Application du modèle Entité-Composant-Système à la programmation d'interactions

Thibault Raffaillac, Stéphane Huot

Application du modèle Entité-Composant-Système à la programmation d'interactions

Applying the Entity-Component-System Model to Interaction Programming

Thibault Raffaillac
Inria, Univ. Lille, UMR 9189 - CRIStAL, Lille, France
thibault.raffaillac@inria.fr

Stéphane Huot
Inria, Univ. Lille, UMR 9189 - CRIStAL, Lille, France
stephane.huot@inria.fr

ABSTRACT

This paper introduces a new GUI framework based on the Entity-Component-System model (ECS), where interactive elements (Entities) can acquire any data (Components). Behaviors are managed by continuously running processes (Systems) which select entities by the components they possess. This model facilitates the handling and reuse of behaviors. It allows to define the interaction modalities of an application globally, by formulating them as a set of Systems. We present *Polyphony*, an experimental toolkit implementing this approach, detail our interpretation of the ECS model in the context of GUIs, and demonstrate its use with a sample application.

CCS CONCEPTS

• **Human-centered computing** → *User interface programming*; *User interface toolkits*;

KEYWORDS

User Interface Toolkit, Interaction Programming, Entity-Component-System, ECS

RÉSUMÉ

Cet article présente un nouveau cadre de conception d'IHM basé sur le modèle Entité-Composant-Système (ECS). Dans ce modèle, les éléments interactifs (Entités) acquièrent librement des données (Composants). Les comportements sont régis par des processus communs s'exécutant continuellement (Systèmes), qui sélectionnent les entités par les composants qu'elles possèdent. Ce modèle favorise la manipulation et la réutilisation des comportements. Il permet de définir globalement les modalités d'interaction d'une application, en les formulant par un ensemble de systèmes. Nous présentons *Polyphony*, une boîte à outils expérimentale implémentant cette approche, détaillons notre interprétation du modèle ECS en contexte IHM, et l'illustrons avec un exemple d'application.

MOTS-CLEFS

boîte à outils d'IHM, programmation d'interactions, Entité-Composant-Système, ECS

1 INTRODUCTION

Dans un système interactif, programmer les comportements et les réactions aux événements utilisateurs est une tâche qui s'avère souvent difficile [29]. Plusieurs raisons ont été mises en avant, telles que la complexité des interfaces liée au nombre d'éléments qui les composent [31, 41] ou le besoin de connaissances et outils techniques spécifiques à la création d'interactions avancées [6, 15, 17, 33]. De fait – et malgré de nombreuses propositions de modèles, architectures logicielles et boîtes à outils –, le prototypage, l'implémentation et la diffusion de nouvelles techniques d'interaction restent des problèmes d'actualité.

Parmi les solutions possibles et peu approfondies dans ce contexte, nous proposons d'appliquer le modèle *Entité-Composant-Système* (ECS) [28] à la programmation d'interactions utilisateur. Dans ce modèle, les éléments ou objets de l'application – les *Entités* – ne possèdent aucun comportement propre. Leurs comportements sont gérés par des *Systèmes*, opérant chacun sur les Entités possédant certaines données, les *Composants*. Ce modèle, issu du domaine des jeux vidéos, permet par exemple de modéliser la gravité comme un Système qui opère sur tout élément (Entité) possédant des Composants masse et position. Il permet donc d'exprimer une interface graphique non comme la réaction d'éléments à leur environnement, mais comme l'action de règles universelles sur des ensembles d'éléments – telles la répulsion, le contact, et la friction.

Pour illustrer et explorer les apports de ce modèle à la programmation d'interactions, nous avons réalisé la boîte à outils *Polyphony*. Dans cette bibliothèque, les boutons, zones de texte, mais aussi la souris, la fenêtre, sont des Entités. Leurs propriétés sont des Composants pouvant être acquis et retirés dynamiquement. Les Systèmes sont des fonctions itérant en séquence à chaque événement d'entrée/sortie, qui appliquent chacun un type de comportement interactif (affichage des bordures, drag & drop, saisie de texte, ...). Par exemple, un Système dessine une image pour chaque Entité possédant les composants `position`, `drawableIcon` et `iconPosition`, sur chaque Entité (fenêtre) possédant un composant `graphicsContext`. Les Systèmes sont eux-mêmes des Entités, permettant d'utiliser des Composants pour représenter leurs comportements tels que des relations de précédence ou des abonnements aux événements.

La première partie de cet article décrit un exemple d'application de dessin vectoriel construite avec Polyphony et détaille les aspects notables de la programmation d'IHM avec ECS. La section suivante expose les choix d'implémentation d'ECS à partir de l'analyse de trois frameworks majeurs, et en détaille notre adaptation au contexte IHM. Nous comparons ensuite la programmation des comportements interactifs avec ECS à l'état de l'art, pour leur réutilisation, leur orchestration, et leur manipulation à l'exécution. Enfin nous concluons sur les limites de ce modèle, ainsi que des pistes de validation pour les travaux futurs.

2 UNE APPLICATION DE DESSIN VECTORIEL BASÉE SUR ECS

Le développement en Java de la boîte à outils Polyphony s'est fait de pair avec celui d'une application de dessin vectoriel, l'objectif étant de mettre en valeur les atouts d'ECS dans le contexte IHM.

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. Ce modèle est apparu dans le domaine du développement de jeux vidéos [9, 27, 28]. 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 utilise les Composants comme interface entre les deux mondes: les programmeurs définissent les Composants disponibles et créent les Systèmes itérant dessus; les concepteurs instancient les différentes Entités et leurs comportements par assemblage de Composants choisis.

Les éléments constituant le modèle sont donc:

Entités - Ce sont des identifiants uniques (souvent de simples entiers) représentant les éléments du programme. Ils sont similaires aux *objets* mais ne possèdent aucune donnée.

Composants - Ils représentent les données du programme (comme `TextPosition` ou `BackgroundColor`), 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 modulable du programme. Les Entités ne s'enregistrent pas directement auprès des Systèmes, mais acquièrent les Composants nécessaires pour être "vus" par ceux-ci.

Enfin, nous ajoutons deux 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:

Sélections - Ce sont des groupes d'Entités, constitués en fonction des Composants possédés. Les Sélections sont le mécanisme fondamental par lequel les Systèmes obtiennent des listes d'Entités sur lesquelles opérer.

Contexte - Il représente le *monde* qui contient toutes les Entités, stocke les variables globales, et offre une interface pour obtenir des Sélections.

2.2 Description du prototype

L'application de dessin vectoriel est un exemple courant et bien approprié pour illustrer la conception de techniques d'interaction [1, 20]. Elle combine l'implémentation d'objets graphiques, d'outils (e.g. *dessin de formes*, *pipette à couleurs*), de commandes (e.g. *copier/coller*, *undo*), et offre une large palette de tâches possibles permettant de nombreuses améliorations et combinaisons entre éléments.

Notre application de base (figure 1) permet ainsi de:

- dessiner des rectangles et des ovales
- déplacer les formes créées
- enregistrer le résultat au format SVG
- réinitialiser l'espace de travail

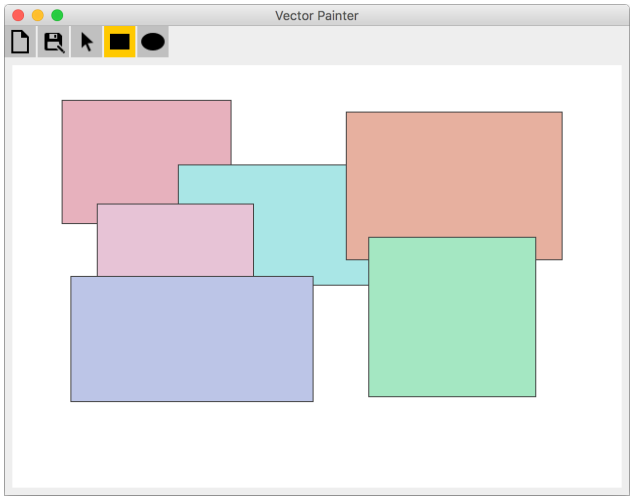

Figure 1: Espace de travail de l'application, avec au centre le canevas de dessin et en haut la barre d'outils.

Tous les éléments de l'interface sont donc des Entités: les boutons, les formes dessinées, et même la zone de dessin en arrière-plan. Les périphériques d'interaction (souris et clavier) sont aussi représentés par des Entités, ainsi que la fenêtre d'affichage. Enfin, dans la continuité des fonctions *lambda*, qui sont des fonctions réifiées en tant qu'objets, nous avons implémenté les Systèmes par des Entités. Leurs dépendances sont ainsi représentées par des données stockées dans leurs Composants.

Les Entités sont instanciées par des fabriques d'Entités, qui leur attribuent des Composants prédéterminés. Un bouton, par exemple, est instancié par la fabrique `Button`, selon le pseudo-code suivant :

```
function Button(x, y, icon)
   r = Rectangle(0, 0, 8+i.width, 8+i.height)
   e = Entity()
   e.add("position", Point(x, y))
   e.add("drawableShape", r)
   e.add("backgroundColor", LIGHT_GRAY)
   e.add("drawableIcon", icon)
   e.add("iconPosition", Point(4, 4))
   e.add("targetableShape", r)
   return e
end
```

Composants	Button	Canvas	Figure	Mouse	Keyboard	View	*System
position	×	×	×	×		×	
parent			~				
children		~					
depth		~	~				
isRecordable			~				
drawableShape	×	×	×				
backgroundColor	×	×	×				
borderColor			×				
drawableIcon	×	×	×	×			
iconPosition	×	×	×	×			
selectableTool	~						
targetableShape	×		×				
isDragable			×				
onDrop	~						
isPointer				×			
buttonStates				×			
keyStates					×		
graphicsContext						×	
execute							×
priority							×
shortcut							~
selection							~
readsMouseMotion							~
readsMouseButton							~
readsKeyboard							~
readsDisplaySync							~
readsEverything							~
isDisabled							~

Tableau 1: Matrice des Composants définis dans l'application, en fonction des fabriques d'Entités les attribuant. Les Entités issues de chaque fabrique reçoivent tous les Composants indiqués par ×, et éventuellement ceux indiqués par ~ (en fonction de leur spécialisation).

La figure 2 représente toutes les Entités actives dans l'application de dessin, chacune identifiée par la fabrique qui l'a instanciée. Contrairement à des boîtes à outils répandues (e.g., Qt, JavaFX, HTML), elles n'appartiennent pas nécessairement à un graphe de scène. Dès leur création, elles sont accessibles aux Systèmes, donc visibles et interactives. Les liens de parenté, représentés sur cette figure par des flèches, sont définis uniquement lorsqu'il faut établir une relation d'ordre (profondeur d'affichage, précédence de ciblage).

Les Composants déterminent les comportements pouvant être acquis par chaque Entité. Ils sont lus par chaque Système qui met en application ces comportements. La table 1 dresse la liste des Composants de l'application de dessin, ainsi que les fabriques qui les attribuent. Pour les définir, nous nous sommes inspirés de CSS 1 [40], qui synthétise la plupart des comportements communs aux différentes balises HTML, et permet par exemple de mimer un élément `button` avec un conteneur `div`, uniquement par acquisition de propriétés CSS. Nous avons délibérément laissé de côté les notions de *layout*, et de décoration de texte.

2.3 Utilisation d'ECS en pratique

Nous détaillons ici les aspects du développement d'interfaces qui nous ont semblé notables avec ECS, par rapport aux boîtes à outils les plus répandues.

D'abord, la mise en place de Systèmes permet de centraliser la définition des interactions et des comportements du système, à la manière des machines à état de SwingStates [2], et donc de réduire l'utilisation de *callbacks* dont le nombre peut être problématique [31]. Le sélecteur d'outils de notre application de dessin illustre ce principe. Un Système `ToolSelectionSystem` observe à chaque clic de souris si un des boutons possédant le Composant `selectableTool` a été activé. Si c'est le cas il désactive l'outil précédent et fait de `selectableTool` le nouvel outil actif. Ainsi, aucun des boutons de sélection d'outils n'implémente de callback `onClick`, ce comportement commun étant géré dans un Système de plus haut niveau. De plus si le concepteur souhaite ajouter un nouvel outil a posteriori, il lui suffit d'ajouter le Composant `selectableTool` à l'Entité concernée pour que le Système `ToolSelectionSystem` le prenne en compte. Il n'y a donc pas besoin d'ajouter de nouveaux callbacks, ni de modifier le code de l'application en plusieurs endroits.

Grâce au principe des Sélections, les Entités ne stockent pas de listes de pointeurs entre elles. Par exemple, le Système `ToolSelectionSystem` ne conserve pas une liste des boutons activables, mais chacun de ces boutons possède le Composant `selectableTool`, par lequel il peut être retrouvé avec une Sélection. On limite ainsi le nombre de références entre objets dans l'application, qui compliquent généralement la suppression et la mutation des objets référencés.

Les Entités *périphériques* (`Mouse`, `Keyboard`, `View` – voir figure 2) sont l'interface avec le système d'exploitation. La souris `Mouse`, par exemple, possède le Composant `position` qui stocke à tout instant les coordonnées du curseur système. Ces coordonnées sont modifiables, ce qui force en retour la position du curseur. Dans l'application de dessin, ce mécanisme sert à contraindre le curseur à rester dans le canevas lors du dessin d'une nouvelle forme.

Avec ECS les périphériques sont caractérisés par leurs Composants: `isPointer`, `buttonStates` et `position` pour la souris, `keyStates` pour le clavier, et `graphicsContext` pour l'écran. Ce principe permet d'abstraire les périphériques en conservant leurs caractéristiques individuelles. Des données comme la présence de boutons supplémentaires sur une souris sont conservées, mais simplement ignorées par tout Système traitant des pointeurs à deux boutons.

La matérialisation des périphériques en Entités offre une représentation à la fois flexible et persistante entre les différentes couches du programme. La structure d'une Entité comme `Mouse` étant extensible, il n'est pas nécessaire de créer un nouvel objet pour introduire de nouvelles données. Ce mécanisme est illustré en figure 3. L'Entité représentant la souris est initialisée par la fabrique `Mouse` avec trois Composants. Puis chaque événement `mouseButton` déclenche une série de Systèmes, chacun ajoutant et retirant des Composants à la souris. Les données sont ainsi partagées entre les différent Systèmes, utilisant l'Entité souris comme support de stockage.

Enfin, grâce à l'utilisation systématique des Sélections pour retrouver les Entités, les techniques d'interaction sont exposées en amont à la possibilité d'utiliser des périphériques multiples. Bien que nous ne l'ayons pas démontré dans notre prototype, l'utilisation de plusieurs périphériques d'entrée et sortie (physiques ou virtuels), ainsi que leur remplacement est facilité car implicite, tant qu'ils présentent les Composants nécessaires aux Systèmes qui les utilisent. Un Système implémentant une technique de pointage pourra par exemple fonctionner avec toute Entité qui fournit les composants `position`, `isPointer` et `buttonStates`, que ce soit une souris, une tablette ou un dispositif virtuel produisant ces données (e.g. un "robot" reproduisant des enregistrements d'entrées).

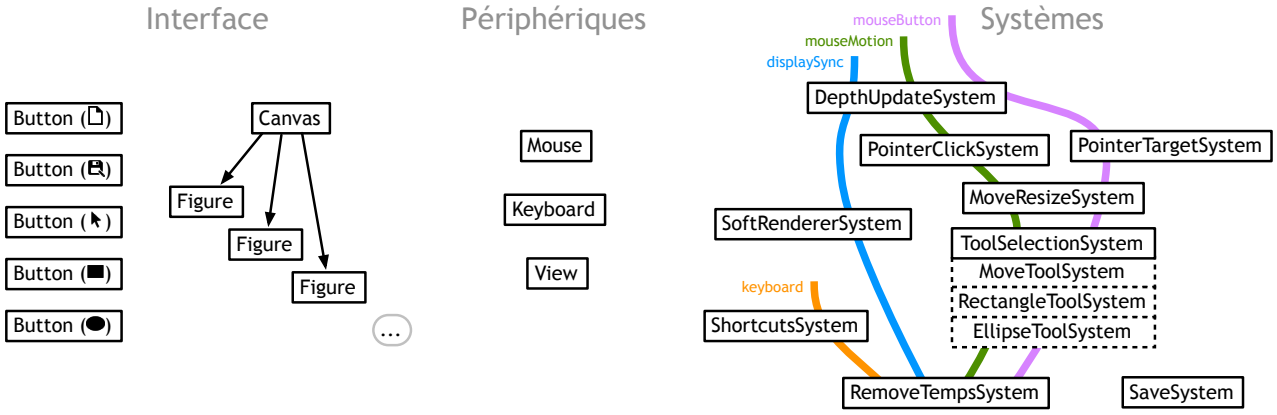

Figure 2: Liste des Entités définies dans l'application de dessin. Les flèches à gauche représentent les liens directs de parenté entre Entités. Les lignes colorées à droite indiquent les Systèmes s'exécutant pour chaque type de déclencheur, de haut en bas.

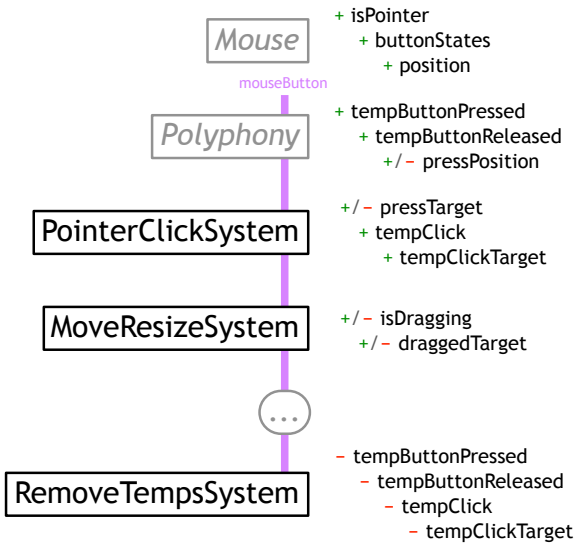

Figure 3: Évolution des Composants de l'Entité souris, à travers les Systèmes réagissant à l'événement `mouseButton`.

L'utilisation d'Entités pour matérialiser les Systèmes nous permet d'utiliser des Composants booléens pour modéliser plusieurs types de déclencheurs – après *quels événements* ils s'exécutent. Pour les besoins de notre application d'exemple, nous avons défini quatre types de déclencheurs:

- le rendu graphique (`readsDisplaySync`)
- les clics de souris (`readsMouseButton`)
- les mouvements de souris (`readsMouseMotion`)
- les appuis clavier (`readsKeyboard`)

Ces déclencheurs représentent les événements issus du système d'exploitation et ont été définis pour que les Systèmes n'utilisent pas de ressources en s'exécutant lorsqu'ils n'ont pas lieu d'être. Les Systèmes acquièrent ces Composants pour indiquer s'ils doivent s'exécuter pour chacun des déclencheurs. Des *Méta-Systèmes* se chargent alors de retrouver les Systèmes possédant les Composants requis et d'exécuter leurs champs `execute`. Les Systèmes possèdent en outre des Composants `priority` pour ordonner leur exécution.

Grâce à l'utilisation de Composants, nous avons aussi défini des déclencheurs internes à l'application. Le Composant `shortcut` permet par exemple à un Système d'être déclenché par un raccourci clavier. Le Méta-Système `ShortcutsSystem` détecte l'appui d'un raccourci clavier, et se charge de retrouver et d'exécuter le(s) Système(s) réagissant à ce raccourci.

L'organisation des Systèmes permet d'ordonner les comportements par type de traitement dans l'application. Cependant certains traitements nécessitent d'être ordonnés entre Entités, comme l'affichage. Dans ce cas, le dessin de bordures, d'images et de formes d'arrière plan, sont des traitements distincts. Or il faut aussi ordonner les traitements entre Entités, comme dessiner la bordure d'une Entité en arrière plan avant l'ombre d'une Entité au premier plan. Dans ce cas la programmation par Systèmes est mise en défaut : il faut implémenter tous les traitements dans un même Système itérant sur les Entités. L'application des principes d'ECS nécessite donc de minimiser les relations d'ordre d'exécution entre Entités. Dans le cas du rendu graphique, l'utilisation d'un *depth-buffer* permet de dessiner tous les éléments sans ordre nécessaire.

2.4 Exemple d'implémentation du drag & drop

Pour illustrer l'utilisation de Polyphony, nous détaillons un exemple souvent difficile à implémenter dans la plupart des bibliothèques actuelles : le drag and drop. Cette technique d'interaction à la souris consiste à déplacer un objet et le déposer sur un autre. Ensuite, si les deux objets sont compatibles, une commande s'exécute, qui dépend des deux objets et de l'endroit du dépôt.

Dans l'application de dessin, il s'agit de supporter le déplacement des formes dessinées sur les boutons de la barre d'outils, pour les faire changer de forme, ou les supprimer. Ainsi l'outil de réinitialisation du canevas supprime la forme déplacée, l'outil de rectangles la transforme en un rectangle (en conservant ses dimensions), et l'outil d'ellipses le transforme en ellipse.

Les combinaisons de comportements sont synthétisées dans le tableau 2. L'axe vertical représente le type de forme déplacée, et l'axe horizontal le bouton sur lequel elle est relâchée.

La séquence d'actions à exécuter est la suivante:

- le curseur survole l'objet à déplacer;
- le curseur clique sur l'objet à déplacer;
- le curseur se déplace et l'objet avec lui;
- le curseur survole un objet de dépôt (un mécanisme de *feedforward* peut suggérer que l'action de dépôt aura un effet);
- le curseur relâche l'objet déplacé (une commande dépendant des deux objets s'exécute alors).

	🗋	■	●
Rectangle	suppression		→ Ellipse
Ellipse	suppression	→ Rectangle	

Tableau 2: Matrice des combinaisons de drag and drop.

Les difficultés liées à cette technique sont de plusieurs natures. D'abord, déplacer un objet implique de le retirer de son emplacement actuel, ce qui peut mettre en défaut les contraintes de positionnement locales. Ensuite, les différentes étapes nécessitent d'attendre des actions de l'utilisateur, et peuvent donc difficilement s'exprimer par un bloc de code continu (se sont souvent des callbacks multiples). 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 est fonction du produit du nombre de types déplaçables par le nombre de types receveurs, et peut être très important. En fonction des cas, les comportements résultants peuvent donc appartenir aux objets déplacés, ou aux objets de dépôt.

Dans Polyphony, le Système `MoveResizeSystem` est dédié à l'action d'appui-déplacement, à la manière de l'Interacteur Move-Grow de Myers [30]. Celui-ci prend en entrée les clics de souris générés par le système d'exploitation, ainsi que les cibles potentielles générées par le Système `PointerTargetSystem`. Il génère en sortie les Composants `isDragging` et `draggedTarget` sur la souris (c.f. figure 3), ainsi que `isDragged` et `originalPosition` pour chaque cible possédant le Composant `isDraggable`. L'état dans la séquence d'actions est mémorisé grâce à ces Composants temporaires: `draggedTarget` indique quel objet est en cours de déplacement et `originalPosition` conserve sa position initiale pour la rétablir éventuellement.

Les comportements de *drop* étant ici uniquement déterminés par le receveur, nous les avons implémentés par des *callbacks* sur les boutons, avec des Composants `onDrop`. `MoveResizeSystem` s'occupe de vérifier si la cible du dépôt possède ce Composant et l'exécute le cas échéant. Ainsi le comportement de drag and drop est entièrement contenu dans un Système et les comportements spécifiques aux receveurs sont stockés sur ceux-ci.

Dans notre application exemple, les Composants générés par `MoveResizeSystem` sont ensuite utilisés par les Systèmes `MoveToolSystem`, `RectangleToolSystem` et `EllipseToolSystem`. Le premier permet de déplacer les formes géométriques, les deux autres de dessiner de nouvelles formes par tracé de boîtes.

Ce séquencement illustre la composition des comportements avec ECS. L'outil de tracé de rectangles `RectangleToolSystem` se base sur le Système de détection d'appui-déplacement `MoveResizeSystem`, qui dépend lui-même de `PointerTargetSystem` pour la détection de la cible pointée. La dépendance se modélise par l'ordre d'exécution dans la chaîne des Systèmes. Chacun insère des Composants sur l'Entité souris, en se basant sur les Composants insérés par les Systèmes précédents. Ainsi, `MoveResizeSystem`, `RectangleToolSystem` et `EllipseToolSystem` sont simplement positionnés en aval dans la chaîne d'exécution des Systèmes.

3 CLARIFICATION DU MODÈLE ENTITÉ-COMPOSANT-SYSTÈME

ECS est un paradigme de haut-niveau, en ce qu'il promeut des principes sans spécifier précisément la manière dont il faut les réaliser. Cette formalisation des Systèmes et des Composants relativement floue a donc donné lieu à différentes interprétations et implémentations que nous détaillons ici.

3.1 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 clarifier ce point, nous présentons ici un *espace de conception* d'ECS, à partir de l'étude de 3 frameworks majeurs, ainsi que notre expérience dans le développement d'une nouvelle variante dédiée à la programmation de l'interaction.

Pour constituer un espace de conception et réaliser notre propre prototype, nous avons d'abord sélectionné un ensemble de frameworks à étudier basés sur ECS. Nous avons considéré en priorité les travaux les plus cités comme référence[1], et ceux bénéficiant des documentations les plus complètes. Tous les frameworks que nous avons trouvés ciblent le développement de jeux vidéos, et nous n'en avons trouvé aucun dédié aux interfaces graphiques.

Artemis (Java) est une des premières implémentations du modèle ECS. La version originale n'étant pas documentée et plus maintenue, nous avons sélectionné son *fork* le plus populaire, Artemis-odb [34].

Entitas [37] (C#) propose une interprétation distincte d'Artemis en plus de fournir un méta-langage ainsi qu'un préprocesseur pour C#. Sa très grande popularité ainsi que sa documentation abondante en font une variante majeure d'ECS.

Enfin, il existe un grand nombre de travaux qui n'implémentent pas les comportements dans des Systèmes distincts, mais directement sur les Composants. Ils sont souvent qualifiés d'*Entity-Component systems*, ce qui alimente la confusion quant à leur adhérence au modèle ECS original. Parmi ces travaux, nous pouvons citer Unity [39] (C#), Google CORGI [21] (C++), Nez [35] (C#), et Apple GameplayKit [4] (Swift). Nous avons sélectionné ce dernier pour sa documentation complète.

L'analyse présentée dans la table 3 n'est pas exhaustive. Elle met en évidence les choix de conception sur lesquels les interprétations diffèrent principalement. Nous y avons aussi inclus nos propres choix, et les expliquons dans le reste de la section.

3.2 Adaptation d'ECS au contexte IHM

Toutes les boîtes à outils implémentant ECS que nous avons étudiées sont dédiées exclusivement au jeu vidéo. Or ce type d'applications a des besoins très particuliers, parfois similaires, mais souvent différents de la programmation d'interactions:

[1] en particulier à l'aide de sites comme http://entity-systems.wikidot.com/

Bibliothèque	Artemis-odb (Java)	Entitas (C#)	GameplayKit (Swift)	Polyphony (Java)
Représentation des Entités	entier, ou entier encapsulé dans un objet	objet	objet	objet
Définition d'un Composant	sous-classe de `Component` avec valeurs par défaut	sous-classe de `IComponent`	sous-classe de `GKComponent` avec valeurs par défaut	*non défini*
Définition d'un Système	sous-classe de `BaseSystem`, peut être lié à une sélection d'Entités, exécution périodique	sous-classe de `ISystem`, peut être lié à une sélection d'Entités, exécution périodique ou unitaire	méthode de Composant `updateWithDeltaTime:`	Entité avec Composant `execute`
Structuration du Contexte	objet(s) matérialisant le Contexte	objet(s) matérialisant le Contexte	arbre de scène	singleton matérialisant le Contexte
Sélection d'Entités	par Composants possédés, avec algèbre *all/one/none*	par Composants possédés, avec algèbre *all/one/none*	par Composants possédés, ou par conditions programmables	par Composants possédés, ou par conditions programmables
Comportements transitoires	*listeners* sur Sélections d'Entités	*listeners* sur Sélections d'Entités	*non explicitement supportés*	accesseurs sur Sélections d'Entités
Stockage des Composants	par Composant, avec dictionnaire Entité → Valeur	par Entité, avec emplacements prédéterminés	par Entité, avec dictionnaire Composant → Valeur	par Entité, avec dictionnaire Composant → Valeur
Ordonnancement des Systèmes	dynamiquement par priorité	dynamiquement par insertion dans une liste `Systems`	statiquement	dynamiquement par priorité
Réutilisation des Entités	objet *factory*, chargement depuis un fichier	chargement depuis un fichier	*non explicitement supporté*	objet *factory*
Extension aux langages	post-traitement de l'exécutable compilé	précompilation d'une surcouche du langage		

Tableau 3: Analyse de trois variantes d'ECS et de Polyphony selon notre espace de conception.

- l'état du jeu est généralement mis à jour par une unique chaîne de Systèmes, à un *tickrate* défini souvent diviseur de 60Hz (fréquence commune de rafraîchissement des écrans). Les interfaces graphiques étant généralement mises à jour par réaction à diverses sources d'événements, nous avons introduit de multiples chaînes de Systèmes;
- les éléments du jeu ont des relations peu structurées entre eux, ils se déplacent, apparaissent, disparaissent, et sont difficilement organisables en parents et enfants, alors que les interfaces ont des relations de contenance et de proximité qui se structurent mieux par des arbres de scène;
- nombre de jeux obéissent au principe *un joueur par machine*, et supportent un seul couple clavier/souris (mais parfois plusieurs manettes), ce qui est limitant pour le prototypage de techniques d'interaction qui motive notre travail. Nous avons introduit l'usage d'Entités *périphériques* pour y pallier;
- le *pipeline* d'exécution des différents comportements est généralement fixe, ce pourquoi les implémentations d'ECS ordonnent les Systèmes par simples listes. Pour permettre le prototypage par dessus des applications existantes, nous représentons les Systèmes par des Entités, et assouplissons ainsi les déclencheurs grâce aux Composants;
- les scènes sont des "niveaux", souvent indépendants, se succédant en séquence, et sérialisés dans des fichiers chargés au démarrage, alors que la navigation dans des interfaces est beaucoup plus complexe que séquentielle. ECS se prête déjà bien à ce changement de contexte.

3.3 Polyphony: choix de conception

Dans cette section, nous détaillons chacun de nos choix pour Polyphony, selon l'espace de conception présenté plus haut.

Tout d'abord nous avons choisi de représenter les Entités à l'aide d'objets plutôt que des entiers. En effet la programmation d'IHM est très liée au modèle objet aujourd'hui, et le but de notre travail sur ECS est de l'en rapprocher, afin qu'il puisse à terme être combiné avec des paradigmes existants.

Ainsi, la gestion des composants se fait à l'aide d'un protocole de *Méta Objet* inspiré de JavaScript [19] :

- `e.set("c",v)` ajoute le composant `c` de valeur `v` à l'Entité `e`
- `e.add("c",v)` ajoute `c` s'il n'existe pas déjà
- `e.get("c")` renvoie la valeur du composant (ou `null`)
- `e.has("c")` renvoie `true` si et seulement si `e` possède `c`
- `e.remove("c")` retire `c` de l'Entité `e`

L'intérêt de la méthode `add` est de permettre à une éventuelle fabrique d'Entités de recevoir une Entité existante et de compléter uniquement les champs qui lui manquent.

Pour faciliter le prototypage et l'ajout de nouveaux Composants, nous avons choisi de ne pas nécessiter leur définition explicite. Les Composants sont gérés par simple association d'une Valeur avec un Nom. Ce choix rapproche ECS des langages permettant l'ajout dynamique de variables aux objets[2], tels JavaScript et Python – `e.set("c",v)` équivaut alors à `e.c = v`.

Artemis-odb et Entitas définissent chacun une classe pour stocker les Entités actives, et représenter le Contexte. Ils permettent ainsi d'instancier plusieurs *mondes*, par exemple pour charger un prochain niveau, ou gérer un second joueur sur une même machine. Polyphony contient un Contexte global unique. En effet, il nous est toujours possible de gérer plusieurs *mondes* à l'aide de Composants – on peut alors considérer que la variable `context` de chaque Entité est aussi un Composant.

Pour la Sélection d'Entités, nous avons introduit des conditions de filtrage basées sur des prédicats *accept* : $Entity \rightarrow Boolean$. Les algèbres de combinaison de Sélections *all/one/none* peuvent alors être implémentées à l'aide des opérateurs logiques *et/ou/non* dans ces fonctions. De plus, `e.has` accepte les expressions régulières, ce qui permet par exemple au Système `RemoveTempsSystem` de parcourir toutes les Entités possédant des champs de la forme `"temp*"`, et de retirer ces champs.

Les comportements transitoires sont des processus s'exécutant lorsque des variables changent de valeur. Dans les applications basées sur ECS, on peut les opposer aux comportements *systématiques*, qui sont implémentés par les Systèmes. Dans notre exemple d'application, ces transitions sont à l'œuvre lorsqu'une Entité acquiert le Composant `depth`, et que `SoftRendererSystem` doit trier à nouveau la liste des Entités affichables. Artemis-odb et Entitas permettent l'enregistrement de *listeners* sur des Sélections. D'autres implémentations fournissent des types de Systèmes déclenchés par les mises à jour de Sélections. Nous avons choisi de fournir des accesseurs sur les Sélections, pour les Entités ajoutées et retirées depuis leur dernière lecture.

Pour l'ordonnancement des Systèmes, la pluralité des dépendances et le besoin de flexibilité nous ont poussé à utiliser le mécanisme des Composants, donc à faire des Systèmes des Entités. Ainsi tous les modèles d'exécution pouvant s'exprimer par des données attribuées aux processus, peuvent s'exprimer à l'aide de Composants. C'est le cas par exemple des relations *A s'exécute après B* et *A s'exécute si* `v` *change*. Une exception notable est le *binding* de djnn [36], qui lie deux processus à l'aide d'un objet intermédiaire, mais peut néanmoins s'implémenter à l'aide d'une troisième Entité. Lorsque les Composants des Systèmes ont été définis, des Méta-Systèmes (Systèmes exécutant des Systèmes) les parcourent, les ordonnent, et les exécutent.

Enfin, la réutilisation d'Entités avec Composants prédéterminés se fait à l'aide du patron *fabrique*. Nous rapprochons ainsi la création d'une Entité de la construction d'un objet. La fabrique fait alors office de *descripteur* d'Entité, tel un type qui pourrait être acquis ou perdu en fonction des Composants possédés.

[2]https://rosettacode.org/wiki/Add_a_variable_to_a_class_instance_at_runtime

3.4 Liens avec le langage hôte

Polyphony est développé en Java, langage au typage statique et fort. Les types de ses objets sont déterminés avant l'exécution, et il n'est pas possible d'ajouter des champs aux objets durant l'exécution. Ces limitations nous ont contraint à développer un protocole d'Entités indépendant du langage hôte, ce qui a permis en retour d'expérimenter librement des ajouts à ECS, sans dépendre de primitives réflexives de bas niveau.

ECS ne promeut pas de modèle d'héritage (classes ou prototypes), étant donné que chaque Entité est créée avec sa propre copie de chaque Composant. Dans notre application, les Entités coexistent avec les objets de Java Swing de plus bas niveau (`Point`, `Rectangle`, `Color`, etc), qui utilisent un héritage de classes. Cependant pour les Entités, nous n'avons pas encore pu étudier de cas où l'héritage puisse être utile.

La plupart des langages orientés-objets (Java, C#, Python, etc) utilisent un mécanisme de ramasse-miettes pour supprimer automatiquement les objets non-référencés. D'autres langages orientés-objets (C++, Rust) utilisent le comptage de références pour les supprimer. ECS se distingue par le besoin de supprimer manuellement les Entités – même si elles sont toujours référencées. Or la suppression manuelle sûre d'objets est pour l'instant difficilement accessible [24]. Il faut attendre que de tels langages se développent avant de pouvoir y intégrer ECS.

Aujourd'hui, l'intégration d'ECS dans un langage "compatible" est un travail difficile. D'autant que le principe des Sélections nécessite d'avoir accès à tous les objets actifs, ce qu'aucun langage ne permet simplement aujourd'hui. De même, la validation d'*interfaces* (pour les Sélections) basée uniquement sur les champs possédés (ou *typage structurel*) n'est à notre connaissance disponible que dans le langage Go[3]. Il est donc aussi envisageable à terme de développer un langage dédié orienté objet, au typage dynamique et fort, qui implémente directement les principes d'ECS.

4 GESTION DES COMPORTEMENTS INTERACTIFS

Les comportements interactifs comme nous les considérons sont définis à deux niveaux. À haut niveau, ils désignent les réactions observables des éléments à différents stimuli. Les éléments d'intérêt sont ici les objets manipulés par les bibliothèques d'interaction. Les stimuli peuvent être des événements externes (clic de souris, front montant d'horloge), ou internes (la souris survole un élément). À bas niveau, les comportements désignent les blocs de code déclenchés par ces événements et qui donnent lieu aux réactions observées. Par exemple, pour un comportement de haut niveau "permettre l'édition de texte", le code correspondant s'abonne au clavier, et pour chaque saisie de caractère l'ajoute en bout d'une chaîne de caractères dont l'affichage est mis à jour. Dans cette section, nous nous intéressons aux approches existantes pour exprimer et programmer de tels comportements interactifs et situons le modèle ECS que nous avons adopté par rapport à ces travaux.

[3]voir https://gobyexample.com/interfaces

4.1 Mutualisation et réutilisation des comportements

La plupart des boîtes à outils basées sur le modèle Objet utilisent le principe d'héritage pour organiser les comportements individuels et partagés, et peuvent être qualifiées de "monolithiques" [7]. Chaque comportement est incarné par un ensemble de variables et de méthodes dans une classe (ou un prototype), et tout objet en hérite à condition d'être un descendant de sa classe (ou prototype). Ce principe associe la *nature* même des éléments à leurs comportements (propres ou hérités): par exemple une classe `TextField` est spécifique en ce qu'elle contient du texte, et peut être éditée.

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 [7, 26, 32], et aussi bien de manière statique (à la compilation) que dynamique (à l'exécution). Par exemple, il est difficile d'ajouter le comportement d'édition de texte aux champs *label* d'une interface (e.g. `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 permette 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 [14] ou les mixins [11]. Pour un programmeur n'ayant pas accès au code source, ajouter un comportement à un élément revient à changer sa *nature*, ce qui relève parfois de l'impossible. Comme le remarque Lecolinet, "*behaviors and other features are not seen as general services that could be used by any widget*" [26].

De nombreuses bibliothèques d'interaction dites "polylithiques" [7] ont été proposées, suivant le paradigme de *Composition*. Taylor et al. [38] ont présenté une architecture par agents indépendants, échangeant des messages entre couches sans avoir connaissance des agents avec lesquels ils communiquent. Cela facilite la réutilisation de comportements à grande échelle mais pas la gestion des comportements de faible granularité (comme de l'édition de texte).

Jazz [8] modélise les comportements par des nœuds insérés entre les éléments d'interface dans un *graphe de scène*. Les comportements sont ainsi hérités par tous les enfants du graphe (qui est en fait un arbre). Un principe similaire est repris dans MaggLite [23], suivant un modèle de *graphes combinés* qui associe des comportements interactifs réutilisables aux nœuds d'un graphe de scène [22]. Dans Polyphony les Systèmes sont comparables aux graphes d'interaction de MaggLite, ainsi qu'aux Interacteurs de Garnet [30]. Cependant il se distingue de ces travaux en ce que les Systèmes sont instanciés une fois pour toutes les Entités, plutôt qu'une fois par Entité, et facilitent donc l'expression de comportements de groupe – contraintes de positionnement linéaires [5], dessin d'ombres réalistes.

4.2 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 (e.g., *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 ci-dessus: 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 [18] 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 [12] 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 [32] propose un modèle de programmation orienté Prototype au dessus de C++, dans lequel les objets peuvent ajouter, remplacer ou supprimer des variables et méthodes à tout moment. Notre implémentation basée sur ECS se rapproche beaucoup de ce travail, cependant nous étendons la notion d'Entités aux périphériques d'interaction ainsi qu'aux Systèmes, afin de fournir une plus grande flexibilité à l'exécution dans l'ordonnancement des comportements.

4.3 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'événements 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" [31].

Pour palier à ces limitations, différents modèles ont été proposés. Les modèles basés sur le formalisme états-transitions (machines à états) de HsmTk [10] et SwingStates [2] 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 [16] 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 [3] afin de tirer partie des deux formalismes aux niveaux où ils sont les plus adaptés. Dans le contexte des séquences d'actions, Proton [25] est notable pour l'utilisation d'expressions régulières pour représenter des gestes complexes, et ainsi épargner la définition de chaînes de fonctions dépendantes.

Dans ECS, l'orchestration des comportements de fait à haut niveau, non sur les données mais sur les processus. Le modèle s'aligne sur le fonctionnement des algorithmes liés aux jeux vidéos (rendu 3D, simulation physique), qui traitent de grands volumes de données à répétition. Il invite à considérer les comportements comme des processus transformant des événements d'entrée en événements de sortie. Ainsi que l'écrivent Chatty et al. pour présenter djnn, "*Like computations can be described as the evaluation of lambda expressions, interactions can be described as the activation of interconnected processes*" [13]. Polyphony se distingue des travaux présentés en attachant des attributs (Composants) aux processus (Systèmes), afin de traiter explicitement l'inversion de contrôle à l'aide des Méta-Systèmes, sans toutefois contraindre le choix des attributs à donner. L'orchestration d'une application peut ainsi être rendue plus complexe par l'introduction de nouveaux Composants et Méta-Systèmes.

5 CONCLUSION

Nous avons discuté dans cet article de la conception d'une bibliothèque d'interaction basée sur le modèle Entité-Composant-Système. Nous avons clarifié ce modèle, détaillé un exemple d'application de dessin vectoriel basé dessus, et énuméré les aspects différenciants de son utilisation en pratique. Nous avons ensuite analysé les choix de conception d'une implémentation d'ECS selon trois variantes majeures, et détaillé nos choix pour la conception d'une boîte à outils pour l'IHM, Polyphony. Enfin, nous avons comparé notre approche à des travaux de référence, en nous attachant particulièrement à la gestion des comportements dans une application interactive.

Les apports de ce modèle pour la programmation d'interactions sont le polymorphisme des éléments grâce aux Entités, un mécanisme puissant de Sélections grâce aux Composants, et une gestion poussée des comportements interactifs grâce à une programmation "orientée Systèmes". Cependant la flexibilité de ce modèle est à double tranchant, beaucoup de comportements de haut niveau peuvent y être exprimés de plusieurs manières, et nous manquons encore de recul pour les distinguer.

5.1 Limites et travaux futurs

Parmi les atouts suggérés du modèle ECS pour concevoir des IHM, il en reste un certain nombre que nous n'avons pas implémentés pour cet article. Ce sont en particulier l'utilisation de périphériques multiples et changeant dynamiquement, la gestion de contraintes de positionnement linéaires, et l'utilisation d'un moteur de rendu graphique *hardware* implémenté avec plusieurs Systèmes. En outre Polyphony étant une preuve de concept implémentée dans un langage peu compatible avec ECS, sa syntaxe est verbeuse à l'utilisation. L'implémentation d'un langage dédié à ECS est une piste que nous considèrerons.

Enfin, bien que nous ayons détaillé ECS et présenté une implémentation fonctionnelle pour la programmation d'IHM dans cet article, cette approche manque encore de validation quant à son utilisation pratique face aux boîtes à outils existantes. Nos pistes de validation pour le futur sont :

- la conception des mêmes applications interactives *canoniques* dans différentes boîtes à outils, selon les styles de programmation recommandés, afin de comparer quantitativement les codes obtenus;
- le prototypage de techniques d'interaction novatrices tirant parti des atouts du modèle ECS;
- l'observation qualitative d'utilisateurs programmant un ensemble d'exercices à l'aide de Polyphony.

Nous espérons aussi que ce travail inspirera le développement de boîtes à outils "hybrides" complémentant la programmation par *widgets* avec les mécanismes de Systèmes et Sélections issus d'ECS.

REMERCIEMENTS

Les auteurs remercient Damien Pollet pour ses conseils et les nombreuses discussions autour du projet.

REFERENCES

[1] Georg Apitz and François Guimbretière. 2004. CrossY: A Crossing-Based Drawing Application. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '04)*. ACM, New York, NY, USA, 3–12. DOI: http://dx.doi.org/10.1145/1029632.1029635

[2] C. Appert and M. Beaudouin-Lafon. 2008. SwingStates: Adding State Machines to Java and the Swing Toolkit. *Software: Practice and Experience* 38, 11 (Sept. 2008), 1149–1182. DOI: http://dx.doi.org/10.1002/spe.867

[3] Caroline Appert, Stéphane Huot, Pierre Dragicevic, and Michel Beaudouin-Lafon. 2009. FlowStates: Prototypage D'Applications Interactives Avec Des Flots De DonnÉEs Et Des Machines À ÉTats. In *Proceedings of the 21st International Conference on Association Francophone D'Interaction Homme-Machine (IHM '09)*. ACM, New York, NY, USA, 119–128. DOI: http://dx.doi.org/10.1145/1629826.1629845

[4] Apple Inc. 2016. GameplayKit Programming Guide: Entities and Components. https://developer.apple.com/library/content/documentation/General/Conceptual/GameplayKit_Guide/EntityComponent.html. (March 2016).

[5] Greg J. Badros, Alan Borning, and Peter J. Stuckey. 2001. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Trans. Comput.-Hum. Interact.* 8, 4 (Dec. 2001), 267–306. DOI: http://dx.doi.org/10.1145/504704.504705

[6] Michel Beaudouin-Lafon and Henry Michael Lassen. 2000. The Architecture and Implementation of CPN2000, a post-WIMP Graphical Application. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST '00)*. ACM, New York, NY, USA, 181–190. DOI: http://dx.doi.org/10.1145/354401.354761

[7] Benjamin B. Bederson, Jesse Grosjean, and Jon Meyer. 2004. Toolkit Design for Interactive Structured Graphics. *IEEE Trans. Softw. Eng.* 30, 8 (Aug. 2004), 535–546. DOI: http://dx.doi.org/10.1109/TSE.2004.44

[8] Benjamin B. Bederson, Jon Meyer, and Lance Good. 2000. Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST '00)*. ACM, New York, NY, USA, 171–180. DOI: http://dx.doi.org/10.1145/354401.354754

[9] Scott Bilas. 2002. A Data-Driven Game Object System. (March 2002).

[10] Renaud Blanch and Michel Beaudouin-Lafon. 2006. Programming Rich Interactions Using the Hierarchical State Machine Toolkit. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '06)*. ACM, New York, NY, USA, 51–58. DOI: http://dx.doi.org/10.1145/1133265.1133275

[11] Gilad Bracha and William Cook. 1990. Mixin-Based Inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP '90)*. ACM, New York, NY, USA, 303–311. DOI: http://dx.doi.org/10.1145/97945.97982

[12] Marcellin Buisson, Alexandre Bustico, Stéphane Chatty, Francois-Régis Colin, Yannick Jestin, Sébastien Maury, Christophe Mertz, and Philippe Truillet. 2002. Ivy: Un Bus Logiciel Au Service Du Développement De Prototypes De Systèmes Interactifs. In *Proceedings of the 14th Conference on L'Interaction Homme-Machine (IHM '02)*. ACM, New York, NY, USA, 223–226. DOI: http://dx.doi.org/10.1145/777005.777040

[13] Stéphane Chatty, Mathieu Magnaudet, and Daniel Prun. 2015. Verification of Properties of Interactive Components from Their Executable Code. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '15)*. ACM, New York, NY, USA, 276–285. DOI: http://dx.doi.org/10.1145/2774225.2774848

[14] Gael Curry, Larry Baer, Daniel Lipkie, and Bruce Lee. 1982. Traits: An Approach to Multiple-Inheritance Subclassing. In *Proceedings of the SIGOA Conference on Office Information Systems*. ACM, New York, NY, USA, 1–9. DOI: http://dx.doi.org/10.1145/800210.806468

[15] Brian Dorn, Adam Stankiewicz, and Chris Roggi. 2013. Lost While Searching: Difficulties in Information Seeking Among End-User Programmers. In *Proceedings of the 76th ASIS&T Annual Meeting: Beyond the Cloud: Rethinking Information Boundaries (ASIST '13)*. American Society for Information Science, Silver Springs, MD, USA, 21:1–21:11.

[16] Pierre Dragicevic and Jean-Daniel Fekete. 2001. Input Device Selection and Interaction Configuration with ICON. In *People and Computers XV—Interaction without Frontiers*. Springer, London, 543–558. DOI: http://dx.doi.org/10.1007/978-1-4471-0353-0_34

[17] E. Duala-Ekoko and M. P. Robillard. 2012. Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study. In *2012 34th International Conference on Software Engineering (ICSE)*. 266–276. DOI: http://dx.doi.org/10.1109/ICSE.2012.6227187

[18] James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2011. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 225–234. DOI: http://dx.doi.org/10.1145/2047196.2047226

[19] Ecma International. 2015. ECMAScript 2015 Language Specification – ECMA-262 6th Edition. http://www.ecma-international.org/ecma-262/6.0/index.html. (June 2015).

[20] Alix Goguey, Géry Casiez, Thomas Pietrzak, Daniel Vogel, and Nicolas Roussel. 2014. Adoiraccourcix: Multi-Touch Command Selection Using Finger Identification. In *Proceedings of the 26th Conference on L'Interaction Homme-Machine (IHM '14)*. ACM, New York, NY, USA, 28–37. DOI: http://dx.doi.org/10.1145/2670444.2670446

[21] Google Inc. 2015. CORGI: Main Page. http://google.github.io/corgi/. (2015).

[22] Stéphane Huot, Pierre Dragicevic, and Cédric Dumas. 2006. Flexibilité Et Modularité Pour La Conception D'Interactions: Le ModèLe D'Architecture Logicielle Des Graphes Combinés. In *Proceedings of the 18th Conference on L'Interaction Homme-Machine (IHM '06)*. ACM, New York, NY, USA, 43–50. DOI: http://dx.doi.org/10.1145/1132736.1132742

[23] Stéphane Huot, Cédric Dumas, Pierre Dragicevic, Jean-Daniel Fekete, and Gérard Hégron. 2004. The MaggLite post-WIMP Toolkit: Draw It, Connect It and Run It. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '04)*. ACM, New York, NY, USA, 257–266. DOI: http://dx.doi.org/10.1145/1029632.1029677

[24] Piyus Kedia, Manuel Costa, Matthew Parkinson, Kapil Vaswani, Dimitrios Vytiniotis, and Aaron Blankstein. 2017. Simple, Fast, and Safe Manual Memory Management. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 233–247. DOI: http://dx.doi.org/10.1145/3062341.3062376

[25] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. 2012. Proton: Multitouch Gestures As Regular Expressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. ACM, New York, NY, USA, 2885–2894. DOI: http://dx.doi.org/10.1145/2207676.2208694

[26] Eric Lecolinet. 2003. A Molecular Architecture for Creating Advanced GUIs. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology (UIST '03)*. ACM, New York, NY, USA, 135–144. DOI: http://dx.doi.org/10.1145/964696.964711

[27] Tom Leonard. 1999. Postmortem: Thief: The Dark Project. https://www.gamasutra.com/view/feature/131762/postmortem_thief_the_dark_project.php. (July 1999).

[28] Adam Martin. 2007. Entity Systems Are the Future of MMOG Development – Part 1. (Sept. 2007).

[29] B. Myers, S. Y. Park, Y. Nakano, G. Mueller, and A. Ko. 2008. How Designers Design and Program Interactive Behaviors. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. 177–184. DOI: http://dx.doi.org/10.1109/VLHCC.2008.4639081

[30] Brad A. Myers. 1990. A New Model for Handling Input. *ACM Trans. Inf. Syst.* 8, 3 (July 1990), 289–320. DOI: http://dx.doi.org/10.1145/98188.98204

[31] Brad A. Myers. 1991. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology (UIST '91)*. ACM, New York, NY, USA, 211–220. DOI: http://dx.doi.org/10.1145/120782.120805

[32] B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. 1997. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering* 23, 6 (June 1997), 347–365. DOI: http://dx.doi.org/10.1109/32.601073

[33] Brad A. Myers and Mary Beth Rosson. 1992. Survey on User Interface Programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '92)*. ACM, New York, NY, USA, 195–202. DOI: http://dx.doi.org/10.1145/142750.142789

[34] Adrian Papari. 2018. Artemis-Odb: A Continuation of the Popular Artemis ECS Framework. (April 2018).

[35] prime31. 2018. Nez Is a Free 2D Focused Framework That Works with MonoGame and FNA. (April 2018).

[36] Stéphanie Rey, Stéphane Conversy, Mathieu Magnaudet, Mathieu Poirier, Daniel Prun, Jean-Luc Vinot, and Stéphane Chatty. 2015. Using the Djnn Framework to Create and Validate Interactive Components Iteratively. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '15)*. ACM, New York, NY, USA, 230–233. DOI: http://dx.doi.org/10.1145/2774225.2775438

[37] Simon Schmid. 2018. Entitas-CSharp: Entitas Is a Super Fast Entity Component System (ECS) Framework Specifically Made for C# and Unity. (March 2018).

[38] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., and Jason E. Robbins. 1995. A Component- and Message-Based Architectural Style for GUI Software. In *Proceedings of the 17th International Conference on Software Engineering (ICSE '95)*. ACM, New York, NY, USA, 295–304. DOI: http://dx.doi.org/10.1145/225014.225042

[39] Unity Technologies. 2017. Unity - Scripting API: MonoBehaviour. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html. (2017).

[40] W3C. 1996. Cascading Style Sheets, Level 1. https://www.w3.org/TR/CSS1/. (Dec. 1996).

[41] M. F. Zibran, F. Z. Eishita, and C. K. Roy. 2011. Useful, But Usable? Factors Affecting the Usability of APIs. In *2011 18th Working Conference on Reverse Engineering*. 151–155. DOI: http://dx.doi.org/10.1109/WCRE.2011.26