smalltalk d'exécuter ces fonctions. Par exemple, la fonction native `glfwCreateWindow` se transpose en la méthode Smalltalk suivante :

```
createWindow: title width: width height: height monitor: monitor share: window
    ^self ffiCall: #(GLFWwindow glfwCreateWindow(int width, int height,
        String title, GLFWmonitor monitor, GLFWwindow window))
```

Nous avons conçu des scripts permettant de générer automatiquement les bindings pour Smalltalk à partir des définitions de fonctions pour le langage C de chacune des bibliothèques. Une fois les bindings réalisés, nous pouvions ouvrir une fenêtre supplémentaire avec GLFW, et dessiner directement dedans avec OpenGL et FreeType. Or à cause de l'intégration étroite de Morphic dans Pharo, notre fenêtre devrait cohabiter avec une fenêtre principale. Comme le système d'exploitation

envoyait un flux unique d'évènements pour les deux fenêtres, l'exécution de notre système *asphyxiait* la fenêtre principale qui ne recevait plus d'évènements d'entrée. Nous avons donc pu expérimenter à l'aide de Pharo, mais au prix de *beaucoup* d'efforts à bas niveau.

Dans le cadre de notre travail, un inconvénient majeur de la plateforme Pharo était son support insuffisant de l'interaction à bas-niveau, et qu'elle ne destine pas ce support à un autre usage qu'interne à la plateforme. Les bindings obsolètes nécessitaient un travail d'ingénierie conséquent pour utiliser des bibliothèques plus récentes. Nous y avons donc investi beaucoup de temps et d'apprentissage, que nous avons moins investis dans de la recherche de haut niveau. De plus, nous avons trouvé un faible intérêt dans la communauté pour des contributions à l'interaction à bas-niveau, probablement car nous améliorions l'existant plutôt que d'apporter des fonctionnalités majeures. L'exploration de concepts de haut-niveau nous ayant amenés à dépendre de fonctionnalités avancées de bas-niveau, il nous semble essentiel aujourd'hui d'**utiliser un langage de programmation avec un support de l'interaction à bas-niveau le plus complet possible**, pour qu'on n'ait pas à s'en occuper. C'est le cas par exemple avec Java, qui fournit par l'intermédiaire du framework Swing des fonctionnalités avancées, comme par exemple le contrôle explicite de la souris avec la classe `Robot`.

2.4.7.3 Compatibilité et l'extensibilité des outils de programmation d'interactions

À la lumière des discussions précédentes, nous énonçons donc un Essentiel d'Interaction avec les caractéristiques suivantes :

- fournit un mécanisme de notification de l'affichage global au système
- encourage les conventions de nommage pour les accesseurs et mutateurs des bibliothèques logicielles
- favorise un langage de programmation avec une métaprogrammation très spécifiée et accessible
- favorise un langage de programmation avec un support complet du bas niveau

Ces caractéristiques faciliteraient d'abord le développement d'applications interactives grâce à l'apprentissage moindre des règles spécifiques à chaque framework. Avec des conventions communes, la compatibilité entre bibliothèques serait facilitée, et les programmeurs pourraient réutiliser leurs connaissances entre toutes. Ensuite, les développeurs auraient moins d'efforts à faire pour accéder à des fonctionnalités de bas niveau, si celles-ci étaient d'emblée incluses dans le langage. Enfin, nous conjecturons que la possibilité de modifier le sens des éléments du langage et d'en ajouter de nouveaux permettrait de contrôler plus finement l'*expérience utilisateur* du développement de nouvelles interactions (par rapport à l'utilisation des fonctions d'une API). La finalité serait ici de s'affranchir des concepts de programmation hérités du calcul, pour imaginer une expérience de la programmation dédiée à l'interaction. Nous en sommes encore loin aujourd'hui, car la création d'un langage de programmation est une tâche très lourde, qui nous oblige à modifier des langages existants plutôt que travailler à partir d'une copie vierge. Nous espérons cependant que l'accumulation de concepts de programmation dédiés à l'interaction facilitera la création à terme d'un tel langage. Notre troisième Essentiel d'Interaction est donc :

Essentiel d'Interaction n°3

Favoriser la compatibilité et l'extensibilité des bibliothèques logicielles, à l'aide de mécanismes et conventions standards à bas niveau, et d'un langage à la syntaxe flexible

Chapitre 3. La boîte à outils Polyphony

Dans les chapitres précédents, nous avons discuté de l'adéquation des outils actuels avec les besoins émis par les programmeurs de nouvelles techniques d'interaction, et constaté que la documentation et la flexibilité des frameworks étaient cruciales pour favoriser leur utilisation dans des contextes non prioritaires. Nous souhaitions aussi expérimenter l'utilisation d'une technologie du Web (ici le langage JavaScript) dans la conception d'une architecture d'interaction, pour permettre un éventuel transfert de connaissances vers le Web. Nous avons donc choisi de contribuer avec la création d'un nouveau framework, *Polyphony*, conçu pour les interfaces ad hoc (simples, peu robustes, et créées pour un usage précis) et flexibles (dont le code et l'apparence sont simples à modifier).

Ce travail illustre nos Essentiels d'Interaction avec un *modèle de programmation original*, dans lequel tous les traitements de l'interface (leur ordre, leurs déclencheurs) sont clairement explicités dans le programme, et manipulables. Il est possible de les observer, les réordonner, les remplacer, et d'en insérer de nouveaux, afin de modifier le comportement d'une interface existante. Nous supportons ainsi une pratique de développement incrémentale, qui permet d'incorporer et de raffiner les besoins à mesure qu'ils émergent dans un projet. Nous présentons aussi une *réification des dispositifs d'interaction en entités programmables et extensibles*, dont l'introspection est facilitée (pour en découvrir les variables disponibles), et les variables sont dynamiques (pour y accumuler des données de plus haut niveau). Grâce à un mécanisme de variables temporaires (existant durant un court laps de temps), nous fournissons une alternative à la programmation évènementielle, en utilisant les entités réifiant les périphériques comme supports de données des techniques d'interaction. Cette représentation nous permet en retour d'intégrer les traitements des techniques d'interaction dans ceux de l'interface, et ainsi d'*unifier le fil d'interface avec le fil d'interaction*. Nous parvenons ainsi à une représentation simple du flux global d'exécution, qu'il est possible de représenter graphiquement dans son intégralité, ce qui en facilite la compréhension. Enfin, l'expérimentation d'un langage dynamique comme JavaScript consiste à tirer parti du caractère dynamique des objets (leurs variables ne sont pas contraintes à la définition d'une classe). Nous utilisons cette fonctionnalité pour symboliser l'ajout de comportement aux éléments de l'interface, et ainsi constituer de nouveaux éléments par Composition plutôt d'Héritage. Ce travail consiste donc à *traduire la dynamicité du langage en dynamicité du framework*, ce qui nous amène à proposer de nouveaux besoins et évolutions pour ce langage.

Au cours de nous recherches bibliographiques et technologiques, nous avons considéré l'étude et l'application du modèle Entité-Composant-Système (ECS) à la programmation d'interfaces graphiques et d'interactions. Ce modèle architectural est issu du domaine du jeu vidéo, y est utilisé depuis plus d'une décennie, et n'a à notre connaissance pas été utilisé pour modéliser des inferfaces. Il nous a semblé être pertinent pour le prototypage de nouvelles techniques d'interaction. En effet, ECS est basé sur le principe de Composition, qui est un besoin récurrent en IHM, pour lequel de nombreuses contributions ont été proposées, comme les Web Components [Coo14], UBit [Lec03], ou encore Jazz [Bed04]. De plus, ECS accorde une priorité importante à la modélisation du flux d'exécution, qui est conçu comme un *pipeline* — modèle commun dans les jeux vidéo, qui cherchent à maximiser

l'utilisation des ressources de la machine. Il fournit donc une solution au problème de fragmentation de la logique énoncé en section 2.2.4. Enfin, ECS a été appliqué avec succès dans des jeux commerciaux tels que Thief: The Dark Project [Leo99], Operation Flashpoint 2 [Mar07], ou encore Overwatch [For17]. La communauté de développeurs autour de ce modèle est très active aujourd'hui, autant comme utilisateurs de bibliothèques basées sur ECS, que comme développeurs de ces bibliothèques. Nous avons identifié en amont de notre travail sur ECS un besoin, exprimé principalement sur des forums de discussions en ligne, de voir ce modèle appliqué à la construction des interfaces dans les jeux. Ces conditions nous ont donc incités à adapter le modèle ECS à la programmation d'interactions dans un contexte de recherche.

Dans ce chapitre, nous présentons la première étape de notre travail, qui a été d'appliquer ECS à la programmation d'IHM, d'en fournir une architecture claire et non-ambiguë, et d'en dégager les contributions principales à la programmation d'interfaces graphiques et d'interactions. La première section de ce chapitre présente un état de l'art sur les bibliothèques logicielles dédiées au prototypage et à la programmation de nouvelles techniques d'interaction. Dans la section suivante, nous présentons Polyphony, une boîte à outils basée sur ECS, et introduisons son utilisation du point de vue des programmeurs d'applications. Ensuite, nous détaillons l'architecture de Polyphony, à des fins de réplicabilité. Enfin nous énumérons les choix d'implémentation caractéristiques de toute variante d'ECS, et expliquons ceux de Polyphony.

3.1 État de l'art des outils de programmation pour la recherche de nouvelles IHM

Cette section est dédiée à la présentation de l'état de l'art et au positionnement de notre travail sur la boîte à outils Polyphony. Nous ciblons en particulier le contexte de la programmation de techniques d'interaction, thème central de ce travail de thèse, bien que nous considérons que Polyphony pourrait servir dans de nombreux contextes d'applications. Dans le chapitre 1, nous avons discuté des problèmes rencontrés par les chercheurs en IHM, dans l'utilisation de frameworks d'interaction. Ils relatent principalement que leurs usages ne sont pas les cibles prioritaires des développeurs de frameworks, et que leurs besoins sont mal reconnus et supportés. Ils manquent ainsi de documentations et d'exemples pertinents pour illustrer l'utilisation des frameworks dans un contexte de recherche. Les frameworks majeurs rendent difficile la création d'interfaces non stéréotypées, à la fois en termes d'apparence et d'interactivité. À cet effet, nous avons observé que les chercheurs exprimaient le besoin de s'approprier leurs outils, voire de les détourner pour créer des artefacts pour lesquels ils n'étaient pas prévus. Enfin, pour intégrer des contributions à des interfaces existantes, comme de nouvelles techniques d'interaction, les chercheurs se sont heurtés à un manque de flexibilité des frameworks. L'état de l'art présenté ici doit nous permettre de relever les solutions qui ont été proposées en réponse aux problèmes ci-dessus. Nous avons analysé les travaux selon les questions suivantes :

- Quels nouveaux usages ont-ils introduits/permis ?
- Quels concepts et nouvelles manières de programmer ont-ils proposés ?
- À quel point sont-ils flexibles, dans le changement de comportements prédéfinis ?
- Comment s'intègrent-ils avec leur langage de programmation ou bibliothèque d'interaction ?

La recherche en IHM a conduit à un grand nombre de boîtes à outils, couvrant de nombreux usages. Nous n'avons pas la prétention de les énumérer toutes ici, d'autant que notre objet d'étude peut amener à en considérer beaucoup. Les lecteurs intéressés par un état de l'art récent et très complet peuvent se référer au manuscrit de thèse de Vincent Lecrubier [Lec16]. Nous présentons ici les travaux les plus significatifs, qui nous permettent de situer les apports de Polyphony à la programmation de techniques d'interaction.

3.1.1 djnn et Smala

djnn est un framework complet de construction d'interfaces graphiques, développé au LII de l'ENAC depuis 2014 [Mag14, Cha15, Rey15, Cha16]. Son contexte d'application initial est la spécification d'interfaces dans le domaine de l'aéronautique, pour lequel la vérification et la certification des programmes sont courants et nécessaires. De plus, il est caractérisé par des équipes pluridisciplinaires, dans lesquelles il est courant que l'apparence des interfaces, leur logique, et leurs techniques d'interaction soient développés par des équipes distinctes. djnn s'est construit autour d'une conception itérative des interfaces, à l'aide d'un paradigme de développement rigoureux : chaque programme est un arbre de composants interactifs, dont les interactions définissent l'exécution du programme. *Tous* les éléments interactifs sont modélisés par ces composants (les widgets, les périphériques d'entrée/sortie, ou encore les formules de calcul), et sont inclus dans une hiérarchie de composants. La représentation au format XML de cet arbre est utilisée comme interface entre les différentes équipes.

djnn se base sur le modèle théorique I* développé par Stéphane Chatty quelques années auparavant [Cha07, Cha08, Cha12]. Ce modèle est centré autour de la définition de *processus*, et des relations de causalité entre eux. Il définit le concept de *binding*, qui relie un processus source à un processus dépendant, en propageant toute activation de l'un à l'autre. Ce concept est repris et développé dans Smala, un langage de programmation basé sur le framework djnn, et transpilé vers C [Mag18]. Il introduit deux opérateurs de causalité, le couplage de processus et le lien *data-flow* (voir figure 20), supportés par une syntaxe concise. De plus, il introduit une manière d'exprimer les interfaces à mi-chemin entre exécution immédiate de code, et alternance de modèles déclaratifs.

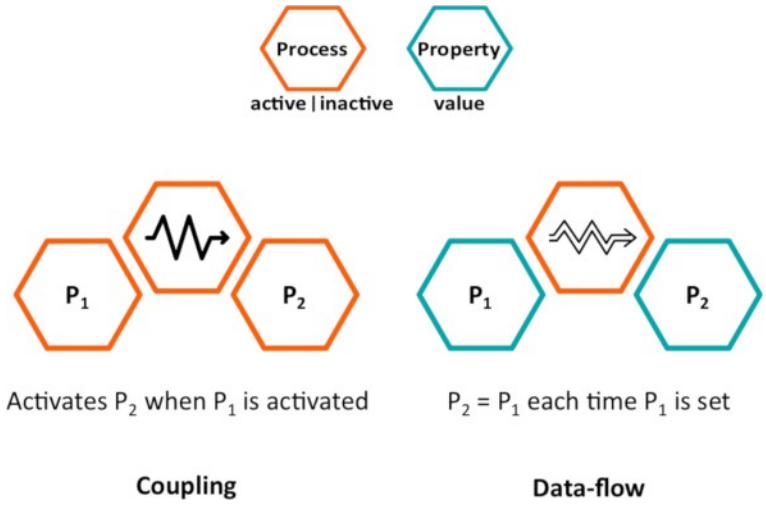


Figure 20 : Les opérateurs de causalité de djnn/Smala (figure extraite de [Mag18]).

djnn fournit un ensemble réduit de concepts unificateurs, avec lesquels ses auteurs parviennent à exprimer des interfaces complexes. Ce framework invite les programmeurs à raisonner sur une interface comme un ensemble de processus interconnectés, et provoquant l'activation ou la désactivation des nœuds du graphe de scène. Par la définition de relations de causalité, il est facile de propager des changements d'état, ce qui est d'autant plus facilité par la syntaxe de Smala. La distinction ainsi que la relation étroite entre djnn et Smala illustre parfaitement notre idée d'intégration entre framework et langage. Ici, djnn est le framework, auquel on accède par une API en C. Smala est le langage (ici spécialement conçu pour djnn), dont l'intérêt est d'être une syntaxe pour djnn. Le travail sur Smala a permis d'associer des éléments de syntaxe concis et clairs à des concepts autrefois exprimés par des appels de fonctions (en particulier les bindings). Cependant, bien que le langage Smala soit particulièrement adapté pour définir une nouvelle interface, il ne démontre pas la modification d'une interface existante. Ce type de flexibilité fait partie des besoins liés au prototypage de nouvelles techniques d'interaction, et nécessite l'introspection de tous les éléments de l'interface. Or il est à craindre que modéliser toutes les relations de causalité de l'interface en objets de bindings, en génère un grand nombre, et rende difficile leur introspection.

3.1.2 UBit

UBit (*Ubiquitous Brick Interaction Toolkit*) est une boîte à outils basée sur le langage C++, implémentant une architecture “moléculaire” pour construire des IHMs [Lec99, Lec03]. Elle a été conçue à partir d'un modèle simple offrant beaucoup de flexibilité, afin de supporter le développement de nouvelles techniques d'interaction, dans des contextes variés comme l'interaction distribuée ou à plusieurs utilisateurs. Ce modèle répond au caractère “monolithique” des interfaces basées sur des classes de widgets, pour lesquelles les comportements de chaque widget sont prédéfinis dans sa classe.

UBit utilise un arbre de scène, dans lequel les widgets de l'interface composent leur apparence et leur interactivité à l'aide de *briques*, de petites unités sémantiques agissant comme les mots d'une phrase (voir figure 21). Elle utilise avantageusement les relations de fratrie dans l'arbre de scène, souvent peu mises en valeur, et permet aussi la “réplication récursive” par laquelle un nœud possède plusieurs parents pour lesquels il est dupliqué. De plus, UBit s'intègre étroitement avec le C++ en redéfinissant les opérateurs arithmétiques + et /, qui lui permettent d'initialiser un arbre de scène de manière récursive, sans avoir à construire les nœuds dans des variables intermédiaires.

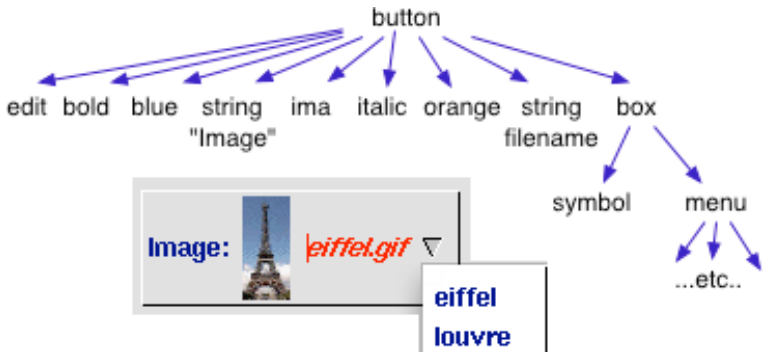


Figure 21 : Illustration d'un assemblage de briques dans UBit (figure extraite de [Lec03]).

Cependant, la granularité fine du concept de briques “atomiques” fait qu'en contrepartie toute interface de taille modeste contient un grand nombre de briques. Il peut être alors difficile de représenter visuellement le graphe de scène, et de le manipuler directement, par exemple pour étendre une application existante ou en modifier des parties. De plus, UBit ayant été conçue pour reproduire des interfaces WIMP, le choix des différents types de briques et leurs combinaisons possibles ont été conçus pour reproduire des contrôles standards (boutons, cases à cocher, menus, etc.). L'ajout d'un nouveau type de brique doit prendre en compte les interactions possibles avec les briques existantes, ce qui rend difficile l'extension d'UBit à de nouveaux types de contrôles. Dans le cas de nouveaux types d'interfaces (ex. Réalité Virtuelle, interfaces ubiquitaires), il faudrait potentiellement reconcevoir l'ensemble des briques.

3.1.3 Amulet/Garnet

Amulet est un framework de construction d'interfaces graphiques et d'interactions basé sur le langage C++, qui a fait figure de pionnier dans l'innovation en programmation d'IHMs [Mye97]. Développé sur plus d'une décennie, il se caractérisait par des widgets basés sur un modèle d'objets à prototypes, l'intégration d'un système de contraintes pouvant s'attacher à toutes les variables, la gestion des entrées utilisateur par des objets “*Interactors*” réutilisables, et des possibilités avancées d'inspection et de modification de l'interface à l'exécution. Ces caractéristiques sont encore aujourd'hui en avance sur les frameworks courants de programmation d'IHM. Amulet est le successeur de Garnet (illustré en figure 22), un framework et environnement de développement basé sur Common Lisp et X11 [Mye90].

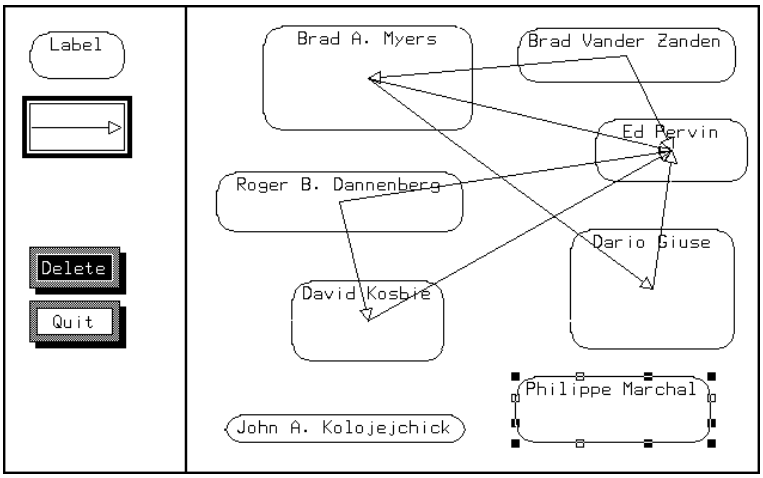


Figure 22 : Exemple d'interface construite avec Garnet (figure extraite de [Mye90])

Amulet a été spécifiquement développé pour soutenir la recherche sur les interfaces graphiques. La flexibilité et l'extensibilité étaient donc des thèmes centraux, intégrés dans son modèle innovant de construction des interfaces. Il a servi à explorer des sujets tels que l'interaction distribuée sur le réseau Internet, le support et l'expression des animations, le support de l'action *undo*, le déboguage graphique des applications interactives, la création d'un outil graphique de définition d'interface (Gilt), l'instanciation de widgets par le dessin à main levée (SILK), et la programmation par démonstration (Gamut). Le modèle à prototypes d'Amulet consiste pour chaque objet à avoir un parent (son

prototype), qui implémente les méthodes de l'objet (tout comme une classe le ferait). Cependant, à la différence des classes il est possible d'ajouter ou supprimer des champs de chaque objet (indépendamment des autres), et de changer de prototype. On peut ainsi considérer la programmation orientée prototype comme une alternative aux classes plus flexible. Amulet est un des rares travaux ayant exploré l'usage d'un langage dynamique (au sens moderne, nous entendons le typage et la topologie des objets), pour la construction d'interfaces graphiques. Le modèle à prototypes était implémenté en C++ à l'aide de fonctions, sans faire usage de fonctionnalités de métaprogrammation, ce qui explique qu'il ait souffert d'une mauvaise performance.

Aujourd'hui, certaines des fonctionnalités innovantes qu'Amulet a contribué à développer ont acquis une diffusion plus large (gestion des contraintes de Cassowary [Bad01], introspection des interfaces avec les navigateurs Web), tandis que les autres sont restées du domaine de la recherche. Il est à envisager que l'essor de langages dynamiques comme Python et JavaScript permette le développement de frameworks exploitant les objets à prototypes, dont nous nous sommes également inspirés dans ce travail de thèse.

3.1.4 SwingStates et HsmTk

SwingStates est une boîte à outils basée sur le langage Java, s'intégrant avec le framework natif Swing pour y appliquer le formalisme des machines à états [App06]. Elle a été conçue pour structurer le flux de contrôle des applications graphiques, dont la complexité est depuis longtemps reconnue comme problématique, pour la maintenance et l'introspection des interfaces [Mye91]. Elle est notable aussi pour le développement des notions de *tags*, inspirés de Tcl/Tk, et popularisés par les classes de CSS1 [W3C96]. Les machines à état (ou *automates finis*) sont un modèle mathématique permettant de décrire le fonctionnement d'un système comme un enchaînement d'états, et l'exécution de transitions entre ces états. Appliqué à la programmation d'IHMs, elles permettent de décrire des techniques d'interaction de façon rigoureuse et systématique — en particulier pour repérer et clarifier les interactions inattendues (ex. appui simultané des deux boutons sur la figure 23).

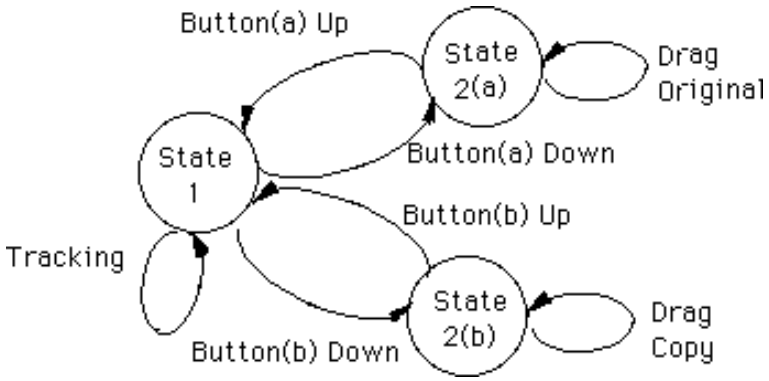


Figure 23 : Illustration d'une technique d'interaction à deux boutons modélisée par une machine à états (figure extraite de [Bux90]).

HsmTk est une autre boîte à outils, basée sur le C++, et appliquant cette fois le formalisme des machines à états hiérarchiques [Bla06] (voir figure 24). Elle s'intègre à la fois comme une extension du langage C++ (pour définir la logique des automates), et comme une extension du format SVG (pour

l'intégration de la logique à la spécification de l'interface). HsmTk est notable pour chercher à s'extraire du modèle de widget, réutilisant des technologies existantes, éprouvées, et bien optimisées (SVG et la syntaxe du C++), pour concevoir des interfaces graphiques innovantes.

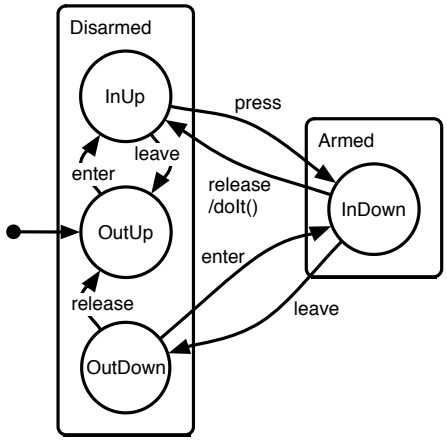


Figure 24 : Exemple d'une machine à états hiérarchique (figure extraite de [Bla06]).

SwingStates et HsmTk ont comme points communs d'avoir été toutes deux développées — indépendamment — au LRI (Université Paris-Sud), et d'appliquer le principe d'*interaction comme objet de première classe* de Beaudoin-Lafon [Bea04]. Les machines à état sont un paradigme robuste et éprouvé pour modéliser des techniques d'interaction, cependant elles ont rarement été utilisées pour du prototypage — y compris pour la recherche académique. En effet les automates s'expriment au mieux par des diagrammes visuels, et se transposent en texte souvent de façon verbeuse. De plus, il sont assez vite limités dans la complexité des interactions qu'ils peuvent modéliser, en particulier lorsqu'il faut considérer la combinaison de plusieurs états, les transitions continues entre états (animations), ou l'expression de *feedback* et *feedforward*. Ces besoins impliquent de rajouter des états, et augmentent donc la complexité de façon exponentielle. Pour ces mêmes raisons, les formalismes à états sont généralement très limités dans leurs capacités à faire évoluer le nombre d'états et de transitions. Les paradigmes à états sont cependant très utilisés dans les domaines où l'interaction doit être formellement spécifiée et déterministe, comme les systèmes critiques, avec ICO/Petshop qui est basé sur des réseaux de Petri [Nav09]. Enfin, InterState est notable pour avoir fourni une représentation graphique unifiée entre machines à états et données des objets (voir figure 25), afin de spécifier des comportements réutilisables sous forme de machines à états [One14].

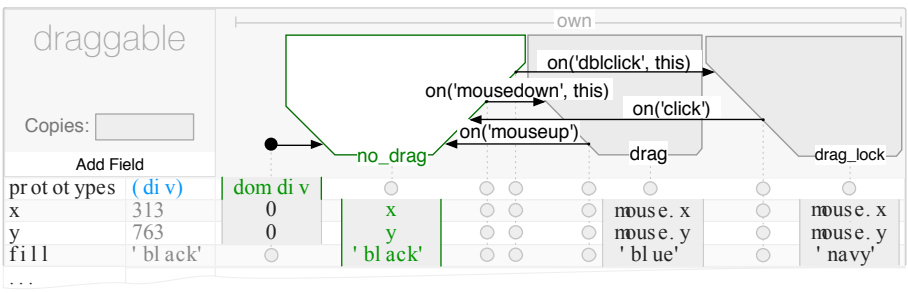


Figure 25 : Exemple du comportement de glisser-déposer implémenté dans InterState (figure extraite de [One14]).

3.1.5 ICON et MaggLite

ICON est une boîte à outils dédiée à la manipulation des périphériques d'entrée d'une application, et la spécification de techniques d'interaction [Dra01, Dra04]. Elle est centrée autour d'une représentation graphique interactive en flots de données (*dataflow*), dans laquelle on connecte des slots d'entrée et de sortie pour établir des flux permanents (voir figure 26). ICON a été créée pour *adapter* la richesse des périphériques d'entrée aux modalités d'interaction limitées des logiciels de bureau, afin de permettre l'utilisation efficace de périphériques déviant du cadre stéréotypé du couple souris/clavier. Elle est construite principalement pour le framework Swing, mais fonctionne aussi partiellement pour des applications externes.

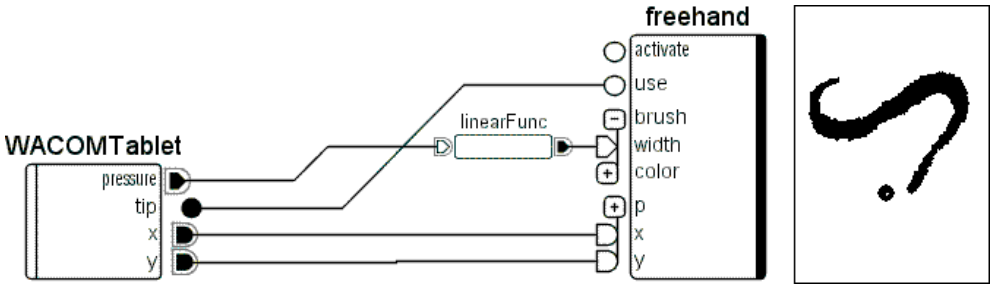


Figure 26 : Exemple d'une configuration d'entrées dans ICON (figure extraite de [Dra04]).

ICON est notable pour avoir été utilisée pendant plus d'une décennie pour gérer les entrées sur le mur d'écran WILD du LRI [Bea12]. De plus, elle a démontré l'applicabilité de la définition de techniques relativement complexes, par un langage de programmation visuel basé sur des diagrammes de flots (boîtes, slots et connexions). L'apparence visuelle très structurée lui permet en effet d'afficher une densité d'informations importante en restant lisible, chose généralement difficile avec des systèmes comme Max/MSP ou Pure Data. Cependant, bien que l'approche par flux de données soit adaptée aux aspects continus de l'interaction, elle l'est moins pour décrire des états et transitions entre états. Ainsi, l'approche hybride FlowStates a été proposée, qui combine les flux de données d'ICON avec les machines à états de SwingStates [App09]. Les interfaces construites avec ICON se distinguent par la flexibilité des connexions entre boîtes. Il est possible de lier les connecteurs à la volée, et de les déconnecter, sans perturber le fonctionnement du système. Les développeurs peuvent ainsi brancher un périphérique, et réaliser les connexions nécessaires pour l'adapter à une application existante, le tout durant l'exécution de l'application. Enfin, ICON étant un système de programmation visuelle, il n'est pas question d'intégration à un langage, puisqu'on ne l'utilise pas à partir des fonctions d'une API.

MaggLite est une boîte à outils de construction d'interfaces à main levée (illustrée en figure 27), développée conjointement à ICON et l'utilisant comme base de gestion des évènements d'entrée [Huo04]. Elle est dédiée explicitement à l'exploration et la définition de nouvelles techniques d'interaction. Par un modèle de *graphes combinés* [Huo06] faisant cohabiter graphe d'interaction (issu d'ICON) et graphe de scène, elle sépare clairement l'apparence et la logique d'une application, en donnant une importance accrue à la gestion des interactions (par rapport à un système basé sur la

propagation d'évènements). MaggLite est aussi notable pour avoir mis en lumière l'aspect stéréotypé des interfaces WIMP, en démontrant l'utilisation à la fois de périphériques non conventionnels, de techniques d'interaction issues de la recherche, et de visuels inhabituels.

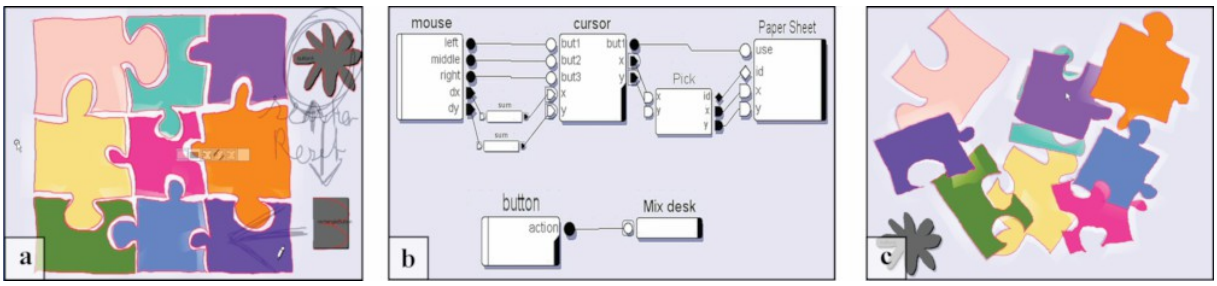


Figure 27 : Illustration du principe "*Draw it, Connect it and Run it*" de MaggLite (figure extraite de [Huo04]).

Bien que ICON et MaggLite aient contribué à défaire le caractère inflexible et conventionnel des interfaces, le principe des graphes d'interaction a été peu répliqué par la suite. Pourtant, les périphériques et techniques d'interaction n'ont cessé de se diversifier, mais aussi de gagner en degrés de liberté et paramètres à contrôler. Les graphes d'interaction montrent leurs limites dans l'expression de techniques complexes qui y sont difficilement lisibles, ainsi que le manque d'intégration dans les langages couramment utilisés pour prototyper des interfaces graphiques.

3.1.6 Proton et Proton++

Proton est une boîte à outils facilitant l'expression de techniques d'interaction *multi-touch* sur écrans tactiles [Kin12]. Elle répond à la difficulté de prototyper rapidement de nouvelles interactions au doigt, en particulier lorsqu'elles mettent en oeuvre des séquences de gestes et combinaisons à plusieurs doigts. Proton modélise les techniques au doigt par des expressions régulières, qui sont formées par assemblage de trois types d'actions (appui, déplacement, relâchement), auxquels sont ajoutés des attributs de numéro de doigt et d'objet ciblé (voir figure 28). Elle fournit de plus un éditeur graphique de *tablatures*, inspirées de la notation musicale. Proton++ est la continuation du travail effectué sur Proton, qui permet de définir de nouveaux types d'actions paramétriques, et de spécifier leurs paramètres par une extension aux expressions régulières de Proton [Kin12].

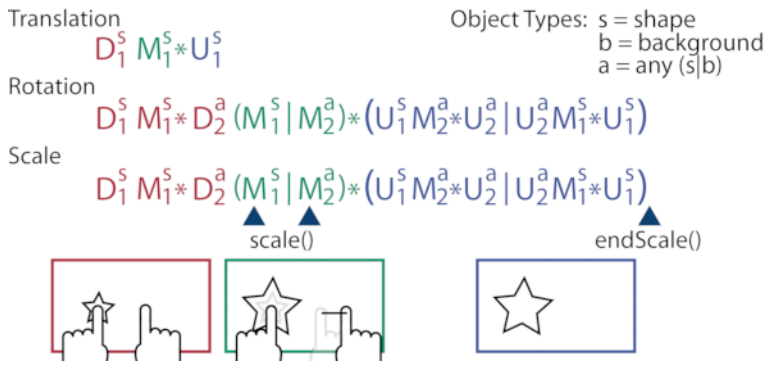


Figure 28 : Illustration des expressions régulières de Proton (figure extraite de [Kin12]).

Proton et Proton++ sont notables pour avoir rapproché l'expression de techniques d'interaction à la définition d'expressions régulières, apportant de nouvelles manières pour les développeurs de raisonner sur ces artefacts. De plus, grâce à leur définition d'une syntaxe concise et inambiguë pour décrire des gestes complexes, elles permettent de raisonner à plus haut niveau pour prototyper du contrôle gestuel, sans avoir avoir à se perdre dans du code souvent verbeux.

Cependant, les deux boîtes à outils reposent fortement sur l'absence d'ambiguïtés dans les gestes atomiques. En effet il est facile de différencier un appui du doigt d'un déplacement du doigt, à la fois pour le programme et pour l'utilisateur. Les gestes ajoutés par Proton++ respectent ces mêmes propriétés : ils sont fortement différenciables les uns avec les autres. Or les ambiguïtés sont courantes dans les interfaces gestuelles, pour lesquelles les systèmes de reconnaissance classiques quantifient une probabilité de reconnaissance. De façon générale, nous arguons que chaque technique d'interaction peut nécessiter un contrôle fin, en particulier lorsqu'elle évolue avec les usages courants. Par exemple, il est possible que les utilisateurs réalisant un geste de pincement (*pinch*) sur un objet pour l'agrandir, commencencent le geste en l'air et positionnent ainsi les premiers points de contact en dehors de l'objet. Pour une technique d'interaction robuste à ce type d'usages, il convient de détecter les gestes ambigus, or la rédution à des expressions régulières de Proton et Proton++ limite ce contrôle.

3.1.7 subArctic/Artkit

subArctic est un framework complet de programmation d'interfaces graphiques et d'interactions, construit principalement autour de la notion d'*extensibilité* [Hud05]. Développé comme Garnet sur plus d'une décennie, et pour soutenir la recherche, il a été utilisé pour explorer des sujets tels que l'abstraction et l'implémentation des animations, l'implémentation de contraintes de positionnement propagées localement (μConstraints), le déboguage visuel interactif des applications graphiques, et plus généralement l'implémentation d'interfaces non-standard. Ces travaux ont contribué à améliorer les fonctionnalités et l'apparence des interfaces graphiques, et sont aujourd'hui présents, sous différentes formes, dans de nombreux frameworks. subArctic est le successeur d'Artkit, un framework d'interaction basé sur le C++ [Hen90].

En ce qui concerne la programmation de nouvelles techniques d'interaction, subArctic se caractérise principalement par un modèle complet d'entrées extensibles (voir figure 29). Ce modèle permet la définition et l'ajout de nouveaux *interacteurs*, à la compilation comme à l'exécution, qui interprètent en séquence des comportements de plus haut niveau. Il est ainsi possible d'intégrer de nouvelles techniques d'interaction à des interfaces existantes, ou de remplacer des modalités d'interaction existantes. Ce modèle est analogue à celui que nous implémentons dans Polyphony, et que nous présentons dans les sections suivantes. Enfin, la boîte à outils subArctic n'ayant pas fait usage de techniques d'intégration à un langage, elle s'utilise par une API standard pour Java.

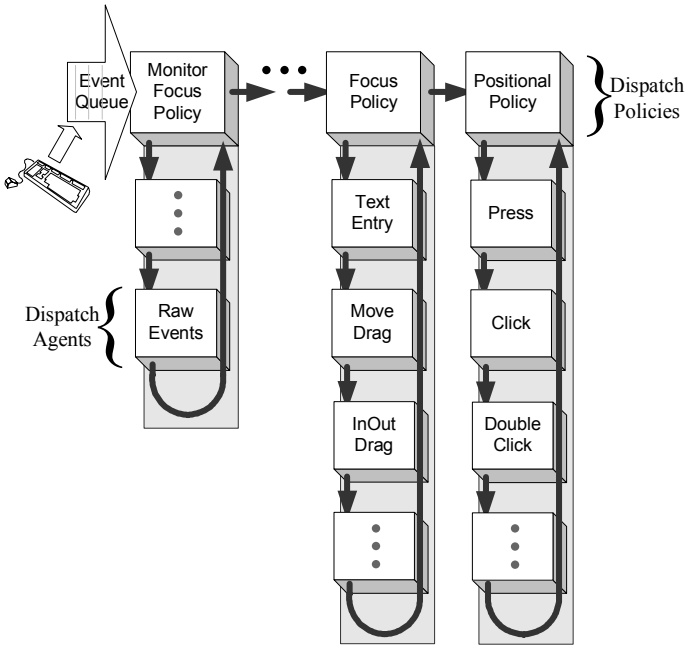


Figure 29 : Illustration du modèle d'évènements de subArctic (figure extraite de [Hud05]).

3.1.8 D3/Protovis

D3 est une bibliothèque conçue pour créer des visualisations interactives sur le Web, qui fournit un langage dédié basé sur JavaScript, permettant de manipuler du SVG inclus dans le DOM d'une page Web [Bos11]. Elle se base sur une approche pragmatique de la création de visualisations : être la plus *efficiente* (moins d'efforts pour produire une visualisation), *accessible* (plus facile à apprendre), *expressive* (variété des visualisations possibles, voir figure 30), et *performante* (temps d'exécution). D3 est notable pour son intégration voulue à des technologies existantes (SVG, DOM), sans introduire de nouvelle représentation ni étendre de format existant. Les auteurs de cette bibliothèque critiquent ouvertement l'usage de représentations intermédiaires de contrevenir aux objectifs précédents. Ils justifient leur choix pour : le faible apprentissage nécessaire pour les utilisateurs familiers du Web, la grande base d'utilisateurs du Web qui peuvent adopter rapidement D3, les documentations déjà nombreuses et complètes sur DOM et SVG, le bénéfice des outils de débogage du DOM présents dans les navigateurs Web, et l'absence de réduction d'expressivité due à la traduction depuis un format différent.

D3 a été un succès d'usage, puisqu'elle a été adoptée massivement pour produire des visualisations interactives en ligne, et a contribué à défaire le caractère stéréotypé des visualisations, en ramenant leurs conceptions aux dessins de formes simples en SVG. Le concept de données *liées* aux éléments graphiques relie la flexibilité des données à celle des visualisations. Cependant, D3 ne cherche pas expressément à étendre les capacités des périphériques d'interaction, autrement qu'en proposant de nouvelles manières de manipuler les données présentes dans le DOM. Son intérêt pour la programmation de nouvelles techniques d'interaction est donc moindre, bien que son approche pragmatique de validation de contribution par la communauté soit un exemple à suivre. D3 succède à Protovis, un framework complet de visualisation basé sur JavaScript [Bos09].

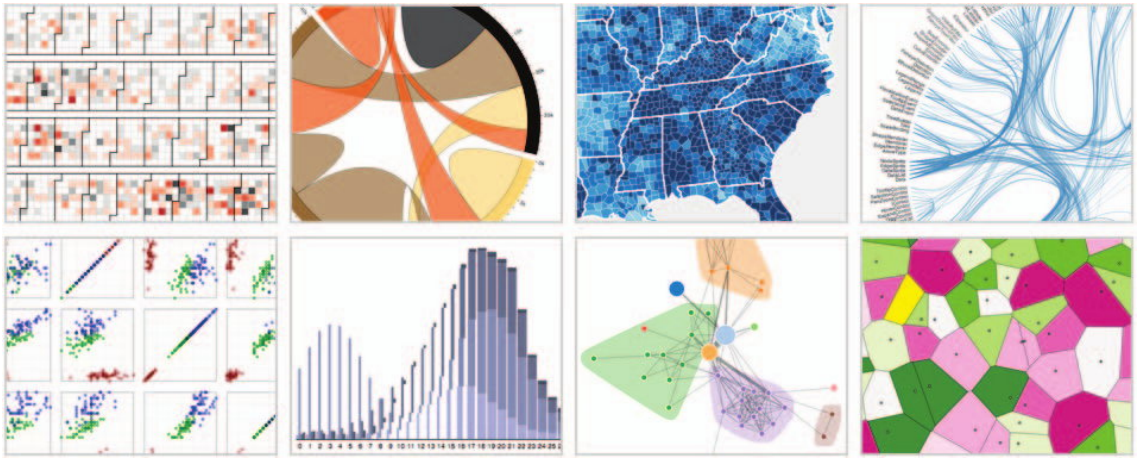


Figure 30 : Exemples de visualisations construites avec D3 (figure extraite de [Bos11]).

3.1.9 Limites de l'état de l'art et opportunités de contributions

À partir des travaux présentés ci-dessus, nous observons un certain nombre de motifs récurrents en nous concentrant sur les questions posées au début de cette section. Tout d'abord, la plupart répondent à des **besoins de propagation d'informations**. Ces informations peuvent être des données, ou de façon plus abstraite des "survenues d'évènements". Il peut s'agir de propager une information de clic de souris vers un déclenchement de la commande d'un bouton, ou de propager l'information de déplacement de la souris sur une contrainte de positionnement durant un appui-relâchement. Ces besoins ne correspondent pas exactement au paradigme de mise à jour d'états en mémoire mis en avant dans les architectures matérielles et la programmation impérative, car les informations doivent transiter entre des étapes de traitement, plutôt qu'être archivées en mémoire. C'est pourquoi de nombreuses boîtes à outils ont introduit des paradigmes de propagation d'informations (*dataflow*, *bindings*, systèmes de contraintes). Les boîtes à outils étant pour la plupart implémentées avec des langages impératifs, ces paradigmes doivent cohabiter avec des types de raisonnements différents. Nous considérons que cette situation est source de confusions pour les développeurs, et **avons choisi au contraire d'adapter la propagation d'information au modèle impératif** (en particulier avec le stockage des changements d'états en "variables temporaires" présentées en section 3.2.5).

Ensuite, la majorité des travaux que nous avons étudiés nécessitent la **construction d'un arbre de scène**. Ce type de structure est très répandu dans les frameworks d'interaction, et très utile pour fournir des comportements globaux cohérents (ex. les règles de flux de positionnement des éléments dans leurs conteneurs en HTML), pour mutualiser la modélisation de relations complexes (ex. l'ordre d'affichage et l'ordre de réception des évènements de la souris), et plus généralement pour maîtriser la complexité des interfaces réalistes. L'arbre de scène impose de raisonner sur la construction d'interfaces par imbrications récursives d'éléments rectangulaires, et en retour permet la réutilisation de contrôles standards (les *widgets*), implémentés comme des nœuds de l'arbre de scène. Cependant, l'arbre de scène n'est pas strictement nécessaire pour produire de petites applications. Par exemple, pour dessiner un objet suivant le curseur à l'écran, il faudrait dans la plupart des boîtes à outils fournir un canevas de dessin à la taille de l'écran, et y exécuter des instructions de dessin. La nécessité d'un arbre de scène gène ainsi le caractère *incrémental* de l'activité de prototypage, évoqué en section 1.3.1.2.

Selon les Dimensions Cognitives de Notations [Gre96], ce point représente un coût important de *Premature commitment*, en ce que les développeurs doivent d'abord construire un arbre de scène minimal (ainsi que le code de base pour initialiser la bibliothèque) avant de pouvoir créer avec. Ce coût est accentué par l'introduction de notions spécifiques aux arbres de scène, dont il faut acquérir une certaine compréhension au préalable. Enfin, nous conjecturons que l'utilisation d'un arbre de scène, ainsi que les concepts visant à simplifier les différents types de nœuds existant, résulte en l'instanciation d'un plus grand nombre de nœuds. Sans pouvoir étayer cette affirmation, **nous avons néanmoins cherché à remettre en question la nécessité d'un arbre de scène, et étudié les différents types de relations qui en tirent parti**.

De nombreux travaux proposent une **intégration étroite avec un langage de programmation** (djnn avec Smala, UBit avec C++, D3 avec JavaScript, voire Proton avec les expressions régulières), au delà de fonctions accessibles par une API. Ils font souvent usage de fonctionnalités de métaprogrammation, pour *tordre* les règles liant la syntaxe et la sémantique, et en superposer de nouvelles. Par exemple, UBit redéfinit les opérateurs + et /, qui expriment respectivement l'addition et la division arithmétiques, pour leur faire exprimer respectivement l'ajout d'un frère dans un arbre, et la relation d'attachement d'un *callback* à un type de déclencheur. D'autres boîtes à outils comme InterState, ICON, ou encore Proton, fournissent des interfaces graphiques pour programmer visuellement de l'interaction. Ils possèdent ainsi un contrôle plus fin des éléments de syntaxe qu'utiliseront les développeurs pour exprimer des comportements interactifs, sans dépendre des règles préexistantes d'un langage. Durant la conception de Polyphony, nous avons **cherché à appuyer l'intégration avec le langage JavaScript, faisant coïncider la notion d'Entité avec celle des objets, afin que la syntaxe de Polyphony semble "native" à JavaScript**, plutôt que "superposée" comme peut l'être par exemple UBit.

Nous allons discuter maintenant de points saillants, pour lesquels nous nous sommes volontairement distingués de l'état de l'art présenté ici, et plus généralement des frameworks d'interaction utilisés pour le prototypage en IHM. Nous en dégageons des opportunités de contributions, qui forment l'origine de notre travail sur Polyphony.

3.1.9.1 Mutualisation et réutilisation des comportements

Une manière de différencier les bibliothèques de construction d'IHM est d'analyser leur gestion des *comportements* interactifs. Nous considérons cette notion de comportements à deux niveaux interdépendants. À haut niveau, ils désignent les réactions observables des Entités à différents stimuli. Ces stimuli peuvent être des évènements externes (clic de souris, front montant d'horloge), ou internes (la souris survole un élément). À bas niveau, les comportements désignent les variables et le code déclenchés par ces évènements, et qui donnent lieu aux réactions observées. Par exemple, le comportement "permettre l'édition de texte" réfère à haut niveau à l'affichage de caractères en séquence lors de chaque appui de clavier, et à bas niveau au code qui après chaque entrée du clavier obtient l'objet ciblé par le clavier, et ajoute chaque caractère en bout d'une chaîne de caractères dont l'affichage est mis à jour.

La plupart des bibliothèques basées sur des objets utilisent la notion d'*arbre d'héritage* pour organiser les comportements individuels et partagés, et peuvent être décrits comme "monolithiques" [Bed04]. L'arbre d'héritage établit une hiérarchie de types entre les objets, selon

laquelle tout objet descendant d'un autre en hérite les définitions de méthodes et de variables (donc les comportements). Grâce à cette propriété, le type descendant peut être utilisé là où on attend le type parent — c'est le *polymorphisme*. Dans les applications d'IHM, les comportements partagés par cet arbre sont principalement : l'apparence des widgets, leur propagation des contraintes de dimensionnement aux voisins/enfants, et leurs réactions aux évènements souris/clavier. Cette réutilisation est particulièrement utile pour la création de nouveaux types de widgets, qui peuvent se baser sur des widgets existants (hériter de leur classe), et ajouter des fonctionnalités plutôt que tout réimplémenter.

Cependant, les hiérarchies entre ancêtres sont généralement prédéfinies, ce qui complique l'attribution de comportements à des objets qui n'ont pas été spécifiquement conçus et implémentés pour les recevoir [Bed04, Lec03, Mye97], aussi bien de manière statique (à la compilation) que dynamique (à l'exécution). Par exemple, il est souvent difficile d'ajouter le comportement d'édition de texte aux *labels* d'une interface (ex. `JLabel`). En effet, les fonctions pour *recevoir les appuis clavier*, *afficher un curseur de texte clignotant* et *modifier une chaîne de caractères*, appartiennent souvent à une autre famille de composants, dédiée à l'édition de texte.

Pour un programmeur ayant accès au code source de l'interface, ce changement peut se faire de deux manières : soit avec une sous-classe d'un *champ de texte* modifiée pour ressembler à un label, soit avec une sous-classe d'un *label* modifiée pour permettre l'édition de texte. Dans les deux cas un des deux comportements est recréé car il ne peut pas être hérité. Ce problème a donné lieu à des architectures d'objets favorisant la *Composition* plutôt que l'*Héritage*, comme les Traits [Cur82] ou les mixins [Bra90]. Cependant, ces architectures ont été peu utilisées pour développer des bibliothèques d'IHM, ce que nous imputons aux contraintes qu'elles imposent pour maintenir leur cohérence. Pour un programmeur n'ayant pas accès au code source de l'interface, ajouter un comportement à un élément revient à changer sa *nature*, ce qui relève parfois de l'impossible. Comme le remarque Lecolinet [Lec03], « *behaviors and other features are not seen as general services that could be used by any widget* ».

De nombreuses bibliothèques dites "polylithiques" [Bed04] ont été proposées, liées au paradigme de Composition, pour partager des comportements sans arbre d'héritage, statiquement ou dynamiquement. UBit [Lec03] le fait en synthétisant des comportements et des propriétés composables dans les "briques" d'un arbre de scène, dont l'interprétation récursive matérialise l'interface. De même, Jazz [Bed00] modélise les comportements par des nœuds insérés entre les éléments de l'arbre de scène. Les comportements sont ainsi hérités par tous les enfants de l'arbre. MaggLite [Huo04] reprend un principe similaire, ajoutant un modèle de *graphes combinés* qui lie le graphe d'interaction de ICON à celui du graphe de scène [Huo06].

La majorité de ces approches se base sur l'existence d'un *arbre de scène* pour supplanter l'arbre d'héritage proposé par les langages à objets. L'arbre de scène, qui est déjà utilisé pour structurer les éléments visuellement, se prête bien à l'héritage des styles visuels — propriétés graphiques de UBit, nœuds de Jazz. Cependant, il ne se prête pas à *toutes* les propriétés. CSS, par exemple, indique pour chaque type de propriété si elle est héritée par défaut ou non [Moz19]. En effet, la couleur de texte (`color`) peut être partagée avec les enfants d'un élément, mais pas ses marges (`margin`/`padding`). L'arbre de scène n'est donc pas une solution universelle, d'autant que son utilisation exclusive requiert

de nombreux nœuds, qui rendent son inspection visuelle difficile. Comme nous l'avons énoncé précédemment, nous avons pour objectif d'**explorer des approches qui ne se basent pas exclusivement sur un arbre de scène, notamment à l'aide de la composition**.

3.1.9.2 Dynamicité des comportements

La plupart des bibliothèques d'IHM ont un support limité pour l'ajout de nouveaux comportements à des objets, à l'exécution. Il est aussi difficile d'altérer des comportements à la volée (ex. *activer/désactiver*), à moins qu'ils aient été prévus pour. Dans une interface graphique, par exemple, les menus déroulants ne permettent pas de réordonner leurs éléments à la souris. Rendre les éléments d'un menu ordonnables durant l'exécution du programme, est un exemple de dynamicité des comportements.

Il est important de noter que cette dynamicité est indépendante du choix de Composition ou d'Héritage pour partager les comportements. Il s'agit en fait de pouvoir changer *à l'exécution* les composants ou parents d'un objet donné.

Une solution commune est pour chaque élément de posséder le comportement visé, et de le désactiver par défaut. C'est le cas par exemple dans Qt pour l'exemple cité: la classe de base `QAbstractItemView` contient une méthode `setDragEnabled` qui permet à chaque liste d'activer la possibilité de déplacer les éléments à la souris. Cette approche est toutefois limitée par le fait que l'ajout de nouveaux comportements doit se faire par la classe de base.

Cette limite a conduit les chercheurs à développer diverses techniques pour attribuer des comportements à la volée. Scotty [Eag11] permet par exemple d'analyser la structure des composants d'interaction d'une application Cocoa existante et d'y injecter du code pour en modifier les comportements. Bien que spectaculaire et efficace pour le prototypage rapide de nouveaux comportements interactifs dans des applications existantes, cette approche présente des défauts de robustesse et de persistance qui ne permet pas de l'utiliser à plus grande échelle que pour du "hacking" temporaire. Ivy [Bui02] est un bus de messagerie sur lequel des agents peuvent envoyer du texte et s'enregistrer pour écouter des messages (sélectionnés par expressions régulières). De cette manière, de nouveaux comportements sont ajoutés par le biais de nouveaux agents, et les agents se remplacent en envoyant des messages compatibles. Amulet [Mye97] se base sur un modèle d'objets à *slots* dynamiques (variables et méthodes), qui semblent permettre l'ajout de comportements à la volée. Cependant, spécifier une nouvelle méthode ne garantit pas qu'elle soit exécutée automatiquement, il faut hériter du bon prototype et la hiérarchie des prototypes n'est pas modifiable.

La plupart des travaux gérant l'attribution de nouveaux comportements (non-prédéterminés) à la volée se heurtent à la *nécessité d'une spécification*. En effet, les objets sont passifs par nature, ils ne s'exécutent que si on les "appelle", par une fonction ou un envoi de message. Les programmes sont passifs aussi, le système d'exploitation les lance en exécutant une fonction attendue (généralement `main`). Pour s'exécuter, un comportement/objet/programme doit donc fournir une spécification (ou interface), qui indique comment l'exécuter, et avec quels arguments. Lorsqu'il est défini *avant* le code qui doit l'appeler, il peut imposer sa spécification, et ce sera à l'appelant de se conformer en appelant les bonnes fonctions. Lorsqu'il est défini *après* le code appelant, les spécifications ont nécessairement

été déjà fixées. Par exemple, pour qu'un objet s'affiche dans JavaFX il doit implémenter la méthode `onDraw`, qui sera exécutée par le framework pour tout objet dans le graphe de scène. Toute autre méthode de dessin sera ignorée, car le système ne peut pas savoir autrement qu'elle sert à dessiner.

Les travaux étudiés sont limités par l'extensibilité des spécifications qu'ils fournissent — Ivy par le format des messages échangés sur le bus, et Scotty par l'architecture inchangeable des applications Cocoa. **La définition ad-hoc de nouvelles spécifications, et l'adaptabilité de ces spécifications à des objets préexistants**, sont des fonctionnalités peu présentes dans les langages à objets (hormis la notion de *typage structurel* implémenté dans le langage Go). Elles ont été peu explorées dans les bibliothèques d'IHM, et fournissent des opportunités de contributions pour ce travail de thèse.

3.1.9.3 Orchestration des comportements

Le déclenchement et l'ordre d'exécution des fonctions sur des éléments interactifs est important. Il peut s'agir d'exécuter une fonction systématiquement avant/après une autre, après un changement d'état donné, ou bien sur réception d'évènements d'un ou plusieurs périphériques d'entrée. La conception d'IHM regorge de tels cas d'utilisation complexes. Par exemple, le rendu graphique des éléments d'une interface Web met en œuvre le dessin d'arrière-plans, de bordures, de texte, voire d'ombres portées. En utilisant l'algorithme du peintre, les étapes de rendu pour chaque élément doivent s'exécuter dans un ordre précis: l'arrière-plan *avant* la bordure et le texte, et l'ombre portée *avant* l'arrière plan. Autre exemple, certaines techniques d'interaction s'appuient sur des séquences d'actions, provenant potentiellement de plusieurs périphériques d'entrée, et pouvant mettre en œuvre des délais de pause (comme *CTRL + clic de souris + attente de 500ms*). Là encore, les langages et bibliothèques ont un support limité de ces besoins. Les appels de fonctions offrent une orchestration séquentielle et statique des différents blocs de code, et les mécanismes de callbacks enregistrent des centaines, voire des milliers de liens pour des interfaces “réalistes” [Mye91].

Pour palier à ces limitations, différents modèles ont été proposés. Les modèles basés sur le formalisme états-transitions (machines à états) de SwingStates [App06] et HsmTk [Bla06] permettent de limiter “l'éclatement” du code définissant les comportements interactifs, tout en offrant une représentation de l'interaction proche de celle des concepteurs et des programmeurs. Des modèles en flot de données ont aussi été proposés, qui exécutent des blocs de code lorsque toutes les données dont ils dépendent sont disponibles. Parmi ceux-ci, ICon [Dra01] réifie les liens de dépendances en des arcs représentés graphiquement, que les utilisateurs peuvent manipuler directement, par exemple pour ajouter ou remplacer un périphérique d'entrée. Les deux paradigmes ont même été unis dans FlowStates [App09] afin de tirer partie des deux formalismes aux niveaux où ils sont les plus adaptés.

Des approches d'Ingénierie Dirigée par les Modèles comme MARIA [Pat09] et UsiXML [Lim05] ont également été utilisées pour abstraire la définition des interfaces par rapport aux modalités d'interaction disponibles sur chaque machine (périphériques, système d'exploitation), grâce à des transformations entre modèles de différents niveaux d'abstraction. Elles permettent d'adapter l'interface au contexte dans laquelle on l'utilise, qu'il s'agisse de la taille de l'écran (ex. les sites Web *responsive*), de l'état physique et émotionnel de l'utilisateur [Gal17, Gaj10], ou simplement pour explorer un espace de conception de l'interface [Lei15]. Dans le contexte des gestes et des séquences

d'actions, Proton [Kin12] est notable pour son utilisation d'expressions régulières pour représenter les séquences de gestes, et GestIT [Spa13] est notable pour sa modélisation de séquences d'actions complexes à l'aide d'opérateurs de composition.

La plupart des approches basées sur des modèles ont en commun qu'elles représentent des *micro-dépendances* au sein des techniques d'interaction — elles rendent les relations “faire B après A” explicites, plutôt qu'avec le séquencement implicite du code. Ce faisant, elles ont tendance à nécessiter beaucoup de code, et à passer difficilement à l'échelle pour gérer des interfaces et interactions complexes. Dans ECS, l'orchestration des comportements se fait au niveau *macroscopique*, non pas sur les données mais sur les processus. En ce sens, nous le considérons comme complémentaire aux approches d'ingénierie par les modèles, qui pourraient l'utiliser comme cible de moindre niveau d'abstraction. Le modèle ECS s'aligne en effet sur l'exécution des algorithmes liés aux jeux vidéo (rendu 3D, simulation physique), qui traitent rapidement et régulièrement de grands volumes de données. Il incite à considérer les comportements comme des processus qui transforment les évènements d'entrée en évènements de sortie. Comme Chatty et al. l'écrivent pour présenter djnn, « *Like computations can be described as the evaluation of lambda expressions, interactions can be described as the activation of interconnected processes* ». Polyphony se distingue par l'ajout d'attributs (Composants) aux processus (Systèmes), de manière à traiter explicitement les dépendances entre eux, sans contraindre le choix des attributs à donner. L'orchestration d'une application peut ainsi être étendue et améliorée par l'introduction de nouveaux composants.

Enfin, peu de travaux ont réussi à pallier la “fragmentation” de la logique dans les applications interactives, mise en avant par Myers [Mye91]. Beaucoup d'entre eux mettent en oeuvre des *petites* fonctions, qui réalisent des comportements minimes se réduisant à quelques lignes de code (ex. les transitions des machines à états, ou les propagations de contraintes entre variables). L'utilisation de ces petites fonctions est encouragé par la pratique des *callbacks*, qui consiste à enregistrer une fonction à appeler lorsqu'un évènement se produit. Les programmes effectuent alors de nombreux “sauts” entre des fonctions, qui rendent leur fonctionnement difficilement observable, en plus d'être peu efficace sur les architectures modernes de processeurs. Ces problèmes justifient l'**étude de modèles basés sur des processus macroscopiques, et écartant l'usage de callbacks pour réduire la fragmentation de la logique dans les applications interactives**.

3.2 Programmation d'interactions avec Polyphony

Dans cette section, nous présentons la boîte à outils Polyphony, basée sur le modèle de programmation ECS, et adaptée au prototypage d'interfaces graphiques et de techniques d'interaction. Nous décrivons d'abord les concepts de base d'ECS, puis illustrons l'utilisation de Polyphony du point de vue d'un programmeur d'application.

3.2.1 Le modèle Entité-Composant-Système

ECS (parfois appelé CES) est un paradigme conçu pour structurer le code et les données dans un programme, apparu dans le domaine du développement de jeux vidéos [Leo99, Bil02, Mar07]. Dans ce contexte, les équipes sont typiquement séparées entre programmeurs concevant la logique et les outils, et concepteurs produisant du contenu scénaristique ou multimédia. ECS conçoit les

Composants comme l'interface entre ces deux mondes : les programmeurs définissent les Composants disponibles et créent les Systèmes qui vont itérer dessus ; les concepteurs instancient le contenu avec des Entités et leur associent des comportements par assemblage de Composants choisis. Les éléments constituant le modèle sont donc :

Entités
: Ce sont des identifiants uniques pour chaque élément du programme (souvent de simples entiers). Ils sont similaires aux objets, mais ne possèdent *ni données ni code*.

Composants
: Ils représentent les données du programme (comme `bounds` ou `children`), et sont associés dynamiquement aux Entités. Selon les interprétations, un Composant seul peut désigner le *type* de donnée attachable à toutes les Entités, ou son *instance* attachée à une Entité.

Systèmes
: Ils s'exécutent continuellement en suivant un ordre prédéfini, et représentent chacun un comportement réutilisable du programme. Les Entités ne s'enregistrent pas auprès des Systèmes, mais acquièrent plutôt les Composants nécessaires pour être "vus" par ceux-ci.

Le fonctionnement général d'une application basée sur ECS est illustré en figure 31. Les Composants sont le coeur du programme, ils définissent à la fois les attributs (`shape`, `backgroundColor`) et les capacités des Entités (`clickable`, `draggable`). Les Entités acquièrent du comportement par l'acquisition de Composants. Enfin ce sont les Systèmes qui exécutent chaque type de comportement, sélectionnant les Entités à partir de leurs Composants. Par exemple, un Système de dessin d'arrière plan sélectionnerait toutes les Entités possédant les Composants `bounds`, `shape` et `backgroundColor`, et pour chacune dessinerait une forme d'arrière-plan à l'écran. Nous représentons les Systèmes dans un ordre d'exécution commun à la plupart des frameworks d'interaction : gestion des périphériques d'entrée, interprétation des techniques d'interaction, traitements spécifiques aux différents widgets, traitements spécifiques à l'application, et rendu des sorties (en particulier à l'écran).

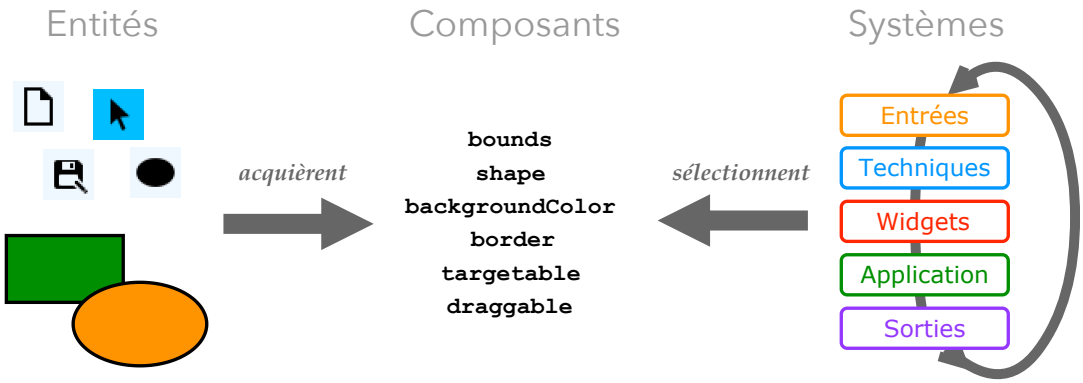


Figure 31 : Illustration des échanges à haut niveau entre Entités, Composants, et Systèmes.

Aux trois éléments de base du modèle s'ajoutent quatre éléments récurrents dans la majorité des implémentations d'ECS, bien qu'ils ne fassent pas partie du modèle de base et n'aient pas de terminologies fixes :

Descripteurs
Ce sont des conditions basées sur les Composants possédés par chaque Entité. Ils sont analogues à des *interfaces* évaluées dynamiquement, et sont le mécanisme fondamental par lequel les Systèmes savent sur quelles Entités opérer.

Sélections
Ce sont des conteneurs basés sur les Descripteurs, et mis à jour dynamiquement à mesure que des Entités acquièrent ou perdent des Composants.

Contexte
Il représente le *monde* courant auquel appartiennent toutes les Entités, stocke les Composants et les variables globales, enregistre et exécute les Systèmes, et fournit une interface pour obtenir des Sélections.

Fabriques d'Entités
Ce sont des modèles prédéterminés, utilisés pour instancier des Entités avec un ensemble de Composants et de valeurs par défaut.

3.2.2 Présentation de Polyphony et du prototype de développement

Polyphony est une boîte à outils logicielle expérimentale basée sur ECS, dédiée à la construction d'interfaces graphiques et d'interactions (voir figure 32). Elle se compose d'un noyau implémentant les notions d'Entités, de Composants, de Systèmes, de Descripteurs et de Sélections. Elle fournit de plus des *bindings* vers l'une ou l'autre de deux bibliothèques de bas-niveau , SDL [Lan98] et libpointing [Cas11]. Notre objectif était de pouvoir utiliser le support avancé de la souris de libpointing (souris multiples, remplacement des fonctions de transfert, interaction *subpixel* [Rou12]), avec SDL pour l'affichage. Cependant, libpointing empêchait la réception des évènements du clavier par SDL, ce qui nous a obligés à utiliser *l'un ou l'autre* des bindings. À un niveau supérieur dépendant du premier, Polyphony fournit des Composants, Systèmes, Sélections et fabriques prédéfinis, qui peuvent être utilisés par les applications pour construire rapidement des interfaces.

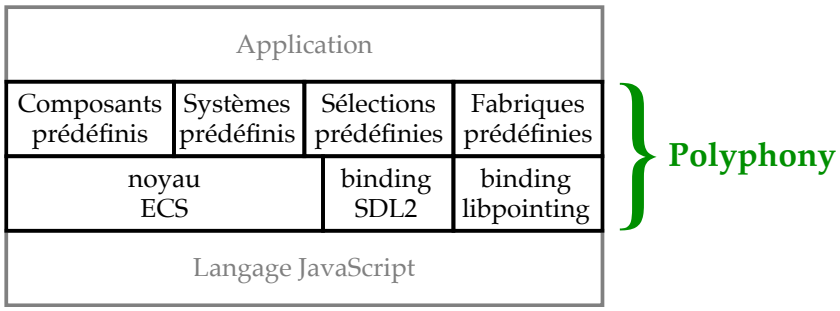


Figure 32 : Structure en modules de Polyphony

L'application de dessin vectoriel est un exemple courant et bien approprié pour illustrer la conception de techniques d'interaction [Gog14, Api04]. Elle combine l'implémentation d'objets graphiques, d'outils (ex. dessin de formes, pipette à couleurs), de commandes (ex. copier/coller, undo), et offre une large palette de tâches possibles avec de nombreuses améliorations et combinaisons entre éléments.

Notre application de base de dessin (voir figure 33) permet ainsi de :

- dessiner des rectangles et des ovales ;
- déplacer, supprimer et changer le type des formes créées ;
- enregistrer le résultat au format SVG ;
- réinitialiser l'espace de travail.

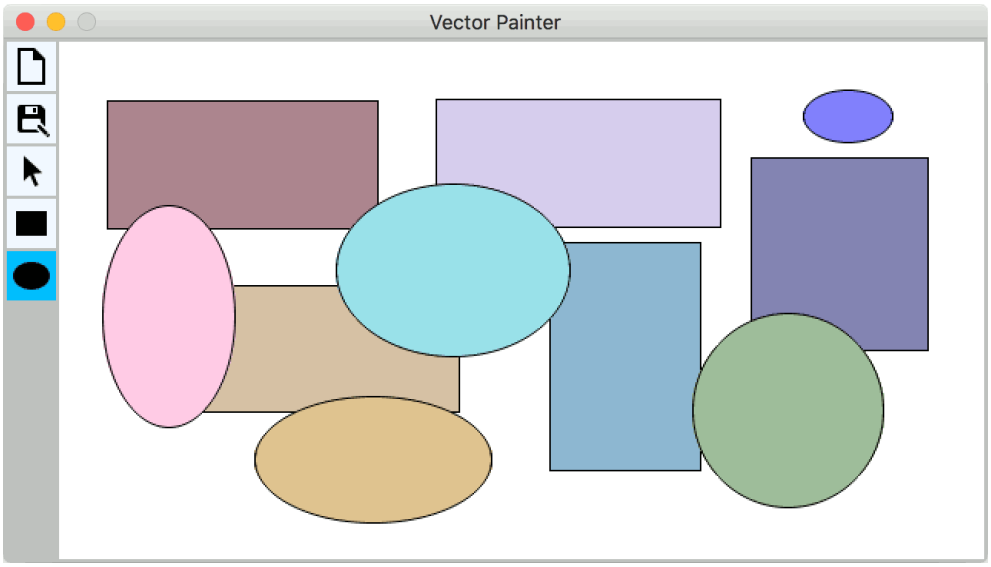


Figure 33 : Espace de travail de notre exemple d'application de dessin vectoriel

3.2.3 Illustration avec le code de l'application

Tous les éléments de l'interface sont des Entités : boutons, formes dessinées, et zone de dessin en arrière plan. Les périphériques d'entrée sont aussi représentés par des Entités, ainsi que la zone d'affichage.

3.2.3.1 Création d'Entités

Les Entités sont essentiellement des identifiants qui peuvent être créés à la volée et sans Composants avec la fonction `Entity` : `let e = Entity()`. Dans la pratique, il peut être souhaitable de créer des Entités avec des Composants initiaux. C'est pourquoi la fonction `Entity` peut recevoir en paramètre optionnel un objet JavaScript pour ajouter ces Composants initiaux. Par exemple, dans notre application de dessin, une Entité pour un simple rectangle sera créée avec :

```
let e = Entity({
    bounds: new Bounds(0, 0, 100, 50),
    shape: SHAPE_RECTANGLE,
})
```

À ce stade, l'Entité est visible par tous les Systèmes qui sélectionnent les Entités avec `bounds` ou `shape` (ex. “Système de *layout*”), sans avoir besoin de l'enregistrer quelque part. Dans notre cas, le “Système de rendu d'arrière-plan” sélectionnera et affichera les Entités qui possèdent au moins les

Composants `bounds`, `shape`, et `backgroundColor`. Ainsi, l'ajout du Composant approprié à notre Entité la fera apparaître à l'écran :

```
e.backgroundColor = rgba(0, 0, 255)
```

L'Entité devient alors visible par le "Système de rendu d'arrière-plan", qui dessine un rectangle bleu à l'écran. En pratique, Polyphony fournit des fabriques d'Entités qui permettent d'instancier des widgets standards avec des Composants prédéfinis. Ainsi, dans l'application de dessin, les boutons de la barre d'outils ainsi que le canevas sont créés à partir des fabriques `Button` et `Canvas` :

```
let y = 2
let resetButton = Button(new Icon('icons/reset.bmp'), 2, y)
let saveButton = Button(new Icon('icons/save.bmp'), 2, y += 34)
let moveButton = Button(new Icon('icons/move.bmp'), 2, y += 34, {toggleGroup: 1})
let rectButton = Button(new Icon('icons/rect.bmp'), 2, y += 34, {toggleGroup: 1})
let ovalButton = Button(new Icon('icons/oval.bmp'), 2, y += 34, {toggleGroup: 1})
let canvas = Canvas(36, 2, 602, 476)
```

Chaque fabrique prend en arguments un ensemble de valeurs nécessaires à la construction de l'Entité de base. En dernier argument optionnel, on peut passer un dictionnaire contenant un ensemble de Composants additionnels pour initialiser la nouvelle Entité. Si ce dictionnaire est déjà une Entité, elle est directement complétée plutôt que d'en créer une nouvelle. Par exemple, la fabrique `Button` est définie par :

```
function Button(imgOrTxt, x, y, e = {}) {
    e.depth = e.depth || 0
    e.bounds = e.bounds || new Bounds(x, y, imgOrTxt.w + 8, imgOrTxt.h + 8)
    e.shape = e.shape || SHAPE_RECTANGLE
    e.backgroundColor = e.backgroundColor || rgba(240, 248, 255)
    e[imgOrTxt instanceof Image ? 'image' : 'richText'] = imgOrTxt
    e.targetable = true
    e.actionable = true
    return Entity(e)
}
```

Lorsque la fabrique reçoit une Entité à compléter, elle n'ajoute les Composants que s'ils n'ont pas été déjà définis. C'est ce que réalise l'opération `e.composant = e.composant || valeur` — lorsqu'un Composant n'est pas défini il est évalué à `undefined`, ce qui est équivalent à `false` pour les opérations booléennes.

Enfin, dans ECS les Entités sont supprimées manuellement. Comme elles sont accessibles globalement grâce aux Sélections, elles ne peuvent jamais être qualifiées de "hors de portée" pour des mécanismes de portée locale ou de ramasse-miettes. La suppression d'une Entité s'effectue avec `e.delete()`, ses Composants étant alors automatiquement retirés, ainsi que toutes références pointant vers `e` dans le système.

3.2.3.2 Création de Composants

Dans Polyphony, les Composants sont les constructeurs d'objets de JavaScript :

```
function Bounds(x, y, w, h) {
    this.x = x
    this.y = y
    this.w = w
    this.h = h
}
```

Conformément au modèle ECS, ils ne contiennent pas normalement de code. Une exception à cette règle est la définition d'accesseurs, pour laquelle nous utilisons alors des méthodes d'instance — implémentées par l'héritage *prototypal* de JavaScript. Un *setter*, par exemple, se crée avec :

```
Bounds.prototype = {
    setX(x) {
        this.x = x
        return this
    }
}
```

Par convention, les *setters* dans Polyphony renvoient toujours l'objet ciblé (`this`), afin de pouvoir enchaîner plusieurs appels dans une seule instruction et rendre ainsi le code plus concis : (`e.setX(10).setY(20)`).

3.2.3.3 Création de Systèmes

Comme pour les fonctions lambda, qui sont réifiées en tant qu'objets du langage, nous avons implémenté les Systèmes en tant qu'Entités. Leurs dépendances sont donc représentées par des données stockées sous forme de Composants. Les systèmes sont instanciés une seule fois avec la même fonction `Entity`, qui prend comme paramètres une fonction suivie de ses Composants :

```
let ResetSystem = Entity(function ResetSystem() {
    if (resetButton.tmpAction) {
        for (let c of canvas.children)
            c.delete()
        canvas.children = []
    }
    ...
}, { runOn: POINTER_INPUT, order: 60 })
```

Dans notre exemple d'application, ce Système simple va vérifier si le bouton `resetButton` défini plus haut a été activé (cliqué) par la souris. Lorsque c'est le cas, il supprime tous les enfants de l'Entité canevas. Le Composant `runOn` de ce Système indique qu'il sera déclenché lors de chaque évènement

de pointage (souris). Le Composant `order` distingue et ordonne les classes de Systèmes (représentés en figure 31) : Entrées (0 à 19), Techniques d'interaction (20 à 39), Widgets (40 à 59), Application (60 à 79), et Sorties (80 à 99). `ResetSystem` fait donc partie des Systèmes d'application.

3.2.3.4 Création de Sélections

Lorsqu'on souhaite itérer sur des groupes d'Entités possédant des Composants en commun, on utilise généralement les Sélections de Polyphony. Par exemple, pour mettre en surbrillance toutes les Entités pouvant être ciblées par la souris :

```
for (let t of Targetables)
    t.border = new Border(1, rgba(0, 255, 0))
```

Ici, on itère sur la Sélection `Targetables`, qui contient toutes les Entités avec les Composants `bounds`, `depth` et `targetable`. Pour chacune de ces Entités, on remplace la bordure par une bordure verte épaisse d'un pixel. Une Sélection est définie avec la fonction `Selection`, qui prend comme paramètre une fonction *accept : Entité → booléen* pour filtrer les entités par condition programmable. Un deuxième argument optionnel fournit un critère d'ordre *compare : Entité × Entité → nombre* lors de l'itération sur la Sélection. Par exemple, la Sélection `Targetables` est définie avec :

```
let Targetables = Selection(e => 'bounds' in e && 'depth' in e && e.targetable,
    (a, b) => b.depth - a.depth) // trier par profondeur décroissante
```

Les Sélections dans Polyphony sont implémentées avec de simples tableaux. Lorsqu'une Entité subit une modification de Composant (ajout, remplacement, ou suppression), elle est ajoutée à une liste interne d'Entités à "soumettre aux Sélections". À l'issue de l'exécution de tout Système (et avant l'exécution du suivant), chaque Entité de cette liste est soumise à chaque Sélection (elle est passée en argument de la fonction *accept*) qui l'inclut ou non. Grâce à ce mécanisme, nous ne courons pas le risque de modifier une Sélection alors qu'on itère dessus, ce qui est une source de bugs répandue. Enfin, le tri d'une Sélection est effectué avant d'itérer dessus, à l'aide d'un booléen `dirty` indiquant si la Sélection a été modifiée et doit être triée à nouveau.

3.2.4 Modélisation d'une application interactive avec ECS

La figure 34 présente toutes les Entités de notre exemple d'application de dessin, chacune identifiée par la fabrique qui l'a instanciée. Contrairement aux frameworks les plus courants (ex. Qt, JavaFX, HTML), les objets graphiques n'appartiennent pas nécessairement à un graphe de scène. Dès leur création, les Entités sont accessibles aux Systèmes, donc déjà visibles et interactives. Les relations d'arbres, représentées dans cette figure par des flèches entre Entités, ne sont définies que lorsqu'il est nécessaire d'établir un ordre entre Entités — comme la profondeur d'affichage avec le Composant `depth`. Les périphériques d'interaction sont également matérialisés par des Entités (`Pointer`, `Keyboard`, et `View`), afin de fournir un stockage persistant et flexible pour les techniques d'interaction. Enfin, les Systèmes sont représentés par ordre d'exécution, leur couleur différenciant le type de traitement qu'ils exécutent.

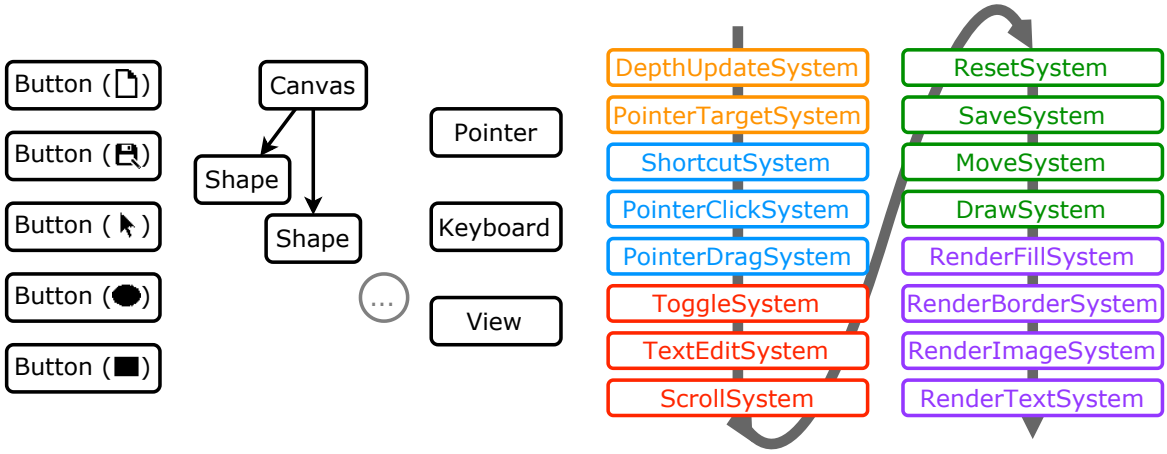


Figure 34 : Liste des Entités (incluant les Systèmes) dans l'application de dessin. Les flèches à gauche représentent les relations d'arbre. Les Systèmes à droite sont présentés dans leur ordre d'exécution, leur couleur de bordure indiquant leur type de traitement : Entrées, Techniques d'interaction, Widgets, Application, Sorties.

Le flux d'exécution dans Polyphony fonctionne comme une machine réactive [Dra01]. Toutes les 25*ms* (40*Hz*), Polyphony vérifie tous les évènements en attente du système d'exploitation et met instantanément à jour les Entités claviers et souris. Lorsqu'au moins un évènement clavier ou souris se produit, une liste ordonnée des Systèmes est recueillie, par le biais d'une Sélection `Systems`. Polyphony les ordonne ensuite selon le Composant `order`, et exécute chaque Système dont le Composant `runOn` correspond avec l'un des types d'évènements produits. Nous avons choisi de synchroniser le déclenchement de tous les Systèmes (y compris l'affichage) avec les évènements d'entrée (mouvement de souris, appui clavier, ou tout autre périphérique d'entrée), en raison du manque d'accessibilité des évènements *VSync* à bas-niveau. À l'avenir nous envisageons d'améliorer le modèle d'exécution réactive de Polyphony, avec une machine réactive plus générique et extensible, indépendante des évènements en entrée.

Les Composants déterminent les comportements qui peuvent être acquis par chaque Entité. Ils sont lus par chaque Système mettant en œuvre ces comportements. Le tableau 5 liste les Composants de l'application de dessin, ainsi que les fabriques qui les assignent. Ces Composants ont été inspirés par CSS1 [W3C96], qui synthétise la plupart des comportements communs aux différentes balises HTML, permettant par exemple de mimer un `button` avec un conteneur `div`, uniquement en acquérant des propriétés CSS.

Composants	Button	Canvas	Shape	Pointer	Keyboard	View	*Systems*
children		×					
depth	×	×	×				
bounds	×	×	×			×	
shape	×	×	×				
backgroundColor	×	×	×				
border			×				
image	~						
richText	~		~				
targetable	×	×	×				
actionable	×						
toggleGroup	~						
draggable			×				
textEditable			×				
cursorPosition				×			
buttons				×			
keyStates					×		
focus					×		
origin						×	
scrollable						~	
runOn							×
order							×

Tableau 5 : Composants et fabriques d'Entités les assignant. Les Entités de chaque fabrique reçoivent tous les Composants indiqués par ×, et selon leur spécialisation reçoivent ensuite ceux indiqués par ~.

Polyphony permet l'implémentation de contrôles standards d'interface utilisateur, avec les Systèmes et Composants. Trois types de contrôles sont illustrés dans notre exemple d'application :

- Les boutons activables (*toggle buttons*) sont des Entités avec les composants `bounds`, `shape`, `backgroundColor`, `image`/`richText`, `targetable`, et `actionable` (comme les boutons ordinaires). Le comportement d'activation/désactivation nécessite un autre Composant, `toggleGroup`. Un Système `ToggleSystem` sélectionne toutes les Entités avec ce Composant, et recherche un clic de pointeur sur l'une d'elles. Quand c'est le cas, l'Entité activée reçoit un nouveau Component `toggled` valant `true`, et son Composant `backgroundColor` est modifié, en même temps que toutes les autres Entités du même `toggleGroup` sont désactivées.
- Les champs de texte nécessitent un Composant `focus` sur chaque Entité clavier, et un Composant `richText` pour afficher le texte formaté à l'intérieur des limites d'une Entité. Ce dernier Composant contient une chaîne de caractères, des marges internes, et des informations de polices des caractères. Un Système `TextEditSystem` observe chaque caractère tapé sur le clavier et, en fonction du `focus`, met à jour le Composant `richText` de l'Entité ciblée, tout en gérant un curseur clignotant.

- Les vues déroulantes étendent les Entités `View` avec des Composants `viewport` et `scrollable`. Un Système `ScrollSystem` ajoute et gère une Entité enfant pour chaque vue avec ces Composants, afin d'afficher une barre de défilement. Le système détecte les actions de glisser-déposer de la barre, et les mouvements de la molette à l'intérieur de la vue, pour mettre à jour le composant `origin` à l'intérieur des bornes autorisées par `viewport`.

Plus généralement, la mise en œuvre de nouveaux types de widgets nécessite l'ajout de nouveaux Composants décrivant leurs capacités, et l'insertion de nouveaux Systèmes dans la plage des widgets de `order` (entre 40 et 59). Ces Systèmes forment des comportements réutilisables, qui peuvent à leur tour être composés pour la conception de futurs widgets.

3.2.5 Réification des périphériques en Entités

Polyphony s'interface avec les ressources du système d'exploitation via un ensemble extensible d'Entités *périphériques* (`Pointer`, `Keyboard`, `View`). Leurs Composants stockent l'état courant de chaque périphérique. `Pointer`, par exemple, possède le Composant `buttons` qui stocke l'état des boutons à tout instant, ainsi que le Composant `cursorPosition` qui stocke les coordonnées du curseur système. Ces coordonnées sont évidemment accessibles, mais aussi modifiables afin de contrôler la position du curseur de manière simple. Dans notre application de dessin, ce mécanisme est utilisé pour forcer le curseur à rester dans les bornes du canevas lors du dessin d'une forme par appui-déplacement.

La modélisation des périphériques en tant qu'Entités offre une représentation flexible et persistante entre les différentes couches du programme. La structure d'une Entité étant extensible, de nouvelles données peuvent être introduites sans créer de nouvel objet, mais en ajoutant de nouveaux Composants qui permettront à l'Entité d'être traitée par les Systèmes correspondants. Ce mécanisme est une forme de propagation d'évènements, qui ne crée pas d'objets d'évènements temporaires puisque les données sont centralisées dans les Composants de l'Entité.

La réification du pointeur en Entité est illustrée en figure 35. L'Entité représentant le curseur système est initialisée par la fabrique `Pointer` avec deux Composants, `cursorPosition` et `buttons`. Tant que c'est le seul pointeur disponible, nous l'appelons l'Entité `Pointer`. Ensuite, le *binding* d'entrées/sorties de Polyphony ajoute des Composants temporaires `tmpPressed`, `tmpReleased`, `tmpMotion` ou `tmpWheel` — qui stockent les coordonnées relatives ou absolues de l'évènement de pointage. Les Systèmes en aval qui dépendent d'une action de la souris observeront l'Entité `Pointer` par le biais de la Sélection `Pointers` — au cas où plusieurs pointeurs ont été instanciés. Ces Systèmes peuvent ensuite ajouter ou retirer des Composants pour les Systèmes suivants. Par exemple, `PointerClickSystem` uilise les Composants `tmpPressed/Released` pour détecter les clics de souris, et ajoute ensuite un Composant `tmpClick` avec la valeur de `target` à l'Entité lorsque c'est le cas. Enfin, les Composants temporaires (dont le nom commence par `tmp`) sont automatiquement supprimés des Entités à l'issue de chaque chaîne d'exécution des Systèmes.

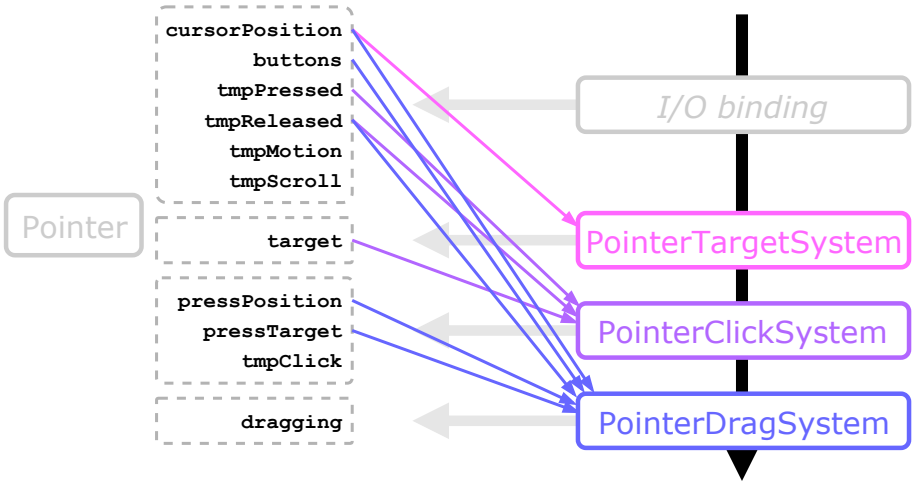


Figure 35 : Évolution des Composants de l'Entité `Pointer`, à travers les Systèmes réagissant aux évènements de pointage.

3.2.6 Un exemple pratique : implémentation du glisser-déposer

Notre exemple d'application repose principalement sur des techniques de manipulation directe (dessin, déplacement, modification de formes), et en particulier sur le glisser-déposer. Cette technique d'interaction est notoirement difficile à mettre en œuvre dans la plupart des bibliothèques d'interaction actuelles, et illustre bien les avantages de Polyphony pour l'implémentation de techniques d'interaction. Le glisser-déposer consiste à déplacer un objet et à le déposer sur un espace vide ou un autre objet. Dans le second cas, si les deux objets sont compatibles, une commande est exécutée, qui dépend des deux objets et de l'emplacement du dépôt.

Dans l'application de dessin, les objets graphiques peuvent être glissés et déposés sur les boutons de la barre d'outils, afin de modifier leur forme ou les supprimer : Le bouton/outil de réinitialisation de la toile supprime la forme déposée ; Le bouton/outil de rectangles la transforme en rectangle (en conservant ses dimensions) ; Le bouton/outil d'ellipses la transforme en ellipse. Les combinaisons possibles sont résumées dans le tableau 6.

Forme déplacée	**Bouton de dépôt**		
			
Rectangle	suppression	*sans effet*	→ Ellipse
Ellipse	suppression	→ Rectangle	*sans effet*

Tableau 6 : Matrice des combinaisons de glisser-déposer et commandes exécutées dans l'application

La séquence d'actions pour exécuter et modéliser le glisser-déposer est :

- le curseur survole l'objet à déplacer (un mécanisme de *feedforward* peut indiquer que l'objet peut être déplacé) ;
- un bouton est pressé au dessus de l'objet à déplacer ;
- le curseur se déplace et l'objet le suit ;
- le curseur survole un objet de dépôt (un *feedforward* peut suggérer que le dépôt aura un effet) ;
- le bouton est relâché (une commande dépendant de la nature des deux objets est exécutée s'ils sont compatibles).

Les difficultés associées à cette technique sont de plusieurs natures. Tout d'abord, déplacer un objet implique de le retirer de son emplacement actuel, ce qui peut casser les contraintes de positionnement locales. Ensuite, les différentes étapes nécessitent d'attendre des actions de l'utilisateur, et peuvent donc difficilement être exprimées par un bloc de code continu (elles sont généralement implémentées dans plusieurs *callbacks* à différents endroits du code). Enfin, les commandes et leur *feedforward* ne dépendent pas de l'un ou l'autre des objets, mais de la *combinaison* des deux. Le nombre de possibilités peut donc être très important puisqu'il dépend du produit du nombre d'objets qui peuvent être déplacés par le nombre d'objets destinataires. Selon le cas, les comportements résultants peuvent appartenir aux objets déplacés ou aux objets de dépôt.

Dans notre exemple d'application de dessin, un Système `PointerDragSystem` est dédié à la détection et à la gestion du glisser-déposer. Il dépend des Composants `cursorPosition`, `buttons` et `tmpReleased`, ainsi que `pressPosition` et `pressTarget` ajoutés par `PointerClickSystem`. Son code complet est présenté ci-dessous :

```
let PointerDragSystem = Entity(function PointerDragSystem() {
    for (let pointer of Pointers) {
        let position = pointer.cursorPosition
        let target = pointer.pressTarget
        let dragging = pointer.dragging // Entité en cours de déplacement
        if (!dragging && pointer.buttons[0] && target && target.draggable &&
          position.distance(pointer.pressPosition) > 10) {
            pointer.dragging = dragging = target
            dragging.draggedBy = pointer
            delete dragging.targetable // ne peut plus être pointé
        }
        if (dragging) {
            dragging.bounds.setX(position.x).setY(position.y)
            if (pointer.tmpReleased == BUTTON_PRIMARY) {
                delete pointer.dragging
                pointer.tmpDrop = dragging
                delete dragging.draggedBy
                dragging.targetable = true
            }
        }
    }
}, { runOn: POINTER_INPUT, order: 22 })
```

Au début d'une action de déplacement valide, ce Système ajoute les Composants `dragging` et `draggedBy`, respectivement au pointeur et à l'élément déplacé. Tant que le pointeur possède l'attribut `dragging`, il actualise la position de l'objet déplacé avec les coordonnées du curseur. Lorsque le Système détecte un relâchement de bouton, il supprime les Composants précédents et ajoute `tmpDrop` au pointeur (son Composant `target` contient déjà la cible du dépôt). Les Systèmes insérés après `PointerDragSystem` détecteront la technique du glisser-déposer en observant les Composants des Entités pointeurs. Dans notre application de dessin vectoriel, les Systèmes `ResetSystem`, `MoveSystem` et `DrawSystem` détectent et exécutent les actions spécifiées dans le tableau 6. De plus, le déplacement des formes sur le canevas utilise les Composants ajoutés par `PointerDragSystem` pour gérer cette action.

Ce séquencement des Systèmes illustre la composition des comportements avec ECS. L'outil de déplacement des formes est basé sur le Système de détection du glisser-déposer, `PointerDragSystem`, qui lui-même dépend de `PointerClickSystem` pour détecter la cible cliquée. Les dépendances entre Systèmes sont modélisées par leur ordre dans la chaîne d'exécution. Chaque Système insère des Composants sur les Entités des périphériques, en se basant sur les Composants insérés par les Systèmes précédents. Ainsi, construire des techniques d'interaction de plus haut niveau implique de les positionner en aval dans la chaîne d'exécution.

3.3 Architecture de Polyphony

Cette section est dédiée à la construction d'une architecture logicielle pour Polyphony. Le but de ce travail est de faciliter la *réplication* de Polyphony à des fins de recherche. En effet, nous considérons que le modèle ECS est destiné à évoluer, à mesure que des applications plus complexes seront construites avec. Polyphony étant un travail relativement jeune, nous manquons encore de recul pour juger de certains choix de conception. Polyphony étant un travail relativement préliminaire et exploratoire, certains de nos choix de conception devront encore être évalués et mis à l'épreuve de situations plus écologiques, afin de mieux en cerner les avantages et inconvénients. Il est donc important pour nous de documenter aussi clairement que possible son fonctionnement, afin que d'autres chercheurs puissent le reproduire et le faire évoluer.

Nous commençons par détailler la “philosophie” de la programmation avec ECS, à la lumière de ses différences avec la Programmation Orientée Objet (POO). Ensuite nous présentons *l'architecture* de Polyphony, c'est-à-dire l'ensemble des structures qui la composent et leurs interactions, qui rendent les applications interactives vis-à-vis des utilisateurs finaux. Enfin, comme il existe de nombreuses manières de concevoir les Systèmes et les Composants d'une interface graphique avec ECS, nous mettons en évidence ces choix et expliquons nos décisions pour Polyphony.

3.3.1 Fondements d'une architecture logicielle basée sur ECS

Les origines d'ECS sont étroitement liées au modèle objet. En effet, le domaine du développement de moteurs de jeux vidéo (hors *scripting*) est notoirement dominé par C++ et la programmation par objets. ECS y a émergé initialement comme une alternative à la POO, avec comme point de départ l'utilisation des données (Composants) comme interfaces entre programmeurs et designers/artistes. Comme décrit par Leonard [Leo99], « *Programmers specified the available properties and relations, and the*

interface used for editing, using a set of straightforward classes and structures. Using GUI tools, the designers specified the hierarchy and composition of game objects independent of the programming staff. In Thief there was no code-based game object hierarchy of any kind ».

Ensuite, les consoles de 7e génération (Playstation 3 et Xbox 360) ont contribué au développement et à la popularisation de la *conception orientée données* (*data-oriented design*) pour l'optimisation des jeux [Act14]. Ces consoles étant très sensibles à la fragmentation des accès mémoire, il fallait pouvoir contrôler précisément le stockage des différentes données, pour laquelle la POO était perçue comme limitante [Llo09]. Grâce à son utilisation centrale des données, ECS s'est développé de pair avec la conception orientée données, acquérant ensuite une distinction plus précise entre l'abstraction des Composants et leur stockage en pratique.

De par son origine comme alternative à la programmation par objets, ECS se comprend au mieux à partir de ses différences avec la POO. Nous les synthétisons dans le tableau 7, et les utilisons comme points de départ pour décrire l'architecture de Polyphony ci-dessous. Dans les descriptions qui suivent, les “éléments” désignent à la fois les objets et les Entités.

	POO	**ECS**
Liens entre éléments et données	l'objet *stocke* ses données	les données sont *attachées* aux Entités
Structure de données des éléments	structure/*record* (langages statiques), dictionnaire (langages dynamiques)	base de données (Composants)
Localisation du code	global (fonctions), local (méthodes d'objets)	global (Systèmes)
Modèle de flux d'exécution	messages (appels de méthodes) entre objets	séquence de Systèmes
Contrôle des accès aux données	variables privées/publiques (encapsulation)	Composants publiques
Réutilisation de code et variables	héritage (classes ou prototypes), composition (interfaces, mixins, ...)	composition (Systèmes et Composants)
“Nature” d'un élément	types des classes/prototypes dans sa chaîne d'héritage	ensemble des Descripteurs l'évaluant positivement à tout instant
Visibilité d'un élément non-global	en conservant une référence vers l'objet	en conservant une référence vers l'Entité ou en l'obtenant par une Sélection
Suppression d'un élément	implicite (portée lexicale, ramasse-miettes), explicite (C++)	explicite

Tableau 7 : Comparaison des caractéristiques discriminantes entre POO et ECS.

3.3.1.1 Liens entre éléments et données, et structure de données des éléments

Dans ECS, on sépare clairement l'attachement des données aux Entités, de leur stockage en pratique. Ainsi, si une Entité “possède” des Composants `bounds` et `backgroundColor`, il n'y a aucune raison de penser que ces deux données seront stockées conjointement dans une même structure ou dictionnaire. Pourtant, ces données seront bien accessibles avec une syntaxe d'accesseurs commune aux objets, `e.bounds` et `e.backgroundColor` — le point symbolise alors la relation d'appartenance “Composant *de* l'Entité”. Dans Polyphony nous avons choisi d'utiliser des dictionnaires, pour nous concentrer sur l'utilisation d'ECS à haut niveau plutôt que son

implémentation. Cependant, de nombreuses implémentations utilisent des bases de données sophistiquées dans le but d'améliorer la performance des accès mémoire, en particulier du point de vue des Systèmes (voir la section 3.4 pour une analyse d'implémentations existantes).

Il faut donc noter que la modification d'*un* Composant sur *une* Entité est une opération non triviale, en particulier dans les implémentations avec bases de données. Le choix du stockage des Composants est un compromis entre fréquence des modifications d'Entités et fréquence des accès par les Systèmes, les derniers étant typiquement privilégiés car normalement plus fréquents. Ce point explique que la description de l'architecture de Polyphony représente séparément Entités et Composants, alors que leur lien est en surface analogue à celui des objets et leurs données.

3.3.1.2 Localisation du code

Contrairement aux objets, les Entités ne possèdent pas de code propre. Le code de l'application est intégralement contenu dans les Systèmes, et leur exécution “fait vivre” l'interaction avec les utilisateurs. Dans une architecture d'interaction basée sur des objets comme MVC [Kra88], le fonctionnement d'une interface est modélisé par des échanges de messages (appels de méthodes) entre objets. Par exemple, tout objet visible à l'écran possèderait une méthode `onDraw`, le rafraîchissement de l'affichage consisterait à parcourir tous les objets visibles dans un ordre prédéterminé, et pour chacun appeler la méthode `onDraw`. On dit ainsi qu'un objet “sait s'afficher”.

Avec ECS, tout élément visible à l'écran possède des attributs (Composants) de couleur de fond, de couleur et épaisseur de bordure, d'image affichable, ou de texte. Ainsi, il ne “sait” pas s'afficher, mais compose son affichage grâce à ses attributs. Plusieurs Systèmes se chargent d'exécuter les instructions d'affichage :

- Un “Système de rendu d'arrière-plan” obtient toutes les Entités possédant une couleur de fond et une forme, et pour chacune dessine une forme pleine à l'écran.
- Un “Système de rendu de contours” obtient toutes les Entités possédant une bordure et une forme, et pour chacune dessine un contour de forme à l'écran.
- Un “Système de rendu d'images” obtient toutes les Entités possédant une image à afficher, et pour chacune dessine leur image à l'écran.
- Enfin, un “Système de rendu de texte” obtient toutes les Entités possédant un attribut de texte, une police d'affichage et une forme délimitante, et pour chacune dessine le texte à l'écran à l'intérieur de la forme délimitante.

Dans certains cas un comportement peut ne pas être composable à l'aide des Systèmes existants. Par exemple, une Entité peut nécessiter le dessin d'une figure en 3 dimensions. Que ce comportement soit spécifique à *une* Entité, ou réutilisable par plusieurs Entités, il sera toujours implémenté avec un nouveau Système (plutôt qu'une méthode attachée à chaque Entité). Avec ce mécanisme, chaque “type” de comportement est normalement contenu dans un Système. En effet, lorsqu'un Système implémente déjà un comportement donné, le fait qu'il soit accessible publiquement promeut son utilisation. Il est tout à fait possible que les développeurs ne conçoivent pas un Système pour qu'il soit réutilisé, mais nous conjecturons que le modèle ECS encourage la réutilisabilité dans le programme (sans toutefois l'imposer).

3.3.1.3 Modèle de flux d'exécution

L'approche de conception orientée données fait qu'avec ECS on se représente une application interactive comme un *pipeline* transformant continuellement des données d'interaction en retours utilisateurs (voir figure 36). Les Systèmes sont les étages de ce pipeline. Leur ordre est normalement fixe, ou change rarement dans le temps. Les données qu'ils traitent sont les Composants des Entités visibles et interactives, ainsi que des Entités des périphériques. Comme chaque Système effectue généralement une opération spécifique à un ensemble d'Entités, il se prête souvent bien à des optimisations de parallélisation (ex. *multithreading* sur CPU, ou rendu sur la carte graphique).

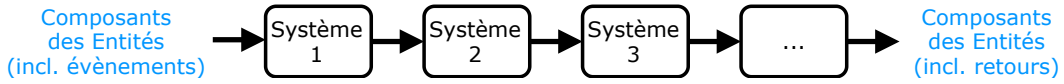


Figure 36 : Schématisation du *pipeline* d'exécution des Systèmes dans Polyphony, qui transforme les données d'interaction en entrée en données de retours utilisateurs en sortie.

Certaines implémentations comme Unity [Uni19] font de ce *pipeline* un graphe plutôt qu'une chaîne linéaire, afin de permettre la parallélisation de certains traitements. Cependant le modèle d'une chaîne reste commun à la majorité des implémentations d'ECS, et nous nous en sommes contentés dans ce travail.

3.3.1.4 Contrôle des accès aux données

En POO, de nombreux langages supportent l'*encapsulation* des données des objets — c'est-à-dire que seul l'objet propriétaire peut accéder à ses données. L'encapsulation peut faire partie intégrante du langage (ex. les slots privés de Smalltalk), être supportée de manière optionnelle (ex. les mots clefs `protected`/`private` de Java), ou être une pratique recommandée (ex. le préfixe __ en Python). Sa conséquence principale est qu'il faut "demander" à un objet l'accès à une de ses données, et qu'il est *responsable* de tout traitement fait avec ses données. Dans un modèle comme PAC [Cou87] ou MVC [Kra88], les objets contenant les données d'intérêt (respectivement l'Abstraction et le Modèle) occupent ainsi une position centrale dans les descriptions d'architecture, les autres facettes ayant pour rôles de leur transmettre les données.

Dans Polyphony, les Composants sont les supports du stockage des données, et occupent une position secondaire. De même, les Entités servent uniquement à retrouver des Composants en commun. Ce sont les Systèmes qui occupent la position centrale, et interagissent avec tous les autres acteurs de l'architecture.

3.3.1.5 Réutilisation de code et variables

Dans ECS les Composants sont de simples structures, dont les champs sont prédéfinis et indivisibles. Par exemple, si une Entité possède une valeur d'intensité rouge d'arrière-plan, alors elle possèdera généralement aussi des valeurs d'intensité verte et bleue, formant le composant `backgroundColor`. Les Composants sont ainsi le modèle de réutilisation de variables d'ECS (tout comme les structures du C). Ils permettent de raisonner à plus haut niveau que les variables

atomiques des Entités — qui peuvent en outre être très nombreuses. Dans la plupart des implémentations, chaque Composant est attaché à une seule Entité, cependant certaines implémentation permettent d'attacher un Composant à plusieurs Entités. Par leur attachement dynamique aux Entités qui met en oeuvre l'ajout de comportements, ils sont analogues aux interfaces et *mixins* en POO, en étant cependant contraints à **ne définir que des données**.

La mise à disposition de code qui puisse être réutilisé se fait par les Systèmes. Chaque Système est un bloc de code, qui *matérialise* un comportement du programme. Ainsi, la détection des clics de souris est un Système, de même que l'affichage d'images à l'écran, ou la sérialisation d'un groupe d'Entités en texte. Tous les Systèmes ne sont pas nécessairement des comportements destinés à être réutilisés dans le programme. Par exemple, dans une barre d'outils, le déclenchement des commandes pour chaque clic sur un bouton sera typiquement implémenté dans un Système spécifique à la barre d'outils. ECS décourage donc l'utilisation de *callbacks* dans l'application, qui sont reconnus pour fragmenter la logique du programme, et en compliquent la maintenance [Mye91]. Ce regroupement des comportements se fait au prix d'un éclatement du point de vue des Entités. En effet, pour énumérer les comportements observables d'une Entité, il faut énumérer ses Composants, et pour chacun repérer les Systèmes qui ont une influence dessus. Le choix des Composants disponibles est donc important, car ils doivent rendre *implicites* les comportements qui seront observés pour les Entités qui les acquièrent. Ils forment ainsi un ensemble *cohérent* d'attributs composables, qui s'ils sont bien choisis permettront aux développeurs d'anticiper les comportements observables des Entités à partir de leurs Composants.

3.3.1.6 Nature des éléments

En POO l'objet qui a servi à initialiser une instance d'objet (sa classe ou son prototype) définit son *type*. Ce type caractérise la nature même de l'objet, par exemple une voiture aura le type `Car`, et ce dernier est lui-même un objet, qui définit les propriétés intrinsèques d'une voiture comme `height` ou `nbDoors`. Le type est utilisé pour vérifier (statiquement ou à l'exécution) qu'un objet possède des variables et méthodes attendues. Par exemple, une fonction définie en Java par `int open_door(Car)` s'attend à opérer sur une voiture, donc un appel avec un entier comme `open_door(42)` déclenchera une erreur de type lors de la compilation.

Avec ECS les Entités ne possèdent aucune propriété intrinsèque, c'est-à-dire qui soit toujours présente. La nature d'une Entité est une notion variable dans le temps, qui change à mesure que des Composants sont ajoutés et retirés de l'Entité. La fabrique ayant instancié une Entité ne peut pas présager de sa nature plus tard, et ne peut donc pas être utilisé comme un type. L'association d'un ou plusieurs types à une Entité est donc nécessairement dynamique, c'est-à-dire qu'ils doivent être évalués chaque fois à l'exécution, et que leur validité court tant qu'aucune modification de Composants n'intervient. C'est par l'intermédiaire des Descripteurs qu'on teste la *validité* d'une Entité pour une condition booléenne donnée.

3.3.1.7 Visibilité d'un élément non-global, et suppression d'un élément

Comme en POO, on accède à une Entité à partir du moment où on possède une référence vers celle-ci, et le fait de posséder une référence permet d'accéder librement à l'Entité, sans restriction sur l'origine de l'accès. Avec ECS, un mécanisme supplémentaire (la Sélection), permet d'obtenir les

références d'Entités répondant à un Descripteur donné. Ce mécanisme est analogue aux sélecteurs de CSS [W3C96]. Dans Polyphony il est utilisé au coeur de l'architecture d'interaction, pour permettre aux Systèmes d'énumérer les Entités sur lesquelles opérer.

En pratique, ce principe favorise une structuration moins forte de l'application. La plupart des frameworks orientés objets se basent en effet sur la construction d'un *arbre de scène* [Bed00, Lec03, Huo04]. Cette structure de données contient tous les éléments interactifs de l'application. Elle est utilisée à la fois pour énumérer les éléments interactifs (par parcours récursif), et pour représenter des relations de parenté, utiles aux contraintes de positionnement et à la propagation de propriétés [Bed00]. Dans Polyphony, l'arbre de scène est utilisé uniquement pour les relations de parenté, et pour propager des informations de style (opacité, alignement de texte, etc.). Toutes les Entités n'en font pas nécessairement partie (y compris celles visibles et interactives), d'où l'absence d'arbre de scène dans la description de l'architecture.

La contrepartie de ce principe est que contrairement à la POO, on ne peut plus induire qu'un objet est hors de portée si aucune référence ne pointe vers lui. Avec ECS il est normal qu'une Entité ne soit référencée par aucune autre, mais qu'elle soit néanmoins toujours visible et interactive. Les Entités doivent donc être supprimées explicitement. L'invalidation des références (en les assignant à `null`, ou en retirant les Composants d'autres Entités qui les référencent) est du ressort du programmeur dans Polyphony, cependant il est envisageable dans le futur d'automatiser cette opération.

3.3.2 Description de l'architecture

Dans cette partie, nous présentons les éléments constitutifs de Polyphony, ainsi que les relations qu'ils entretiennent entre eux. Cette description de l'architecture est destinée à en donner une vue d'ensemble, ainsi qu'à mettre en lumière l'enchaînement des étapes d'exécution du programme.

Le principe de base de Polyphony est synthétisé en figure 37. Lorsqu'un Système est activé durant la chaîne d'exécution des Systèmes, il effectue plusieurs types d'opérations :

- Il récupère les listes d'Entités et de périphériques sur lesquels opérer, par l'intermédiaire de Sélections (non représentées sur la figure).
- Il lit et combine les Composants des Entités avec les périphériques. Par exemple, la combinaison des bornes d'une Entité avec la position d'une souris crée une information de ciblage de la souris vers l'Entité.
- Enfin, il modifie éventuellement les Composants des Entités et périphériques énumérés, afin de propager les données pour les Systèmes suivants.

Dans les descriptions qui suivent, nous formulons les *services* nécessaires au fonctionnement de Polyphony, et les illustrons avec leur syntaxe en code JavaScript. Les descriptions de cette section doivent permettre à quiconque de réimplémenter la première moitié de bas-niveau de Polyphony, en JavaScript ou tout autre langage.

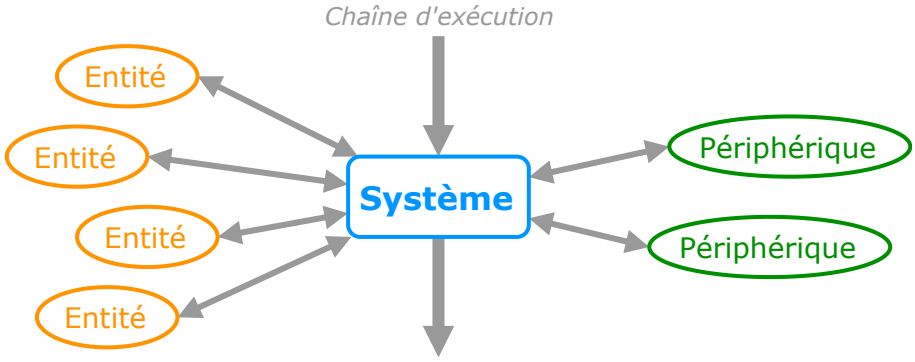


Figure 37 : Illustration du fonctionnement d'un Système dans la chaîne d'exécution de Polyphony.

3.3.2.1 Entités et Systèmes

Les Entités sont le coeur de Polyphony. Elles représentent tous les éléments actifs et interactifs de l'application, du point de vue des utilisateurs. Dans Polyphony, chaque Entité se comporte comme un dictionnaire associant des noms à des références de Composants. Une Entité peut être créée à partir d'une fonction, auquel cas elle est exécutable et représente un Système. Cette fonction peut exister sous la forme d'un Composant spécial, ou à l'aide d'un type d'objet spécifique du langage (dans notre cas le type natif `function`). Les services relatifs aux Entités et Systèmes sont :

Création d'une Entité — `let e = Entity(obj, fct_params)`
: Une Entité est instanciée, et sa référence est stockée dans la liste des Entités actives. Un objet peut éventuellement être passé en premier paramètre, auquel cas il est utilisé pour fournir des Composants initiaux à l'Entité. Cet objet peut aussi être une fonction, auquel cas un second paramètre fournit les Composants initiaux pour cette Entité. La référence de l'Entité créée est aussi ajoutée à un ensemble (`Set`) interne des Entités modifiées, qui servira à mettre à jour les Sélections avant l'exécution de Chaque Système. Les Sélections seront notifiées de ces modifications à l'issue de l'exécution de ce Système.

Association/Remplacement d'un Composant à une Entité — `e.comp = reference`
: Un lien est créé entre une Entité et un Composant donnés. Chaque Entité peut être associée à plusieurs Composants, mais un Composant n'est associé qu'à une seule Entité. Ainsi, lorsqu'une des variables d'un Composant est modifiée, l'Entité à laquelle il est attaché est automatiquement ajoutée à la liste des Entités modifiées. Une exception à cette règle concerne les Composants *immutables* (dont les variables internes ne peuvent pas être modifiées), qui peuvent être partagés entre plusieurs Entités. C'est le cas par exemple pour les couleurs, dont les structures sous-jacentes sont ainsi réutilisées. Chaque Composant est en fait une association entre un nom et un emplacement (non typé), dont toute modification (ajout, remplacement, modification interne, suppression) entraîne l'ajout de l'Entité à la liste des Entités modifiées. Lorsque le nom du Composant est préfixé par `tmp`, le lien est *temporaire*, c'est-à-dire qu'il sera détruit à l'issue de la chaîne d'exécution courante des Systèmes. Ce type de Composants permet de faire transiter des informations des Systèmes en amont vers les Systèmes en aval, en stockant ces informations sur des Entités (comme `Pointer` et `Keyboard`). Enfin la référence de l'Entité associée est ajoutée à l'ensemble interne des Entités modifiées.

Lecture d'un Composant d'une Entité — `let reference = e.comp`

Le lien entre une Entité et un Composant est retrouvé, et la référence vers le second est renvoyée. S'il n'y a pas de tel lien défini, une valeur invalide est renvoyée (`undefined` en JavaScript).

Vérification de l'association d'un Composant avec une Entité — `if ('comp' in e)`

L'expression renvoie un booléen indiquant si l'Entité est attachée à un Composant avec ce nom. Cette opération n'est pas strictement nécessaire dans une implémentation de Polyphony, car elle peut être remplacée par `if (e.comp != undefined)`.

Itération sur les Composants d'une Entité — `for (let [c, r] of Object.entries(a))`

Un bloc de code est exécuté autant de fois qu'il y a de Composants attachés à une Entité, deux variables (`c` et `r`) renseignant à chaque itération le nom du Composant itéré ainsi que sa valeur. Cette opération est utile en particulier pour la sérialisation d'une Entité en texte, lorsqu'il faut débugger le programme ou bien enregistrer son état courant dans un fichier.

Suppression d'un Composant d'une Entité — `delete e.comp`

Le lien entre une Entité et un Composant est détruit, et la structure de données du second est effacée de la mémoire du programme (si aucune autre Entité n'y réfère, dans le cas des Composants immutables). Si aucun Composant de ce nom n'était défini pour l'Entité, l'opération n'a aucun effet et ne renvoie pas d'erreur. La référence de l'Entité est ajoutée à l'ensemble interne des Entités modifiées.

Suppression d'une Entité — `e.delete()`

Tous les Composants liés à une Entité sont supprimés, et cette Entité est retirée de la liste des Entités actives. Elle est aussi ajoutée à un ensemble interne des Entités supprimées. Les Sélections seront notifiées de ce changement lors de l'exécution du prochain Système. Si des références vers l'Entité à supprimer existent encore dans l'application, elles sont conservées (l'Entité pointée est alors une coquille vide). Il serait aussi envisageable d'invalider automatiquement ces références, tel que décrit par Kedia [Ked17].

Exécution d'un Système — `e(...)`

Toutes les Sélections sont d'abord mises à jour avec les ensembles des Entités modifiées et supprimées. Ces ensembles sont ensuite réinitialisés dans leur état vide. L'Entité qu'on tente d'exécuter est normalement associée à une fonction (stockée comme un Composant caché), qui est exécutée en relayant les éventuels paramètres de l'appel de fonction. L'absence d'une telle fonction déclenche une erreur lors de l'exécution du programme.

Exécution du Méta-Système — `MetaSystem(eventType)`

Le Méta-Système est un Système particulier, dont la fonction est d'ordonner (selon le Composant `order`) et d'exécuter les autres Systèmes. Il utilise une Sélection lui permettant d'itérer sur tous les Systèmes (à l'exception de lui-même). Cette Sélection étant mise à jour à chaque exécution de Système, une copie en est faite au préalable afin de ne pas risquer de la modifier pendant qu'on itère dessus. Le Méta-Système reçoit en argument le type d'évènement déclencheur lui permettant de sélectionner les Systèmes à inclure dans la chaîne d'exécutions (en fonction de leurs Composants `runOn`). À la fin de son exécution, les Composants temporaires sont automatiquement supprimés.

3.3.2.2 Composants et environnement

Les Composants décrivent les attributs que les Entités peuvent acquérir (comme `shape` ou `backgroundColor`), ainsi que leurs capacités (comme `actionable` ou `draggable`). Dans Polyphony, le Composant désigne une structure attachée à une Entité, plutôt que le type de donnée duquel il est instancié. L'environnement, par l'intermédiaire des *bindings* avec le système d'exploitation, manipule les Composants des Entités matérialisant les périphériques d'interaction (souris, clavier, et écran), au début et à la fin de chaque chaîne d'exécution des Systèmes. Les services relatifs aux Composants et à l'environnement sont :

Création d'un type de Composant — `function Comp(i) { this.i = i }`
: Un type global et permanent est créé, qui s'apparente à une déclaration de *structure* du langage hôte. Cette structure définit un nombre fixe de variables, dans un ordre déterminé. Chaque variable possède un nom, un type si le langage hôte le permet, et est accessible par tous sans restriction. La définition d'un nouveau type de Composant implique l'existence d'une syntaxe permettant d'instancier de nouveaux Composants.

Instanciation d'un Composant pour une Entité — `e.comp = new Comp(...)`
: Un Composant est créé, et sa mémoire est initialisée avec les valeurs fournies en arguments (plus d'éventuelles variables calculées à partir des arguments). Un lien est immédiatement créé entre ce Composant et l'Entité.

Modification d'une valeur d'un Composant — `e.comp.i = val`
: Le Composant voit une de ses variables en mémoire recevoir une nouvelle valeur. Par le lien qui le lie à une Entité, celle-ci est automatiquement ajoutée à l'ensemble interne des Entités modifiées, afin que les Sélections puissent réagir à ce changement.

Sérialisation un Composant en code — `serialize(reference)`
: Il s'agit de transformer un Composant existant en une chaîne de caractères, telle que l'interprétation de cette chaîne comme du code source génère le même Composant initial. Cette opération n'est pas strictement nécessaire à l'implémentation de Polyphony, cependant elle nous permet de charger et sauvegarder les Entités d'une scène en code JavaScript, sans recourir à un format externe comme XML.

Actualisation des Composants des Entités périphériques
: Cette opération dépend des fonctions logicielles utilisées pour récupérer les évènements du système d'exploitation. Lorsqu'un évènement externe se produit (action sur la souris, le clavier, interruption du programme), le type de l'évènement est d'abord testé pour actualiser les bons Composants de chaque Entité périphérique (ex. `pointer.cursorPosition` pour un mouvement de souris). Ensuite, le Méta-Système est lancé, en lui passant en paramètre le type d'évènement ayant déclenché l'actualisation.

Initialisation des éléments de l'environnement
: Cette opération correspond au démarrage de l'application. Les *bindings* avec le système d'exploitation sont d'abord initialisés s'il y a lieu. Ensuite une fenêtre graphique est créée, ainsi que les Entités prédéfinies `pointer`, `keyboard`, et `view`. Il se peut aussi que ces Entités soient créées par *hot-plug*, c'est-à-dire après leur branchement alors que l'application a déjà démarré. C'est le cas notamment avec l'utilisation de libpointing [Cas11]. Enfin, le Méta-Système est lancé une première fois pour générer un premier affichage à l'écran des Entités déjà présentes.

3.3.2.3 Descripteurs et Sélections

Les Sélections sont la première interface entre les Systèmes et les Entités. Elles permettent aux Systèmes de récupérer les identifiants des Entités sur lesquelles opérer, avant d'utiliser ces identifiants pour récupérer les Composants. Une Sélection est un ensemble d'Entités basé sur une condition booléenne (Descripteur), dont le contenu est régulièrement mis à jour pour inclure *toutes* les Entités validant la condition, et *uniquement* ces Entités. Les services relatifs aux Descripteurs et Sélections sont :

Création d'un Descripteur — `let desc = (e) => 'comp' in e && e.comp < extVal`
: Une référence vers une fonction est renvoyée, qui prend en argument une Entité et renvoie un booléen. C'est donc une condition programmable, qui peut se baser sur les Composants attachés, leurs valeurs, ou une combinaison incluant des valeurs externes.

Interrogation d'un Descripteur — `if (desc(e)) {...}`
: L'exécution de la fonction référencée par le Descripteur renvoie un booléen, qui permet de vérifier à un instant donné si une Entité appartient ou non à une certaine catégorie. Cette opération est la base de la construction et de la mise à jour des Sélections.

Création d'un Sélecteur — `let sel = new Selection(desc, comp)`
: Une Sélection globale et permanente est créée, à partir d'un Descripteur fourni en argument. Une relation binaire d'ordre peut être fournie en second argument (fonction *Entité* × *Entité* → *nombre*), auquel cas la Sélection est garantie d'être toujours triée selon cette relation. Le Descripteur n'est pas modifiable après création de la Sélection, cependant il est publiquement accessible en lecture.

Mise à jour des Entités pour une Sélection — `sel.update(modified, deleted)`
: Une Sélection est notifiée des listes d'Entités ayant été modifiées et supprimées depuis la dernière exécution d'un Système. Elle insère les Entités validant pour la première fois son Descripteur, conserve les Entités l'ayant déjà validé, retire les Entités ne le validant plus, et ignore les Entités ne le validant toujours pas. Les Entités de la liste `deleted` sont systématiquement retirées, quel que soit le résultat du Descripteur.

Itération sur les Entités d'une Sélection — `for (let e of sel) {...}`
: Un bloc de code est exécuté autant de fois qu'il y a d'Entités dans une Sélection, une variable (`e`) renseignant à chaque itération la référence de cette Entité. Si une relation d'ordre existe pour la Sélection, elle est triée avant l'exécution de la boucle. Cette opération permet à un Système d'exécuter un comportement donné sur chaque Entité, indépendamment des autres.

Accès aléatoire à toute Entité d'une Sélection — `let e = sel[n]`
: L'Entité en `n`-ième position dans la Sélection est renvoyée. Plus généralement, toute Sélection est utilisable comme un tableau, à l'exception des opérations de modification du tableau. Cet accès permet à un Système d'exécuter un comportement transverse à plusieurs Entités, pour lequel l'itération en séquence serait fastidieuse. La résolution globale de contraintes de positionnement linéaires en est un exemple, puisqu'il s'agit de considérer l'ensemble des contraintes comme un problème d'Optimisation Linéaire, et de le résoudre globalement [Bor97].

De nombreuses implémentations d'ECS proposent des algèbres de combinaison des Sélections, pour par exemple “itérer sur les entités qui sont dans S1 et S2, mais pas dans S3”. Ces algèbres correspondent généralement aux opérateurs booléens *ET*, *OU*, et *NON*. Or en pratique, ils sont déjà implémentés dans le langage JavaScript, avec les opérateurs `&&`, `||`, et `!`. Nous avons donc conçu les descripteurs comme des fonctions (plutôt que des listes de Composants), afin de réutiliser l'algèbre de composition intégrée, et d'être cohérents avec le langage. De plus, nous souhaitions avoir le plus de flexibilité pour exprimer des descripteurs, car nous ne pouvions pas anticiper les opérations algébriques qui seraient les plus utilisées en pratique. En effet, la majorité des implémentations d'ECS expriment les descripteurs de base (hors combinaisons algébriques) comme des listes de Composants, qui doivent tous être possédés par une Entité pour l'inclure. Ces descripteurs implémentent en fait l'opération de base *ET*, et impliquent donc que c'est l'opération qui sera la plus utilisée. Comme nous voulions observer avant de prendre une telle orientation, nous avons choisi l'option la plus flexible, avec des descripteurs spécifiés en code.

3.3.3 Choix de conception des Systèmes, Composants et périphériques

Cette partie illustre le deuxième niveau de Polyphony (représenté en figure 32), construit par dessus l'implémentation d'ECS présentée dans la section précédente. Ce niveau est constitué des éléments réutilisables fournis par Polyphony, pour aider les programmeurs à construire rapidement des interfaces complètes. Le choix des Systèmes proposés, des Composants disponibles ainsi que l'usage des Sélections, résultent de compromis qu'il est important d'expliquer afin de soutenir les futures itérations de boîtes à outils d'IHM basées sur ECS.

3.3.3.1 Choix des Systèmes

La conception des Systèmes dans Polyphony résulte d'un compromis entre comportements simples et comportements complexes. Prenons par exemple le dessin des bordures de widgets (rectangulaires ou polygonales) dans l'application. Nous avons quatre alternatives, classées de la plus simple à la plus complexe :

- Créer un Système dessinant des segments, avec lequel une bordure est formée de plusieurs Entités positionnées en polygone.
- Créer un Système dessinant des rectangles, qui conviendra pour la majorité des besoins.
- Créer un Système dessinant des formes géométriques arbitraires, pour lequel le dessin d'une bordure revient à fournir un tableau de points.
- Créer un Système dessinant des widgets de base (fond, bordure, image, et texte), pour lequel le dessin d'une bordure sera un sous-ensemble de ses capacités.

Avec des types de comportements simples, il est facile de les assembler de façon créative. Avec un Système dessinant des segments, on pourrait ainsi changer la couleur d'un des bords, indépendamment des autres, ce qui n'est pas réalisable avec les Systèmes alternatifs plus complexes (s'ils se basent sur une seule couleur). À l'opposé avec des types de comportements complexes, on définit globalement moins de Systèmes, donc la chaîne d'exécution des Systèmes est plus courte, et il est plus facile d'en avoir une vue d'ensemble. De façon générale, nous voulons faciliter la programmation d'applications complexes, et rendre la structure des Systèmes la plus simple possible, donc nous favorisons les Systèmes complexes et en faible nombre.

Dans l'exemple ci-dessus, le premier Système dessinant un segment par Entité est trop simple. Pour dessiner une bordure rectangulaire complète (sans “trous”), il nécessite de créer quatre Entités, et de les positionner relativement à l'Entité du widget qu'on souhaite dessiner. De plus, ce Système n'apporte pas un comportement de “grande valeur”, car il peut être recréé rapidement si un développeur avait besoin de dessiner des lignes.

Le second Système est plus avancé, mais stéréotypé. En effet, chaque widget qui nécessite une bordure n'aura qu'à acquérir quelques Composants (sans nécessiter de nouvelles Entités), donc il sera simple et rapide d'ajouter des bordures rectangulaires aux widgets. Cependant, des chercheurs expérimentant la conception de contrôles non-rectangulaires ne pourront pas utiliser ce Système, et devront en créer un autre.

Le troisième Système est plus avancé et moins stéréotypé, mais complexe pour des usages simples. Des bordures pour contrôles non-rectangulaires seraient simples à créer, et leurs coordonnées seraient incluses dans un Composant, donc ne nécessiteraient pas de multiplier les Entités. Pourtant, elles rendent plus complexe le dessin de bordures pour les contrôles rectangulaires, qui devraient alors spécifier quatre points.

Enfin, le quatrième Système est trop complexe. En effet, bien qu'il soit particulièrement adapté pour dessiner des contrôles standards, ce Système est “monolithique” et ne fait aucun usage des capacités de composition d'ECS. La création de nouveaux types de contrôles se ferait en multipliant les Entités, chacune utilisant un sous-ensemble des capacités du Système. De plus ce type de Système est le plus difficile à remplacer, donc contribue à la cristallisation des types de contrôles disponibles, et plus généralement de l'expérience utilisateur.

L'exemple que nous avons choisi est volontairement simple, de manière à bien illustrer les choix qui s'offrent aux programmeurs pour concevoir des Systèmes selon leurs besoins. En pratique il est possible de faire un Système qui couvre plusieurs usages simultanément (ex. qui permette de spécifier une bordure complète, ou chaque côté séparément). Lors de l'élaboration d'un Système, nous pourrions être tentés de multiplier ces options, et d'inclure des besoins hypothétiques futurs, pour faciliter le prototypage de nouvelles interactions. Or l'extrapolation des besoins crée des Systèmes plus complexes, car il existe de nombreuses pistes de développements, dont certaines ne se développeront jamais. Dans l'exemple des bordures ci-dessus, de telles pistes seraient : des bordures polygonales, des bordures en courbes paramétriques, des bordures d'épaisseurs/couleurs non-uniformes, ou encore des bordures animées dans le temps. À partir de cette remarque, notre première recommandation est de **supporter uniquement les usages contemporains**. Le modèle ECS est conçu pour faciliter la création de nouveaux Composants et Systèmes, ce qui est préférable à des Systèmes complexes supportant des besoins peu communs. Enfin, de la même manière qu'on quantifie le nombre d'éléments que peut retenir la mémoire de travail (*empan mnésique*) autour de 7 [Mil56], nous préférons réduire le nombre de Systèmes pour permettre aux développeurs d'en avoir une “vue d'ensemble” (voire de les représenter graphiquement). Il faut alors identifier des comportements indépendants (ex. le dessin de bordures, d'arrière-plans, de texte, ou d'images), et les répartir en un nombre raisonnable. Le choix final est un compromis qui dépend du contexte, et n'a pas une unique solution. Nous recommandons donc de **diviser les Systèmes en un faible nombre de comportements indépendants**.

3.3.3.2 Choix des Composants

La fonction première des Composants est d'aggréger des données. Ces données peuvent toujours être divisées en des valeurs atomiques — par exemple, une coordonnée `x`, une coordonnée `y`, une largeur `width`, une hauteur `height`. D'un autre côté, ces données peuvent aussi être regroupées dans des ensembles partageant un sens commun — par exemple, `x`, `y`, `width` et `height` forment un rectangle, ou des bornes. À l'extrême, les données peuvent être regroupées dans de grands ensembles avec un sens large — comme les classes de *widgets de base* définies dans la plupart des frameworks. Il existe donc un équilibre à trouver entre Composants légers (voire atomiques), et Composants lourds.

Les programmeurs interagissent directement avec les Composants, pour gérer les comportements ajoutés aux diverses Entités. L'enjeu de ce choix de conception est donc la facilité qu'auront les programmeurs à s'abstraire du bas-niveau, et à manipuler peu de Composants à la fois. Si les Composants sont trop lourds (comme des widgets complets), alors le mécanisme de composition d'ECS ne peut pas être exploité à son maximum, puisqu'on ajouterait des comportements en activant des options dans un Composant, plutôt qu'en acquérant de nouveaux Composants. Si les Composants sont trop légers, alors les différents comportements nécessitent beaucoup de Composants, et les programmeurs doivent écrire plus de code pour les ajouter aux Entités.

La performance de l'application influence aussi ce choix de conception. En faveur des Composants lourds, l'accès à une donnée d'un Composant (en lecture et écriture) est plus rapide que l'accès à un Composant d'une Entité, à cause de la flexibilité du stockage des Composants d'une Entité. En faveur des Composants légers, le regroupement en mémoire de toutes les données utilisées par un même Système favorise l'utilisation des caches du processeur, donc les données inutilisées d'un Composant diminuent globalement la performance de l'application.

Pour le choix des Composants de Polyphony, nous avons donc appliqué ces recommandations :

- **les données requises par *un* Système devraient appartenir à *un* seul Composant**
- **les données partagées par *plusieurs* Systèmes devraient être réparties pour que chaque Système dépendant d'un Composant en utilise *toutes* les variables**

En exemple d'application de ces recommandations, nous avons observé que tous les éléments affichables de l'interface possédaient nécessairement les variables `x`, `y`, `width` et `height` — y compris ceux non-rectangulaires qui possèdent alors un rectangle englobant. Seul le pointeur système possède uniquement des coordonnées `x` et `y`, cependant ses comportements ne sont pas destinés à être composés avec ceux des widgets de l'interface. En conséquence, Polyphony propose un Composant `Bounds` incluant les quatre variables.

3.3.3.3 Choix liés aux Sélections

En programmation par objets (et Entités), les objets se “connaissent” les uns les autres en stockant des références (généralement des adresses en mémoire) :

- un objet en “connaît” un autre en stockant une référence vers celui-ci
- un objet en “connaît” plusieurs autres en stockant un tableau de références vers ceux-ci

Avec l'utilisation des Sélections, de nouvelles manières de référer aux objets apparaissent :

- on peut référer à une Entité qu'on sait être le premier élément d'une Sélection (ex : le premier pointeur, `Pointers[0]`)
- on peut référer à plusieurs Entités en leur attribuant un Composant les différenciant des autres Entités, et en les récupérant par une Sélection

Dans Polyphony, toutes ces manières de référer aux Entités coexistent, et offrent souvent plusieurs manières de faire. Dans l'exemple de notre application de dessin, pour référer aux boutons activables mutuellement exclusifs (les outils de déplacement, rectangles, et ellipses), faut-il conserver un tableau de références, ou attribuer aux boutons un même Composant `toggleGroup` ? De même, pour définir les relations de parenté dans un arbre de scène, chaque noeud stocke-t-il un tableau d'enfants, ou les enfants portent-ils un Composant `parent` ?

Pour répondre à ces questions, il faut rappeler que les Sélections sont des structures coûteuses en performance, car Polyphony les maintient à jour en leur présentant *chaque* Entité qui est modifiée. Il faut donc éviter de les utiliser abusivement. Dans le premier cas, on utilisera une sélection `ToggleGroups` qui réfère à toutes les Entités possédant un Composant `toggleGroup`, et on filtrera lors de leur énumération les Entités avec un même `toggleGroup` (car on sait qu'elles sont peu nombreuses). Ce choix permet en outre d'ajouter facilement un nouveau bouton à un groupe, en lui attribuant le même Composant `toggleGroup`. Dans le second cas, on suppose que de grands arbres de scène risquent d'être utilisés. Avec une Sélection de toutes les Entités ayant un Composant `parent`, on aurait beaucoup d'Entités à ignorer pour énumérer celles avec un `parent` donné. De plus, l'ordre des enfants d'un noeud est important (qui ne serait pas garanti dans une Sélection sans critère de tri). On utilisera donc plutôt un tableau d'enfants sur chaque noeud parent de l'arbre. Ainsi, pour différencier les deux cas, nous recommandons d'**utiliser une Sélection lorsque le nombre d'Entités à écarter est faible**, car chaque Entité ignorée dans une Sélection est beaucoup plus coûteuse qu'une Entité ignorée dans un tableau.

Enfin, lors de la détection de techniques d'interaction sur les périphériques, il se pose la question des Entités auxquelles attribuer les Composants générés. Prenons par exemple la technique du glisser-déposer : lors du dépôt d'un objet sur un autre, un *évènement* de dépôt temporaire est généré, qui doit être communiqué aux Systèmes en aval dans la chaîne d'exécution. Les Composants à créer peuvent être stockés :

- sur l'objet receveur, qui indique quel objet il a reçu (`tmpDropOf`), et éventuellement quel pointeur l'a déposé (`tmpDropBy`)
- sur l'objet déposé, qui indique sur quel objet il est relâché (`tmpDropOn`), et éventuellement quel pointeur l'a déposé (`tmpDropBy`)
- sur le pointeur à l'origine du dépôt, qui indique quel objet a été déplacé (`tmpDropOf`), et sur quel objet il l'a déposé (`tmpDropOn`)
- sur une combinaison des trois

L'interaction entre les périphériques et les Entités induit la possibilité de stocker l'information d'un côté ou de l'autre (ou des deux côtés). Dans le cas du glisser-déposer, on suppose que *beaucoup* d'Entités seront déplaçables, *beaucoup* pourront recevoir des dépôts, mais qu'il existera *peu* de pointeurs (généralement un seul). Il sera donc judicieux d'ajouter les Composants sur le pointeur, car

les Systèmes en aval n'auront qu'à énumérer la Sélection `Pointers` pour détecter l'action de dépôt (et écarteront peu d'Entités). Ainsi, nous recommandons de **stocker les Composants temporaires sur les Entités les moins nombreuses** (c'est-à-dire accessibles à partir des Sélections les plus réduites).

3.4 Implémentation du modèle Entité-Composant-Système

Dans la section précédente, nous avons décrit les principes essentiels d'ECS, ainsi que l'architecture de Polyphony à haut niveau. Nous n'y avons *pas* décrit nos choix de conception à bas niveau (Les Entités sont-elles des objets ? Où sont stockés les Composants ? etc.), afin de faciliter l'application de Polyphony à tout type de langage de programmation (orienté-objet, impératif, statique ou dynamique). La présente section est dédiée à l'implémentation de Polyphony. Nous commençons par étudier trois implémentations majeures d'ECS, afin d'identifier un espace de conception où nous positionner. Ensuite nous étudions les différences entre les domaines du Jeu Vidéo et des IHM, qui justifient nos écarts par rapport aux implémentations existantes. Enfin, nous présentons en détails les choix d'implémentation de Polyphony.

3.4.1 Analyse d'implémentations existantes

Il existe de nombreux frameworks basés sur ECS, chacun avec des variations des mêmes concepts. Cette multiplicité rend difficile le choix d'une implémentation plutôt qu'une autre, pour quiconque ayant besoin d'utiliser ce modèle de programmation. Pour mieux informer documenter et renseigner l'utilisation d'ECS comme modèle de programmation, et en particulier dans le cas des IHM et techniques d'interaction, nous avons construit un *espace de conception* des implémentations d'ECS. Il est basé sur l'étude de 3 frameworks parmi les plus référencés [Mar14] et mieux documentés, ainsi que notre expérience dans le développement d'une nouvelle variante dédiée à la programmation d'interactions, *Polyphony*. Tous les frameworks que nous avons trouvés ciblent le développement de jeux vidéo, et nous n'en avons trouvé aucun dédié aux interfaces graphiques.

Nous avons sélectionné :

- **Artemis** (Java), une des premières implémentations historiques d'ECS. La version originale n'étant plus documentée et plus maintenue, nous avons sélectionné son successeur le plus populaire, Artemis-odb [Pap18].
- **Entitas** (C#), une interprétation populaire d'ECS pour Unity, qui inclut un méta-langage et un préprocesseur pour C# [Sch18]. Sa base d'utilisateurs ainsi que sa documentation complète en font une implémentation majeure d'ECS.
- **GameplayKit** (Swift), un des nombreux frameworks inspirés d'ECS qui implémentent les comportements directement sur les Composants, plutôt que dans des Systèmes distincts — avec par exemple CORGI (C++) et Nez (C#) [Goo15, Pri18]. Ils sont parfois qualifiés d'*Entity-Component systems* (systèmes à Entités-Composants), ce qui alimente la confusion avec le modèle ECS original. Unity (C#) implémentait aussi cette approche initialement, et s'est converti plus tard au modèle original, mais n'est pas encore suffisamment documenté [Uni17].

L'analyse présentée dans le tableau 8 n'est pas exhaustive, mais met en lumière les choix de conception sur lesquels les interprétations d'ECS diffèrent principalement. Nous avons aussi inclus nos propres choix pour l'implémentation de Polyphony, et les discutons à la fin de la section. Les

critères de notre analyse sont :

- **Représentation des Entités** — comment les Entités sont matérialisées dans le framework
- **Représentation des Composants** — comment les Composants sont matérialisés, et définis par les programmeurs
- **Représentation des Systèmes** — comment les Systèmes sont matérialisés, et définis par les programmeurs
- **Structuration du Contexte** — comment l'environnement est matérialisé dans le framework, pour instancier de nouveaux Systèmes/Composants/Entités, et pour obtenir des Sélections
- **Sélection d'Entités** — comment les Entités sont regroupées à partir de leurs Composants
- **Changements d'états** — quels mécanismes sont disponibles pour réagir aux changements d'états
- **Stockage des Composants** — comment et où sont stockés les Composants
- **Ordonnancement des Systèmes** — quel mécanisme permet d'ordonner les Systèmes dans la chaîne d'exécution
- **Fabriques d'Entités** — comment définit-on des modèles d'Entités à réutiliser
- **Extensions de langage** — le framework implémente-t-il des extensions de *méta-langage*, c-à-d de nouvelles syntaxes non permises dans le langage hôte

Framework (Langage)	Artemis-odb (Java)	Entitas (C#)	GameplayKit (Swift)	Polyphony (JavaScript)
Représentation des Entités	entier, ou entier dans un objet	objet	objet	objet
Représentation des Composants	sous-classes de `Component` avec valeurs par défaut	sous-classes de `IComponent`	sous-classes de `GKComponent` avec valeurs par défaut	tout objet
Représentation des Systèmes	sous-classes de `BaseSystem`, attachées à une Sélection, exécution périodique	sous-classes de `ISystem`, attachées à une Sélection, exécution périodique ou unitaire	méthode de Composant `updateWithDeltaTime:`	Entité initialisée avec une fonction
Structuration du Contexte	objet `World`	objet `Context`	arbre de scène	*global*
Sélection d'Entités	par liste de Composants, avec algèbre *all/one/none*	par liste de Composants, avec algèbre *all/one/none*	par liste de Composants, ou condition programmable	par condition programmable
Changements d'états	*listeners* sur Sélections	*listeners* sur Sélections	*non explicitement supportés*	Composants temporaires
Stockage des Composants	par Composant, avec table Entité→Valeur	par Entité, avec *slots* prédéterminés	par Entité, avec table Composant→Valeur	comme propriétés d'objets natifs
Ordonnancement des Systèmes	dynamiquement par priorité	dynamiquement par insertion dans une liste	statiquement	dynamiquement par priorité
Fabriques d'Entités	objet *fabrique*, chargement de fichier	chargement de fichier	*non explicitement supporté*	fonction *fabrique*
Extensions de langage	post-traitement de l'exécutable compilé	pré-compilation d'une surcouche du langage	*aucune*	syntaxe des Entités par objets *Proxy*

Tableau 8 : Analyse de trois variantes d'ECS et de Polyphony selon notre espace de conception.

3.4.2 Polyphony : choix de conception

Les principales dimensions de conception pour implémenter ECS ayant été présentées, nous devons à présent les appliquer à la programmation d'IHM. Il convient d'abord d'étudier l'utilisation d'un paradigme de composition dans les jeux vidéo, et de justifier son application à la construction d'interfaces graphiques et d'interactions. Nous soulignons ensuite les différences entre le développement de jeux vidéos et d'interfaces graphiques, puis discutons nos choix au regard de ces différences.

3.4.2.1 Adaptation d'ECS au contexte IHM

Toutes les boîtes à outils que nous avons étudiées sont dédiées exclusivement au développement de jeux vidéos. Or ce type d'applications a des besoins très particuliers, parfois similaires, mais souvent différents de la programmation d'interactions, que nous détaillons ici.

Multiples déclencheurs de Systèmes
L'état du jeu est généralement mis à jour par une unique chaîne de Systèmes, avec un *tickrate* fixe (souvent 60Hz, une fréquence de rafraîchissement commune des écrans). Comme les interfaces graphiques sont généralement mises à jour en réponse à des sources d'évènements multiples et variées, nous avons introduit de multiples déclencheurs de la chaîne de Systèmes, et le filtrage des Systèmes à exécuter en fonction de l'évènement déclencheur.

Arbre de scène des Entités
Les éléments d'un jeu ont des relations peu structurées entre eux. Ils se déplacent, apparaissent, disparaissent, et nécessitent peu de relations entre eux — mise à part l'utilisation de structures de partitionnement pour optimiser l'affichage [Bis98]. D'un autre côté, les éléments d'une interface graphique ont des relations d'ordre d'affichage, de positionnement relatif, et de styles hérités, qui s'expriment au mieux par des arbres de scène.

Entités matérialisant les périphériques d'interaction
Nombre de jeux obéissent au principe *un joueur par machine*, et supportent un seul couple clavier/souris (mais parfois plusieurs manettes). Dans le cadre de la recherche et du prototypage de techniques d'interaction, nous voulons supporter une plus grande variété de périphériques, en multiples exemplaires, et supporter leur ajout/retrait à l'exécution. Nous avons introduit les Entités *périphériques* pour permettre une gestion poussée des entrées.

Définition de Composants temporaires
Dans un jeu, la plupart des changements d'états observables sont liés soit aux actions du joueur, soit à l'environnement. Dans le premier cas, un mécanisme global d'évènements sert généralement à signaler les changements d'états. Dans le second cas, la plupart des implémentations que nous avons étudiées intègrent des *listeners* (donc des callbacks *en plus* des Systèmes) sur les Sélections, afin d'observer les changements de groupes d'Entités. Dans les interfaces graphiques, les changements d'états sont courants, et on y associe de nombreux types d'actions — techniques d'interaction, commandes, positionnement. Pour matérialiser et manipuler de nombreux types de changements d'état *sans introduire de listeners*, nous avons introduit l'utilisation de Composants temporaires, principalement pour les Entités périphériques.

Représentation des Systèmes en Entités

Le *pipeline* d'exécution des différents Systèmes est généralement fixe, ce pourquoi les implementations d'ECS les ordonnent dans des simples listes. Pour faciliter le prototypage et la manipulation des Systèmes à l'exécution, nous avons adopté une approche récursive et représentons les Systèmes par des Entités. Le déclenchement des Systèmes est ainsi rendu plus flexible, grâce à la gestion des déclencheurs par des Composants, et à la mise en place d'un Méta-Système qui ordonne et exécute les Systèmes.

Abandon des objets de Contexte

Les jeux sont souvent organisés en “niveaux”, indépendants les uns des autres, et déclenchent des séquences de chargement entre eux. Ils stockent les Entités, Composants, et Systèmes relatifs à chaque niveau dans des “objets de Contexte”, qui permettent de séparer clairement les niveaux. À l'opposé, la navigation dans les interfaces graphiques implique des allers-retours entre vues, onglets ou modes, avec de nombreux éléments en commun, et en évitant les séquences de chargement qui pourraient interrompre l'interaction. À cet effet, nous avons écarté l'utilisation d'objets de Contexte , préférant la flexibilité des Composants pour partager des éléments entre vues.

3.4.2.2 Implémentation des Entités et Composants

Dans Polyphony nous représentons les Entités par des objets natifs, encapsulés dans des objets *Proxy* de JavaScript. Les objets Proxy nous permettent d'intercepter toutes les opérations natives qu'ils recoivent, ce qui nous permet de proposer des Entités accessibles avec la syntaxe native de JavaScript [Ecm15], mais se comportent différemment. Ces opérations sont :

- `e.c` renvoie la valeur du Composant `c` pour l'Entité `e` (ou `undefined`) ;
- `e.c = v` associe le Composant `c` de valeur `v` à l'Entité `e` ;
- `'c' in e` renvoie `true` si et seulement si `e` est associée à `c` ;
- `delete e.c` dissocie `c` de l'Entité `e`.

Les raisons de la réutilisation d'une syntaxe d'objets sont principalement car les développeurs en IHM sont très familiers du modèle objet, donc de la syntaxe `obj.property` d'accès à un champ d'un objet. Nous réutilisons donc un concept connu et bien maîtrisé. L'autre raison de ce choix est que la représentation des Entités par des entiers (dans Artemis) présume qu'on va les stocker dans des tableaux, ce qui casse l'abstraction entre les Entités et leur stockage effectif. Enfin, comme nous ne pouvions pas anticiper les usages qui allaient être faits des Entités, nous avons choisi de permettre un maximum de flexibilité dans l'évolution de notre modèle, en faisant des Entités de simples dictionnaires. De même nous n'avons imposé aucune contrainte sur les Composants, pour expérimenter différents choix de conception : utilisation des méthodes *setter* sur les Composants, distinction des types et noms des Composants (ex. `backgroundColor` pour un type `Color`, alors que les implémentations font souvent coïncider les deux), réutilisation de Composants immutables (ex. `Color`), ou encore stockage dynamique dans les Composants pour que les Systèmes ajoutent des variables privées.

Pour ce qui est du stockage effectif des Composants, les implémentations que nous avons étudiées mettent en œuvre des bases de données sophistiquées, afin d'optimiser spécifiquement certains types de traitements. Ainsi, Artemis optimise les traitements des Systèmes, car à partir de chaque

Composant, les Systèmes peuvent obtenir toutes les valeurs qui les intéressent. À l'opposé, Entitas stocke les Composants ensemble au sein d'une même Entité, et optimise ainsi les traitements effectués à la suite sur une même Entité. Dans notre cas, nous avons considéré que l'optimisation du stockage des objets est déjà un problème majeur pour JavaScript, donc nous avons choisi de réutiliser les objets natifs.

Pour ce qui est des fabriques d'Entités, dans les premières versions de Polyphony nous initialisions chaque Entité par du code écrit à la main. Lorsque nous avons pu dessiner des formes dans notre application d'exemple, nous avons implémenté l'enregistrement et le chargement des Entités du canevas. Plutôt que d'importer un format externe et en compliquer l'apprentissage, nous avons choisi de *sérialiser* les Entités en code JavaScript, et de “charger” un tel fichier en l'exécutant simplement.

3.4.2.3 Implémentation des Systèmes

Dans Polyphony, les Systèmes sont matérialisés par des fonctions de JavaScript, encapsulées comme des Entités dans des objets *Proxy*. JavaScript permet en effet d'assigner des propriétés à des fonctions, tout en leur attribuant un type natif distinct (`function` plutôt qu'`object`). Nous pouvons donc distinguer une Entité exécutable d'une Entité qui ne l'est pas, tout en supportant la gestion des Composants d'une manière commune.

Nous avons initialement choisi d'implémenter les Systèmes par de simples fonctions (plutôt que des méthodes d'objets) pour nous rapprocher d'un concept de “fonctions avec attributs” (qui stockeraient des variables). Notre idée était alors d'*augmenter* la déclaraction d'une fonction avec des initialisations d'attributs, qui renseigneraient en particulier les déclencheurs extérieurs de la fonction. Nous pouvions ainsi nous passer de l'enregistrement explicite de ces fonctions comme des *callbacks*, puisque le système se chargerait automatiquement de rediriger les évènements externes vers les fonctions avec les attributs correspondants.

Contrairement à Entitas, Polyphony supporte uniquement les Systèmes exécutés périodiquement, et contrairement à Artemis et Entitas, les Systèmes ne sont pas attachés explicitement à une Sélection. Dans ces frameworks, il est courant que chaque Système opère sur une Sélection d'Entités. Ils favorisent donc ces usages en offrant des Systèmes qui implémentent déjà l'itération sur une Sélection, et ne nécessitent qu'une fonction qui sera exécutée pour chaque Entité itérée. Dans notre cas, la combinaison des widgets (Entités) et des périphériques (Entités) fait que de nombreux Systèmes itèrent sur plusieurs Sélection. De plus, nous souhaitions ne pas limiter l'exécution d'un Système à une fonction exécutée par Entité, mais permettre des traitements de groupes si nécessaire. Nous n'avons donc pas attaché explicitement les Systèmes aux Sélections qu'ils traitent. Ce point pourrait changer à l'avenir, car il est difficile actuellement pour une Entité d'énumérer les Systèmes qui sont actifs pour elle, ce qui peut compliquer leur utilisation par les développeurs.

Pour l'ordonnancement des Systèmes, nous avons souhaité pouvoir *représenter* visuellement la chaîne des Systèmes, voire à terme proposer une interface d'édition. Ce point nous a incités à limiter le nombre de Systèmes, et à opter pour une chaîne linéaire plutôt qu'un graphe de dépendances qui aurait été plus complexe à représenter. Étant donné le faible nombre de Systèmes, il était envisageable de les ordonner à la main, plutôt qu'exprimer des dépendances explicites entre Systèmes, et les

ordonner avec un parcours dans un graphe de dépendances. Les Systèmes possèdent donc un Composant `order` pour ordonner manuellement leur liste. Cependant, le support des Composants sur les Systèmes n'exclut pas l'ajout de dépendances entre Systèmes ultérieurement.

3.4.2.4 Implémentation du Contexte et des Sélections

Artemis-odb et Entitas définissent chacun une classe pour stocker les Entités actives et matérialiser le Contexte. Ils créent de nouvelles Entités, leur lient des Composants, et enregistrent de nouveaux Systèmes en appelant des méthodes sur l'objet Contexte. Ils permettent également d'instancier plusieurs de ces Contextes pour représenter plusieurs *mondes*, par exemple pour charger le niveau suivant ou gérer un second joueur sur la même machine. Polyphony à l'inverse ne fournit pas de Contexte explicite, et rend tout global, sans qu'il soit nécessaire de conserver une référence contextuelle. A la place, on peut gérer plusieurs *mondes* en ajoutant un Composant `mondes` à chaque Entité, afin de gérer les Contextes parents de chaque Entité.

Comme expliqué en section 3.3.2.3, nous avons choisi d'implémenter chaque Descripteur par une fonction (plutôt qu'une liste de Composants à posséder). De plus nous ne fournissons pas d'algèbre de composition des Sélections, afin de réutiliser de façon cohérence l'algèbre booléenne du JavaScript, et pour ne pas présupposer de l'utilisation qu'en auront les développeurs. Jusqu'à présent, nos besoins se sont principalement résumés à des listes de Composants, quelques rares exceptions (des conditions sur les valeurs internes des Composants) pouvant se ramener à des Composants. Il est donc envisageable à l'avenir d'intégrer Polyphony avec une implémentation existante d'ECS, pour bénéficier d'une exécution plus optimisée.

3.4.2.5 Intégration avec le langage JavaScript

Comme discuté dans les Essentiels d'Interaction, nous avons voulu souligner le rapport étroit entre un framework et le langage de programmation qu'on utilise. Dans Polyphony, ce rapport se manifeste par la réutilisation de la syntaxe d'accès aux objets, pour accéder aux Entités. En particulier, l'initialisation des Entités réutilise et étend celle des objets de JavaScript :

```
let e = Entity({
    depth: 0,
    bounds: new Bounds(0, 0, 100, 50),
    shape: SHAPE_RECTANGLE,
    backgroundColor: rgba(0, 0, 255),
})
```

La fonction `Entity` crée et renvoie un objet `Proxy` de JavaScript, qui intercepte tous les accès de lecture et écriture, et nous permettent de le faire se *comporter* comme une Entité. Par extension, la définition d'un arbre d'Entités se fait avec une déclaration récursive, que la syntaxe de déclaration des objets de JavaScript permet déjà :

```
let root = Entity({
    children: [
        Entity({
            depth: 0,
            bounds: new Bounds(0, 0, 100, 50),
            shape: SHAPE_RECTANGLE,
            backgroundColor: rgba(0, 0, 255),
        }),
        Entity({
            depth: 1,
            bounds: new Bounds(50, 20, 80, 50),
            shape: SHAPE_RECTANGLE,
            backgroundColor: rgba(255, 0, 0),
        }),
    ],
}
```

À titre d'illustration, nous avons synthétisé dans le tableau 9 les différences de syntaxe de Polyphony, entre JavaScript et une version antérieure en Java [Raf18]. Chacune de ces différences est permise par l'utilisation de métaprogrammation par l'objet `Proxy`. La différence majeure entre les deux versions est dans la lisibilité de la syntaxe, plus courte et moins ponctuée. Grâce à ce changement de langage, nous avons pu réduire la taille des applications utilisant Polyphony, et ainsi construire plus aisément des cas d'étude plus complexes.

	Java	JavaScript

Initialisation d'Entité	`Entity e = new Entity()`
	`  .add("property", value)`
	`  ...`
		`let e = Entity({`
		`  property: value,`
		`  ...})`

Lecture de propriété	`(Type)e.get("property")`	`e.property`
Écriture de propriété	`e.set("property", value)`	`e.property = value`
Test de propriété	`e.has("property")`	`'property' in e`
Ajout de propriété	`e.add("property", value)`	`e.property = value`
Suppression de propriété	`e.remove("property")`	`delete e.property`

Tableau 9 : Comparaison des syntaxes d'utilisation de Polyphony entre Java et JavaScript

Pour bénéficier d'une syntaxe “intégrée” à celle du langage JavaScript (c'est-à-dire qui réutilise et détourne des éléments de syntaxe, plutôt que de proposer une API par fonctions), nous avons fait des compromis dans les choix de conception de Polyphony. Ainsi, notre framework a été influencé en partie par certaines caractéristiques de JavaScript :

- L'implémentation des objets comme dictionnaires nous a naturellement incités à utiliser cette structure de données pour les Entités, plutôt qu'une base de données.
- L'absence de types pour les champs des objets nous a incités à ne pas en imposer a priori dans Polyphony, bien qu'à l'avenir nous n'excluons pas d'expérimenter leur utilisation.

- Une Entité ne "porte" pas son identifiant, à l'inverse de frameworks comme djnn [Mag14] ou le DOM [WHA15]. Les identifiants sont portés par les propriétés des objets ainsi que les variables du langage, qui *référencent* les Entités. Ainsi chaque Entité peut être référencée sous plusieurs noms. Cette caractéristique est intégrée dans JavaScript, et partagée par la majorité des langages de programmation, donc par cohérence nous l'avons adoptée dans Polyphony.

En pratique, Polyphony peut être implémenté dans de très nombreux langages. Bien que les choix de conception aient été *orientés* pour être hautement compatibles avec des langages orientés objet dynamiques, l'implémentation dans un langage moins compatible est possible. Le niveau de compatibilité d'un langage avec Polyphony dépendra du support de différentes caractéristiques. Pour clarifier ce dernier point, nous énumérons dans le tableau 10 les caractéristiques définissant cette compatibilité pour cinq langages impératifs.

JavaScript et Python ont la meilleure compatibilité, on peut y implémenter Polyphony en se basant sur les objets natifs du langage. Smalltalk ne permet pas d'étendre les variables d'une instance en particulier. Cependant il permet d'intercepter les accès aux propriétés (qui sont de simples envois de messages), donc d'implémenter les Entités comme des objets, mais avec des dictionnaires. Java et C ne permettent pas d'intercepter les lectures/écritures de propriétés, nous ne pouvons donc pas utiliser la syntaxe native d'accès aux objets (voir le tableau 9). Enfin, pour les langages qui ne supportent pas la déclaration et l'initialisation simultanées d'objets, on ne pourra pas initialiser les Entités comme les objets de façon récursive.

	JavaScript	Python	Java	Smalltalk	C
Dictionnaires	oui	oui	oui	oui	non
Déclaration/initialisation d'objet simultanées	oui	oui	non	non	non
Initialisation d'objets récursive	oui	tableaux	non	oui	oui
Typage	nominatif, dynamique	nominatif, dynamique	nominatif, statique	nominatif, dynamique	nominatif, statique
Références vers fonctions	objets	objets	objets	objets	pointeurs
Fonctions avec propriétés	oui	oui	non	non	non
Ajout/suppression de variable	oui	oui	non	non	non
Ajout/suppression de méthode	oui	verbeux	non	non	non
Itérateurs extensibles	oui	oui	oui	n/a	non
Interception d'accès à une propriété	oui	oui	non	n/a	non
Énumération de propriétés	oui	oui	oui	oui	non

Tableau 10 : Compatibilités de cinq langages pour l'implémentation de Polyphony

3.5 Conclusions

Nous avons discuté dans ce chapitre de la conception d'une bibliothèque d'IHM basée sur le modèle Entité-Composant-Système, et orientée vers le prototypage de nouvelles techniques d'interaction. Nous avons présenté l'état de l'art des bibliothèques liées au prototypage d'interfaces, et dégagé les opportunités de recherche qui ont motivé ce travail de thèse. Nous avons présenté la boîte à outils Polyphony, et illustré son utilisation à l'aide d'une application de dessin vectoriel. Nous avons ensuite détaillé l'architecture de Polyphony, en nous focalisant sur des descriptions de haut niveau

permettant de la répliquer dans d'autres contextes. Enfin, nous avons analysé les choix de conception de l'implémentation d'ECS à l'aide de trois variantes majeures, et expliqué nos propres choix pour la conception d'une boîte à outils pour l'IHM.

3.5.1 Contributions et limites

Nous discutons à présent des apports et inconvénients d'ECS et Polyphony à la programmation d'interfaces graphiques et d'interactions, à partir de notre expérience dans leur conception et leur implémentation. Lors de la conception de Polyphony, nous avons été confrontés à de nombreux problèmes d'implémentation pour lesquels les solutions choisies pouvaient fortement influencer l'utilisabilité de la boîte à outils. En particulier, nous aurions pu utiliser les patrons de conception courants, tels que les *listeners* ou les *delegates*. Au lieu de cela, nous avons essayé de garder le modèle ECS aussi *pur* que possible en ne le mêlant pas à d'autres paradigmes, afin de souligner ses forces et ses faiblesses dans le contexte des IHMs.

3.5.1.1 Application d'un modèle de composition à l'interaction

Les frameworks d'interaction s'appuient communément sur deux hiérarchies en arbres pour structurer le code et les données dans une application : un *arbre de types* (classes) pour partager et réutiliser le code, et un *arbre de scène* pour structurer l'interface. Les arbres de types propagent les méthodes et les variables de n'importe quel type d'objet à ses enfants, et permettent d'utiliser n'importe quel enfant là où un parent est attendu. C'est le polymorphisme des types, et c'est le mécanisme par lequel les comportements sont généralement partagés entre différents types de widgets. Les arbres de scènes sont utilisés pour structurer l'interface, et permettent le couplage de nombreux aspects des interfaces graphiques avec un seul arbre : ordre d'affichage, positionnement relatif, et propagation des styles. Certains travaux introduisent également un troisième *graphe d'interaction*, pour exprimer les dépendances entre les techniques d'interaction [Hud05, Huo04, Mye97].

Les jeux vidéo utilisent souvent les arbres de types, tandis que les éléments du jeu nécessitent généralement moins de structuration qu'un arbre de scène. Dans ce domaine, l'application d'un modèle de composition comme ECS s'est avérée utile, en partie parce que de nombreux comportements (visibilité, physicalité, contrôlabilité) peuvent être décorrélés et composés à volonté. Par exemple, on peut instancier un pot de fleurs qui soit visible, qui ne bloque pas physiquement le joueur, et qui soit inerte, mais on peut aussi donner au joueur le contrôle de cette plante, et lui attribuer une masse et une boîte de collision.

Pour les interfaces graphiques, de nombreux aspects peuvent être déconnectés des arbres de types et de scène, et donc bénéficier d'un modèle de composition. Dans notre application d'exemple, le parcours d'arbre typiquement effectué pendant la sélection à la souris (*picking*) et le rendu graphique des widgets, est synthétisé dans un seul système `DepthUpdateSystem`. De même, la composition de l'apparence graphique des widgets par des formes simples (polygones pleins, lignes, images, et texte) permet un rendu plus rapide en les envoyant en groupes à la carte graphique. Un autre exemple est la spécification de contraintes de positionnement élastiques entre widgets dans l'interface [Bad01], au lieu du positionnement relatif des widgets par rapport aux parents dans l'arbre de scène. Enfin, nous

avons montré avec l'exemple du glisser-déposer comment chaque technique d'interaction peut être composée à partir des résultats de techniques d'interaction plus simples (ciblage de widget, clic sur widget, etc.).

Avec Polyphony, le modèle de composition d'ECS remplace intégralement l'arbre des types, tout en s'appuyant partiellement sur un arbre de scène. Celui-ci nous est encore utile pour :

- référer explicitement à des enfants plutôt qu'en utilisant une Sélection (ex. les formes du canevas)
- exprimer des relations d'ordre d'affichage entre Entités
- propager des styles visuels entre entités (ex. police de caractères), bien que nous n'ayons pas encore implémenté cette fonctionnalité

À l'avenir, nous comptons expérimenter différentes alternatives, et nous cherchons toujours un bon compromis entre ces principes structurants.

3.5.1.2 Réutilisation non-hiérarchique

ECS met en œuvre un mécanisme puissant de réutilisation : il évite la *duplication de code* en permettant aux Entités de déléguer leurs comportements à des Systèmes, et il limite également la *duplication d'exécution* en exécutant chaque Système une fois pour toutes les Entités. La duplication d'exécution est fréquente pour les widgets effectuant des traitements périodiques dans des méthodes d'instance. En effet lorsque plusieurs widgets du même type partagent la même méthode, elle est exécutée autant de fois qu'il y a de widgets. Les Systèmes, d'autre part, exécutent un même traitement sur tout un ensemble d'Entités. Ils peuvent dupliquer l'exécution en traitant chaque entité à la fois, mais ils ont également la possibilité de les exécuter en parallèle lorsque cela est possible.

L'énumération des Entités intégrée à ECS permet de ne pas dépendre du parcours récursif d'un arbre de scène, et de choisir l'ordre dans lequel on les parcourt. Par exemple, on peut résoudre les contraintes de mise en page en utilisant un solveur d'optimisation linéaire tel que Cassowary [Bad01], plutôt que de propager des contraintes avec des messages entre entités [San93]. Dans les implémentations ECS, les Systèmes sont également utiles pour optimiser implicitement le rendu des composants graphiques en les envoyant par paquets (*batches*) à l'API graphique. De telles optimisations existent dans la plupart des frameworks d'interaction, mais elles nécessitent des gestionnaires d'affichage complexes, qui gèrent des caches d'affichage et compilent les multiples instructions de dessin en paquets pour l'API graphique.

Avec les Systèmes, tous les comportements sont séparés des éléments auxquels ils se rapportent (à l'inverse des méthodes d'instance ou des callbacks). Bien que ce soit utile pour des comportements partagés, qui sont écrits et exécutés une seule fois pour toutes les Entités, cela n'apporte aucun avantage pour les comportements individuels — écrits pour une Entité et jamais réutilisés. Des cas d'utilisation et des interfaces plus réalistes doivent être conçus et implémentés avec notre approche ECS avant que nous puissions tirer des conclusions fermes, mais nous pourrions envisager une approche mixte utilisant des systèmes pour les comportements partagés et des mécanismes existants pour les comportements individuels.

3.5.1.3 Dynamicité des interfaces

Avec les Descripteurs, on peut vérifier qu'une Entité respecte un protocole donné. En ce sens ils sont analogues à des *interfaces*, à la différence qu'ils sont définis dynamiquement, évalués dynamiquement (puisque les Entités peuvent changer en cours d'exécution), et ne se limitent pas aux Composants possédés (ce sont des conditions programmables). Avec les Sélecteurs, on peut obtenir une liste de toutes les Entités vérifiant un Descripteur. Ce mécanisme permet de découvrir des Entités en fonction de leurs capacités, et de les filtrer finement afin de sélectionner celles sur lesquelles agir.

Nous avons ainsi conçu des interfaces aussi flexibles que possible, afin de permettre l'ajout de nouveaux comportements, aussi bien à la compilation qu'à l'exécution. Ces comportements peuvent aussi bien s'appliquer à des Entités futures, qu'à des Entités déjà initialisées. Par ce mécanisme, nous voulons permettre la modification de toute application existante à des fins de recherche et de prototypage. Nous pensons par exemple à l'intégration d'ExposeHK [Mal13] à une application existante, qui consiste à afficher sous chaque bouton une *tooltip* avec son raccourci clavier, lors d'un appui sur la touche Cmd/Ctrl. L'implémentation d'une telle technique d'interaction nécessiterait d'abord d'énumérer tous les boutons, ce qui est réalisable à l'aide d'une Sélection sur les Entités possédant le Composant `clickable`. Nous énumérons ainsi les widgets qui sont des boutons par leur *caractéristique*, plutôt que par les noms de classe qui sont clickables.

Ensuite, il faudrait associer chaque bouton à la commande qu'il déclenche, et de même chaque raccourci clavier à la commande qu'il déclenche, afin d'associer ensuite les boutons aux raccourcis clavier. Si l'exécution d'une commande après un clic ou un raccourci clavier est gérée directement en code, alors il faudra inspecter le code des Systèmes pour retrouver cette association. Dans notre application de dessin, elle prendrait la forme suivante :

```
let saveButton = Button(new Image('icons/save.png'), 2, y += 34)
...
let SaveSystem = Entity(function() {
    if (saveButton.tmpClick || Keyboards[0].tmpShortcut === SDLK_s) {
        // sauvegarde du canevas
        ...
    }
}, { runOn: POINTER_INPUT | KEYBOARD_INPUT, order: 61 })
```

Alternativement, l'instruction `if` pourrait être séparée en deux instructions `if` appelant une même fonction `saveCanvas()`, ou le code de sauvegarde pourrait être dupliqué (ce qui est peu envisageable si la commande est complexe). Ici, ECS ne permet pas facilement de retrouver l'association entre boutons et clavier, car la relation de *causalité* n'est pas explicitée en un objet qui puisse être énuméré, comme peuvent l'être les *bindings* de djnn [Mag17], ou la classe `Action` de Swing. En leur absence, on devra inspecter le *bytecode* de chaque Système pour détecter les instructions `if`, ou comme les auteurs d'ExposeHK, d'instrumenter les appels de fonctions au niveau du système. Pour éviter d'avoir recours à ces techniques très complexes (et non portables entre systèmes), il est important d'inclure un niveau d'*indirection* dans le déclenchement de toute

commande, qui permette de retrouver l'association entre une commande et ses déclencheurs. Dans notre cas, il s'agirait d'ajouter des attributs à `SaveSystem`, et de déléguer son activation au Méta-Système (ou à un Système tiers) :

```
let SaveSystem = Entity(function() {
    // sauvegarde du canevas
    ...
}, { runOn: SHORTCUT_S, order: 61 })
```

Plus généralement, nous arguons que l'utilisation d'indirections dans les liens entre éléments (objets ou processus) contribue à la *flexibilité* et l'*observabilité* des applications. L'indirection pourrait donc être un principe de conception, applicable à toutes les applications interactives, et que nous envisageons d'étudier et de promouvoir à l'avenir.

3.5.1.4 Centralisation de la logique du programme

Les systèmes aident à centraliser la définition des interactions et des comportements d'une manière similaire aux machines à états de SwingStates [App06]. Le sélecteur d'outils de notre application de dessin illustre bien ce principe. Le Système `ToggleSystem` détecte les clics sur n'importe quelle Entité `togglable`, et désactive toutes les autres Entités avec le même `toggleGroup`. Ainsi, aucun des boutons de sélection d'outils n'implémente un callback `onClick` pour activer un outil et désactiver les autres. Ce comportement commun est géré dans un système dédié à plus haut niveau. De plus, l'ajout d'un nouvel outil ne nécessite que d'ajouter les Composants `togglable` et `toggleGroup` au nouveau bouton, pour que `ToggleSystem` le prenne en compte. Il n'est donc pas nécessaire d'ajouter de nouveaux callbacks ou de modifier le code de l'application à plusieurs endroits. Pour chaque outil, il est courant d'avoir du code à exécuter lors de l'activation ou la désactivation. Dans notre application, ce code est contenu dans le Système de chaque outil activable (`MoveTool` et `DrawTool`), qui détecte l'activation avec le Composant temporaire `tmpToggled == true`.

Un autre exemple est l'implémentation du glisser-déposer dont nous avons parlé précédemment. La logique de cette technique d'interaction est contenue dans un seul Système, malgré les nombreuses actions requises pour effectuer un glisser-déposer complet. Des techniques d'interaction multimodales pourraient également être implémentées, en déclenchant un même Système sur plusieurs types d'évènements, par exemple avec `runOn : MOUSE_INPUT | KEYBOARD_INPUT`.

Le regroupement de toutes les actions d'un type de comportement en un unique Système implique aussi que les traitements de plusieurs Systèmes ne peuvent pas s'entremêler. Or certaines opérations requièrent d'ordonner les traitements par Entités, plutôt que par type de comportement, comme le rendu graphique. Dans ce cas, le dessin des formes de fond, des bordures et des images sont des opérations distinctes exécutées dans l'ordre pour chaque Entité. Ces opérations doivent également être ordonnées entre Entités, c'est-à-dire dessiner l'image d'une Entité d'arrière-plan avant la bordure d'une Entité d'avant-plan. Ceci est illustré dans la figure 38, où le regroupement des processus dans des Systèmes réorganise leur exécution.

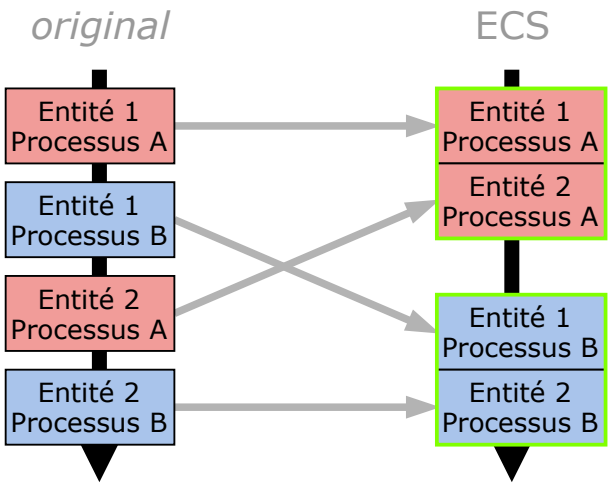


Figure 38 : Ordonnancement de deux processus avec deux Entités. Le regroupement des exécutions par type de processus altère l'ordre des exécutions.

La programmation avec les Systèmes nous oblige à rendre les processus réordonnés insensibles à l'ordre d'exécution. Dans le cas du rendu graphique, l'utilisation d'un *depth-buffer* permet de dessiner tous les éléments opaques dans un ordre quelconque. Pour les éléments avec transparence, il faut normalement dessiner les éléments d'arrière-plan avant ceux d'avant-plan. Notre solution est de rediriger les instructions de dessin de chaque élément transparent vers une texture privée, et de les dessiner dans l'ordre dans un dernier Système. L'application des principes d'ECS exige donc de réduire au minimum les relations d'ordre entre les entités.

3.5.1.5 Matérialisation persistante des périphériques

Dans Polyphony, les Entités des périphériques sont les supports du stockage de données relatives aux techniques d'interaction. Les différents Systèmes interprétant les techniques en séquence n'ont pas à se passer les données en arguments, au risque de perdre des données de Systèmes antérieurs. Toutes les données sont stockées sur les Entités périphériques, et permettent donc aux Systèmes d'accéder à la fois à des données de bas-niveau et de haut-niveau.

De plus avec ECS, les périphériques d'entrée sont caractérisés par leurs Composants : `cursorPosition` et `buttons` pour les souris, `bounds` et `origin` pour les vues, et `keyStates` pour les claviers. Ce principe permet d'abstraire les dispositifs tout en conservant leurs caractéristiques individuelles. Les données telles que la présence de boutons supplémentaires sur une souris sont conservées, mais simplement ignorées par tout système traitant des pointeurs à deux boutons.

Grâce à l'utilisation systématique des Sélections pour rassembler les Entités, les techniques d'interaction sont implicitement exposées à la possibilité d'utiliser de multiples périphériques. Bien que nous ne l'ayons pas démontré dans notre prototype, l'utilisation de plusieurs instances des mêmes périphériques d'entrée ou de sortie (physiques ou virtuels), ainsi que leur remplacement, est facilitée car elle est implicite, tant qu'ils présentent les bons Composants. Par exemple, un Système

implémentant une technique de pointage pourrait fonctionner avec n'importe quelle Entité qui fournit les composants `cursorPosition` et `buttons`, que ce soit une souris, une tablette ou un dispositif virtuel produisant ces données (par exemple, un “robot” qui rejoue des entrées).

En pratique, nous avons observé que les Composants se divisaient en deux catégories :

- **attributs** — qui ne sont que des propriétés des Entités, sont souvent partagées par plusieurs Systèmes, et n'apportent aucun comportement spécifique (ex. `bounds`, `keyStates`, `order`)
- **activateurs** — qui sont souvent requis par *un* Système, donc relatifs à un comportement spécifique (ex. `richText`, `targetable`, `focus`).

Lors de la conception de l'ensemble des Composants disponibles et des Systèmes correspondants, nous devons rendre l'activation de chaque Système pour chaque Entité *prévisible*. À cette fin, l'activation d'un Système par un seul Composant est de toute évidence le plus prévisible (que la combinaison de plusieurs Composants), d'où l'émergence des activateurs. Dans notre application de dessin, par exemple, `View` partage le Composant `bounds` avec d'autres types d'Entités (c'est un attribut). Son Composant activateur est ici `origin`, et c'est donc lui qui octroie la capacité à “représenter les Entités contenues dans un rectange donné”. L'association d'un activateur à chaque Système est ce qui permet aux développeurs de choisir les Systèmes qui vont influer sur une Entité. Si un activateur est partagé par deux Systèmes, il sera impossible de faire fonctionner l'un sans l'autre pour toute Entité.

De façon générale le modèle ECS de base ne possède pas de liens clairs entre les Systèmes et les Composants qu'ils utilisent. Leur séparation est pourtant importante, car elle permet de remplacer les Systèmes à tout moment, que ce soit pour optimiser leur implémentation ou l'augmenter. En suivant la description faite par Leonard [Leo99], les Composants seraient en fait une *spécification des comportements* (en particulier les activateurs), qui sont ensuite implémentés dans les Systèmes. Leur nommage est donc essentiel, car il doit informer le comportement acquis avec chacun, et épargne la nécessité d'une documentation complémentaire pour expliquer leurs rôles (bien que ce soit utile pour les comportements complexes). À l'avenir, nous envisageons donc de spécifier deux types de Composants, les *attributs* et les *comportements*, afin d'inciter les développeurs à nommer les seconds comme des comportements (ex. `clickable`, `draggable`, `drawableAs`). Ils pourraient bénéficier en outre de mots clés de déclaration, et de colorations syntaxiques pour les distinguer dans du code.

Conclusion

Durant ce travail de thèse, nous nous sommes intéressés à la programmation de prototypes de recherche en IHM, que nous avons cherché à rendre moins difficile et moins stéréotypée. Nous avons commencé par étudier l'activité de programmation dans ce contexte, et avons pour cela réalisé une série d'interviews de chercheurs ayant développé de nouvelles techniques d'interaction. Les observations ont mené à trois classifications, des problèmes rencontrés par les participants, des outils qu'ils avaient jugés utiles, et des techniques qu'ils avaient employées pour surmonter leurs difficultés. Nous avons ensuite proposé un questionnaire en ligne pour évaluer la pertinence des classes de problèmes et techniques auprès d'un plus grand nombre de participants en IHM, et étudier les critères selon lesquels ils choisissent les bibliothèques. Nous en avons déduit que (i) les chercheurs utilisent principalement des frameworks dans le cadre de leur travail, (ii) ils choisissent en priorité les plus documentés et les plus utilisés, (iii) les problèmes de documentation sont les plus fréquents et les problèmes de fiabilité les plus critiques, et (iv) les caractéristiques supportant le mieux leurs techniques avancées sont l'extensibilité, la réutilisabilité, et la transparence des frameworks.

Nous avons choisi d'explorer des contributions logicielles pour améliorer la programmation de prototypes de recherche en IHM, et avons au préalable étudié des principes de conception pour guider notre travail. Leur synthèse a donné lieu à trois *Essentiels d'Interaction*, qui clarifient et différencient notre démarche de recherche par rapport à l'état de l'art. Nous cherchons ainsi à (i) **augmenter des capacités d'orchestration des comportements interactifs**, (ii) **fournir un environnement d'entrées/sorties utilisateurs complet pour toute application**, et (iii) **mettre à profit des conventions de programmation et possibilités d'intégration aux langages**.

Nous avons illustré l'application des Essentiels d'Interaction avec deux réalisations. La première est une extension du langage Smalltalk pour exprimer les transitions animées par adjonction d'une durée aux appels de fonctions. Elle réduit le code nécessaire pour exprimer une animation par rapport à l'état de l'art, et met en commun dans le langage la gestion des animations autrefois spécifique à chaque framework. Ce travail nous a permis de formuler le troisième Essentiel d'Interaction, en analysant notre travail en lien étroit avec le langage de programmation. La deuxième réalisation est un framework d'interaction, qui applique le modèle Entité-Composant-Système issu du jeu vidéo, à la programmation d'interfaces graphiques. Il répond principalement aux besoins de réutilisabilité (grâce au mécanisme de composition intégré au modèle), et d'extensibilité (en facilitant la création de nouvelles techniques d'interaction et types de widgets).

Avec ces travaux, nous avons cherché à améliorer les modèles de programmation des frameworks et langages, pour le prototypage de nouvelles interactions. Nous avons proposé des concepts de bas niveau applicables aux langages, et facilitant l'implémentation de frameworks d'interaction (animation de fonctions, typage et énumération dynamique d'objets à partir de leurs champs). Les travaux de l'état de l'art ont principalement proposé des modèles de réutilisation basés sur des arbres de types (ex. HTML, Qt, JavaFX), ou de scène (ex. Jazz, UBit), mais qui fragmentent l'exécution du code (le flux

d'exécution “saute” fréquemment en mémoire, le rendant difficile à suivre et visualiser). Nous proposons un modèle de réutilisation basé sur la composition, qui organise le flux d'exécution de façon linéaire, et rend le fonctionnement interne de l'application transparent, pour les développeurs souhaitant le manipuler. De plus, l'état de l'art s'est principalement basé sur une propagation des entrées utilisateurs en couches, qui communiquent les données de couche à couche, sans nécessairement conserver les données brutes. Nous proposons une représentation des périphériques d'interaction en objets extensibles et persistents, sur lesquels les différentes techniques d'interaction viennent ajouter des données de plus haut niveau (voire des données temporaires pour les transitions d'états). Cette représentation centralise les données d'interaction, et évite aux développeurs d'avoir à les aggréger depuis différentes couches. Enfin, de nombreux travaux de l'état de l'art ont proposé des syntaxes de programmation réduites (par rapport à l'utilisation des fonctions d'une API), grâce à une intégration partielle aux langages de programmation. Nous avons appuyé et approfondi cette démarche, par le développement de concepts intégrables aux langages, et l'utilisation de métaprogrammation pour les implémenter.

Limites

Nous avons privilégié le développement de nouveaux concepts qui puissent faciliter le développement d'interfaces moins stéréotypées, plutôt que des outils qui facilitent l'utilisation de concepts existants. De plus, nous avons consacré beaucoup d'efforts à raffiner leur conception afin d'explorer et démontrer au mieux la pertinence de nos Essentiels d'Interaction. En contrepartie, la faible maturité de ces concepts ne nous a pas permis de les tester en conditions réelles. Dans le cas de l'animation de fonctions, il conviendrait d'adapter ce concept à d'autres langages, pour encourager son intégration plus cohérente avec leur syntaxe (ex. `object.setProperty(target) during 2s`). Dans le cas de Polyphony, il conviendrait de valider nos travaux avec des interfaces plus complexes, mettant en œuvre la gestion du positionnement, des styles visuels, des périphériques multiples et changeant dynamiquement, ou encore du rendu optimisé sur carte graphique. Nous pourrions alors diffuser Polyphony auprès de chercheurs en IHM, et recueillir leurs retours, ainsi que les forces et faiblesses du modèle pour le prototypage de nouvelles interactions.

En outre, dans ce travail nous n'avons pas abordé les problèmes de documentation soulevés à l'issue des interviews et questionnaires. Bien que ce problème soit le plus fréquent pour les utilisateurs de frameworks d'interaction, il bénéficie de travaux de recherche abondants. Ceux-ci nous laissent penser que la solution ne peut pas se résumer à un nouvel outil de documentation, et qu'elle pourrait très bien dépasser le cadre d'un travail de thèse. Enfin, bien que nous ayons observé la faible utilisation des frameworks issus de la recherche pour prototyper de nouvelles interactions, nous n'avons pas exploré la piste de leur promotion auprès des chercheurs. En effet, la complexité de bas niveau des systèmes informatiques vis-à-vis de l'interaction nous a incités à concentrer nos efforts pour la réduire, plutôt que promouvoir des outils qui la cachent. Nous espérons ainsi accélérer le développement de futures boîtes à outils, et par là contribuer au développement de nouvelles interactions avec les machines.

Promouvoir Polyphony pour les utilisateurs d'ECS

En amont de notre travail sur Polyphony, nous avons relevé des discussions sur des forums de développeurs de jeux vidéo, qui montraient un intérêt pour l'application d'ECS à la programmation d'interfaces graphiques. En effet, alors qu'ils appliquent déjà ECS à l'architecture de leurs jeux, de nombreux développeurs souhaitent l'appliquer aux interfaces des mêmes jeux, principalement pour ne pas le mélanger avec un modèle orienté objet au sein d'un même programme. Sur ces mêmes forums, nous avons observé de nombreuses réponses déconseillant l'utilisation d'ECS pour les interfaces, arguant principalement que les outils existants sont très bien adaptés. Nous avons donc travaillé sur un sujet avec une demande explicite, et avons démontré qu'ECS peut s'appliquer à la conception d'interfaces graphiques. Nous proposons en outre d'introduire le support de Composants temporaires, pour permettre la réification des périphériques en Entités.

Perspectives de travaux futurs sur ECS : À ce stade de notre travail, nous considérons le modèle de base de Polyphony suffisamment stable, cependant les choix des Composants et Systèmes sont encore susceptibles de changer. De plus, nous considérons d'autres langages pour l'implémentation de Polyphony. La première version était réalisée en Java, qui s'est révélé verbeux à l'usage à cause de l'incompatibilité entre son modèle objet et ECS (nous obligeant à passer par des appels de fonctions). La version courante est réalisée en JavaScript (avec Node.js), or son support de l'interaction à bas niveau s'est rélévé très limité, à cause du faible nombre de bindings disponibles, d'une obsolescence des bibliothèques tierces très rapide, et de l'instabilité générale de la plateforme. Nous considérons actuellement le développement sous Python, qui permet une métaprogrammation équivalente, tout en offrant un support de bas niveau suffisant (avec PyGame). Il est aussi envisageable que nous travaillions avec un framework existant pour ECS, tel que Unity. Enfin, nous sommes actuellement en contact avec la communauté d'ECS, pour promouvoir notre architecture d'interaction basée dessus, et la confronter aux discussions avec de potentiels utilisateurs. À l'avenir, nous souhaitons voir ECS évoluer pour supporter explicitement la programmation d'interactions, et à plus long terme contribuer à l'évolution des langages à objets/entités, et la conception future d'un langage dédié et optimisé pour ECS.

Améliorer le support de bas niveau de l'interaction

Comme suggéré plus haut, nous avons eu beaucoup de difficultés pour développer les deux projets présentés dans ce manuscrit, principalement à bas niveau. Nous rejoignons en partie les problèmes rencontrés par les participants de nos études, en ce que nous avons utilisé des bibliothèques (Node.js, OpenGL) pour des besoins rarement exprimés, et avons manqué de documentation et d'exemples contextuels pour supporter notre travail. À la différence de nos participants, nos difficultés ne sont pas liées à l'utilisation de frameworks, mais de bibliothèques de bas niveau. Ces difficultés ont consisté en :

- le choix des bibliothèques logicielles à utiliser pour les entrées de clavier/souris, et pour le dessin à l'écran (à cause de leur nombre, l'incertitude de l'existence de bindings fonctionnels pour un langage particulier, l'incertitude quant à leur performance, la découverte tardive de fonctionnalités manquantes, ou encore la découverte tardive de nouvelles bibliothèques)

- le choix d'un binding approprié pour une bibliothèque donnée (la plupart redéfinissent les noms des fonctions, altèrent les concepts d'utilisation de l'API, omettent certaines fonctionnalités, ou ne fonctionnent simplement plus)
- l'implémentation d'un binding pour une bibliothèque donnée (chaque langage définissant sa propre syntaxe pour la définition de bindings, avec un support souvent non-officiel et peu stable dans le temps, et ne supportant pas toujours toute l'Interface Binaire-Programme du système)
- le manque de puissance des bibliothèques de bas niveau, qui nécessite plus de développement qu'anticipé (ex. l'écriture de texte multi-lignes dans FreeType, le dessin de lignes avec épaisseur dans OpenGL, ou l'entrée de texte avec SDL)
- le manque de flexibilité des bibliothèques de haut niveau, lorsqu'on souhaite les utiliser pour des besoins simples (à cause des procédures d'installation complexes, des incompatibilités entre bibliothèques, des modèles complexes requérant un apprentissage conséquent, ou encore des mauvaises performances)

Les outils de bas niveau sont donc peu stables, ce qui est cohérent avec l'idée que peu de programmeurs sont censés y accéder. Or cette situation ne favorise pas le développement de nouvelles boîtes à outils, et contribue à pérenniser les frameworks majeurs, qui peuvent investir les efforts de développement pour tester et supporter de nombreuses alternatives sur chaque système d'exploitation. Dans la continuité du deuxième Essentiel d'Interaction (*un environnement d'interaction minimal et initialisé au démarrage de toute application*), nous souhaitons étudier à l'avenir le support de l'interaction à bas niveau, pour faciliter son accès dans les boîtes à outils d'interaction, et en utilisation directe par les programmeurs. Ce support prendrait la forme d'une API d'interaction *essentielle*, qui puisse être utilisée facilement dans tout langage, en utilisant un sous-ensemble de l'Interface Binaire-Programme, et en fournissant une interface en C facilitant la génération automatique de bindings. Les bibliothèques comme SDL existent déjà pour fournir un support complet de l'interaction. Cependant elles souffrent du manque de puissance évoqué ci-dessus, qui rend impossible leur utilisation seule.

Perspectives de travaux futurs sur les outils de bas niveau : Nous pensons qu'il serait nécessaire d'abord de clarifier l'*espace de conception* d'une bibliothèque de bas niveau, à partir de l'étude des nombreuses bibliothèques disponibles. Cet espace pourrait se représenter comme un graphe de dépendances, où les noeuds sont les fonctions de l'API, et les arêtes les dépendances entre elles. Par exemple, la fonction `dessinTexteMultiLigne` dépendrait de la fonction `dessinMot`, qui elle-même dépendrait de `dessinLettre`. Chaque fonction aurait comme poids sa complexité d'implémentation si elle n'était pas incluse dans une API. Il s'agirait alors de maximiser la somme des poids inclus dans une API, tout en minimisant le nombre de fonctions à inclure. Nous espérons ainsi contribuer à stabiliser les APIs de bas niveau, voire à terme les rapprocher des langages de programmation.

Vers la programmation d'interactions semi-structurée

La quasi-totalité des frameworks que nous avons étudiés (avec l'exception notable d'ImGUI) nécessitent l'initialisation d'un arbre de scène pour créer une quelconque application interactive. Qu'on souhaite réaliser une interface complexe comportant de multiples widgets et diverses modalités d'interaction, ou au contraire dessiner quelques formes à l'écran, il est toujours demandé de *structurer* l'interface. Une telle structure implique pour les utilisateurs d'un framework :

- d'apprendre les concepts d'arbre de scène, d'ordre d'affichage, de positionnement relatif, de propagation des évènements (*event bubbling*), voire de parcours récursif
- d'exprimer leur application à l'aide des widgets réutilisables de l'interface (ex. un canevas pour dessiner, une vue transparente pour intercepter les données d'entrée)
- d'initialiser un arbre de scène, avec au moins une vue et un conteneur, et placer leur code dans des *callbacks*

Ces efforts sont pondérés par la simplicité avec laquelle on peut exprimer des interfaces complexes. Cependant, ils sont un obstacle majeur lorsqu'il n'est pas question de créer une interface qui soit un assemblage de widgets. Il semble qu'il y ait un fossé entre ces interfaces “assemblées”, et des interfaces minimales (ou moins stéréotypées), pour lesquelles un arbre de scène serait superflu. Nous arguons qu'il devrait exister un juste milieu, qui permettrait d'utiliser des frameworks reconnus, pour développer des interfaces structurées différemment. Dans Polyphony, nous avons réduit l'utilisation de l'arbre de scène, rendant son utilisation non-systématique. Nous pouvons qualifier ce modèle de *semi-structuré*, car l'application possède une structure entre certaines Entités, mais pas entre toutes.

Perspectives de travaux futurs sur la programmation semi-structurée : Tout au long de ce manuscrit de thèse nous avons cherché à appréhender la nature de la “programmation d'interactions”. Nous l'avons caractérisée à partir des travaux des praticiens en IHM, observée lors des interviews de chercheurs, et pratiquée dans le développement de nos contributions logicielles. Or nous avons principalement observé des *interfaces graphiques*. L'idée de la programmation d'interactions que nous avons distillée dans ce manuscrit est en fait biaisée par l'environnement où nous l'avons étudiée. Cette thèse traite donc *réellement* de la programmation d'interfaces graphiques, bien que ce soient les interactions qui nous motivent. À partir de ce constat, nous souhaitons à l'avenir acquérir une connaissance plus complète des domaines où l'interaction se programme, et où on lui confère une structure originale, voire pas de structure. Entre tous ces domaines, la programmation semi-structurée pourrait être un concept unificateur, qui contribuerait à combler leurs éventuelles différences, et apporter une vision plus complète de la programmation d'interactions.

Bibliographie

[Act14] Acton, M. (2014). *Data-Oriented Design and C++*. Consulté à l'adresse https://www.youtube.com/watch?v=rX0ItVEVjHc

[Api04] Apitz, G., & Guimbretière, F. (2004). CrossY: A Crossing-based Drawing Application. *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, 3–12. https://doi.org/10.1145/1029632.1029635

[App06] Apple Inc. (2006). About Core Animation. Consulté 26 avril 2017, à l'adresse http://developer.apple.com/documentation/Cocoa/Conceptual/CoreAnimation_guide/

[App06] Appert, C., & Beaudouin-Lafon, M. (2006). SwingStates: Adding State Machines to the Swing Toolkit. *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, 319–322. https://doi.org/10.1145/1166253.1166302

[App08] Appert, C., & Beaudouin−Lafon, M. (2008). SwingStates : Adding state machines to Java and the Swing toolkit. *Software: Practice and Experience*, *38*(11), 1149-1182. https://doi.org/10.1002/spe.867

[App09] Appert, C., Huot, S., Dragicevic, P., & Beaudouin-Lafon, M. (2009). FlowStates: Prototypage D'Applications Interactives Avec Des Flots De DonnÉEs Et Des Machines À ÉTats. *Proceedings of the 21st International Conference on Association Francophone D'Interaction Homme-Machine*, 119–128. https://doi.org/10.1145/1629826.1629845

[Ase16] Asenov, D., Hilliges, O., & Müller, P. (2016). The Effect of Richer Visualizations on Code Comprehension. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 5040–5045. https://doi.org/10.1145/2858036.2858372

[Bad01] Badros, G. J., Borning, A., & Stuckey, P. J. (2001). The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Trans. Comput.-Hum. Interact.*, *8*(4), 267–306. https://doi.org/10.1145/504704.504705

[Bai08] Bailly, G., Lecolinet, E., & Nigay, L. (2008). Flower Menus: A New Type of Marking Menu with Large Menu Breadth, Within Groups and Efficient Expert Mode Memorization. *Proceedings of the Working Conference on Advanced Visual Interfaces*, 15–22. https://doi.org/10.1145/1385569.1385575

[Bai16] Bailly, G., Lecolinet, E., & Nigay, L. (2016). Visual Menu Techniques. *ACM Comput. Surv.*, *49*(4), 60:1–60:41. https://doi.org/10.1145/3002171

[Bar18] Baron, D., Jackson, D., & Birtles, B. (2018, octobre 11). *CSS Transitions*. Consulté à l'adresse https://www.w3.org/TR/css-transitions-1/#animatable-css

[Bau08] Bau, O., & Mackay, W. E. (2008). OctoPocus: A Dynamic Guide for Learning Gesture-based Command Sets. *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*, 37–46. https://doi.org/10.1145/1449715.1449724

[Bau13] Baudart, G., Mandel, L., & Pouzet, M. (2013). Programming Mixed Music in ReactiveML. *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, 11–22. https://doi.org/10.1145/2505341.2505344

[Bea00] Beaudouin-Lafon, M., & Lassen, H. M. (2000). The Architecture and Implementation of CPN2000, a post-WIMP Graphical Application. *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*, 181–190. https://doi.org/10.1145/354401.354761

[Bea00] Beaudouin-Lafon, M. (2000). Instrumental Interaction : An Interaction Model for Designing post-WIMP User Interfaces. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 446–453. https://doi.org/10.1145/332040.332473

[Bea00] Beaudouin-Lafon, M., & Mackay, W. E. (2000). Reification, Polymorphism and Reuse : Three Principles for Designing Visual Interfaces. *Proceedings of the Working Conference on Advanced Visual Interfaces*, 102–109. https://doi.org/10.1145/345513.345267

[Bea04] Beaudouin-Lafon, M. (2004). Designing Interaction, Not Interfaces. *Proceedings of the Working Conference on Advanced Visual Interfaces*, 15–22. https://doi.org/10.1145/989863.989865

[Bea08] Beaudouin-Lafon, M. (2008). Interaction is the Future of Computing. In T. Erickson & D. McDonald (Éd.), *HCI Remixed, Reflections on Works That Have Influenced the HCI Community* (p. 263-266). Consulté à l'adresse http://www.visi.com/~snowfall/HCIremixed.html

[Bea12] Beaudouin-Lafon, M., Huot, S., Nancel, M., Mackay, W., Pietriga, E., Primet, R., … Klokmose, C. (2012). Multisurface Interaction in the WILD Room. *Computer*, *45*(4), 48-56. https://doi.org/10.1109/MC.2012.110

[Bed99] Bederson, B. B., & Boltman, A. (1999). Does animation help users build mental maps of spatial information? *1999 IEEE Symposium on Information Visualization, 1999. (Info Vis '99) Proceedings*, 28-35. https://doi.org/10.1109/INFVIS.1999.801854

[Bed00] Bederson, B. B., Meyer, J., & Good, L. (2000). Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*, 171–180. https://doi.org/10.1145/354401.354754

[Bed04] Bederson, B. B., Grosjean, J., & Meyer, J. (2004). Toolkit design for interactive structured graphics. *IEEE Transactions on Software Engineering*, *30*(8), 535-546. https://doi.org/10.1109/TSE.2004.44

[Ber87] Berry, G., Couronné, P., Gonthier, G., & BANATRE, J.-P. (1987). *Programmation Synchrone Des Systèmes Réactifs : Le Langage Esterel*. *6*(4), 305–316.

[Ber14] Berry, G., & Serrano, M. (2014). Hop and HipHop : Multitier Web Orchestration. In R. Natarajan (Éd.), *Distributed Computing and Internet Technology* (p. 1-13). Consulté à l'adresse https://link.springer.com/chapter/10.1007/978-3-319-04483-5_1

[Ber17] Béra, C., Miranda, E., Felgentreff, T., Denker, M., & Ducasse, S. (2017). Sista : Saving Optimized Code in Snapshots for Fast Start-Up. *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, 1–11. https://doi.org/10.1145/3132190.3132201

[Bet00] Bétrancourt, M., & Tversky, B. (2000). Effect of computer animation on users' performance : A review. *Le Travail Humain: A Bilingual and Multi-Disciplinary Journal in Human Factors*, *63*(4), 311-329.

[Bie93] Bier, E. A., Stone, M. C., Pier, K., Buxton, W., & DeRose, T. D. (1993). Toolglass and Magic Lenses: The See-through Interface. *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, 73–80. https://doi.org/10.1145/166117.166126

[Bil02] Bilas, S. (2002, mars). *A Data-Driven Game Object System*. Présenté à Programming, San Francisco, CA, USA. Consulté à l'adresse https://www.gdcvault.com/play/1022543/A-Data-Driven-Object

[Bis98] Bishop, L., Eberly, D., Whitted, T., Finch, M., & Shantz, M. (1998). Designing a PC game engine. *IEEE Computer Graphics and Applications*, *18*(1), 46-53. https://doi.org/10.1109/38.637270

[Bla04] Blanch, R., Guiard, Y., & Beaudouin-Lafon, M. (2004). Semantic Pointing: Improving Target Acquisition with Control-display Ratio Adaptation. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 519–526. https://doi.org/10.1145/985692.985758

[Bla06] Blanch, R., & Beaudouin-Lafon, M. (2006). Programming Rich Interactions Using the Hierarchical State Machine Toolkit. *Proceedings of the Working Conference on Advanced Visual Interfaces*, 51–58. https://doi.org/10.1145/1133265.1133275

[Blo06] Bloch, J. (2006). How to Design a Good API and Why It Matters. *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, 506–507. https://doi.org/10.1145/1176617.1176622

[Bor97] Borning, A., Marriott, K., Stuckey, P., & Xiao, Y. (1997). Solving Linear Arithmetic Constraints for User Interface Applications. *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, 87–96. https://doi.org/10.1145/263407.263518

[Bor05] Bore, C., & Bore, S. (2005). Profiling software API usability for consumer electronics. *2005 Digest of Technical Papers. International Conference on Consumer Electronics, 2005. ICCE.*, 155-156. https://doi.org/10.1109/ICCE.2005.1429764

[Bos09] Bostock, M., & Heer, J. (2009). *Protovis: A Graphical Toolkit for Visualization*. *15*(6), 1121–1128. https://doi.org/10.1109/TVCG.2009.174

[Bos11] Bostock, M., Ogievetsky, V., & Heer, J. (2011). D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, *17*(12), 2301-2309. https://doi.org/10.1109/TVCG.2011.185

[Bou91] Boussinot, F. (1991). Reactive C : An extension of C to program reactive systems. *Software: Practice and Experience*, *21*(4), 401-428. https://doi.org/10.1002/spe.4380210406

[Bra90] Bracha, G., & Cook, W. (1990). Mixin-based Inheritance. *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, 303–311. https://doi.org/10.1145/97945.97982

[Bra08] Brandt, J., Guo, P. J., Lewenstein, J., & Klemmer, S. R. (2008). Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice. *Proceedings of the 4th International Workshop on End-user Software Engineering*, 1–5. https://doi.org/10.1145/1370847.1370848

[Bra09] Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., & Klemmer, S. R. (2009). Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1589–1598. https://doi.org/10.1145/1518701.1518944

[Bra09] Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., & Klemmer, S. R. (2009). Writing Code to Prototype, Ideate, and Discover. *IEEE Software*, *26*(5), 18-24. https://doi.org/10.1109/MS.2009.147

[Bro88] Brotman, L. S., & Netravali, A. N. (1988). Motion Interpolation by Optimal Control. *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, 309–315. https://doi.org/10.1145/54852.378531

[Bru09] Bruch, M., Monperrus, M., & Mezini, M. (2009). Learning from Examples to Improve Code Completion Systems. *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 213–222. https://doi.org/10.1145/1595696.1595728

[Bui02] Buisson, M., Bustico, A., Chatty, S., Colin, F.-R., Jestin, Y., Maury, S., … Truillet, P. (2002). Ivy: Un Bus Logiciel Au Service Du Développement De Prototypes De Systèmes Interactifs. *Proceedings of the 14th Conference on L'Interaction Homme-Machine*, 223–226. https://doi.org/10.1145/777005.777040

[Bux90] Buxton, W. A. S. (1990). *A Three-State Model of Graphical Input*.

[Cao10] Cao, J., Riche, Y., Wiedenbeck, S., Burnett, M., & Grigoreanu, V. (2010). End-user Mashup Programming : Through the Design Lens. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1009–1018. https://doi.org/10.1145/1753326.1753477

[Car19] Carnegie Mellon University. (2019). Natural Programming. Consulté 12 septembre 2019, à l'adresse https://www.cs.cmu.edu/~NatProg/

[Cas87] Caspi, P., Pilaud, D., Halbwachs, N., & Plaice, J. A. (1987). LUSTRE : A Declarative Language for Real-time Programming. *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 178–188. https://doi.org/10.1145/41625.41641

[Cas11] Casiez, G., & Roussel, N. (2011). No More Bricolage!: Methods and Tools to Characterize, Replicate and Compare Pointing Transfer Functions. *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, 603–614. https://doi.org/10.1145/2047196.2047276

[Cha07] Chatty, S., Lemort, A., & Vales, S. (2007). Multiple Input Support in a Model-Based Interaction Framework. *Second Annual IEEE International Workshop on Horizontal Interactive Human-Computer Systems, 2007. TABLETOP '07*, 179-186. https://doi.org/10.1109/TABLETOP.2007.27

[Cha08] Chatty, S. (2008). Programs = Data + Algorithms + Architecture: Consequences for Interactive Software Engineering. In *Lecture Notes in Computer Science*. *Engineering Interactive Systems* (p. 356-373). https://doi.org/10.1007/978-3-540-92698-6_22

[Cha12] Chatty, S. (2012). RÉConcilier Conception D'Interfaces Et Conception Logicielle: Vers La « Conception OrientÉe-SystÈMes ». *Proceedings of the 2012 Conference on Ergonomie Et Interaction Homme-machine*, 73:73–73:80. https://doi.org/10.1145/2652574.2653412

[Cha14] Chatty, S., & Conversy, S. (2014, juin 17). *What programming languages for interactive systems designers?* pp 47-51. Consulté à l'adresse https://hal-enac.archives-ouvertes.fr/hal-01024013/document

[Cha15] Chatty, S., Magnaudet, M., & Prun, D. (2015). Verification of Properties of Interactive Components from Their Executable Code. *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 276–285. https://doi.org/10.1145/2774225.2774848

[Cha16] Chatty, S., Magnaudet, M., Prun, D., Conversy, S., Rey, S., & Poirier, M. (2016). Designing, developing and verifying interactive components iteratively with djnn. *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. Présenté à TOULOUSE, France. Consulté à l'adresse https://hal.archives-ouvertes.fr/hal-01292291

[Che98] Chell, E. (1998). Critical incident technique. In *Qualitative methods and analysis in organizational research : A practical guide* (p. 51-72). Thousand Oaks, CA: Sage Publications Ltd.

[Che16] Chevalier, F., Riche, N. H., Plaisant, C., Chalbi, A., & Hurter, C. (2016). Animations 25 Years Later : New Roles and Opportunities. *Proceedings of the International Working Conference on Advanced Visual Interfaces*, 280–287. https://doi.org/10.1145/2909132.2909255

[Cla06] Claypool, M., & Claypool, K. (2006). Latency and Player Actions in Online Games. *Commun. ACM*, *49*(11), 40–45. https://doi.org/10.1145/1167838.1167860

[Con98] Conversy, S., Janecek, P., & Roussel, N. (1998). Factorisons La Gestion Des Événements Des Applications Interactives ! *Actes Des 10ème Journées Francophones Sur l'Interaction Homme Machine (IHM'98)*, 141–144. Consulté à l'adresse http://insitu.lri.fr/ roussel/publications/IHM98.pdf

[Con08] Conversy, S., Barboni, E., Navarre, D., & Palanque, P. (2008). Improving Modularity of Interactive Software with the MDPC Architecture. In J. Gulliksen, M. B. Harning, P. Palanque, G. C. van der Veer, & J. Wesson (Éd.), *Engineering Interactive Systems* (p. 321–338). https://doi.org/10.1007/978-3-540-92698-6_20

[Con12] Conversy, S. (2012, mars 23). *A visual perception account of programming languages : Finding the natural science in the art*. Consulté à l'adresse https://hal.inria.fr/hal-00737414

[Con13] Conversy, S. (2013). Existe-t-il Une DiffÉRence Entre Langages Visuels Et Textuels En Termes De Perception? *Proceedings of the 25th Conference on L'Interaction Homme-Machine*, 53:53–53:58. https://doi.org/10.1145/2534903.2534911

[Con14] Conversy, S. (2014). Unifying Textual and Visual : A Theoretical Account of the Visual Perception of Programming Languages. *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 201–212. https://doi.org/10.1145/2661136.2661138

[Coo14] Cooney, D. (2014, juillet 24). *Introduction to Web Components* (D. Glazkov & H. Ito, Éd.). Consulté à l'adresse https://www.w3.org/TR/components-intro/

[Cou87] Coutaz, J. (1987). PAC: AN OBJECT ORIENTED MODEL FOR IMPLEMENTING USER INTERFACES. *SIGCHI Bull.*, *19*(2), 37–41. https://doi.org/10.1145/36111.1045592

[Cur82] Curry, G., Baer, L., Lipkie, D., & Lee, B. (1982). Traits: An Approach to Multiple-inheritance Subclassing. *Proceedings of the SIGOA Conference on Office Information Systems*, 1–9. https://doi.org/10.1145/800210.806468

[Cyp93] Cypher, A., Halbert, D. C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B. A., & Turransky, A. (Éd.). (1993). *Watch What I Do : Programming by Demonstration*. Cambridge, MA, USA: MIT Press.

[Dam97] van Dam, A. (1997). Post-WIMP User Interfaces. *Commun. ACM*, *40*(2), 63–67. https://doi.org/10.1145/253671.253708

[Dan11] Dann, W. P., & Pausch, R. (2011). *Learning to Program with Alice* (3 edition). Boston: Pearson.

[Deb15] Deber, J., Jota, R., Forlines, C., & Wigdor, D. (2015). How Much Faster is Fast Enough? : User Perception of Latency & Latency Improvements in Direct and Indirect Touch. *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, 1827–1836. https://doi.org/10.1145/2702123.2702300

[Deh13] Dehouck, M., Bhatti, M. U., Bergel, A., & Ducasse, S. (2013, septembre 10). *Pragmatic Visualizations for Roassal : A Florilegium*. Présenté à International Workshop on Smalltalk Technologies. Consulté à l'adresse https://hal.inria.fr/hal-00862065/document

[Dra01] Dragicevic, P., & Fekete, J.-D. (2001). Input Device Selection and Interaction Configuration with ICON. In *People and Computers XV—Interaction without Frontiers* (p. 543-558). https://doi.org/10.1007/978-1-4471-0353-0_34

[Dra04] Dragicevic, P. (2004). *Un modèle d'interaction en entrée pour des systèmes interactifs multi-dispositifs hautement configurables*. Consulté à l'adresse https://tel.archives-ouvertes.fr/tel-00426037

[Dra04] Dragicevic, P., & Fekete, J.-D. (2004). Support for Input Adaptability in the ICON Toolkit. *Proceedings of the 6th International Conference on Multimodal Interfaces*, 212–219. https://doi.org/10.1145/1027933.1027969

[Dra11] Dragicevic, P., Huot, S., & Chevalier, F. (2011). Gliimpse : Animating from Markup Code to Rendered Documents and Vice Versa. *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, 257–262. https://doi.org/10.1145/2047196.2047229

[Dua11] Duala-Ekoko, E., & Robillard, M. P. (2011). Using Structure-Based Recommendations to Facilitate Discoverability in APIs. *ECOOP 2011 – Object-Oriented Programming*, 79-104. https://doi.org/10.1007/978-3-642-22655-7_5

[Dua12] Duala-Ekoko, E., & Robillard, M. P. (2012). Asking and answering questions about unfamiliar APIs: An exploratory study. *2012 34th International Conference on Software Engineering (ICSE)*, 266-276. https://doi.org/10.1109/ICSE.2012.6227187

[Duc17] Ducasse, S., Zagidulin, D., Hess, N., & Chloupis, D. (2017). *Pharo by Example 5.0*. Square Bracket Associates.

[Dur18] Duruisseau, M., Tarby, J.-C., Le Pallec, X., & Gérard, S. (2018). VisUML : A Live UML Visualization to Help Developers in Their Programming Task. In S. Yamamoto & H. Mori (Éd.), *Human Interface and the Management of Information. Interaction, Visualization, and Analytics* (p. 3-22). Springer International Publishing.

[Eag11] Eagan, J. R., Beaudouin-Lafon, M., & Mackay, W. E. (2011). Cracking the Cocoa Nut: User Interface Programming at Runtime. *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, 225–234. https://doi.org/10.1145/2047196.2047226

[Ecm15] Ecma International. (2015, juin). ECMAScript 2015 Language Specification – ECMA-262 6th Edition. Consulté 14 mai 2018, à l'adresse http://www.ecma-international.org/ecma-262/6.0/index.html

[Efr79] Efron, B. (1979). Bootstrap Methods : Another Look at the Jackknife. *The Annals of Statistics*, *7*(1), 1-26. https://doi.org/10.1214/aos/1176344552

[Fau16] Faulkner, S., Eicholz, A., Leithead, T., & Danilo, A. (2016, novembre 1). HTML 5.1 : 7.8 Animation Frames. Consulté 6 avril 2017, à l'adresse https://www.w3.org/TR/html51/webappapis.html#animation-frames

[Fek96] Fekete, J.-D. (1996, septembre). *Les trois services du noyau sémantique indispensables a l'IHM*. 45-50. Consulté à l'adresse https://hal.inria.fr/hal-00911555

[Fek96] Fekete, J.-D. (1996). Les trois services du noyau sémantique indispensables à l'IHM. *"Séance pleinière des journées du GDR Programmation du CNRS*. Consulté à l'adresse http://insitu.lri.fr/~fekete/ps/ihm96.pdf

[For17] Ford, T. (2017). *Overwatch Gameplay Architecture and Netcode*. Consulté à l'adresse https://www.youtube.com/watch?v=W3aieHjyNvw

[Fow05] Fowler, M. (2005, juin 26). InversionOfControl. Consulté 1 août 2019, à l'adresse https://martinfowler.com/bliki/InversionOfControl.html

[Fra19] Framasoft. (2019). Créez et diffusez vos formulaires facilement | Framaforms.org. Consulté 22 septembre 2019, à l'adresse https://framaforms.org/

[Gai91] Gaines, B. R. (1991). Modeling and forecasting the information sciences. *Information Sciences*, *57-58*, 3-22. https://doi.org/10.1016/0020-0255(91)90066-4

[Gaj10] Gajos, K. Z., Weld, D. S., & Wobbrock, J. O. (2010). Automatically generating personalized user interfaces with Supple. *Artificial Intelligence*, *174*(12), 910-950. https://doi.org/10.1016/j.artint.2010.05.005

[Gal17] Galindo, J. A., Dupuy-Chessa, S., & Céret, E. (2017). Toward a UI adaptation approach driven by user emotions. *Proceedings of the tenth international conference on advances in computer-human interactions (ACHI'2017)*, 12-17. IARIA.

[Get02] Gettys, J., Scheifler, R. W., Adams, C., Joloboff, V., Hiura, H., McMahon, B., … Yamada, S. (2002). *Xlib—C Language X Interface : X Window System Standard*. 476.

[Git13] GitHub. (2013, juillet 15). Electron | Développez des applications desktop multi-plateformes avec JavaScript, HTML et CSS. Consulté 12 août 2019, à l'adresse https://electronjs.org/

[Gla17] Glaser, B. G., & Strauss, A. L. (2017). *La découverte de la théorie ancrée—2e éd. - Stratégies pour la recherche qualitative*. Armand Colin.

[GLF02] The GLFW Development Team. (2002). GLFW—An OpenGL library. Consulté 7 juin 2019, à l'adresse https://www.glfw.org/

[Gog14] Goguey, A., Casiez, G., Pietrzak, T., Vogel, D., & Roussel, N. (2014). Adoiraccourcix: Multi-touch Command Selection Using Finger Identification. *Proceedings of the 26th Conference on L'Interaction Homme-Machine*, 28–37. https://doi.org/10.1145/2670444.2670446

[Gon96] Gonzalez, C. (1996). Does Animation in User Interfaces Improve Decision Making? *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 27–34. https://doi.org/10.1145/238386.238396

[Goo08] Google. (2008). Android. Consulté 26 avril 2017, à l'adresse Android website: https://www.android.com/

[Goo15] Google Inc. (2015). CORGI: Main Page. Consulté 23 mars 2018, à l'adresse http://google.github.io/corgi/

[Gre96] Green, T. R. G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments : A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing*, *7*(2), 131-174. https://doi.org/10.1006/jvlc.1996.0009

[Gre14] GreenSock. (2014, août 3). GSAP, the standard for JavaScript HTML5 animation. Consulté 26 avril 2017, à l'adresse GreenSock website: https://greensock.com/gsap

[Gri09] Grigoreanu, V., Fernandez, R., Inkpen, K., & Robertson, G. (2009). What designers want: Needs of interactive application designers. *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 139-146. https://doi.org/10.1109/VLHCC.2009.5295277

[Gri12] Grill, T., Polacek, O., & Tscheligi, M. (2012). Methods towards API Usability: A Structural Analysis of Usability Problem Categories. *Human-Centered Software Engineering*, 164-180. https://doi.org/10.1007/978-3-642-34347-6_10

[Gro05] Grossman, T., & Balakrishnan, R. (2005). The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 281–290. https://doi.org/10.1145/1054972.1055012

[Hee08] Heer, J., Agrawala, M., & Willett, W. (2008). Generalized Selection via Interactive Query Relaxation. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 959–968. https://doi.org/10.1145/1357054.1357203

[Hee08] Heer, J., Mackinlay, J., Stolte, C., & Agrawala, M. (2008). Graphical Histories for Visualization : Supporting Analysis, Communication, and Evaluation. *IEEE Transactions on Visualization and Computer Graphics*, *14*(6), 1189-1196. https://doi.org/10.1109/TVCG.2008.137

[Hen90] Henry, T. R., Hudson, S. E., & Newell, G. L. (1990). Integrating Gesture and Snapping into a User Interface Toolkit. *Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, 112–122. https://doi.org/10.1145/97924.97938

[Hen09] Henning, M. (2009). API Design Matters. *Commun. ACM*, *52*(5), 46–56. https://doi.org/10.1145/1506409.1506424

[Hic99] Hickson, I. (1999, janvier 26). Acid Tests. Consulté 12 août 2019, à l'adresse https://www.acidtests.org/

[Hor17] Hornbæk, K., & Oulasvirta, A. (2017). What Is Interaction? *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 5040–5052. https://doi.org/10.1145/3025453.3025765

[Hud05] Hudson, S. E., Mankoff, J., & Smith, I. (2005). Extensible Input Handling in the subArctic Toolkit. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 381–390. https://doi.org/10.1145/1054972.1055025

[Huo04] Huot, S., Dumas, C., Dragicevic, P., Fekete, J.-D., & Hégron, G. (2004). The MaggLite post-WIMP Toolkit: Draw It, Connect It and Run It. *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, 257–266. https://doi.org/10.1145/1029632.1029677

[Huo06] Huot, S., Dragicevic, P., & Dumas, C. (2006). Flexibilité Et Modularité Pour La Conception D'Interactions: Le ModèLe D'Architecture Logicielle Des Graphes Combinés. *Proceedings of the 18th Conference on L'Interaction Homme-Machine*, 43–50. https://doi.org/10.1145/1132736.1132742

[Huo13] Huot, S. (2013). *'Designeering Interaction': A Missing Link in the Evolution of Human-Computer Interaction* (Thesis). Consulté à l'adresse https://tel.archives-ouvertes.fr/tel-00823763

[IEE08] IEEE Computer Society. (2008). Opportunistic System Development. *IEEE Software*, *25*(6).

[ISO18] ISO/IEC. (2018, juin). *Information technology—Programming languages—C*. Consulté à l'adresse http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/45/74528.html

[ISO18] ISO/TC 159/SC 4. (2018, mars). ISO 9241-11:2018. Consulté 4 septembre 2019, à l'adresse ISO website: http://www.iso.org/cms/render/live/fr/sites/isoorg/contents/data/standard/06/35/63500.html

[Jac16] Jack, R. H., Stockman, T., & McPherson, A. (2016). Effect of Latency on Performer Interaction and Subjective Quality Assessment of a Digital Musical Instrument. *Proceedings of the Audio Mostly 2016*, 116–123. https://doi.org/10.1145/2986416.2986428

[Jai07] Jaimes, A., & Sebe, N. (2007). Multimodal human–computer interaction: A survey. *Computer Vision and Image Understanding*, *108*(1), 116-134. https://doi.org/10.1016/j.cviu.2006.10.019

[Jan13] Jankowski, J., & Hachet, M. (2013, mai 6). *A Survey of Interaction Techniques for Interactive 3D Environments*. Présenté à Eurographics 2013 - STAR. Consulté à l'adresse https://hal.inria.fr/hal-00789413/document

[Joh88] Johnson, R., & Foote, B. (1988). Designing Reusable Classes. *Journal of Object-Oriented Programming*, *1*(2), 22-35.

[Kal05] Kaltenbrunner, M., Bovermann, T., Bencina, R., & Costanza, E. (2005). *TUIO A Protocol for Table-Top Tangible User Interfaces*. Consulté à l'adresse https://doi.org/files/publications/07a830-GW2005-KaltenBoverBencinaConstanza.pdf

[Ked17] Kedia, P., Costa, M., Parkinson, M., Vaswani, K., Vytiniotis, D., & Blankstein, A. (2017). Simple, Fast, and Safe Manual Memory Management. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 233–247. https://doi.org/10.1145/3062341.3062376

[Ken02] Kennedy, K., & Mercer, R. E. (2002). Planning Animation Cinematography and Shot Structure to Communicate Theme and Mood. *Proceedings of the 2Nd International Symposium on Smart Graphics*, 1–8. https://doi.org/10.1145/569005.569006

[Kic97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. (1997). Aspect-oriented programming. In M. Akşit & S. Matsuoka (Éd.), *ECOOP'97—Object-Oriented Programming* (p. 220-242). Springer Berlin Heidelberg.

[Kin12] Kin, K., Hartmann, B., DeRose, T., & Agrawala, M. (2012). Proton: Multitouch Gestures As Regular Expressions. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2885–2894. https://doi.org/10.1145/2207676.2208694

[Kin12] Kin, K., Hartmann, B., DeRose, T., & Agrawala, M. (2012). Proton++: A Customizable Declarative Multitouch Framework. *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, 477–486. https://doi.org/10.1145/2380116.2380176

[Kle05] Klein, C., & Bederson, B. B. (2005). Benefits of Animated Scrolling. *CHI '05 Extended Abstracts on Human Factors in Computing Systems*, 1965–1968. https://doi.org/10.1145/1056808.1057068

[Ko11] Ko, A. J., & Riche, Y. (2011). The role of conceptual knowledge in API usability. *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 173-176. https://doi.org/10.1109/VLHCC.2011.6070395

[Kra88] Krasner, G. E., & Pope, S. T. (1988). A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80. *J. Object Oriented Program.*

[Lan98] Lantinga, S. (1998). Simple DirectMedia Layer - Homepage. Consulté 26 février 2019, à l'adresse http://libsdl.org/

[Lec99] Lecolinet, E. (1999). A Brick Construction Game Model for Creating Graphical User Interfaces: The Ubit Toolkit. *Proc. INTERACT'99*, 510–518.

[Lec03] Lecolinet, E. (2003). A Molecular Architecture for Creating Advanced GUIs. *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology*, 135–144. https://doi.org/10.1145/964696.964711

[Lec16] Lecrubier, V. (2016). *A formal language for designing, specifying and verifying critical embedded human machine interfaces* (Theses, INSTITUT SUPERIEUR DE L'AERONAUTIQUE ET DE L'ESPACE (ISAE); UNIVERSITE DE TOULOUSE). Consulté à l'adresse https://hal.archives-ouvertes.fr/tel-01455466

[Led18] Ledo, D., Houben, S., Vermeulen, J., Marquardt, N., Oehlberg, L., & Greenberg, S. (2018). Evaluation Strategies for HCI Toolkit Research. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 36:1–36:17. https://doi.org/10.1145/3173574.3173610

[LeG91] LeGuernic, P., Gautier, T., Borgne, M. L., & Maire, C. L. (1991). Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, *79*(9), 1321-1336. https://doi.org/10.1109/5.97301

[Lei15] Leite Neto, N. M., Lenormand, J., du Bousquet, L., & Dupuy-Chessa, S. (2015). Toward testing multiple user interface versions. In T. G. Aho, Pekka; Vos Juan; Bøegh, Jørgen; Rennoch, Axel (Éd.), *Joint research workshop 10th systems testing and validation (STV15) and 1st international workshop on user interface test automation (INTUITEST 2015)* (p. 61-72). Sophia-Antipolis, France: Institute Fraunhofer.

[Leo99] Leonard, T. (1999, juillet 9). Postmortem: Thief: The Dark Project. Consulté 12 mars 2018, à l'adresse https://www.gamasutra.com/view/feature/131762/postmortem_thief_the_dark_project.php

[Let10] Letondal, C., Chatty, S., Philips, G., André, F., & Conversy, S. (2010, septembre 19). *Usability requirements for interaction-oriented development tools*. pp xxx. Consulté à l'adresse https://hal-enac.archives-ouvertes.fr/hal-01022441/document

[Let14] Letondal, C., Pillain, P.-Y., Verdurand, E., Prun, D., & Grisvard, O. (2014, septembre 9). *Of Models, Rationales and Prototypes: Studying Designer Needs in an Airborne Maritime Surveillance Drawing Tool to Support Audio Communication*. pp 8. https://doi.org/10.14236/ewic/hci2014.8

[Lie01] Lieberman, H. (2001). *Your Wish is My Command*. https://doi.org/10.1016/B978-1-55860-688-3.X5000-3

[Lim05] Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., & López-Jaquero, V. (2005). USIXML: A Language Supporting Multi-path Development of User Interfaces. In R. Bastide, P. Palanque, & J. Roth (Éd.), *Engineering Human Computer Interaction and Interactive Systems* (p. 200-220). Springer Berlin Heidelberg.

[Llo09] Llopis, N. (2009, décembre 4). Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP) – Games from Within. Consulté 7 mars 2019, à l'adresse http://gamesfromwithin.com/data-oriented-design

[Mac00] Mackay, W. (2000). Responding to cognitive overload: Co-adaptation between users and technology. *Intellectica*, *30*(1), 177-193. https://doi.org/10.3406/intel.2000.1597

[Mag14] Magnaudet, M., & Chatty, S. (2014). What Should Adaptivity Mean to Interactive Software Programmers? *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 13–22. https://doi.org/10.1145/2607023.2607028

[Mag17] Magnaudet, M., Rey, S., & Conversy, S. (2017). Djnn: A Process Oriented Programming Language for Interactive Systems. *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 159–160. https://doi.org/10.1145/3102113.3102161

[Mag18] Magnaudet, M., Chatty, S., Conversy, S., Leriche, S., Picard, C., & Prun, D. (2018). Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming. *Proc. ACM Hum.-Comput. Interact.*, *2*(EICS), 12:1–12:27. https://doi.org/10.1145/3229094

[Mal95] Maloney, J. H., & Smith, R. B. (1995). Directness and Liveness in the Morphic User Interface Construction Environment. *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, 21–28. https://doi.org/10.1145/215585.215636

[Mal13] Malacria, S., Bailly, G., Harrison, J., Cockburn, A., & Gutwin, C. (2013). Promoting Hotkey Use Through Rehearsal with ExposeHK. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 573–582. https://doi.org/10.1145/2470654.2470735

[Man09] Mandel, L., & Plateau, F. (2009). Interactive Programming of Reactive Systems. *Electronic Notes in Theoretical Computer Science*, *238*(1), 21-36. https://doi.org/10.1016/j.entcs.2008.01.004

[Mar07] Martin, A. (2007, septembre 3). Entity Systems are the future of MMOG development – Part 1. Consulté 9 mai 2018, à l'adresse t-machine.org website: http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/

[Mar14] Martin, A. (2014, novembre 30). What's an Entity System ? - Entity Systems Wiki. Consulté 19 septembre 2019, à l'adresse http://entity-systems.wikidot.com/

[Mar17] Marquardt, N., Houben, S., Beaudouin-Lafon, M., & Wilson, A. D. (2017). HCITools : Strategies and Best Practices for Designing, Evaluating and Sharing Technical HCI Toolkits. *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, 624–627. https://doi.org/10.1145/3027063.3027073

[May12] Mayer, D. (2012, novembre 11). Ratio of bugs per line of code. Consulté 8 août 2019, à l'adresse https://www.mayerdan.com/ruby/2012/11/11/bugs-per-line-of-code-ratio

[McC04] McConnell, S. (2004). *Code Complete : A Practical Handbook of Software Construction, Second Edition* (2nd edition). Redmond, Wash: Microsoft Press.

[McL98] McLellan, S. G., Roesler, A. W., Tempest, J. T., & Spinuzzi, C. I. (1998). Building more usable APIs. *IEEE Software*, *15*(3), 78-86. https://doi.org/10.1109/52.676963

[Mil56] Miller, G. A. (1956). The magical number seven, plus or minus two : Some limits on our capacity for processing information. *Psychological Review*, *63*(2), 81-97. https://doi.org/10.1037/h0043158

[Mir12] Mirlacher, T., Palanque, P., & Bernhaupt, R. (2012). Engineering Animations in User Interfaces. *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 111–120. https://doi.org/10.1145/2305484.2305504

[Moo10] Mooty, M., Faulring, A., Stylos, J., & Myers, B. A. (2010). Calcite: Completing Code Completion for Constructors Using Crowds. *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, 15-22. https://doi.org/10.1109/VLHCC.2010.12

[Moz19] Mozilla. (2019, mars 23). CSS Inheritance. Consulté 18 septembre 2019, à l'adresse MDN Web Docs website: https://developer.mozilla.org/en-US/docs/Web/CSS/inheritance

[Mur05] Muratori, C. (2005). Immediate-Mode Graphical User Interfaces (2005). Consulté 12 février 2019, à l'adresse https://caseymuratori.com/blog_0001

[Mye86] Myers, B. A. (1986). Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 59–66. https://doi.org/10.1145/22627.22349

[Mye90] Myers, B. A., Giuse, D. A., Dannenberg, R. B., Zanden, B. V., Kosbie, D. S., Pervin, E., … Marchal, P. (1990). Garnet: comprehensive support for graphical, highly interactive user interfaces. *Computer*, *23*(11), 71-85. https://doi.org/10.1109/2.60882

[Mye91] Myers, B. A. (1991). Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-backs. *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, 211–220. https://doi.org/10.1145/120782.120805

[Mye94] Myers, B. (1994). Challenges of HCI Design and Implementation. *interactions*, *1*(1), 73–83. https://doi.org/10.1145/174800.174808

[Mye97] Myers, B. A., McDaniel, R. G., Miller, R. C., Ferrency, A. S., Faulring, A., Kyle, B. D., … Doane, P. (1997). The Amulet environment: new models for effective user interface software development. *IEEE Transactions on Software Engineering*, *23*(6), 347-365. https://doi.org/10.1109/32.601073

[Mye04] Myers, B. A., Pane, J. F., & Ko, A. (2004). Natural Programming Languages and Environments. *Commun. ACM*, *47*(9), 47–52. https://doi.org/10.1145/1015864.1015888

[Mye08] Myers, B., Park, S. Y., Nakano, Y., Mueller, G., & Ko, A. (2008). How designers design and program interactive behaviors. *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, 177-184. https://doi.org/10.1109/VLHCC.2008.4639081

[Mye08] Myers, B. A., Ko, A. J., Park, S. Y., Stylos, J., LaToza, T. D., & Beaton, J. (2008). More Natural End-user Software Engineering. *Proceedings of the 4th International Workshop on End-user Software Engineering*, 30–34. https://doi.org/10.1145/1370847.1370854

[Nan14] Nancel, M., & Cockburn, A. (2014). Causality : A Conceptual Model of Interaction History. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1777–1786. https://doi.org/10.1145/2556288.2556990

[Nav01] Navarre, D., Palanque, P., Bastide, R., & Su, O. (2001). A model-based tool for interactive prototyping of highly interactive applications. *Proceedings 12th International Workshop on Rapid System Prototyping. RSP 2001*, 136-141. https://doi.org/10.1109/IWRSP.2001.933851

[Nav09] Navarre, D., Palanque, P., Ladry, J.-F., & Barboni, E. (2009). ICOs : A Model-based User Interface Description Technique Dedicated to Interactive Systems Addressing Usability, Reliability and Scalability. *ACM Trans. Comput.-Hum. Interact.*, *16*(4), 18:1–18:56. https://doi.org/10.1145/1614390.1614393

[Ng14] Ng, A., Annett, M., Dietz, P., Gupta, A., & Bischof, W. F. (2014). In the Blink of an Eye : Investigating Latency Perception During Stylus Interaction. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1103–1112. https://doi.org/10.1145/2556288.2557037

[Nie19] Niedoba, M. (2019, avril 19). Managing complexity. Consulté 6 septembre 2019, à l'adresse Medium website: https://uxdesign.cc/managing-complexity-8094c316a506

[Nor88] Norman, D. A. (1988). *The Design of Everyday Things*. New York: Basic Books.

[Nor10] Norman, D. A. (2010). *Living with Complexity*. Cambridge, Mass: MIT Press.

[Nor10] Norman, D. A. (2010). Technology First, Needs Last : The Research-product Gulf. *Interactions*, *17*(2), 38–42. https://doi.org/10.1145/1699775.1699784

[Obr08] Obrenovic, Ž., Gaševic, D., & Eliëns, A. (2008). Stimulating Creativity through Opportunistic Software Development. *IEEE Software*, *25*(6), 64-70. https://doi.org/10.1109/MS.2008.162

[Oma12] Omar, C., Yoon, Y. S., LaToza, T. D., & Myers, B. A. (2012). Active code completion. *2012 34th International Conference on Software Engineering (ICSE)*, 859-869. https://doi.org/10.1109/ICSE.2012.6227133

[One12] Oney, S., & Brandt, J. (2012). Codelets: Linking Interactive Documentation and Example Code in the Editor. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2697–2706. https://doi.org/10.1145/2207676.2208664

[One14] Oney, S., Myers, B., & Brandt, J. (2014). InterState : A Language and Environment for Expressing Interface Behavior. *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, 263–272. https://doi.org/10.1145/2642918.2647358

[Ora08] Oracle Corp. (2008). Client Technologies : Java Platform, Standard Edition (Java SE) 8 Release 8. Consulté 26 avril 2017, à l'adresse http://docs.oracle.com/javase/8/javase-clienttechnologies.htm

[Pap18] Papari, A. (2018). *artemis-odb: A continuation of the popular Artemis ECS framework* [Java]. Consulté à l'adresse https://github.com/junkdog/artemis-odb (Original work published 2012)

[Pat09] Paterno', F., Santoro, C., & Spano, L. D. (2009). MARIA : A Universal, Declarative, Multiple Abstraction-level Language for Service-oriented Applications in Ubiquitous Environments. *ACM Trans. Comput.-Hum. Interact.*, *16*(4), 19:1–19:30. https://doi.org/10.1145/1614390.1614394

[Pav11] Pavlovych, A., & Stuerzlinger, W. (2011). Target Following Performance in the Presence of Latency, Jitter, and Signal Dropouts. *Proceedings of Graphics Interface 2011*, 33–40. Consulté à l'adresse http://dl.acm.org/citation.cfm?id=1992917.1992924

[Pic13] Piccioni, M., Furia, C. A., & Meyer, B. (2013). An Empirical Study of API Usability. *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 5-14. https://doi.org/10.1109/ESEM.2013.14

[Pla15] Plantec, A. (2015). Bloc : A new Morphic framework. Consulté 7 avril 2017, à l'adresse http://sdmeta.gforge.inria.fr/YouTubeVideos/PharoPreview/BlocSlides-ESUG2015.pdf

[Poo16] Poor, G. M., Jaffee, S. D., Leventhal, L. M., Ringenberg, J., Klopfer, D. S., Zimmerman, G., & Klein, B. A. (2016). Applying the Norman 1986 User-Centered Model to Post-WIMP UIs : Theoretical Predictions and Empirical Outcomes. *ACM Trans. Comput.-Hum. Interact.*, *23*(5), 30:1–30:33. https://doi.org/10.1145/2983531

[Pri18] prime31. (2018). *Nez is a free 2D focused framework that works with MonoGame and FNA* [C#]. Consulté à l'adresse https://github.com/prime31/Nez (Original work published 2016)

[Qt19] The Qt Company. (2019). Qt | Cross-platform software development for embedded & desktop. Consulté 18 avril 2019, à l'adresse https://www.qt.io

[Raf16] Raffaillac, T. (2016). *Show us your Project—Session 2*. Consulté à l'adresse https://youtu.be/ltXFtdhKuM8?t=3m25s

[Raf17] Raffaillac, T. (2017). Language and System Support for Interaction. *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 149–152. https://doi.org/10.1145/3102113.3102155

[Raf17] Raffaillac, T. (2017). *PharoDays17 : Show us your Projects*. Consulté à l'adresse https://youtu.be/1yzc-EKFVs0?t=28m5s

[Raf17] Raffaillac, T., Huot, S., & Ducasse, S. (2017). Turning Function Calls into Animations. *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 81–86. https://doi.org/10.1145/3102113.3102134

[Raf18] Raffaillac, T., & Huot, S. (2018). Application du modèle Entité-Composant-Système à la programmation d'interactions. *Proceedings of the 30th Conference on L'Interaction Homme-Machine*, 42–51. https://doi.org/10.1145/3286689.3286703

[Raf19] Raffaillac, T., & Huot, S. (2019). Polyphony : Programming Interfaces and Interactions with the Entity-Component-System Model. *Proc. ACM Hum.-Comput. Interact.*, *3*(EICS), 8:1–8:22. https://doi.org/10.1145/3331150

[Rea14] Reas, C., & Fry, B. (2014). *Processing : A Programming Handbook for Visual Designers and Artists* (second edition edition). Cambridge, Massachusetts: The MIT Press.

[Res09] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., … Kafai, Y. (2009). Scratch : Programming for All. *Commun. ACM*, *52*(11), 60–67. https://doi.org/10.1145/1592761.1592779

[Rey15] Rey, S., Conversy, S., Magnaudet, M., Poirier, M., Prun, D., Vinot, J.-L., & Chatty, S. (2015). Using the Djnn Framework to Create and Validate Interactive Components Iteratively. *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 230–233. https://doi.org/10.1145/2774225.2775438

[Rob09] Robillard, M. P. (2009). What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, *26*(6), 27-34. https://doi.org/10.1109/MS.2009.193

[Rob11] Robillard, M. P., & DeLine, R. (2011). A field study of API learning obstacles. *Empirical Software Engineering*, *16*(6), 703-732. https://doi.org/10.1007/s10664-010-9150-8

[Rou12] Roussel, N., Casiez, G., Aceituno, J., & Vogel, D. (2012). Giving a Hand to the Eyes : Leveraging Input Accuracy for Subpixel Interaction. *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, 351–358. https://doi.org/10.1145/2380116.2380162

[Roy19] Roy, Q. (2019). *Marking-Menu* [JavaScript]. Consulté à l'adresse https://github.com/QuentinRoy/Marking-Menu (Original work published 2017)

[Sai15] Saied, M. A., Sahraoui, H., & Dufour, B. (2015). An observational study on API usage constraints and their documentation. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 33-42. https://doi.org/10.1109/SANER.2015.7081813

[San93] Sannella, M., Maloney, J., Freeman–Benson, B., & Borning, A. (1993). Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software: Practice and Experience*, *23*(5), 529-566. https://doi.org/10.1002/spe.4380230507

[Sat14] Satyanarayan, A., Wongsuphasawat, K., & Heer, J. (2014). Declarative Interaction Design for Data Visualization. *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, 669–678. https://doi.org/10.1145/2642918.2647360

[Sch07] Schlienger, C., Conversy, S., Chatty, S., Anquetil, M., & Mertz, C. (2007). Improving Users' Comprehension of Changes with Animation and Sound : An Empirical Assessment. *Human-Computer Interaction – INTERACT 2007*, 207-220. https://doi.org/10.1007/978-3-540-74796-3_20

[Sch18] Schmid, S. (2018). *Entitas-CSharp: Entitas is a super fast Entity Component System (ECS) Framework specifically made for C# and Unity* [C#]. Consulté à l'adresse https://github.com/sschmid/Entitas-CSharp (Original work published 2014)

[Sha07] Shanmugasundaram, M., Irani, P., & Gutwin, C. (2007). Can Smooth View Transitions Facilitate Perceptual Constancy in Node-link Diagrams? *Proceedings of Graphics Interface 2007*, 71–78. https://doi.org/10.1145/1268517.1268531

[Sit19] Sitnik, A., & Solovev, I. (2019, août). Easing Functions Cheat Sheet. Consulté 15 septembre 2019, à l'adresse http://easings.net/

[Spa13] Spano, L. D., Cisternino, A., Paternò, F., & Fenu, G. (2013). GestIT : A Declarative and Compositional Framework for Multiplatform Gesture Definition. *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 187–196. https://doi.org/10.1145/2494603.2480307

[Sta93] Stasko, J., Badre, A., & Lewis, C. (1993). Do Algorithm Animations Assist Learning? : An Empirical Study and Analysis. *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, 61–66. https://doi.org/10.1145/169059.169078

[Sty08] Stylos, J., Graf, B., Busse, D. K., Ziegler, C., Ehret, R., & Karstens, J. (2008). A case study of API redesign for improved usability. *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, 189-192. https://doi.org/10.1109/VLHCC.2008.4639083

[Sty09] Stylos, J., Myers, B. A., & Yang, Z. (2009). Jadeite : Improving API Documentation Using Usage Information. *CHI '09 Extended Abstracts on Human Factors in Computing Systems*, 4429–4434. https://doi.org/10.1145/1520340.1520678

[Swe85] Sweet, R. E. (1985). The Mesa Programming Environment. *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, 216–229. https://doi.org/10.1145/800225.806843

[Tio19] TIOBE. (2019, septembre). TIOBE Index | TIOBE - The Software Quality Company. Consulté 15 septembre 2019, à l'adresse https://www.tiobe.com/tiobe-index/

[Tuc04] Tucker, A. B. (2004). *Computer Science Handbook, Second Edition*. Chapman & Hall/CRC.

[Uni17] Unity Technologies. (2017). Unity - Scripting API: MonoBehaviour. Consulté 16 avril 2018, à l'adresse https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

[Uni19] Unity Technologies. (2019). DOTS - Unity's new multithreaded Data-Oriented Technology Stack. Consulté 19 septembre 2019, à l'adresse Unity website: https://unity.com/dots

[Vic12] Victor, B. (2012, septembre). Learnable programming. Consulté 12 septembre 2019, à l'adresse Bret Victor, beast of burden website: http://worrydream.com/#!/LearnableProgramming

[Vid18] Vidal, C., Berry, G., & Serrano, M. (2018). Hiphop.Js : A Language to Orchestrate Web Applications. *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2193–2195. https://doi.org/10.1145/3167132.3167440

[W3C96] W3C. (1996, décembre 17). Cascading Style Sheets, level 1. Consulté 17 mai 2018, à l'adresse https://www.w3.org/TR/CSS1/

[W3C19] W3C. (2019). All Standards and Drafts—W3C. Consulté 10 septembre 2019, à l'adresse https://www.w3.org/TR/

[Wag95] Wagner, A., Curran, P., & O'Brien, R. (1995). Drag Me, Drop Me, Treat Me Like an Object. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 525–530. https://doi.org/10.1145/223904.223975

[Wei97] Weiser, M., & Brown, J. S. (1997). The Coming Age of Calm Technology. In P. J. Denning & R. M. Metcalfe (Éd.), *Beyond Calculation : The Next Fifty Years of Computing* (p. 75-85). https://doi.org/10.1007/978-1-4612-0685-9_6

[Wei10] Weiss, M., & Sari, S. (2010). Evolution of the Mashup Ecosystem by Copying. *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, 11:1–11:7. https://doi.org/10.1145/1944999.1945010

[Wen01] Wengraf, T. (2001). *Qualitative Research Interviewing : Biographic Narrative and Semi-Structured Methods*. London ; Thousand Oaks, Calif: SAGE Publications Ltd.

[WHA15] The WHATWG community. (2015, novembre 19). DOM Standard. Consulté 17 janvier 2019, à l'adresse Web Hypertext Application Technology Working Group website: https://dom.spec.whatwg.org/

[Wik19] Wikipedia. (2019). Hacking. In *Wikipédia*. Consulté à l'adresse https://fr.wikipedia.org/w/index.php?title=Hacking&oldid=162099826

[Wil97] Wiley, D. J., & Hahn, J. K. (1997). Interpolation synthesis of articulated figure motion. *IEEE Computer Graphics and Applications*, *17*(6), 39-45. https://doi.org/10.1109/38.626968

[Yu08] Yu, J., Benatallah, B., Casati, F., & Daniel, F. (2008). Understanding Mashup Development. *IEEE Internet Computing*, *12*(5), 44-52. https://doi.org/10.1109/MIC.2008.114

[Zha12] Zhai, S., & Kristensson, P. O. (2012). The Word-gesture Keyboard : Reimagining Keyboard Interaction. *Commun. ACM*, *55*(9), 91–101. https://doi.org/10.1145/2330667.2330689

[Zho09] Zhong, H., Xie, T., Zhang, L., Pei, J., & Mei, H. (2009). MAPO: Mining and Recommending API Usage Patterns. *ECOOP 2009 – Object-Oriented Programming*, 318-343. https://doi.org/10.1007/978-3-642-03013-0_15

[Zib11] Zibran, M. F., Eishita, F. Z., & Roy, C. K. (2011). Useful, But Usable? Factors Affecting the Usability of APIs. *2011 18th Working Conference on Reverse Engineering*, 151-155. https://doi.org/10.1109/WCRE.2011.26

Annexe A. Plans des études

A.1 Plan des interviews

Introduction

This interview is part of my PhD, where I look at the limits of GUI libraries and frameworks for prototyping and building interactive applications, or advanced interaction techniques (Qt, Cocoa, Swing, SDL, …), and in particular how they are hacked around in actual projects, to get things done. I would like to backtrack with you a few of your past works, where the library could not do everything you intended, so you had to hack your way in. I selected some on the Internet already, but we can review another one if it is more relevant to you.

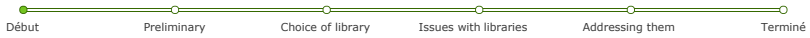
Questions

1. Age? Years of experience? In main language? Frequency of programming? Languages/IDEs/frameworks of choice?
2. Which platform/language/IDE/framework(s) did you use, and why these choices? Approximately how many lines of code is the project? How long did it take to code it? How many versions did you do, w.r.t. refactoring?
3. At which point in the design/prototyping process did you get a working software prototype? What did it implement already? What was left to implement?
4. What were your ambitions at the start of the project? Is there some of your ideas that you could not implement and test because of technological issues/limitations?
5. What was the most difficult thing you had to implement? Would you consider it *hacking*? What would you consider *low-level* there?
6. On a Likert scale (from 1 very dirty to 5 very clean), how “dirty” is it now? If you had the opportunity to recode it, how different would it be?
7. How did you learn the framework(s)? (official doc, book, tutorials, copy/paste examples) How much time did you dedicate to it? Did you have to learn some additional API over the course of the project?
8. With hindsight, what would have helped you best to complete the project? (excluding any library done after) A better framework? A better tutorial?
9. Now if you were to add this code to one of the libraries you used, which one would it be? (higher/lower level, new library) Why?
10. How do you think the framework(s) should have been designed to best suit your need? (may answer weeks later)
11. [Do you have any expectations about my work? :]

Final words

Thank you for your time!
I can send you the results of this study later if you want. Also, it would be nice if we can schedule a short meeting in about two weeks, in case you have some more feedback for this study.

A.2 Plan du questionnaire



The goal of this survey is to better understand the process of programming new interactive artefacts for research and innovation purposes, and the software libraries used to support this process. We are interested in projects where you reached the limits of libraries (e.g. undocumented needs, accessing private functionalities, interfacing with existing applications, combining with other libraries), for the prototyping and implementation of innovative interactive artefacts (e.g. non-standard interaction techniques, data visualisations, UI toolkits, interactive applications).

This research is conducted by Thibault Raffaillac (thibault.raffaillac@inria.fr) and Stéphane Huot (stephane.huot@inria.fr), members of the Inria's team Loki (http://loki.lille.inria.fr/).

This survey takes about 20 minutes to complete. Please read this consent form carefully before proceeding.

Basis to take part in this survey

You must be over 18 years old.
You must have prior experience in developing interactive applications, preferably in a context of research and/or innovation.

Voluntary participation in the project

Your participation to this survey is entirely voluntary, and without any constraints or outside pressure. If you are lacking information needed to make your decision, or have any questions about the project or your rights as a participant, do not hesitate to ask for additional information from the contact person (thibault.raffaillac@inria.fr).

Withdrawal from the project at any time

You are free to terminate your participation to this survey at any time, without any justification, by closing this web page. In this event, the answers you already have responded will not be logged and their will be no trace of your participation in our data.

Anonymity and confidentiality

All the raw information collected from your participation is anonymous and only the members of the research project may have access to it. The data we are collecting is only your answers to the questions (checkboxes and text). We are not collecting any personal data (e.g. email, IP) and the survey system we are using does not allow to retrieve your identity while you are answering the survey or once you have submitted your answers.
Collected data will be used to illustrate the results of our work in scientific communications (articles, posters, oral presentations) or scientific mediation after it has been processed and analyzed. Thus, there will be no way to identify you as a participant. In any case, while processing the data, we will take care to anonymize any collected data that could be used to refer back to you as an individual in your answers.

Informed consent *

☐ I acknowledge that I have read and understood this consent form, and voluntarily consent to participate in this research project

Page suivante >

Début — Preliminary — Choice of library — Issues with libraries — Addressing them — Terminé

Main professional activity*

- Researcher
- Interaction designer
- Project manager
- Engineer
- Software developer
- Other

Other main professional activity

Which kind of institution/company do you work for? *

- University/School
- Public institution
- Startup
- Self-employed
- Small enterprise (< 50 employees)
- Medium enterprise (< 250 employees)
- Big enterprise (≥ 250 employees)

How many people on average (excl. yourself) are working with you on a single project?*

- 0
- 1
- 2
- 3
- 4
- 5+

How do you evaluate your own expertise in writing interactive applications?*

- Basic knowledge
- Novice
- Intermediate
- Advanced
- Expert

How much of your time (professional or leisure) do you devote to programming? *

- 0%~20%
- 20%~40%
- 40%~60%
- 60%~80%
- 80%~100%

< Page précédente | Page suivante >

Début — Preliminary — Choice of library — Issues with libraries — Addressing them — Terminé

In this section we ask you to evaluate the importance of various criteria when choosing a library for a project. Here we consider general-purpose frameworks (e.g. Qt, Cocoa, JavaFX, ReactJS) as well as research toolkits for specific needs (e.g. remote collaboration, wall-sized displays, low-power devices). Below each criterion you may give examples to illustrate your answer.

Which frameworks/toolkits do you use the most? (comma-separated)

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Tool reputation (timely and frequent bug fixes, online comparisons, recommendation from peers, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Developer reputation (number of developers, brand, other tools from the same developer(s)/company, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
User community (activity, size, companies using it, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Documentation quality (abundance, clarity, time to learn, examples, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Range of use cases supported (number, variety, relation to your context, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Quality of the API (coherency, simplicity, adequate paradigm for your needs, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Compatibility with other tools/libraries (bindings already available, extensibility, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Project constraints (already in use, expertise from colleagues, licensing terms, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Personal experience/familiarity with the library *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Technical efficiency (performance, latency, memory use, ...) *	○	○	○	○	○

Example criteria

Are there other important criteria we did not mention here? (comma-separated)

Début — Preliminary — Choice of library — Issues with libraries — Addressing them — Terminé

In this section we ask you to rate the severity of selected issues, that you might have encountered while using interaction libraries. In the table below you may select "Not met" if you never encountered a given issue. Otherwise, indicate how difficult it was to overcome.

Did you encounter any of these issues, and if so how much did they hinder your work at worst? *

	Not met	Met, no trouble	Met, minor trouble	Met, medium trouble	Met, major trouble	Met, insurmountable trouble
Inadequate paradigm for this particular context *	○	○	○	○	○	○
Problems scaling up (in speed/precision/frequency) *	○	○	○	○	○	○
API too complex to use/understand *	○	○	○	○	○	○
Documentation lacking context and examples *	○	○	○	○	○	○
Significant effect/behaviour being undocumented *	○	○	○	○	○	○
Bad compatibility between two libraries *	○	○	○	○	○	○
Forbidden access to functions and data *	○	○	○	○	○	○
Inconsistent behavior across versions/systems *	○	○	○	○	○	○
Documentation requiring too much investment *	○	○	○	○	○	○
Non-deterministic behavior*	○	○	○	○	○	○
Lack of a functionality that would require pulling another library *	○	○	○	○	○	○
Buggy implementation *	○	○	○	○	○	○
Lack of configurability in API *	○	○	○	○	○	○

< Page précédente | Page suivante >

Début — Preliminary — Choice of library — Issues with libraries — Addressing them — Terminé

In this section we consider how you addressed the various issues met in the previous section. We selected a number of coding techniques from previous interviews, and ask you to rate their prevalence in your work. For each technique there is an optional text entry, where you may give examples of the coding techniques you used.

	Never	Rarely	Sometimes	Very often	Always
Using an external mechanism to obtain and process data that is not exposed by an application *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Using accessible raw data to reconstruct/reinterpret a state that you do not have access to *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Aggregating multiple sources of interaction data (input, sensors, events), and fusing them into a single source *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Reimplementing an existing widget/mechanism to gain more control over its appearance/behavior *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Reimplementing low-level system components (e.g. driver) to improve their functionalities or better support specific hardware devices *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Introducing a different programming model, pattern or paradigm on top of the existing framework or toolkit *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Using a visual overlay to add custom functionality *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Reverse-engineering a closed tool or library to acquire understanding of its inner working*	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Modifying the environment of a tool rather than the tool itself to change its behavior *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Purposely setting a parameter outside of its intended/expected range of values *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Reproducing a fake application to control a specific aspect *	○	○	○	○	○

Example techniques

< Page précédente Soumettre