Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model

Thibault Raffaillac, Stéphane Huot

Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model

THIBAULT RAFFAILLAC, Inria - Univ. Lille - UMR 9189 CRIStAL, France
STÉPHANE HUOT, Inria - Univ. Lille - UMR 9189 CRIStAL, France

This paper introduces a new Graphical User Interface (GUI) and Interaction framework based on the Entity-Component-System model (ECS). In this model, interactive elements (Entities) are characterized only by their data (Components). Behaviors are managed by continuously running processes (Systems) which select entities by the Components they possess. This model facilitates the handling of behaviors and promotes their reuse. It provides developers with a simple yet powerful composition pattern to build new interactive elements with Components. It materializes interaction devices as Entities and interaction techniques as a sequence of Systems operating on them. We present *Polyphony*, an experimental toolkit implementing this approach, and discuss our interpretation of the ECS model in the context of GUIs programming.

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

Keywords: User Interface Toolkit, Interaction Programming, GUI, Entity-Component-System, ECS

ACM Reference Format:
Thibault Raffaillac and Stéphane Huot. 2019. Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model. In *Proceedings of the ACM on Human-Computer Interaction*, Vol. 3, EICS, Article 8 (June 2019). ACM, New York, NY. 22 pages. https://doi.org/10.1145/3331150

1 INTRODUCTION

Programming the behaviors and reactions of interactive systems to user input has long been recognized to be a complex and difficult task in many situations [36]. Among the acknowledged causes are the complexity of interfaces due to the number of elements they are made of [34, 49], and the specific knowledge and tools required to build advanced interactions [6, 17, 19, 37]. In fact, despite the availability of numerous architectures and tools, the prototyping, implementation, and dissemination of novel interaction techniques is still challenging today.

Among the unexplored solutions in this context, we propose to apply the *Entity-Component-System* (ECS) model [32] to the design of Graphical User Interfaces (GUI) and advanced interactions. In this model, introduced in Video Games development, the elements of the interface — the *Entities* — do not own or inherit any behavior. They are assigned properties dynamically — the *Components* — which are retrieved by continuously running processes — the *Systems* — that reify different reusable behaviors. For instance, this model allows the gravity force in a physics engine to be modeled as a System operating on every Entity with mass and position Components. Applied to GUIs, it does not model an interface as intertwined elements reacting to their environment, but rather as the combined actions of universal rules — such as the drawing of background shapes and borders, or the highlighting of clickable elements on mouse hover.

We developed the *Polyphony* toolkit to leverage and illustrate the potential benefits of this model to interfaces and interactions programming. In Polyphony, the widgets (buttons, text boxes, etc.), but also the input and output devices (mouse, keyboard, screen, etc.), are Entities. Their

Authors' addresses: Thibault Raffaillac, thibault.raffaillac@inria.fr, Inria - Univ. Lille - UMR 9189 CRIStAL, Lille, France; Stéphane Huot, stephane.huot@inria.fr, Inria - Univ. Lille - UMR 9189 CRIStAL, Lille, France.

properties are Components that can be acquired and removed dynamically. Systems are functions that run in sequence at each input/output event, each one iterating on Entities and applying a type of interactive behavior such as border drawing, drag & drop, text entry, etc. For example, the `RenderImageSystem` will draw an image for each Entity having the Components `position` and `image`. Finally, the Systems are themselves Entities, allowing the use of Components to describe high-level relationships between them such as precedence relations or event subscriptions.

Section 2 illustrates the use of Polyphony with the implementation of a vector drawing application and details the notable aspects of GUI programming with ECS. Section 3 discusses the implementation of Polyphony through a design space based on the analysis of three major ECS frameworks, and details our adaptation of this model from video games to GUIs programming. In section 4, we discuss the pros and cons of our approach, and compare the programming of interactive behaviors in ECS with the state of the art in terms of reuse, dynamicity and orchestration. Finally, we conclude on the limitations of this model, as well as perspectives for future work.

2 A DRAWING APPLICATION BASED ON ECS

This section introduces the Polyphony toolkit, based on the ECS programming model, and adapted to the prototyping and development of user interfaces and interaction techniques. We first describe the basic concepts of ECS. Then, we illustrate the use of Polyphony with the implementation of an interactive drawing application. Finally, we detail the inner structure of this application, and we discuss the implementation of a particular interaction technique (the Drag and Drop).

2.1 The Entity-Component-System model

ECS (sometimes also called CES) is an architectural pattern designed to structure code and data in a program. It appeared in the field of Video Game development [9, 30, 32] where teams are typically separated between programmers designing logic and tools, and designers producing scripts and multimedia content. ECS defines Components as an interface between the two worlds: programmers define the available Components and create the Systems that will operate on them; designers instantiate the different Entities and their behaviors by assembling selected Components.

The main elements defining the model are thus:

Entities are unique identifiers (often simple integers) representing the elements of the program. They are similar to objects in Object-Oriented Programming, but have no data or methods.

Components represent the program data (such as `position` or `color`), and are dynamically associated with the Entities. Depending on the interpretations of ECS, a Component alone can mean the *type* of data that can be bound to Entities, or one *instance* bound to an Entity.

Systems run continuously in a fixed order, each representing a behavior of the program. Entities do not register with Systems, but acquire the necessary Components to be "seen" by them.

The general model of an application based on ECS is illustrated in Fig. 1. Components define both the attributes (e.g., `shape`, `backgroundColor`) and the capabilities (e.g., `targetable`, `draggable`) of the Entities. Entities acquire new behaviors through the acquisition of Components. Systems execute each type of behavior, selecting the Entities through their Components. For instance, a System for drawing background shapes will select all the Entities with `bounds`, `shape` and `backgroundColor` Components, and will draw the corresponding shapes on the screen. Systems are ordered by the type of behavior they execute: input management, interpretation of interaction techniques, widget-specific processing, application-specific processing, and output rendering.

There are also four recurring elements in the majority of ECS implementations, although they are not part of the basic model and do not have fixed terminologies:

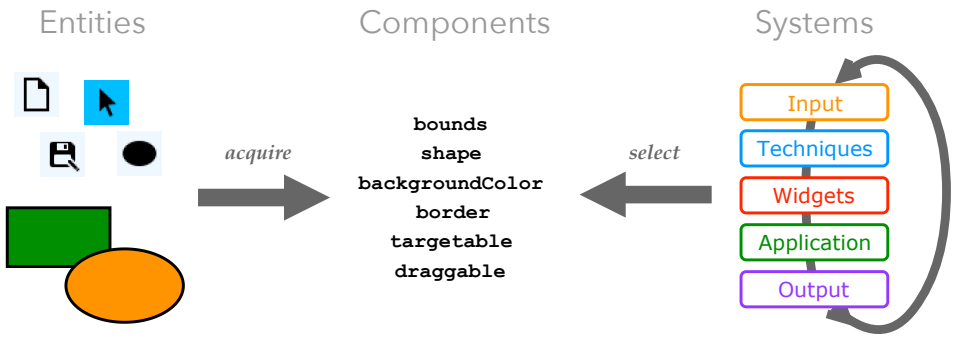

Fig. 1. Illustration of high-level interactions between Entities, Components, and Systems.

Descriptors are conditions based on the Components owned by Entities. They are similar to *interfaces* evaluated at runtime, and are the fundamental mechanism through which Systems know which Entities they operate on.

Selections are groups of Entities based on Descriptors, and updated on the fly as Components are added to or removed from Entities.

Context represents the *world* containing all Entities, storing Components and global variables, and providing an interface to obtain Selections.

Templates or Factories are predefined models used to instantiate Entities with a set of default Components and values.

2.2 Presentation of Polyphony with a sample drawing application

Polyphony is an experimental JavaScript toolkit based on ECS, and dedicated to the implementation of graphical interfaces and interaction techniques (see Fig. 2). It consists of a kernel implementing the notions of Entities, Components, Systems, Descriptors and Selections. It also provides *bindings* to two low-level libraries: SDL [28] and libpointing [14]. At a higher level, Polyphony provides developers with predefined Components, Systems, Selections and Templates to build interfaces.

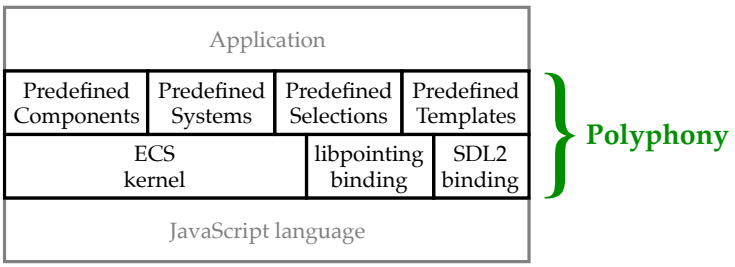

Fig. 2. The internal structure of Polyphony.

Implementing a vector drawing application is a common and appropriate example to illustrate the design of interaction techniques [1, 22]. It combines the implementation of graphical objects with various appearances, tools (e.g., shape drawing, color pipette), commands (e.g., copy/paste, undo), and a wide range of possible tasks with many possible combinations between elements. This is thus a good testbed for prototyping and implementing both basic and advanced interaction techniques. Our basic drawing application (see Fig. 3) allows to:

- draw rectangles and ovals;
- move, delete and change the type of the created shapes;
- save the result as SVG;
- clear the workspace.

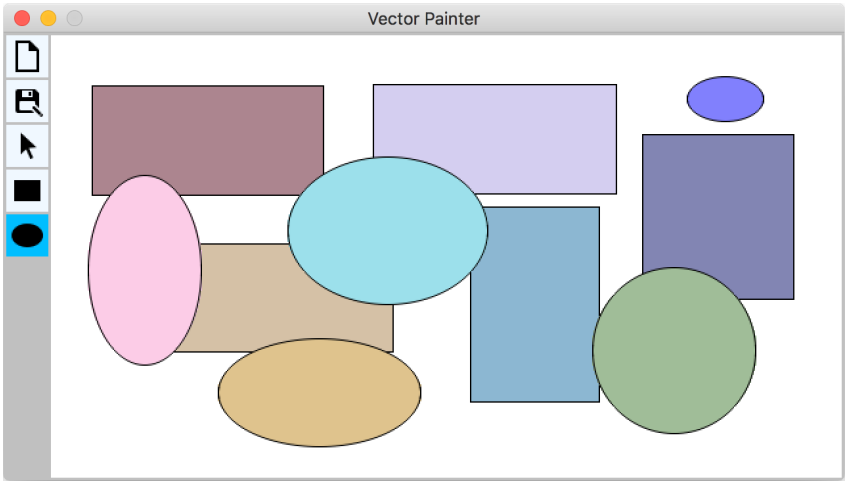

Fig. 3. An example vector drawing application implemented with Polyphony.

2.3 Illustration with the application code

All the elements of this interface (buttons, drawing area, drawn shapes) are Entities. The interaction devices (mouse and keyboard) are also represented by Entities, as well as the display view.

2.3.1 Creating Entities. Entities are basically identifiers created on the fly and without being linked to any Components. In our toolkit, an Entity is created with the `Entity` function, which can take an optional JavaScript object as a parameter in order to add some default Components. For instance, in our drawing application, an Entity for a simple box will be created with:

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

At this point, this Entity is visible to all Systems that process Entities with `bounds` or a `shape` (e.g., a "Layout System"), without the need to register it anywhere. In our case, the "Background Rendering System" will process and display Entities that have at least `bounds`, `shape`, and `backgroundColor` Components. Thus, adding the appropriate Component to our Entity will make it appear on screen:

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

The Entity then becomes visible to the "Background Rendering System" which will draw a blue rectangle on screen. In practice, our toolkit offers helpful *Entity Factories* to instantiate standard widgets with predefined Components. For instance, in the drawing application, the toolbar buttons and the canvas are created from the factories `Button` and `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)
```

Each factory takes a set of necessary values as arguments for constructing the initial Entity. Optionally, a dictionary containing a set of additional Components can be passed to initialize the new Entity. If this dictionary is already an Entity, it is completed with the factory Components instead of creating a new one. For example, the factory `Button` is defined by:

```
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 || new Color(240, 248, 255)
  e[imgOrTxt instanceof Image ? 'image' : 'richText'] = imgOrTxt
  e.targetable = true
  e.clickable = true
  return Entity(e)
}
```

Factories add Components to already existing Entities only if these Components are not already present. This is the purpose of the operation `e.component = e.component || value` in the code above, since when a Component is not defined it is evaluated to `undefined`.

Finally, the Entities are deleted manually. As they are globally accessible through Selections, they can never be considered as "out of reach" for local scope or garbage-collection mechanisms. The deletion of an Entity is done with `e.delete()`. Its Components are then automatically removed, as well as any references pointing to `e` in existing Selections.

2.3.2 Creating Components. In Polyphony, Components are the object constructors from JavaScript:

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

In accordance with the ECS model, Components do not contain any code. An exception is the definition of accessors, for which we use instance methods implemented with the *prototypal* inheritance of JavaScript. A *setter*, for example, is created with:

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

By convention, *setters* return the target object (`this`) so that several calls can be chained in a single instruction (e.g., `e.setX(10).setY(20)`), making the code more concise.

2.3.3 Creating Systems. Similarly to lambda functions, which are reified as objects, we have implemented the Systems as Entities. Their dependencies are thus represented by data stored as

Components. Systems are instantiated once with the function `Entity`, which takes as parameters a function followed by Components:

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

In our sample application, this simple System checks if the button `resetButton` has been activated (e.g., clicked with the mouse). When this occurs, the System removes all children (i.e., shapes) from the `canvas` Entity. The Component `runOn` indicates that the System will be triggered at each pointing event (mouse). The Component `order` distinguishes and orders the System classes (see Fig. 1): Inputs (0 to 19), Interaction Techniques (20 to 39), Widgets (40 to 59), Application (60 to 79), and Outputs (80 to99). `ResetSystem` is therefore part of the "Application Systems".

When a System has to iterate on groups of Entities with common Components, a *Selection* is generally used. For example, to highlight all Entities that can be targeted by the mouse:

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

The iterable Selection `Targetables` contains all Entities with Components `bounds`, `depth` and `targetable`. On each of these Entities, the border is replaced by a thin green border of 1px. A Selection is defined with the function `Selection`, which takes as parameter a function $accept : Entity \longrightarrow boolean$ to filter the entities with a programmable condition. A second optional argument provides an ordering criterion $compare : Entity \times Entity \longrightarrow number$ when iterating over the Selection. For example, the Selection `Targetables` is defined with:

```
let Targetables = Selection(
  e => ('bounds' in e) && ('depth' in e) && e.targetable,
  (a, b) => b.depth - a.depth) // sort by decreasing depth
```

2.3.4 Modeling an interactive application with ECS. Figure 4 presents all the Entities in our example drawing application, each identified by the factory which instantiated it. Unlike popular frameworks (e.g., Qt, JavaFX, HTML), the graphical objects do not necessarily belong to a scene graph. As soon as they are created, Entities are accessible to the Systems, thus already visible and/or interactive. Tree relationships, represented in this figure by arrows, are defined only when it is necessary to establish an ordering between Entities — e.g., display depth with a Component `depth`. Interaction devices are also materialized by Entities (`Pointer`, `Keyboard`, and `View`), in order to provide flexible and persistent representation for implementing interaction techniques. Finally, the Systems are represented in order of execution, their color differentiating the type of processing they perform.

The execution flow in Polyphony works as a reactive machine [18]. Every $25ms$ ($40Hz$), Polyphony checks all pending events from the Operating System, and instantaneously updates the pointers and keyboards Entities. If a pointer event or a keyboard event occurred, the ordered list of Systems is gathered, through a `Systems` Selection. Polyphony then iterates on all Systems, and executes every System whose `runOn` Component matches one of the gathered event types. This `runOn` Component eases the development of multi-input interaction techniques. It takes a bitwise OR of the possible types of triggering events — such as `MOUSE_INPUT`, `KEYBOARD_INPUT`, `DISPLAY_OUTPUT`, etc. —

thus describing the general modalities of an interaction technique. We plan to extend it in the future, to include other kinds of devices, such as voice input or gestural input.

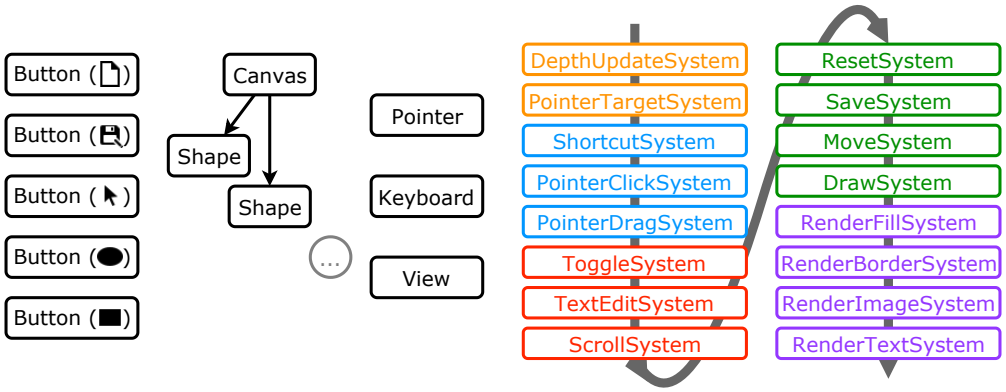

Fig. 4. List of Entities in the drawing application. Arrows on the left represent tree relations. Systems are shown in the order they execute, their colors matching the types of processing shown in Fig 1.

We chose to synchronize Systems triggering and display refresh with input events (e.g., mouse pointer or keyboard, that could be extended to other inputs) because of the lack of low-level support for *VSync* events. We are considering to improve this reactive execution model of the toolkit with a more generic and extensible reactive machine, independent of input events.

Components determine the behaviors that can be acquired by each Entity. They are read by each System implementing these behaviors. Table 1 lists the Components of the drawing application, as well as the factories assigning them. These Components were inspired by CSS1 [48], which synthesizes most of the common behaviors of HTML tags, allowing for example to simulate a `button` with a `div` container only by acquiring CSS properties.

Polyphony enables the implementation of standard UI controls with Systems and Components. Three types of controls are illustrated in our example application:

- Toggle buttons are Entities with `bounds`, `shape`, `backgroundColor`, `image/richText`, `targetable`, and `clickable` Components (like regular buttons). Toggling/untoggling behavior requires another Component: `toggleGroup`. A `ToggleSystem` selects all Entities with this Component and looks for a pointer click on any one of them. When it happens, the toggled Entity receives a new Component `toggled` set to `true`, and its `backgroundColor` is changed, while all other Entities within the same `toggleGroup` are untoggled.
- Text fields require a `focus` Component on each keyboard Entity, and a `richText` Component for displaying formatted text inside an Entity's `bounds`. The latter contains a string, padding sizes, and positional fonts information. A `TextEditSystem` looks for each character typed on the keyboard, and based on the `focus` Component, updates the target Entity's `richText`, while managing its blinking cursor.
- Scrollable views extend `View` Entities with `viewport` and `scrollable` Components. A `ScrollSystem` adds and manages a child Entity on each scrollable view in order to display a draggable scroll bar. The System detects drags on the bar and pointer scrolls in the area in order to update the `origin` Component inside the bounds permitted by `viewport`.

More generally, implementing new kinds of widgets requires adding new Components describing their capabilities, and inserting new Systems in the Widgets `order` range (40 to 59). These Systems form reusable behaviors that may in turn be composed for the design of future widgets.

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

Table 1. Components and the Entity factories assigning them. The Entities from each factory receive all the Components indicated by ×, and possibly those indicated by ~ (depending on their specialization).

2.3.5 Reifying devices as Entities. Polyphony is interfaced with the Operating System resources through an extensible set of *device* Entities (`Pointer`, `Keyboard`, `View`). Their Components store the current state of each device. `Pointer`, for example, has the Component `position` which stores the coordinates of the system cursor at any time. These coordinates can of course be accessed, but also modified in order to control the cursor position in a simple way. In our painter application, this is used to force the cursor position to remain inside of the canvas when drawing a shape.

Modeling devices as Entities offers a flexible and persistent representation between the different layers of the program. The structure of an Entity being extensible, new data can be introduced without creating a new object, but by adding new Components that will make the Entity be processed by the corresponding Systems. This mechanism is a kind of events propagation without neither relying on multiple callbacks nor creating many event objects, since the data at each step is centralized within the Entity's Components and the processing of events in Systems.

This mechanism is illustrated in Fig. 5. An Entity representing the system pointer is initialized by the `Pointer` factory with two Components, `cursorPosition` and `buttons`. As long as this is the only available pointer, we call it the `Pointer` Entity. When a pointing event occurs, the Operating System binding of Polyphony adds temporary Components to the `Pointer` Entity: `tmpPressed`, `tmpReleased`, `tmpMotion` or `tmpScroll` — which store the relative or absolute coordinates for the pointing event. All the subsequent Systems that rely on a mouse action will watch the `Pointer` Entity through the `Pointers` Selection — in the case that several pointers have been instantiated. Then, these Systems can also add or remove Components for subsequent Systems. For example, `PointerClickSystem` uses the Components `tmpPressed/tmpReleased` to detect mouse clicks,

and then adds a Component `tmpClick` with the value of `target` to the pointer when relevant. Finally, temporary Components (which names start with `tmp`) are automatically removed from Entities after all Systems have executed.

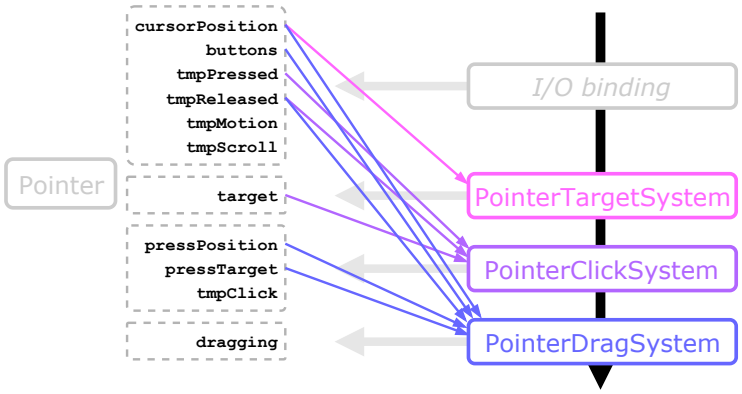

Fig. 5. Evolution of the Components from the `Pointer` Entity, as the Systems react to pointer events. Colored arrows indicate which Components are accessed by each System. Thick horizontal arrows indicate which Components are created by Systems.

2.4 A practical example: implementing Drag and Drop interaction

Our example application mostly relies on direct manipulation techniques — e.g., drawing, moving, or modifying shapes — and especially *Drag and Drop*. Drag and Drop is notoriously difficult to implement in most current frameworks and illustrates well the benefits of Polyphony for implementing such interaction techniques. Drag and Drop consists in moving one object and dropping it onto an empty space or another object. In the latter case, if the two objects are compatible, a command is executed, which depends on the two objects and the location of the drop.

In the drawing application, graphical objects can be dragged and dropped on the toolbar buttons in order to change their shape or to delete them: The canvas resetting button/tool removes the dropped shape; The rectangle button/tool transforms it into a rectangle (keeping its dimensions); The oval button/tool transforms it into an ellipse. Possible combinations are summarized in Table 2.

dragged object	dropping button		
	🗋	■	●
Rectangle	deletion		→ Ellipse
Ellipse	deletion	→ Rectangle	

Table 2. Matrix of the drag and drop combinations and resulting actions in our application.

The sequence of actions to perform and to model a drag and drop technique is:

- the cursor hovers over the object to drag (a *feedforward* mechanism may indicate that the object is draggable);
- a button is pressed on the object to drag;
- the cursor moves and the object follows it;
- the cursor hovers over a depository object (a *feedforward* mechanism may suggest that the drop will have an effect);

- the button is released (a command is then executed, if the two objects are compatible).

The difficulties associated with this technique are of several kinds. First, moving an object involves removing it from its current location, which can break local layout constraints. Then, the different steps require waiting for actions from the user, and can therefore hardly be expressed by a continuous code block (they are commonly implemented in multiple callbacks at different places in the code). Finally, the commands and their *feedforward* do not depend on one or the other of the objects, but on the *combination* of the two. The number of possibilities can thus be very important since it depends on the product of the number of objects that can be moved by the number of recipient objects. Depending on the case, the resulting behaviors may therefore belong to the moved objects or to the repository objects, which results again in splitting the code.

In our drawing application, `PointerDragSystem` is dedicated to the detection and handling of the Drag and Drop. It depends on the `cursorPosition`, `buttons`, and `tmpReleased` Components that are managed by the input binding, as well as `pressPosition` and `pressTarget` added by `PointerClickSystem`. The code of this System is listed below:

```
let PointerDragSystem = Entity(function PointerDragSystem() {
  for (let pointer of Pointers) {
    let position = pointer.cursorPosition
    let target = pointer.pressTarget
    let dragging = pointer.dragging // Entity being dragged
    if (!dragging && pointer.buttons[0] && target && target.draggable &&
      position.distance(pointer.pressPosition) > 10) {
      pointer.dragging = dragging = target
      dragging.draggedBy = pointer
      delete dragging.targetable // cannot be seen by any other pointer
    }
    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 })
```

At the beginning of a valid drag action, `PointerDragSystem` adds the `dragging` and `draggedBy` Components to the pointer and dragged element, respectively. As long as the pointer has the attribute `dragging`, the System updates the position of the moved object with the cursor coordinates. When the System detects a button release, it deletes the previous Components and adds `tmpDrop` to the pointer (its Component `target` already contains the drop target). Systems inserted after `PointerDragSystem` will detect that the Drag and Drop technique is occurring by watching this Component on the pointer Entities. In our drawing application, the Systems `ResetSystem`, `MoveSystem` and `DrawSystem` detect and execute the actions specified in Table 2. Moreover, `MoveSystem` uses the Components added by `PointerDragSystem` to handle the displacement of shapes on the canvas.

This sequencing of Systems illustrates the composition of behaviors with ECS. The drawing tools are based on the drag-release detection System, `PointerDragSystem`, which itself depends on `PointerClickSystem` for detecting the clicked target. Dependencies between Systems are modeled by the order of execution in the Systems chain. Each System inserts Components on the

device Entities, based on the Components inserted by the previous Systems. Thus, building higher level interaction techniques implies positioning them downstream in the execution chain.

3 IMPLEMENTING THE ENTITY-COMPONENT-SYSTEM MODEL

ECS promotes high-level principles without specifying precisely how they should be realized and implemented. This relatively vague formalization of Systems and Components has therefore led to different interpretations and implementations. In this section we highlight the design choices arising when implementing ECS, with the study of three existing implementations. Then, we discuss the differences between Video Games and GUIs, which justify our deviations from existing ECS implementations. Finally, we detail our implementation choices for Polyphony.

3.1 Analysis of existing implementations

There are many frameworks based on ECS, each with variations of the same concepts, which makes it difficult to choose one implementation over another. To better inform the use of ECS as an architectural pattern, and especially in our case of GUI and interaction programming, we have built a *design space* of ECS implementations. This design space, presented in Table 3, is based on the study of three of the most referenced[1] and better documented frameworks, and on our experience in the development of a new variant dedicated to interaction programming, *Polyphony*. All the frameworks we found and discuss target video game development, we did not find any dedicated to graphical user interfaces. They are:

- **`Artemis`** (Java) is one of the first implementations of ECS. The original version is not documented and no longer maintained, however it has a popular continuation, Artemis-odb [39];
- **`Entitas`** [44] (C#) is a renowned interpretation of ECS for Unity, and embeds a meta-language and a preprocessor for C#. Its high popularity and extensive documentation make it a major implementation of ECS;
- **`GameplayKit`** [4] (Swift) is one of the numerous frameworks inspired from ECS that implement behaviors directly on the Components and not in separate Systems (e.g., CORGI [23] (C++), Nez [41] (C#)). They are sometimes referred to as *Entity-Component systems*, which leads to confusion with the original ECS model. Unity [47] (C#) also implemented this approach and later conformed to the original model, although it is not well documented yet.

The analysis presented in Table 3 is not exhaustive, but highlights the design choices on which interpretations of ECS mainly differ. We also included our own choices for the implementation of Polyphony, and we will discuss them in the rest of this section. The criteria of our analysis are:

- **Representing Entities** – how are Entities materialized in the framework
- **Representing Components** – how are Components materialized in the framework and defined by programmers
- **Representing Systems** – how are Systems materialized in the framework and defined by programmers
- **Structuring the Context** – how is the environment materialized in the framework, for registering new Entities/Components/Systems, and obtaining Selections
- **Selecting Entities** – how are Entities gathered from their Components
- **State changes** – which mechanisms are available to react to state changes
- **Storing Components** – how and where are Components stored

[1] see http://entity-systems.wikidot.com/

Framework	Artemis-odb (Java)	Entitas (C#)	GameplayKit (Swift)	Polyphony (JavaScript)
Representing Entities	integer, or integer embedded in an object	object	object	object
Representing Components	`Component` subclass with default values	`IComponent` subclass	`GKComponent` subclass with default values	any object
Representing Systems	`BaseSystem` subclass, bound to a Selection, periodic execution	`ISystem` subclass, bound to a Selection, periodic or one-time execution	Component method `updateWithDeltaTime:`	Entity made from lambda function
Structuring the Context	`World` object	`Context` object	scene tree	*global*
Selecting Entities	by Components, with *all/one/none* algebra	by Components, with *all/one/none* algebra	by Components, or programmable condition	by programmable condition
State changes	*listeners* on Selections	*listeners* on Selections	*not explicitly supported*	temporary Components, *listeners* on Selections
Storing Components	by Component, with table Entity→Value	by Entity, with precompiled slots	by Entity, with table Component→Value	as properties of native objects
Ordering Systems	dynamically by priority	dynamically through insertion in a list	statically	dynamically by priority
Entity Templates	*factory* object, loading from file	loading from file	*not explicitly supported*	*factory* method
Language extensions	post-processing the compiled executable	pre-compilation of language superset	*none*	*none*

Table 3. Analysis of three ECS variants and Polyphony along our design space.

- **Ordering Systems** – which mechanisms allow ordering Systems in the chain of execution
- **Entity Templates** – how can one define Templates of Entities for reuse
- **Language extensions** – does the framework implement *metalanguage* extensions, i.e., new syntaxes not defined in the host language

3.2 Polyphony: design choices

With the main design choices for implementing ECS highlighted, we now need to apply it to GUI programming. We first underline the differences between video games and graphical user interfaces development, then we discuss our choices with regards to these differences. A preliminary version of Polyphony was based on Java [42]. We re-implemented the whole toolkit on top of JavaScript in order to benefit from its native syntax and flexibility, which improved our adaptation of the model.

3.2.1 Adapting ECS to the context of HCI. All the toolkits implementing ECS that we have reviewed are dedicated exclusively to video games development. This domain has very specific needs, usually different from those of GUIs and interaction programming:

- The game state is generally updated by a single chain of Systems, running at a fixed *tickrate* (usually 60Hz, a common refresh rate for displays). Since graphical user interfaces are generally updated in response to multiple and varied event sources, we introduced multiple ways to trigger Systems, and the filtering of Systems depending of the triggering event;
- The elements of video games have loose relationships between them. They move, appear, disappear, and require few relations between them — apart from the use of partitioning structures to optimize the rendering [10]. On the other hand, the elements of a graphical interface have relationships of display order, relative positioning, and inherited styles, that are better structured in scene trees;

- Many games follow the principle of *one player per machine* and support a predefined set of input devices (e.g., a single keyboard/mouse pair, joysticks and gamepads). In the context of prototyping of interaction techniques, we want to support a greater variety of devices, including multiple copies, and support their plugging/unplugging at runtime. We introduced the use of *device* Entities in order to support more advanced input management;
- In a game, most observable state changes are related either to the player's actions or to the environment. In the first case, a global event mechanism is generally used to report state changes, and in the second case, *listeners* on Selections are used to observe changes in groups of Entities. In graphical interfaces, it is common to observe state changes and to associate many types of actions with them — interaction techniques, commands, positioning. To materialize and manipulate state changes, we introduced the use of temporary Components, mainly for device Entities;
- The *pipeline* for executing the different Systems is generally fixed, which is why ECS implementations order Systems by simple lists. To improve prototyping, we adopted a recursive approach and we represent Systems by Entities. The triggering of Systems is thus made more flexible, thanks to the management of triggers by Components, and the introduction and implementation of a "Meta-System" that orders and executes Systems;
- Games are often structured in "levels", which are independent from one another, and trigger loading sequences in-between. Conversely, navigation in interfaces involves going back and forth between views, tabs or modes, with many shared elements, and avoidance of loading sequences that could interrupt interaction. To this end, we discarded the use of Context objects, allowing more flexible sharing of elements using Components.

3.2.2 Entities and Components. We represent the Entities by native objects encapsulated in *Proxy* objects from JavaScript. Proxy objects allow us to intercept all native operations they receive, which makes Entities accessible with the native JavaScript syntax [21], although they behave differently. These operations are:

- `e.c` returns the value of Component `c` from Entity `e` (or `undefined`);
- `e.c = v` sets Component `c` to value `v` on `e`;
- `'c' in e` returns `true` if and only if `e` owns `c`;
- `delete e.c` removes `c` from Entity `e`.

The definition of a new type of Component is equivalent to that of a new prototype object in JavaScript. We do not prevent the violation of the ECS model there, since it is possible to include instance methods in the definition of a Component. Our ECS implementation is not designed to be resilient to errors, but rather to provide minimal syntax, in order to build complex applications that take advantage of the benefits of composition, with less effort.

Similarly, for the implementation of Entity Templates we use *factory* Entities, which are simple functions taking as an optional argument an Entity to "augment" with predefined Components.

3.2.3 State changes. In an interactive application, we can distinguish two kinds of behaviors: (i) *transient* behaviors (or transitions) are processes which execute once on a given state change (e.g., a mouse click), and are the most common in event-based programming; (ii) *periodic* behaviors that execute multiple times at a given timespan (e.g., a display refresh).

With ECS, periodic behaviors are implemented directly with Systems. Transient behaviors needs however some additional support. An example of transient behavior in our drawing application is to save the workspace when clicking on the save button. For this case, we provide temporary Components — e.g., `tmpClick` on buttons — that are automatically deleted at the end of the Systems

chain. Temporary Components are distinguished from permanent ones by naming convention, using the `tmp` prefix. Systems down the chain can look for them to implement transient behaviors.

However, some of these behaviors cannot have a single position in the Systems chain, and should rather process a state change instantaneously. For example, anytime an Entity gets a new value for its `depth` Component — that defines its display order — , both `PointerTargetSystem` and the rendering Systems need to sort their Selections of Entities again before iterating them. Since a `depth` update might happen anywhere in the Systems chain, and has an effect for two Systems, the sorting would have to be duplicated in the chain, before every depending System. For this case instead, we added the ad-hoc ability for Selections to be sorted automatically according to a programmable condition when iterated on.

3.2.4 Systems. In Polyphony, Systems are materialized by JavaScript functions, encapsulated as Entities in *Proxy* objects. JavaScript allows assigning properties to functions, while giving them a distinct native type (`function` rather than `object`). We can therefore distinguish an executable Entity from one which is not, while managing the Components in a common manner.

These functions are necessarily *named* — they are not lambda/anonymous functions. Indeed, for the serialization and debugging of applications, it is important that globally accessible Entities know their own name. Any named function of JavaScript stores its own identifier in an attribute `name`, which is automatically considered as a Component.

3.2.5 Context and Selections. Both Artemis-odb and Entitas define a class to store active Entities and materialize the Context. They create new Entities, bind Components to them, and register new Systems by calling methods on the Context object. They also allow instantiating several of these Contexts for representing multiple *worlds*, for example to load the next level or to manage a second player on the same machine. Polyphony does not provide such an explicit Context and makes everything global, without the need to keep a Context reference around. Instead, one may manage several *worlds* by adding a `world` Component to every Entity in order to share Contexts among Entities.

Finally, we base the Descriptors for Entity Selections on native lambda functions *accept* : $Entity \rightarrow Boolean$. The combination algebras *all/one/none* may therefore be implemented using JavaScript's built-in logical operators *and/or/not*. For example, to filter all Entities containing the Component `bounds` but not `origin`: `e => 'bounds' in e && !('origin' in e)`. This concise syntax avoids having to define an additional algebra on top of the host language like in Artemis-odb and Entitas.

4 DISCUSSION

This section discusses our experience building and implementing a GUI toolkit based on ECS and positions our work with respect to the state-of-the-art. While designing Polyphony, we faced many implementation issues for which the chosen solutions could have strong impact and unforeseeable consequences on the usability of the toolkit. In particular, we could have used the common design patterns of existing frameworks such as *listeners* or *delegates*. Instead, we tried to keep the ECS model as *pure* as possible by not mixing it with other paradigms in order to highlight its strengths and weaknesses in the context of HCI.

4.1 Programming GUIs with ECS: Pros and Cons

4.1.1 Reusing behaviors. ECS implies a powerful reuse mechanism: it avoids *duplicate code* by letting Entities delegate their processing to Systems, and it also limits *duplicate execution* by running each System once for all Entities. Duplicate execution is commonly encountered for widgets doing periodic processing in instance methods. When many widgets of the same type share the same

method, it is still executed as many times as there are widgets. Systems, on the other hand, execute one same processing on a whole set of Entities. They may duplicate execution by processing each Entity at a time, but they also have the opportunity to schedule them in parallel when possible.

Seeing and processing all Entities at once allows Systems to perform more efficiently, or accurately. For instance, one may resolve layout constraints using a linear optimization solver such as Cassowary [5], rather than by propagating constraints with messages between Entities [43]. In ECS implementations, Systems are also useful for optimizing the rendering of graphical elements by converting all visible features to arrays for the GPU and sending them at once. Such optimizations exist in common GUI toolkits, although they require complex display and dirty regions managers to buffer and compile the multiple calls to individual refresh methods of individual Components.

With Systems, all behaviors are kept separate from the elements they relate to (instead of instance methods or callbacks). While this is useful for shared behaviors, which are written and executed once for all Entities, it separates unique behaviors from their elements, making it harder to keep track of the behaviors of unique Entities. More realistic use cases and interfaces need to be designed and implemented with our ECS approach before we can draw firm conclusions, but we envision a mixed approach using Systems for shared behaviors and existing mechanisms (instance methods and/or callbacks) for individual ones.

4.1.2 Applicability of the composition pattern. Interaction frameworks commonly rely on two tree-based hierarchies to structure the code and data in an application: an *inheritance tree* (classes) to share and reuse code, and a *scene tree* to structure the interface. Inheritance trees propagate the methods and variables of any object type to its siblings, and allows any sibling to be used where the ancestor is expected. This is the polymorphism of types, and it is the mechanism by which behaviors are commonly shared among different types of widgets. Scene trees are used to structure the interface, and allow the coupling of many aspects of GUIs as a single tree: display ordering, relative positioning, and propagation of styles. Some works also introduce a third *interaction tree*, to express the dependencies between interaction techniques [24, 26, 35].

Video games generally use inheritance trees, while elements in the game typically require less structuring than a scene tree. In this domain, applying a composition pattern like ECS has proven useful, in part because many behaviors (visibility, physicality, controllability) can be decorrelated and composed at will. For example one might instantiate a flower pot that is visible, does not physically block the player, and is inert, but one could also give the player control to this plant, and give it a mass and a bounding box.

For graphical interfaces, many aspects can be uncorrelated from the inheritance and scene tree, and thus benefit from a composition pattern. In our application, for example, the tree traversal that is typically performed during picking and rendering is synthesized in a single System `DepthUpdateSystem`. Also, composing the graphical appearance of widgets with basic shapes (filled polygons, lines, images) allows faster rendering by sending them as batches to the GPU. Another example is the specification of the global layout of the interface with pairwise constraints, instead of relative positioning in the scene tree [5]. Finally, one may compose an interaction technique with the more basic actions inferred earlier, as shown with the Drag and Drop example.

In Polyphony, the ECS composition pattern breaks the inheritance tree, while partially relying on a scene tree. In practice, not all aspects did benefit from composition, and we are still exploring a good trade-off between these structuring principles.

4.1.3 Centralizing interactions. Systems help to centralize the definition of interactions and behaviors in a manner similar to state machines in SwingStates [2]. It thus reduces the use of numerous *callbacks*, which can be problematic [34]. The tool selector of our drawing application illustrates this principle. A `ToggleSystem` looks for clicks on any Entity with a `toogleGroup`,

and untoggles all other Entities with the same `toggleGroup`. Thus, none of the tool selection buttons implement a callback `onClick` for toggling a tool. This common behavior is managed in a dedicated higher level System. Moreover, adding a new tool only requires adding a Component `toggleGroup` to the new button for the System `ToggleSystem` to take it into account. There is no need to add new callbacks or modify the application code in several places.

Another example is the implementation of the Drag and Drop technique we discussed before. The logic for this interaction technique is contained in a single System, despite the many actions required to perform the whole interaction.

4.1.4 Abstracting capabilities with Components. With ECS, devices are characterized by their Components: `cursorPosition` and `buttons` for the mice, `bounds` and `origin` for the views, and `keyStates` for the keyboards. This principle allows devices to be abstracted while retaining their individual characteristics. Data such as the presence of additional buttons on a mouse is kept, but simply ignored by any System processing two-button pointers.

Also, thanks to the systematic use of Selections to gather Entities, interaction techniques are implicitly exposed to the possibility of using multiple devices. Although we have not demonstrated it in our prototype, the use of several input and output devices (physical or virtual), as well as their replacement, is facilitated because it is implicit, as long as they present the right Components. For example, a System implementing a pointing technique could work with any Entity that provides the Components `cursorPosition` and `buttons`, whether it is a mouse, a tablet or a virtual device producing these data (e.g., a "robot" replaying input records).

In practice, we observed that Components would fall in two categories:

- **attributes** – which are only properties of the Entities, are often shared by several Systems, and do not call for any specific behavior (e.g., `bounds`, `depth`, `order`)
- **activators** – which are often needed by *one* System, thus related to its specific behavior (e.g., `image`, `targetable`, `buttons`)

When designing the set of available Components and the corresponding Systems, we often need to *reliably* control the activation of every System for any Entity. For this purpose, a single triggering Component is arguably safer than a combination of more, hence the emergence of activators. In our application, for instance, the `View` Template shares the `bounds` Component with many other kinds of Entities, and only the `origin` Component is unique to views. The ability to *display elements* is actually given by this Component. However, ECS lacks clear links between Systems and the Components used to activate them. An evolution of ECS should eventually tackle the expression of dependencies among Components and Systems.

4.1.5 Ordering behaviors between elements. The ordering of Systems makes it possible to order the behaviors by type of processing in the application. However, some operations require Entities to be ordered, such as graphical rendering. In this case, the drawing of background shapes, borders and images are separate operations executed in order for each Entity. And these operations also need to be ordered between Entities, i.e., drawing the image of a background Entity before the border of a foreground Entity. This is illustrated in Fig. 6, where the gathering of processes in Systems reorders their execution. Programming with Systems falls short of handling this case: one has to (i) make the reordered processes insensitive to execution order, or (ii) implement all operations in a single System iterating on the Entities. Operations with a strict ordering policy between Entities may not be expressed easily with ECS, and require additional mechanisms to ignore the order. In the case of graphical rendering, the use of a *depth-buffer* allows drawing all the elements in any order.

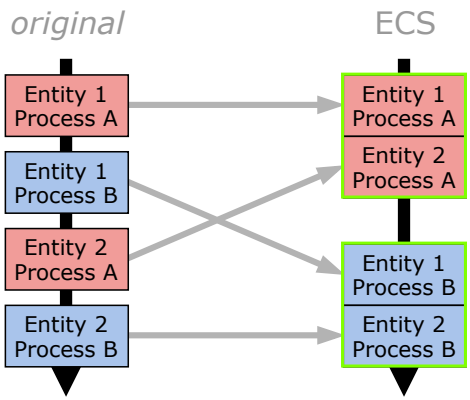

Fig. 6. Ordering of two processes with two Entities. Gathering executions by process type is shown to change the order of executions.

4.2 Related work: Management of interactive behaviors

The management of *interactive behaviors* is the most significant contribution of ECS to the implementation of GUIs. Interactive behaviors can be defined on two levels. At a high level, they refer to the observable reactions of the system to different stimuli. These stimuli can be external events (mouse click, clock tick), or internal (mouse hovering over an element). At a low level, behaviors refer to the code blocks triggered by these events, that generate the observed reactions. For example, for the high-level behavior "editing text", the corresponding code is attached to keyboard presses and it adds each character entry at the end of a string which display is then updated. In this section, we review existing approaches to describe and program such behaviors, and position our ECS-based approach in regards to these works.

4.2.1 Mutualization and reuse of behaviors. Most Object-based toolkits use the principle of inheritance to organize shared behaviors, and can be described as "monolithic" [7]. Each behavior is encapsulated in a class (or prototype) with a set of variables and methods, and any descendant object inherits it. This principle associates the very *nature* of the elements with their behaviors (proper or inherited) — e.g., a class `TextField` is specific in that it contains text and can be edited.

However, hierarchies between ancestors are generally predefined, which prevents the attribution of behaviors to objects that have not been specifically designed and implemented to receive them [7, 29, 35], both statically (at compilation) or dynamically (at runtime). For example, it is difficult to add text editing behavior to the *label* fields of an interface. In fact, the functions for *receiving keyboard support*, *displaying a blinking text cursor* and *updating a character string*, often belong to another family of Components, dedicated to text editing.

For a programmer with access to the interface source code, this change can be done in two ways: either with a subclass of a *text field* modified to look like a label, or with a subclass of a *label* modified to allow text editing. In both cases one of the two behaviors has to be recreated since it cannot be inherited. This problem has led to object architectures that favor *Composition* over *Inheritance*, such as *Traits* [16] or *mixins* [12]. For a programmer who does not have access to the source code, adding a behavior to an element amounts to changing its *nature*, which is sometimes impossible. As Lecolinet notes, "*behaviors and other features are not seen as general services that could be used by any widget*" [29].

Many interaction libraries called "polylithic" [7] have been proposed, following the paradigm of *Composition*. Taylor et al. [46] presented an architecture with independent agents, exchanging messages between layers without knowing the agents with which they communicate. This facilitates the reuse of large-scale behaviors, but not the management of low granularity behaviors (such as text editing). Jazz [8] models behaviors by nodes inserted between interface elements in a *scene graph*. Behaviors are thus inherited by all the children in the graph (which is actually a tree). A similar principle has been adopted in MaggLite [26], following a *mixed graphs* model that links reusable interactive behaviors to the nodes of a scene graph [25]. In Polyphony the Systems are comparable to the interaction graphs in MaggLite, as well as the Interactors in Garnet [33]. However, it differs in that the Systems are instantiated once for all Entities, rather than once per Entity. It therefore facilitates the expression of group behaviors — e.g., linear positioning constraints [5], realistic shadow drawing.

4.2.2 Dynamicity of behaviors. Most GUI frameworks have limited support for adding new behaviors to objects at runtime. It is also difficult to alter behaviors on the fly (e.g., *enable/disable*), unless they have been designed for this purpose. For instance, standard drop-down menus/list widgets do not allow reordering their elements at runtime with the mouse. Making menu items orderable during program execution is an example of dynamic behavior. It is important to mention that this dynamicity is independent from the choice of Inheritance or Composition for sharing behaviors. The point is to be able to change the Components or parents of a given object *at runtime*. A common solution is for each element to possess the intended behavior and to disable it by default. This is for instance the case in Qt for orderable lists: the base class `QAbstractItemView` defines a method `setDragEnabled` that allows each list to enable the ability to move elements with the mouse. However, this approach is limited, as new behaviors must be added through the base class.

This limitation has led researchers to develop various techniques to assign behaviors on the fly. For instance, Scotty [20] allows analyzing the structure of the interactive components in an existing Cocoa application, and injecting code into it to change its behavior. Although spectacular and effective for the rapid prototyping of new interaction techniques in existing applications, this approach has weaknesses in robustness and persistence that prevents it from being used on a larger scale than for temporary "hacking". Ivy [13] is a messaging bus on which agents can send text and register to receive messages (selected by regular expressions). With this mechanism, new behaviors are added through new agents, and agents replace each other by sending compatible messages. Amulet [35] provides a Prototype-oriented programming model above C++, in which objects can add, replace or delete variables and methods at any time. Our ECS-based implementation is very similar to this work, however we extend the notion of Entities to interaction devices as well as Systems, in order to provide greater flexibility in the scheduling of behaviors at runtime.

4.2.3 Orchestration of behaviors. The triggering and execution order of functions on interactive elements is important. It may be the systematic execution of a function before/after another one, after a given state change, or upon receipt of events from one or more input devices. GUI design is full of such complex use cases. For example, the graphical rendering of Web interface elements involves drawing backgrounds, borders, text, or even drop shadows. Using the painter's algorithm, the rendering steps for each element must be performed in a specific order: the background *before* the border and text, and the drop shadow *before* the background. Another example is some interaction techniques that rely on sequences of actions, potentially coming from several input devices, and involving delays (such as *CTRL + mouse press + 500ms pause*). Here again, languages and libraries have limited support for these needs. Function calls offer a sequential and static orchestration of the different code blocks, and callback mechanisms generate hundreds or even thousands of links for "realistic" interfaces [34].

To overcome these limitations, different models have been proposed. Models based on the state-transitions formalism (state machines) in HsmTk [11] and SwingStates [2], limit the "fragmentation" of the code for defining interactive behaviors, while providing a representation of interaction that is close to the designer's and programmer's one. Data flow models have also been proposed, which execute code blocks when all the data on which they depend is available. Among these, ICon [18] reifies dependency links into a visual language, which users can manipulate directly, for example to add or replace an input device. The two models have even been combined in FlowStates [3] in order to take advantage of the two formalisms where they are most appropriate. Model-based approaches such as MARIA [40] and UsiXML [31] have also been used to abstract the definition of user interfaces from the modalities available on each computer (devices, operating systems), and to facilitate the automatic processing of scene trees. In the context of gestures and action sequences, Proton [27] is notable for the use of regular expressions to represent complex gestures, and GestIT [45] is notable for modeling complex action sequences with six composition operators.

Most model-based approaches have in common that they represent *micro-dependencies* inside interaction techniques — they make many relations "do B after A" explicit, instead of the implicit sequencing in code. By doing so, they tend to require a lot of code and to have scalability issues when dealing with complex interfaces or interactions. In ECS, the orchestration of behaviors is carried out at a *macro* level, not on the data but on the processes. The model aligns with the execution of algorithms related to video games (3D rendering, physical simulation), which process large amounts of data repeatedly. It incites to consider behaviors as processes that transform input events into output events. As Chatty et al. write to present djnn, "*Like computations can be described as the evaluation of lambda expressions, interactions can be described as the activation of interconnected processes*" [15]. Polyphony differs from these works by adding attributes (Components) to processes (Systems), in order to explicitly address the dependencies among them, while not constraining the choice of attributes to be given. The orchestration of an application can thus be extended and improved by the introduction of new Components.

5 CONCLUSION

In this article we discussed the design of an interaction toolkit based on the Entity-Component-System (ECS) model. We clarified this model in the perspective of user interfaces and interactions programming, detailed an example of a vector drawing application based on it, and identified the distinctive aspects of its use in practice. We then discussed our design choices for the implementation of a GUI toolkit based on ECS — Polyphony — according to three major ECS implementations. Finally, we positioned our approach against more general reference work, with a particular focus on behavior management in interactive applications.

The main contributions of the ECS model to interaction programming are the polymorphism of elements thanks to Entities, a powerful mechanism of Selections thanks to Components, and an advanced management of interactive behaviors through "Systems oriented" programming. However, the flexibility of this model is two-edged, since many high-level behaviors can be expressed in several ways and we still need to explore varied use cases in order to better characterize them.

5.1 Limits and future work

Among the potential strengths of the ECS model for implementing GUIs, there are a number that we have not explored and implemented yet such as: The use of multiple and dynamically changing devices; The management of layouts and positioning constraints; The implementation of an advanced *hardware* rendering engine with multiple Systems.

Finally, although we detailed and presented a fully functional implementation of ECS for GUI programming in this article, Polyphony is still a prototype and a testbed for adapting ECS to

interaction programming. Our approach still needs to be tested with more advanced and ecological use cases, and ideally to be adopted by a significant user base to be validated [38]. But Polyphony is also as much an effort for designing usable interaction programming toolkits as it is an exploration of the requirements for the design of modern programming languages tailored to interactive applications. We hope this work will foster the development of "hybrid" frameworks complementing the programming of standard *widgets* with the reuse mechanisms from ECS.

ACKNOWLEDGMENTS

The authors would like to thank Damien Pollet for his interest in this work and numerous discussions about the project.

The code of the Polyphony toolkit is available at https://gitlab.inria.fr/Loki/PolyphonyECS.

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. https://doi.org/10.1145/1029632.1029635

[2] Caroline Appert and Michel Beaudouin-Lafon. 2008. SwingStates: Adding State Machines to Java and the Swing Toolkit. *Software: Practice and Experience* 38, 11 (Sept. 2008), 1149–1182. https://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. https://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.

[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. https://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. https://doi.org/10.1145/354401.354761

[7] Benjamin B. Bederson, Jesse Grosjean, and Jon Meyer. 2004. Toolkit Design for Interactive Structured Graphics. *IEEE Transactions on Software Engineering* 30, 8 (Aug. 2004), 535–546. https://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. https://doi.org/10.1145/354401.354754

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

[10] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz. 1998. Designing a PC Game Engine. *IEEE Computer Graphics and Applications* 18, 1 (Jan. 1998), 46–53. https://doi.org/10.1109/38.637270

[11] 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. https://doi.org/10.1145/1133265.1133275

[12] 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. https://doi.org/10.1145/97945.97982

[13] 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. https://doi.org/10.1145/777005.777040

[14] Géry Casiez and Nicolas Roussel. 2011. No More Bricolage!: Methods and Tools to Characterize, Replicate and Compare Pointing Transfer Functions. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 603–614. https://doi.org/10.1145/2047196.2047276

[15] 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. https://doi.org/10.1145/2774225.2774848

[16] 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. https://doi.org/10.1145/800210.806468

[17] 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.

[18] 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. https://doi.org/10.1007/978-1-4471-0353-0_34

[19] Ekwa Duala-Ekoko and Martin P. Robillard. 2012. Asking and Answering Questions About Unfamiliar APIs: An Exploratory Study. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 266–276.

[20] 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. https://doi.org/10.1145/2047196.2047226

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

[22] 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. https://doi.org/10.1145/2670444.2670446

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

[24] Scott E. Hudson, Jennifer Mankoff, and Ian Smith. 2005. Extensible Input Handling in the subArctic Toolkit. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '05)*. ACM, New York, NY, USA, 381–390. https://doi.org/10.1145/1054972.1055025

[25] 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. https://doi.org/10.1145/1132736.1132742

[26] 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. https://doi.org/10.1145/1029632.1029677

[27] 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. https://doi.org/10.1145/2207676.2208694

[28] Sam Lantinga. 1998. Simple DirectMedia Layer - Homepage. http://libsdl.org/.

[29] 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. https://doi.org/10.1145/964696.964711

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

[31] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. 2005. USIXML: A Language Supporting Multi-Path Development of User Interfaces. In *Engineering Human Computer Interaction and Interactive Systems (Lecture Notes in Computer Science)*, Rémi Bastide, Philippe Palanque, and Jörg Roth (Eds.). Springer Berlin Heidelberg, 200–220.

[32] Adam Martin. 2007. Entity Systems Are the Future of MMOG Development – Part 1.

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

[34] 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. https://doi.org/10.1145/120782.120805

[35] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. 1997. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering* 23, 6 (June 1997), 347–365. https://doi.org/10.1109/32.601073

[36] Brad A. Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew Ko. 2008. How Designers Design and Program Interactive Behaviors. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. 177–184. https://doi.org/10.1109/VLHCC.2008.4639081

[37] 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. https://doi.org/10.1145/142750.142789

[38] Dan R. Olsen, Jr. 2007. Evaluating User Interface Systems Research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. ACM, New York, NY, USA, 251–258. https://doi.org/10.1145/1294211.1294256

[39] Adrian Papari. 2018. Artemis-Odb: A Continuation of the Popular Artemis ECS Framework.

[40] Fabio Paterno', Carmen Santoro, and Lucio Davide Spano. 2009. MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments. *ACM Trans. Comput.-Hum. Interact.* 16, 4 (Nov. 2009), 19:1–19:30. https://doi.org/10.1145/1614390.1614394

[41] prime31. 2018. Nez Is a Free 2D Focused Framework That Works with MonoGame and FNA.

[42] Thibault Raffaillac and Stéphane Huot. 2018. Applying the Entity-Component-System Model to Interaction Programming. In *Proceedings of the 30th Conference on L'Interaction Homme-Machine (IHM '18)*. ACM, New York, NY, USA, 42–51. https://doi.org/10.1145/3286689.3286703

[43] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. 1993. Multi-Way versus One-Way Constraints in User Interfaces: Experience with the Deltablue Algorithm. *Software: Practice and Experience* 23, 5 (1993), 529–566. https://doi.org/10.1002/spe.4380230507

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

[45] Lucio Davide Spano, Antonio Cisternino, and Fabio Paternò. 2012. A Compositional Model for Gesture Definition. In *Human-Centered Software Engineering (Lecture Notes in Computer Science)*, Marco Winckler, Peter Forbrig, and Regina Bernhaupt (Eds.). Springer Berlin Heidelberg, 34–52.

[46] 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. https://doi.org/10.1145/225014.225042

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

[48] W3C. 1996. Cascading Style Sheets, Level 1. https://www.w3.org/TR/CSS1/.

[49] Minhaz F. Zibran, Farjana Z. Eishita, and Chanchal Kumar Roy. 2011. Useful, But Usable? Factors Affecting the Usability of APIs. In *2011 18th Working Conference on Reverse Engineering*. 151–155. https://doi.org/10.1109/WCRE.2011.26

Received February 2019; revised March 2019; accepted April 2019