XRSpotlight: Example-based Programming of XR Interactions using a Rule-based Approach

VITTORIA FRAU, Dept. of Mathematics and Computer Science, University of Cagliari, Italy
LUCIO DAVIDE SPANO, Dept. of Mathematics and Computer Science, University of Cagliari, Italy
VALENTINO ARTIZZU, Dept. of Mathematics and Computer Science, University of Cagliari, Italy
MICHAEL NEBELING, University of Michigan, USA

Research on enabling novice AR/VR developers has emphasized the need to lower the technical barriers to entry. This is often achieved by providing new authoring tools that provide simpler means to implement XR interactions through abstraction. However, novices are then bound by the ceiling of each tool and may not form the correct mental model of how interactions are implemented. We present XRSpotlight, a system that supports novices by curating a list of the XR interactions defined in a Unity scene and presenting them as rules in natural language. Our approach is based on a model abstraction that unifies existing XR toolkit implementations. Using our model, XRSpotlight can find incomplete specifications of interactions, suggest similar interactions, and copy-paste interactions from examples using different toolkits. We assess the validity of our model with professional VR developers and demonstrate that XRSpotlight helps novices understand how XR interactions are implemented in examples and apply this knowledge in their projects.

CCS Concepts: • **Human-centered computing** → **Virtual reality**; **Mixed / augmented reality**.

Additional Key Words and Phrases: XR; MR; AR; VR; example-based programming; XR modelling; end-user development;

ACM Reference Format:
Vittoria Frau, Lucio Davide Spano, Valentino Artizzu, and Michael Nebeling. 2023. XRSpotlight: Example-based Programming of XR Interactions using a Rule-based Approach. *Proc. ACM Hum.-Comput. Interact.* 7, EICS, Article 185 (June 2023), 28 pages. https://doi.org/10.1145/3593237

1 INTRODUCTION

Prior studies with novice and professional XR developers highlighted both the need for lowering the technical barriers to entry [3] and the need for new tools for XR prototyping [24]. A common solution to this problem is designing new authoring tools that simplify the development process of XR interactions through abstractions. For example, Flowmatic [57] introduced a reactive visual programming model to design interactive scenes in VR using a data flow programming abstraction. Similarly, Rapido [29] captures the interaction sequences by demonstration in AR and expresses them in a state machine representation that designers can author to adjust the flow. While these tools and their abstractions can make it easier for novice XR developers, this approach has two main problems. First, novice developers do not know how to get started [3]. They need to understand

Authors' addresses: Vittoria Frau, Dept. of Mathematics and Computer Science, University of Cagliari, Via Ospedale 72, Cagliari, Italy, 09124; Lucio Davide Spano, davide.spano@unica.it, Dept. of Mathematics and Computer Science, University of Cagliari, Via Ospedale 72, Cagliari, Italy, 09124; Valentino Artizzu, Dept. of Mathematics and Computer Science, University of Cagliari, Via Ospedale 72, Cagliari, Italy, 09124; Michael Nebeling, nebeling@umich.edu, University of Michigan, Ann Arbor, USA.

2573-0142/2023/6-ART185 $15.00
https://doi.org/10.1145/3593237

which toolkit supports their goals and find good examples to start with. So they inspect different projects created through different technologies before deciding, and they may want to combine interactions found in different examples. Second, simpler tools tend to lock designers in, effectively making them subject to each tool's limitations and harder to transition to more advanced tools [40]. Relying on abstractions also means there is a risk that novice developers form a limited or even incorrect mental model about how XR interactions are implemented.

A better approach might be to provide novices with a rich set of examples of how to use a toolkit to implement certain interactions [10, 19]. Previous research on example-driven design has contributed new programming and web design tools that support the use and adaptation of examples [5, 6, 19, 25, 31, 58]. XR toolkits usually come with example scenes which are often designed to demonstrate a toolkit's flexibility and expressiveness as encouraged in research [41], but—if extended with new tools for inspection and adaptation—could provide a useful way for novices to learn how interactions are implemented. Those example scenes, however, are often too abstract or not problem-oriented, especially when it comes to showing the feature of interest or learning from examples.

In this paper, we present XRSpotlight, a system that aims to empower novice XR developers to find and adapt existing examples to their own XR designs by identifying and representing interactions in natural language. The proposed approach is based on a model abstraction that unifies existing XR toolkit implementations and provides an interface designed to enable novices to understand interactions in a toolkit-agnostic way. XRSpotlight lists all the interactions defined in a Unity scene and presents them as rules in constrained natural language. This enables novices to identify incomplete specifications of interactions (e.g., by highlighting possible missing components such as colliders and rigidbodies), find similar interactions as examples to adapt, and copy-paste interactions from examples using the same or different toolkits. These features allow a simpler understanding of the XR interactions, linking its simplified view with the more complex definition in Unity and guiding the transition of the XRSpotlight support.

The main contributions of our work are: *i)* the design of our abstraction model behind XRSpotlight, which enables example-driven development of XR interactions across different toolkit implementations; *ii)* the demonstration that our abstraction model correctly captures existing toolkit implementations and matches professional VR developers' mental model of how XR interactions are implemented with different XR toolkits; *iii)* our study with novice XR developers showing that XRSpotlight better supports them in inspecting example XR interactions compared to traditional means (using only the Unity Inspector) and enables them to build new XR experiences. The results obtained in the study with novices suggest that XRSpotlight could effectively support them in the transition from our interaction model representation towards programming interactions without having to rely on our tool support.

2 BACKGROUND AND RELATED WORK

Our work adds to a recent stream of research on authoring tools to empower novice XR developers, adapts existing streams of research on example-driven design to XR, and deals with the proliferation of new XR development toolkits.

2.1 Authoring Tools for Novice XR Developers

The entry threshold [23, 36] is one main barrier to the development of XR environments. This was confirmed by two separate studies [3, 24], which also agreed to identify the usage of online resources and the lack of concrete guidelines and examples as open problems for increasing the number of XR developers. The existing research mainly focuses on creating authoring tools for lowering such threshold, usually paying the increased simplicity with a lower ceiling. These tools

are designed for two categories of users: end-user developers and XR designers. In the real world, such distinctions may blend, and often the same person assumes different roles in an XR project [24]. These profiles have limited development skills in common but differ in terms of the artefacts they each want to produce. While end-user developers typically aim to configure or adapt an existing XR experience, designers usually aim to create a prototype of a new XR experience for experimenting or communicating design ideas.

We can find different examples of authoring tools for developers in the literature. We distinguish between *immersive* authoring tool, allowing XR development while immersed environment, and *desktop* authoring, applying the build-test-fix cycle common to development environments. In the immersive authoring [27, 28], developers have the advantage of editing XR interactions *in-situ*, i.e., directly in the environment where the interaction happens. From the early [46] to the latest attempts [55, 56], such approaches support navigation or manipulation interactions having limited consequences in terms of changes in the XR environment state. Desktop authoring tools use a variety of representations for the interaction logic, ranging from graphs [13, 50], block-based programming [9], event-condition-action rules [4, 11, 54], or domain-dependent representations [14, 15]. More recent immersive tools try to increase expressiveness by adapting desktop visualizations in VR. Flowmatic [57] adapted the graph-based representation [13, 50] reaching a ceiling equivalent to 2D authoring tools. Such notation is very close to scripting and difficult to use for novices. Artizzu et al. [2] propose an immersive interface to configure *template* environments, specifying the behaviour of specific objects through natural language rules. We use similar rules but do not use them at runtime. All tools create a different (simpler) representation of the XR environment behaviour, lowering the threshold and the ceiling. This simplified representation enables developers to create new XR experiences, but it also defines the boundary for the types of XR experiences they are able to create. Our representation is an interaction-focused view on top of a game engine (Unity), guiding developers in finding and understanding the relevant information for building full-fledged XR environments.

Related attempts to lower the threshold are available in prototyping tools for designers. Such tools often try to reuse existing practices, like sketching [16, 38, 39, 47], or film-making roles [37] for obtaining interactive prototypes. Pronto [30] supports creating AR interaction through video-capturing and supporting 3D manipulations through a tablet to fine-tune the prototype. In follow-up work on Rapido [29], the authors enhance the tool with a programming by demonstration metaphor for prototyping interactions in AR. The principles are based on video sketching, but the underlying technique uses ARKit to capture the interaction sequences in 6DOF with respect to the environment. Designers may revise the interaction sequences through a state machine representation to author the flow. The opportunity for creating low to middle-fidelity prototypes of the environments is also their main limitation, as creating high-fidelity prototypes will require abandoning the prototyping tool and switching to more advanced tools like Unity [24]. In this phase of their work, they would take advantage of adapting existing examples and focusing on creating the interactions without the need to master specific interaction toolkits.

Resnick et al. [43] identify "wide walls" as a desirable property of user interface development tools, besides low threshold and high ceiling. Having wide walls is a metaphor for identifying tools able for a wide range of projects, even different from those they were designed for. None of the previously discussed tools explicitly position their approach on this aspect. However, prototyping tools [30, 38, 39] leverage existing practices for fostering the designers' creativity, and the papers include diverse design examples, showing a good potential support for wide walls. Instead, authoring tools for end-users [2, 4, 54, 57] focus more on discussing the significance of the interactions and behaviours they simplify for addressing non-programmers. This does not allow us to position such work on their wall wideness. Even though XRSpotlight is designed mainly for inspecting examples,

it supports creative variations through the copy-paste feature. While pasting an interaction on a different object, the tool suggests reconfiguring it to the target scene and even the toolkit. This supports novices in creating variations of the same interaction.

2.2 Example-Driven Design and Programming with Examples

One of the main barriers novice XR developers face is knowing where to start [3]. They need to understand which toolkit better supports the interaction they have in mind and the effort required for reproducing them. In this regard, finding relevant examples is crucial, but there are few experiences they can draw upon compared to web and mobile development [3]. Example-based programming focuses on supporting the development by adapting relevant examples [10]. However, finding a relevant example is not an easy problem for novices. Usually, they lack the knowledge to establish patterns and identify the similarities, which complicates the search problem of online resources [12].

Prior literature presented tools to identify relevant examples for fostering the learning of a language or technology. For instance, Blueprint [5] supports code search directly in the development environment, augmenting queries with code context and retaining a link between copied code and its source. Similarly, Rehearse [6] enable programmers to adapt example code by identifying lines of code relevant to a particular interaction. Hartmann et al. [19] presented a tool called d.mix, which creates web mashups using already-established websites as examples. Ghiani et al. [17] allowed to graphically select snippets from webpages and paste them into a mashup application. Zhang et al. [58] further advanced this approach by supporting developers in glueing together the snippets. Ply [31] supports novice web developers in identifying CSS patterns inside complex stylesheets, hiding rules that are not relevant to the visual feature selected on the current page. Kumar et al. [25] used knowledge discovery techniques to build a design-oriented website repository. AI-based tools appeared recently in this area through embeddable agents suggesting the implementation of entire functions in real-time directly in the code editor [22]. To our knowledge, there is no specific support for finding relevant sample interactions in XR, which motivates this paper's research.

Even though novices can find a relevant example through any search, they need to understand them, i.e., to discover the crucial concept in the example for solving their problem. Ichinco and Kelleher [20] identified different obstacles that prevent the correct usage of the example: how to apply it in the destination context, the programming environment hurdle and the code comprehension. So, novice developers need support in finding an example and a focused representation of the information that makes it relevant. XRSpotlight displays XR interactions as natural-language rules, helping novices identify and understand the information relevant to interactions in an XR example environment. We ease the transfer of its implementation from the example to the experience under development through the copy-paste feature.

2.3 Toolkits for XR Development

While research is currently experimenting with techniques for targeting different prototyping stages and different levels of development expertise, the current practice in creating XR experiences mainly involves commercial AR/VR platforms such as Unity and Unreal [24]. These platforms depend on additional toolkits for handling XR interactions, including components, APIs, debugging tools, documentation and samples. Considering the proliferation of new XR platforms and devices, there is now a large variety of XR toolkits, which makes it difficult for novice XR developers to know about their features and limitations [40]. We provide an overview of the currently available toolkits in Table 1, considering the supported game engine (Unity or Unreal), if they support AR, VR, or both, and which devices they target. Novice developers need to mature knowledge about XR

Table 1. A list of toolkits currently available for XR development.

Toolkit	Game Engine	XR Support	Supported Devices
ARCore [18]	Unity, Unreal	AR	mobile
ARF [49]	Unity	AR	AR headsets, mobile
ARKit [21]	Unity, Unreal	AR	mobile
MRTK [35]	Unity	VR, AR	VR and AR headsets, mobile
Oculus Integration SDK [33]	Unity, Unreal	VR	VR headsets
SteamVR [44]	Unity, Unreal	VR	VR headsets
VirtualGrasp SDK [1]	Unity	VR	VR headsets
VIU [45]	Unity	VR	VR headsets
VRTK [32]	Unity, Unreal	VR	VR headsets
XR Interaction Toolkit [48]	Unity	VR, AR	VR and AR headsets, mobile
XRTK [53]	Unity	VR, AR	VR and AR headsets, mobile

Fig. 1. An overview of the scenes and interaction used in the walkthrough. A: the coffee cup in the MRTK sample scene has a manipulation interaction. B: the kitchen scene with the egg ingredient shows feedback on manipulation. C: the same egg shows the feedback on pointing after changing the rule trigger. D: the feedback replicated on the wrong ingredients. E: the pan drag interaction imported from a SteamVR sample.

interactions to orient themselves in this complex toolkit landscape, e.g., to select the most capable and convenient toolkit for their needs and experience.

In this work, we propose a toolkit abstraction extracting the relevant information for supporting novice developers in understanding the interactions in an XR scene. In the development of our tool, we focused on two representative toolkits: MRTK [35], which covers different types of XR devices (e.g., VR and AR headsets), and SteamVR [44], which focuses on VR headsets and interactions.

3 XRSPOTLIGHT WALKTHROUGH

Before illustrating how XRSpotlight works, we discuss the support it offers to novice developers through concrete usage examples. The first three steps in this walkthrough are a subset of the tasks we used for the user evaluations in Section 8. James is a novice XR developer, so he is familiar with programming languages but has little experience building XR environments or using game engines. He is working on an ongoing project which uses Unity and MRTK to define interactions in a virtual kitchen, where the user can simulate the cooking of simple recipes. The Unity scene is furnished with assets representing kitchen furniture, tools and ingredients and has MRTK imported.

Interpreting and finding examples of interactions in the scene. James' goal is to create guidance for selecting the ingredients in the current recipe. He wants to show feedback (correct or wrong) as long as the user manipulates an ingredient. It is the first time he creates such an interaction, so he needs to understand how to implement it. He starts from a working example by inspecting one of the examples shipped with the MRTK toolkit, the hand manipulation scene. James would like to find an interaction similar to the one he needs to implement, and he picks

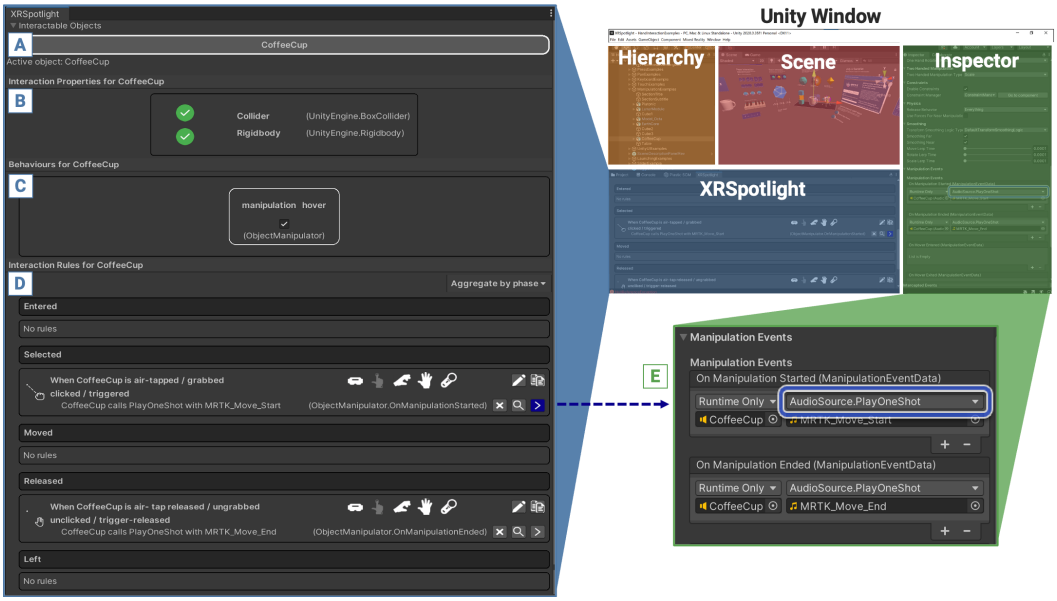


Fig. 2. The XRSpotlight panel in the Unity editor. A: foldout menu showing all the interactable objects in the scene (partially cut for readability). B: information about the support to collision detection and physics. C: interactive behaviours provided by the components associated with the XR object. D: interaction rules, aggregated by phase (see Section 6.1). Each rule entry has icons representing the modalities it supports and a set of buttons allowing (starting from the upper left): to change the trigger, copy the rule, delete the action, find similar actions and show the definition on the Inspector. E: Highlighting the definition of a rule action (consequence) in the Unity Inspector.

the coffee cup object since it provides manipulation affordance (Figure 1-A). By default, the Unity Inspector lists many components, some of them seem related to the interaction, and others seem not, but he needs to figure out the example relevance. To understand the interaction provided by the object, James opens the XRSpotlight window. The interface shows the list of interactions producing consequences in the experience, expressed as trigger-action rules, as Figure 2-D shows. The novice understands that the cup reproduces a sound when a user selects it and stops when released, which is a good starting point for his development task. XRSpotlight helps James identify how MRTK defines this interaction by highlighting the section in the Unity Inspector. James identifies the event related to the interaction in MRTK and the method used for playing the sound (see Figure 2-E).

James has found a relevant example, but he wants to see if there are *better* examples. XRSpotlight supports this task by finding a list of similar rules (see Figure 3). He understands that many objects in the scene behave similarly when moved, even if they have different shapes. Then James realizes that the example he selected is good enough for his purposes.

Example transition to a different scene. The next step is adapting the rules in the MRTK example scene to the kitchen environment. James uses XRSpotlight to copy the first coffee cup rule for the movement and to paste it into one of the ingredients in the kitchen scene, the egg (Figure 4-A). XRSpotlight notifies James that he needs to define the interaction effect since the code defining them in the source scene is not available in the kitchen environment. He wants to show feedback (correct or wrong) as long as the user manipulates an ingredient. Therefore, James writes a script to show the visual feedback and connects it to the interaction for moving the egg, copied

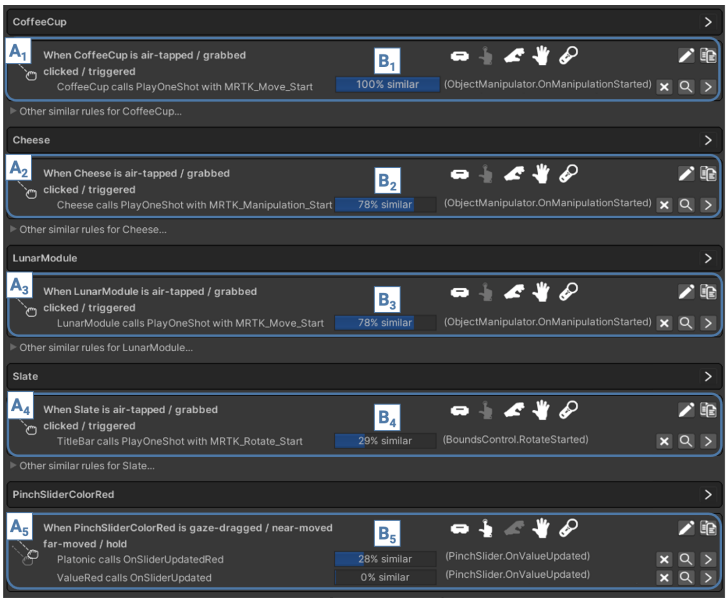


Fig. 3. XRSpotlight shows the interactions similar to the one currently selected by showing an ordered list of rules involving other objects in the scene. For each object (A_i), it shows the most similar rules it supports, including itself (A_1). Next to each row, we show the percentage of the similarity (B_i) and rules similar to the one displayed.

by XRSpotlight and highlighted in the inspector. After that, the kitchen environment contains an interaction in which the egg shows visual feedback as long as the player is moving the egg (Figure 1-B). James follows the same procedure with the second rule in Figure 2-D to hide the feedback when the user releases the egg.

Adapting and Replicating interactions in the scene. Having defined a working interaction, James tests the experience but, even if the feedback on the egg shows and hides correctly, he realises that the selected interaction is not well suited for the task. He thinks it would be better if the user sees the feedback by just pointing at it. Thus, James aims to modify the interaction accordingly. XRSpotlight allows him to change the trigger part of the rules, leaving the effects as they are (Figure 4-B). Changing both the rules' triggers, the egg shows the feedback when pointed and hides it when left (Figure 1-C). Now, the novice developer would like to replicate the interaction for each ingredient on the kitchen table. He uses the copy-paste feature of the previous step (Figure 4-C) and all the ingredients support the feedback interaction (Figure 1-D).

Adapting examples from other toolkits. Suppose that James's next goal consists in implementing an interaction where when a pan is dragged, it makes smoke disappear. He recalls having used a similar interaction in a recent project he found on GitHub. James founds the repository again after a quick search, but *ExamplePan* uses SteamVR, not MRTK. For adapting the example, he would need to master both toolkits, but he cannot find similar projects using MRTK. XRSpotlight allows copying the interaction from the SteamVR scene and pasting it inside the MRTK scene (Figure 1-E). Relying on its modelling abstraction, it searches for the best correspondence between the two toolkits in modelling the considered interaction, if any. The steps are similar to the previous ones and usually require adapting the interaction consequences.

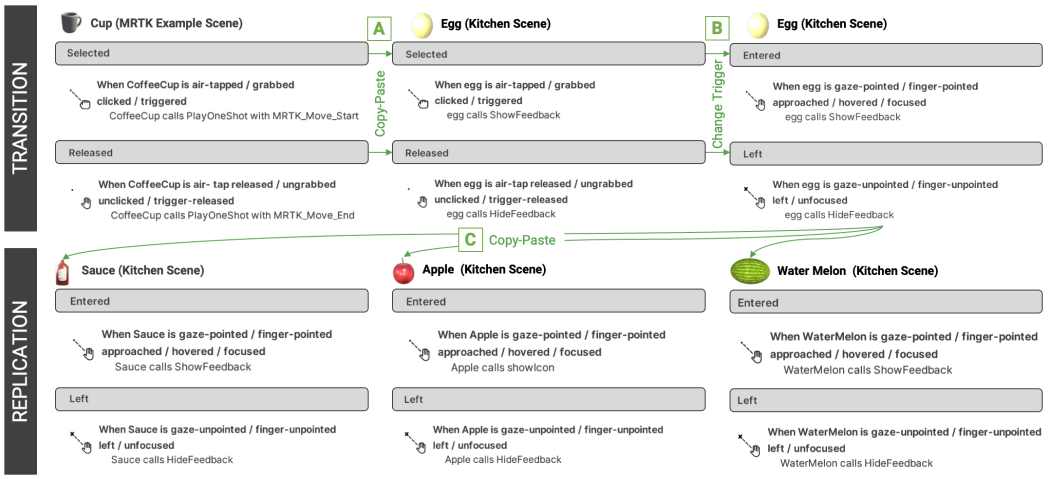


Fig. 4. The import (transition) of an interaction from an example through XRSpotlight, and its replication on other objects. It starts with a copy of the rules Selected and Released from the coffee mug located in the MRTK example on the egg in the kitchen scene (A). Then, the rules are adapted to support a hovering interaction through the change trigger feature (B). After that, the obtained rules are replicated on three other ingredients through copy-paste (C).

4 XRSPOTLIGHT OVERVIEW

In summary, the support provided by XRSpotlight to the tasks depicted in Section 3 consists of three main features: *(1)* expressing interactions in natural language, *(2)* finding examples of similar interactions in the scene, and *(3)* copy-pasting interactions in a toolkit-agnostic way. Figure 5 provides an overview of our XRSpotlight system.

XRSpotlight supports the first feature through a panel providing a rule-based description of the interactions involving the selected game object. In Unity, identifying the interactions and their effects currently requires inspection through different editor panels and/or reading code, presenting an overwhelming amount of information that makes it hard for novices to form the correct understanding. We use trigger-action rules described using natural language sentences to express interactions and communicate when a rule applies (e.g., when the user grabs the object) and its consequences (e.g., support dragging).

The second feature allows developers to use a rule for finding further interaction examples in the current scene, according to a similarity measure provided by XRSpotlights's underlying model to sort and filter interactions in complex scenes. This enables interaction-oriented navigation of the objects in the scene and supports finding the available variants of a relevant interaction and selecting the one that best suits the developer's needs. XRSpotlight facilitates the inspection of these variants by grouping rules by phase (i.e., when the interaction happens), modality (i.e., interaction technique) or action (rule's consequence), which is semantically richer than existing inspection tools.

The third feature supports transferring the interaction definition from an example to a target game object, mimicking the well-known copy-paste function. Behind the scenes, XRSpotlight uses its model of toolkit implementations to identify all the relevant elements (i.e., components, events, and listeners) and recreates them in the target object. The underlying model allows transferring interactive behaviors not only within the same scene, but also between two scenes in different

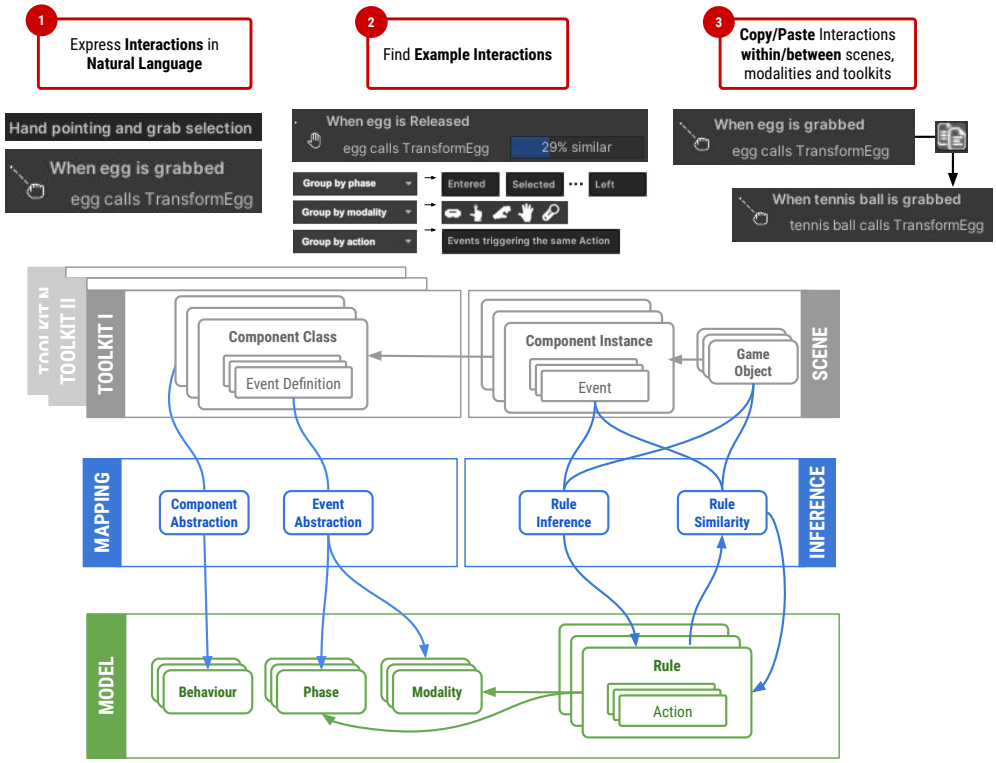


Fig. 5. Using an abstraction of existing toolkit implementations in our model and a mapping from each toolkit to the model, XRSpotlight can express interactions in a toolkit-agnostic way using natural language, find similar interactions for a given example, and copy-paste interactions between scenes to support the use and adaptation of examples provided by different toolkits. At the technical level, toolkit capabilities are captured in terms of behaviours, phases, and modalities, while the interactions defined in a scene are represented as trigger-action rules.

Unity projects. This case is particularly relevant for novices who often try to replicate interactions shipped in the toolkit sample scenes. Finally, the model also supports transferring examples between different toolkits, finding the equivalent elements in the target toolkit.

5 TOOLKIT ANALYSIS AND MODELLING

For deriving the toolkit-agnostic model representing XR interactions, we proceeded to an analysis of the existing toolkits, which informed the three main traits of the resulting model: *(1)* the concepts of interaction phases and modalities, *(2)* the distinction between interactions available and exploited, *(3)* the modelling of rule triggers and their effects. Considering the large number of available XR toolkits (see Table 1), we selected two representatives for proceeding with the in-depth analysis. We eventually chose the *Mixed Reality Toolkit* (MRTK) [35] and *SteamVR* [44]. They are among the most popular toolkits in XR development, provide high-quality documentation (including example experiences), and have an active developer community. They support basically the same interactions, but the differences in how they implement them make the toolkits relevant to our analysis.

Toolkits overview. MRTK started out as an AR toolkit on HoloLens V1 but has since grown into a versatile XR toolkit that unifies the way interactions are supported and implemented on different AR and VR devices. For example, on HoloLens V1, it limits the interaction model to gaze pointing and air-tap selection, while for V2 it supports both near (hand pointing and touch selection) and far manipulation (hand pointing and grab selection). On VR headsets, it supports both motion controllers and hand tracking, if available. Such complexity in the interaction devices and techniques translates into a large set of components, many of which are adaptable for more than one interaction modality. Some of them may be configured for obtaining different and custom interactions, while others provide support to common interactions. It contains also a set of widgets for creating UIs panels.

SteamVR is a popular toolkit focused on VR experiences supporting all the major VR headsets and the corresponding motion controllers. SteamVR offers a small set of components focused on VR interaction. It does not compose them into a widget library. It contains a generic component marking interactive objects in a scene and supports more complex interactions through the composition of additional components. For instance, for dragging and throwing an object the toolkit provides a component for the former and the latter interaction, but there are no predefined interaction sets.

Method. We analysed the main example scene provided with each considered toolkit to understand the toolkit structure and interaction definition better. Such example scenes have a reasonable complexity and aim to cover most of the toolkits' features, making them highly relevant for our analysis. For MRTK, we selected the `HandInteractionExamples` [34]. The scene contains simple 3D objects like polyhedra, mugs etc. demonstrating manipulations in different modalities, virtual objects representing keys or buttons for demonstrating press interactions and panels or widgets for demonstrating the interactions with UI elements. For SteamVR, we focused on the one showcasing interactions with different categories of objects [52]: UI elements, throwable objects, sliders and handles-based manipulation, and virtual remotes. We manually identified and counted the toolkit components associated with the objects in the scene, listing the supported interactions (e.g., focus, grab, hold, press etc.), the interaction technique (e.g., gaze pointing, near or far hand interaction, laser pointing etc.), and how it was defined at the source code level (e.g., events used, listener registration, coding the effects). We used such data for identifying differences and similarities between the toolkit and as ground truth for validating our modelling (see Section 7).

Similarities. We identified the following common traits between the two toolkits:

(1) *Components and Events*. Both toolkits comply with the entity-component-system structure used in all game engines. They consist of a set of components raising events for notifying the interaction sequencing.
(2) *Event Ordering*. Both toolkits define interactions through an explicit ordering among events, splitting them into an ordered sequence. For instance, the manipulation interaction consists of a manipulation start event, followed by a certain number of manipulation changes and then a manipulation end.
(3) *Custom configuration*. Both toolkits include highly customizable components, able to express quite different interactions. They require in-depth analysis to establish which interaction they support on the current object.
(4) *Interaction effects*. In both samples, we noticed that dedicated custom components define the interaction effects, for easing its replication across different game objects.

Differences. The following are the main points where the two toolkits differ:

(1) *Interaction Modalities*. MRTK supports gaze pointing and near and far interaction with both hands or remote controllers. SteamVR instead supports only remotes for near and far interaction.

(2) *Predefined interactions.* MRTK includes many components specifying pre-defined interactions, such as hover, click, push, hold, manipulation etc. In SteamVR, it is possible to support all these interactions, but the developer needs to configure the game object through different components.
(3) *UI widgets.* MRTK contains components and widget objects for creating UIs in the XR experience. SteamVR supports the interaction with objects representing UI elements, but it does not provide any widgets.
(4) *Complexity.* Supporting configurable and pre-defined interaction components, different interaction modalities and UI widgets makes MRTK a complex toolkit aiming at offering out-of-the-box solutions to developers. This results in components defining overlapping interactions (i.e., we have different solutions for creating the same interaction). SteamVR follows a different approach, offering lightweight components supporting simple interactions. Developers obtain more complex ones by composition.

Summary. MRTK and SteamVR follow two different engineering approaches: MRTK cover multiple modalities and provides dedicated components for specific interactions, while SteamVR includes a small set of components but highly customizable. Both approaches rely on event ordering for correctly specifying interactions: dedicated components have dedicated events, customizable ones have polymorphic events. The behaviour of the objects is defined either through event handlers or through specific components (e.g., rigid body, colliders etc.).

5.1 Modelling Toolkits and Interactions in the Scene in XRSpotlight

In our model (the ***Model*** green box in Figure 5), we represent a ***Toolkit*** supporting XR interactions as a set of ***Components*** (similarity 1) that developers attach to interactable objects. The modelling abstraction XRSpotlight uses for representing interactions in XR experiences enables a consistent representation of interactions across different XR devices and toolkits (see Figure 5). We started by leveraging the toolkit similarities to build its foundation and we followed a hybrid approach for modelling different interaction aspects [42]. Each component provides a set of ***Events***, notifying interactions involving the considered object. A toolkit *Component* associated with an XR object provides a set of interactive ***Behaviours*** (e.g., focus, grab, hold, press, rotate, scale, translate, manipulation, hover etc.). Usually, developers exploit a subset of these behaviours to create the interactions (difference 2). The analysis of the two reference toolkits also highlighted differences that challenge the modelling. The first is providing a representation of the event sequencing (similarity 2), which should be understandable for novices without further documentation, but consistent across all the interactions (difference 4). We propose using an abstract point-and-select sequence independent from the pointing device. Inspired by Buxton's model of graphical input [7], we defined the ***Phases*** as the events related to the change of state in such an abstract interaction, which a pointed object should hypothetically raise. The following is the list of the identified *Phases*. We do not use them all in some interactions, but they can consistently cover all sequences in the interactions we analysed in both toolkits. We modelled the phases following the *superset* approach in [42], by providing an abstraction that covers the entire spectrum of the analysed toolkits.

- **Enter**: The user is currently pointing at the object, without confirming the selection.
- **Select**: The user has confirmed the selection.
- **Move**: The user moves the pointer without releasing the selection (e.g., still pressing the selection button).
- **Release**: The user releases the selection.
- **Leave:** The user moves the pointer out of the object.

The second modelling challenge we encountered in creating the abstraction was specifying which type of controller the user is provided with for interacting with the environment (difference 1), without replicating the interaction definition for each device. We introduced the *Modality* concept as a concrete technique for performing the pointing, including a pointing and a selection technique. Currently, our tool identifies the following modalities (but the set may be increased in the future):

- **Gaze-pointing and air-tap selection** (e.g., gaze-and-commit interaction in MRTK)
- **Hand pointing and touch selection** (i.e., selecting by touching with the fingertip)
- **Hand pointing and grab selection** (i.e., selecting by grabbing an object)
- **Laser pointing with hand-gesture selection** (e.g., point-and-commit in MRTK)
- **Laser pointing with remote controller button selection** (i.e., using remotes for pointing and selecting)

To map toolkit components towards the model, we rely on a *toolkit mapping* file containing: 1) the *Behaviours* supported by the components in the toolkit ; 2) the *Phase* and 3) the *Modalities* associated with each interaction event in the toolkit components (see the blue box called Mapping in Figure 5). Such mapping allows managing the overlapping interaction definitions in MRTK and the simpler approach of SteamVR simply and consistently (difference 4). Each toolkit supported by XRSpotlight requires the definition of a mapping file. The effort required for defining and maintaining it is similar to providing metadata required for customizing the appearance of a component in the Unity Inspector (i.e., inserting custom attributes on variables and methods), and it depends on the toolkit's complexity.

We rely on ***Rules*** (Model box in Figure 5) for providing a simplified representation of interactions provided by an XR object. Rules follow the Trigger-Action paradigm [51], including the trigger and a set of *Actions*. We selected such a paradigm given its effectiveness in no-code programming tools [2, 4, 8, 11, 51]. In our model, a trigger is an instance of an *Event* defining an interaction in a toolkit component. The trigger is the event that fires a rule. We know which events define an interaction from the mapping file described in Section 5.1. From a modelling point of view, a trigger is the concrete implementation of a *Phase* (i.e., an abstract event) in a given *Modality*. Therefore, we may identify a trigger through a pair $\{p, m\}$ $p \in Phase, m \in Modality$ (similarity 3). The arrows going from the *Rule* element to the *Phase* and *Modality* represent the trigger in a rule. In the rule visualization panel, we assign specific names to represent each trigger since the name of a phase is quite abstract. For instance, we call *drag* the move phase in the modality hand pointing and touch selection. An ***Action*** corresponds to invoking a method provided by a component on a game object as a reaction to the rule trigger (Similarity 4), which is the lowest common denominator in defining reactions to toolkits events [42]. When the interface supports different interactions to achieve the same effect, the tool produces a different rule for each trigger, with the same action as a consequence. A specific grouping technique allows the identification of multiple triggers resulting in the same action. We decided to avoid introducing UI widgets in our model since it already grasps their interactive part, while the specific appearance and behaviour are not in the scope of this work (difference 3). Overall, the

6 XRSPOTLIGHT IMPLEMENTATION

To demonstrate the effectiveness and validity of the modelling for supporting the features we designed, we implemented them using Unity. After we have seen them in action in the walkthrough (Section 3), we show in this section how they work in the general case. This section follows the ordering of the features in the overview (Section 4). The current implementation consists of a Unity

Editor Panel extension in C# at a research prototype stage, aiming at demonstrating the technical feasibility and the advantages of XRSpotlight. The source code is publicly available on GitHub[1].

6.1 Rule Inference Algorithm

The rule inference algorithm generates the rule-based description using a constrained natural language representation of the interactions on an XR scene. It requires as input a Unity scene and a toolkit mapping file, containing a priori knowledge for mapping the considered toolkit towards our abstract model (currently available for MRTK and SteamVR). It starts by identifying the objects that support interactions by listing those currently contained in the scene. For each of them, it exploits the Unity API for listing its components, searching for instances of those included in the toolkit mapping. If it contains a mapping for at least one of these components, the algorithm marks the object as interactable. For each interactable object, the algorithm lists the *Behaviours* according to the knowledge about the component in the toolkit mapping (e.g., the object is throwable, grabbable etc.). As we already explained in Section 5.1, these are the available interactions. Some of them may be exploited and have associated rules, others may not. For creating the *Rules*, the algorithm loops on the events defined by the interactable components. When it finds one event having one or more attached listeners, it retrieves its *Phase* and *Modalities* from the toolkit mapping for creating the rule trigger (or triggers if the event supports multiple modalities). The action part consists of the listeners attached to the event. Each listener contains the information for invoking a method on a Unity object, including an optional parameter. We generate natural language sentences using a constrained grammar, by composing predefined descriptions for triggers and actions into the rule structure.

For showing an example of the inference procedure, we consider the rule *When the egg is hovered, then the egg calls showIcon*, listed among those in Figure 6. The algorithm recognises the *egg* as an interactable object since it has the `ObjectManipulator` among its components, which is included in the MRTK [35] toolkit mapping file as shown in Table 2. Among the component's events, the file maps the `OnHoverEntered` event to the *enter* phase for all the supported modalities. Since the *egg* object contains a handler for such an event, the algorithm infers one trigger for each modality. Our example relates to the modality of the *laser pointing and hand-gesture selection* and the *enter* phase. The corresponding action describes the Unity Action associated with the event handler, which invokes the `showIcon` method for providing feedback on the ingredient selection.

XRSpotlight provides different views on the initial rule set inferred by this simple procedure. The first is the *Phase* strategy, which creates a group for each phase, by combining their actions, as shown in Figure 6-A. The technique creates a compact representation by aggregating multiple triggers (i.e., phase-modality pair) in a single rule. It shows the differences in the phase handling among the modalities. The disadvantage is the difficulty in relating the changes in the Unity Inspector and the resulting multimodal rules. Indeed, the rule obtained through the grouping combines multiple toolkit events, having the same effect but raised by different interaction techniques.

The second strategy groups the rules by *Modality*. We insert each rule in a separate group for each modality, duplicating those having more than one (see Figure 6-B). This view provides a direct relationship between a rule and the correspondent definition in the Unity Inspector since each rule has exactly one trigger. However, it could create a longer list of rules applying to more than one modality. The last strategy groups the rules by *Action*, see Figure 6-C. If the same action appears in more than one rule, we combine the phases and the modalities in the final rule. The visualization has the purpose of supporting the developer in finding out how many events have the same listener, which is useful for isolating a given dynamic behaviour. It does not support creating a general

[1]https://github.com/cg3hci/XRSpotlight

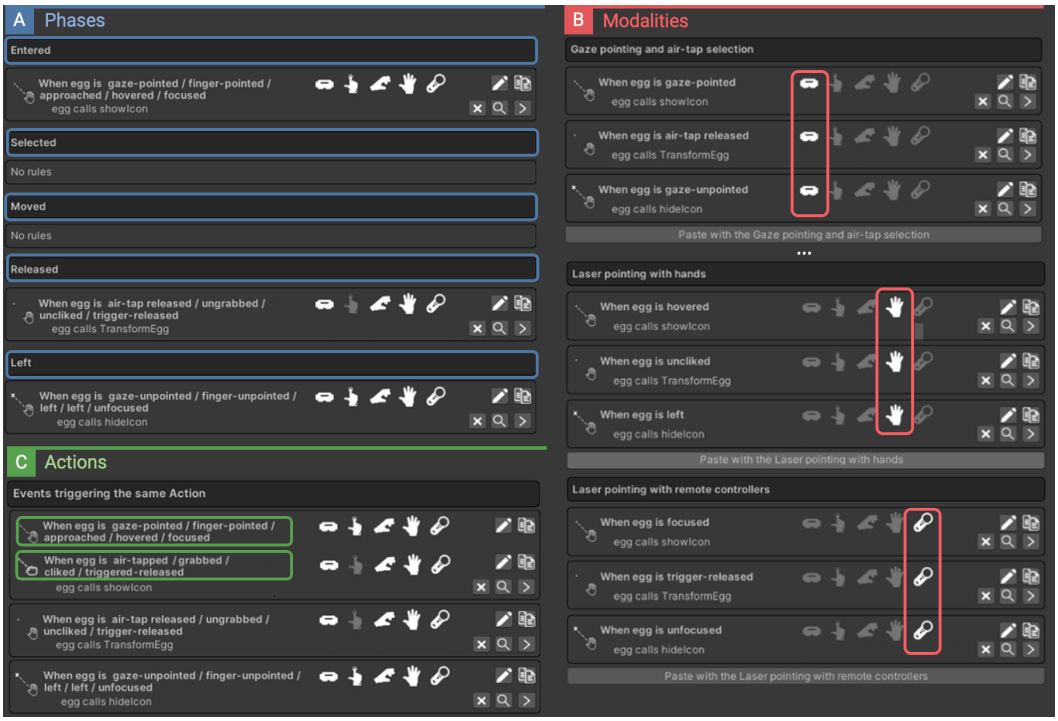


Fig. 6. Different rule groupings for representing the same interaction (the egg hovering feedback described in Section 3). In part A, rules are grouped by phases (blue boxes), showing multimodal reactions to three phases: enter (pointing started), release (the selection disengaged) and left (pointing ended). In part B, rules are grouped by modality (red boxes). Rules working in multiple modalities are reported in each corresponding group, using the specific trigger name. In part C, rules are grouped by action, highlighting the events sharing the same effect (green boxes).

understanding of the entire interaction with the object, but it aims at highlighting the actions having multiple triggers (i.e., different interactions having the same consequences).

6.2 Rule Similarity Algorithm

Mapping the interactions towards the abstract modelling elements allows for defining a similarity relationship, considering which ones are different and establishing a distance function among them. Since a rule consists of a trigger and a set of actions, we define the similarity function considering both elements. If we compare the triggers of two rules r and s, we can distinguish whether they are defined on the same XR object or not ($T^o_{r,s} \in \{0,1\}$), whether they have the same *Phase* or not ($T^p_{r,s} \in \{0,1\}$), the number of modalities which are not in common between the two triggers ($0 \leq T^m_{r,s} \leq 5$). Instead, if we compare two actions α, β, we can check whether they are defined on the same XR object or not ($A^o_{\alpha,\beta} \in \{0,1\}$), whether they invoke the same method of the same component ($A^m_{\alpha,\beta} \in \{0,1\}$), whether the method has the same parameter or not (if any) ($A^p_{\alpha,\beta} \in \{0,1\}$).

$$
\begin{aligned}
t(r,s) &= w_1 T^o_{r,s} + w_2 T^p_{r,s} + w_3 T^m_{r,s} \\
a(r,s) &= \sum_{\alpha_i \in Act_r} \min_{\beta_j \in Act_s} \left(w_4 A^o_{\alpha_i,\beta_j} + w_5 A^m_{\alpha_i,\beta_j} + w_6 A^p_{\alpha_i,\beta_j} \right) \\
d(r,s) &= w_t t(r,s) + w_a a(r,s)
\end{aligned}
\tag{1}
$$

Table 2. A JSON excerpt from the MRTK [35] toolkit configuration file. It contains a list of the components making an XR object interactable. We show the entry for the `ObjectManipulator`, which defines a Unity event in the `OnHoverEntered` property field. Such event corresponds to the *enter* phase, and it supports the five modalities (pointing and selection technique): i) gaze and air tap, ii) hand pointing and touch, iii) hand pointing and grab, iv) laser pointing and hand, v) laser pointing and button. Further events have similar entries in the `event` array, while further JSON objects in the root array describe other interaction components.

```
[  ...,
    {
        "className": "Microsoft.MixedReality.Toolkit.UI.ObjectManipulator",
        "isComponent": true,
        "events": [
          ...
          {
            "reference": [{
              "member": "property",
              "name": "OnHoverEntered"
            }],
            "phase": "enter",
            "modality":  ["gaze", "hand-touch", "hand-grab",
                          "laser-hand", "laser-button"]
          },
          ... ]
    },
]
```

We obtain the final distance value as defined in Equation 1 by weighting the different values we get from the trigger comparison ($t(r,s)$) and aggregating those obtained by the comparison of the actions (Act) belonging to the two rules[2] Such a definition allows limiting the comparison on triggers or actions, which is useful for the copy-paste implementation.

6.3 Copy-Paste Implementation

When a rule is copied, XRSpotlight serialises its definition in a file located in a fixed path, to allow data sharing among objects in different Unity projects. Figure 7 shows the algorithm for pasting the copied, highlighting the different paths XRSpotlight follows when the source and the destination scene are the same (green), different scenes using the same toolkit (blue) or different scenes using different toolkits (red). For pasting the copied rule, the tool searches for the trigger having the highest similarity score (i.e., the minimum distance) in the components provided by the destination XR toolkit, by using the $t(r,s)$ function in Equation 1. In the case of many triggers having the maximum score, the selection process prefers components that the object already contains. Otherwise, it attaches the selected component to the destination. If the tool does not identify any trigger having a similarity higher than zero, the paste fails, and the tool shows an explanation message.

Next, for each action in the source rule, XRSpotlight tries to create an exact copy in the destination rule. This is possible only if the object involved in the source action is also defined in the destination

[2]we use $w_{1...6} = 1$ and $w_a = w_b = 0.5$

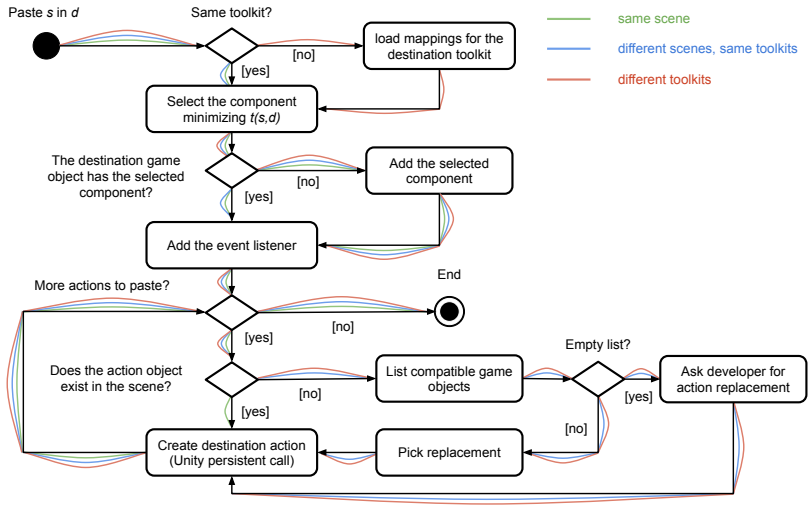


Fig. 7. How to paste a previously copied rule s on a destination d depends on the target scene. In the same scene (green), the trigger uses the same source component and event. The actions do not need to find compatible objects since they are available in the scene. Considering two different scenes (blue) using the same toolkit differs for the action replication, requiring that the involved scripts are available in the target scene, and the developer selects among the compatible objects. Otherwise, the actions require replacement. Finally, the paste between two different toolkits (red) requires establishing a correspondence between the two toolkits through the model. The action replication logic is similar to the previous case. We provide a concrete copy-paste examples in Section 3.

scene, i.e., when the copy and the paste happen in the same scene.[3] Otherwise, the tool shows a list of game objects that may replace the one in the source action since they support the same method. If no object in the scene provides the same method defined in the source action, it is not possible to recreate the action in the destination scene. In this case, the list would be empty and the tool suggests the developer where s/he can add a custom replacement (if needed).

Another version of the paste feature is available when rules are grouped by modality, to support replicating the same interaction using different techniques or devices. In this case, we limit the trigger similarity search only on the components in the destination toolkit that support the selected modality. We use the copy-paste procedure also to move the effects of a rule towards another trigger, eventually deleting the original rule.

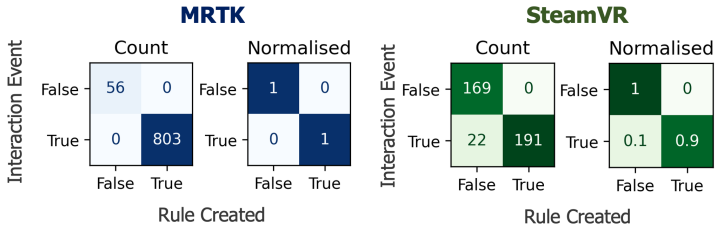
7 TOOLKIT MODEL VALIDATION

We inspected the main example scenes of MRTK [34] and SteamVR[52] using XRSpotlight for validating the underlying model by assessing the accuracy and correctness of the interaction representation.

Method. We proceeded with the assessment in two steps. First, we checked whether XRSpotlight identifies the events defining interactions on XR objects. We consider it as a binary classification problem: given an XR object event, XRSpotlight establishes whether it defines an interaction or not. If the algorithm described in Section 6.1 generates a rule then the event is considered interactive, otherwise it is not. We obtained the ground-truth through manual inspection and labelling of the

[3]In general, the game object involved in an action may be different from the game object where the rule is defined. For instance, in rule `when elevator_button is clicked then the elevator_door calls open`, the rule is defined on the `button_elevator`, while the action involves `elevator_door`.

Table 3. XRSpotlight performance in identifying interactive events in the MRTK and SteamVR sample scenes. Step 1 considers the binary classification problem of whether a given event associated with an XR object represents an interaction. True positives are interaction events correctly identified (rule created), true negatives are events not related to the interaction not generating a rule, true negatives are interaction events not identified, false positives are events not related to the interaction generating a rule. We do not consider events having no associated handler. Step 2 instead reports on the correctness of the rules generated from the identified events (i.e., true positives in Step 1), reporting the count of the correct rules and rate.



Step 1: Identifying Interaction Events	Accuracy	Precision	Recall	F-Score
MRTK	1	1	1	1
SteamVR	0.94	1	0.90	0.94

Step 2: Rule Correctness	MRTK	%	SteamVR	%
Correct Rules (correct trigger and actions)	683	85%	191	100%
Wrong Rules (wrong trigger and actions)	0	0%	0	0%
Wrong Rules (wrong trigger, correct actions)	120	15%	0	0%
Wrong Rules (correct trigger, wrong actions)	0	0%	0	0%

two example scenes. The second step consists in assessing the correctness of the rule identified by XRSpotlight. We consider a rule as correct if it defines the correct trigger and the correct actions, while it is wrong if we have an error in either part. We again used the data we produced through manual inspection as ground truth.

Results. As shown in Table 3, the identification was fully correct with MRTK, while there were 22 interactions not identified in SteamVR. XRSpotlight missed their definitions because it cannot find interactions defined through scripts in code, as this information is not directly exposed in the Unity Inspector. They are explicitly designed to show how developers can customize the Interactable component beyond its standard support, going down into its abstraction. This can be regarded a technical limitation. It has less of a measure of success for our modelling approach since the two ways of defining interactions are equivalent. We would require a code analysis module for finding the ones missed with SteamVR.

On the other hand, the rules identified in SteamVR are all correct. We have lower performance for MRTK, where rules are correct in 85% of cases. The errors are all related to the handling of two components. The first is the `BoundsControl`, which attaches sub-objects representing manipulation handles for resizing and rotating that are not represented in the rule trigger. The second is the `InteractableOnPressReceiver` that has a `InteractionFilter` field which allows filtering the modality between hand pointing with touch selection and laser pointing with gesture selection or both. The generated rules are correct only in case both modalities are allowed.

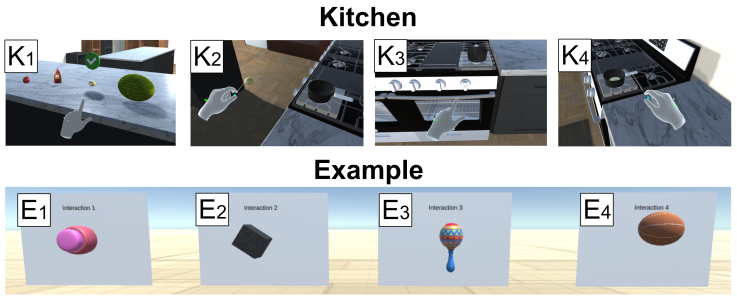


Fig. 8. Overview of both experts' and novices' studies. The tasks inside the kitchen scene are represented in the first row (K1-K4). The second row shows the Example scene containing four example interactions: A button that pops up red particles when clicked (E1), a cube that rotates when focused (E2), a maraca that plays a sound when manipulated (E3), a basketball affected by gravity when grabbed (E4).

Table 4. Summary of the two studies organization. The tasks refer to Figure 8

Study	Solution Type	Tool	Task 1	Task 2	Task 3	Task 4
Expert	Mental Model	None	K1	K2	K3	K4
	Inspection	XRSpotlight, Inspector	K1	K2	K3	K4
			System walkthrough			
Novice	Inspection	XRSpotlight, Inspector	E1	E3	E2	E4
			Comparison			
	Implementation	XRSpotlight Inspector	K1	K3		
			Comparison			
	Mental Model	None	K4			

8 USER EVALUATION

We conducted two user studies, one with expert developers and one with novices with two main goals: *(1)* verifying that the toolkit abstraction aligns with experts' mental models of how interactions should be implemented (calibration), and *(2)* assessing the benefits and shortcomings for novice developers in inspecting and creating XR interactions using our system (validation). After assessing the correctness of the representation with experts, we ensure that its level of complexity is adequate for novices.

Both studies were conducted remotely using Zoom video conferencing. XRSpotlight was presented from the laptop of the moderator, passing screen control to participants during tasks. The two studies lasted one hour, and each participant was compensated with an Amazon gift card for their time ($30 USD for experts and $25 USD for novices).

8.1 Calibration with XR Development Experts

In the calibration with XR experts, we wanted to observe how experienced Unity developers approach the implementation of interactions using a familiar XR toolkit with the goal of establishing their mental model (i.e., the steps they envision for implementing the interaction). Then, we used this mental model to identify opportunities for improvement, both in terms of how we conceptualized interactions in our model and how we facilitated implementation through our XRSpotlight system.

We defined expert developers to have at least two years of experience in professional XR development. We recruited six participants in our collaboration network via email, three with expertise in MRTK and three with expertise in SteamVR. All participants were male, and their age ranged between 26 and 33 years old. They had at least four years of expertise in XR development.

8.1.1 Method. The evaluation was structured into three parts (see Table 4).

Mental Model. In the first part, participants were asked to articulate their mental model about completing four VR implementation tasks conforming to four steps in a recipe to cook fried eggs. As a starting point, we provided a Unity scene called Kitchen, furnished with kitchen-style 3D assets and with their preferred toolkit (MRTK or SteamVR) already imported. Each of the four tasks consisted of implementing an XR interaction (see Figure 8, kitchen row): *(1)* pointing at the egg shows a 3D icon to indicate it is the correct ingredient, *(2)* moving the egg closer to the pan cracks the egg, *(3)* switching the stove button produces the sound of a frying egg, and *(4)* dragging the pan reduces the smoke. Participants were asked to think aloud and edit the Kitchen scene, providing some stub implementation for facilitating the articulation of their mental model. They were also given access to the toolkit documentation and Google. At the end of each task, we asked them about their confidence in their solution on a 7-point Likert scale. Since we recruited experts, the overall goal was not to judge the mental models of the experts. Rather, we wanted to see to what extent the mental models matched the modelling abstraction used in XRSpotlight and to identify commonalities that may later help novices form a correct solution to these tasks.

Inspection. In the second part, participants inspected the scene we prepared with our working implementation as solutions to the previous tasks, using the Unity Inspector window or XRSpotlight (in both cases the provided solution is the same). For each task, we asked them to inspect the scene and to explain 1) how they think our solution works and 2) the difference with respect to their solution. We used the first explanation for ensuring the correct understanding of the solution, in particular, while using XRSpotlight. Instead, listing the differences was useful for establishing the conceptual gaps between a solution supported by XRSpotlight and the one they proposed. Half of the participants inspected the first two interactions using only the Unity Inspector and the remaining two using XRSpotlight. The other half used XRSpotlight first and then the Unity Inspector, to reduce order effects. We kept the order of the tasks. We applied a direct observation strategy [26] since we were interested in collecting qualitative feedback on how they would approach the inspection task. At the end of the inspection session, we asked them to list two advantages and two limitations of solving the tasks with the Unity Inspector alone as compared to using XRSpotlight complementing it.

Walkthrough. The last part consisted of a walkthrough demonstration [26] of XRSpotlight's main features to critique them from the experts' perspective. We pre-recorded videos demonstrating three key features: "finding similar interactions", "copy-and-paste within the same toolkit" and "copy-and-paste across different toolkits." For each feature, we asked them if and how they would include it in their professional workflow and why.

8.1.2 Feedback. **The toolkit abstraction aligns with the expert's mental model**. All experts described correctly the solutions we proposed for the tasks in Figure 8 (kitchen row), with both the Inspector and XRSpotlight. We can summarise the differences the experts identified between their mental model and the proposed solutions into three categories: 1) use of the collider and the rigid body, 2) selection of a different event for triggering the interaction and 3) adding further checks in the triggered method for fine-tuning the effect. All the identified differences would result in a solution XRSpotlight can represent, and the expert did not highlight any gap in the abstraction concepts. Instead, they considered the complexity of the toolkit modelling to be at the right level of abstraction for novices (P2: "*the categorization about move, select, enter [...] are more meaningful to me, but also for novices*").

XRSpotlight provides a useful interaction overview. The experts noted the main advantage of XRSpotlight over the Unity Inspector is its centralized representation of interactions, which allows designers to have a useful overview of the interactive capabilities in the current XR experience. P1

appreciated that "*everything about interactions [is] gathered in one place*", while P5 added that it "*shows how the objects interact in the scene*". In addition, participants also appreciated the balanced display of information, which they considered sufficient for understanding the interaction but not overwhelming. P2 commented that "*in the Inspector I have a lot of Unity properties that I don't care about. I care more about triggers and interactions. In [XRSpotlight] it's very clear to me.*"

Need for fine-grained interaction descriptions for more advanced developers. While the experts expressed that XRSpotlight provides an ideal level of abstraction for novice developers, they identified a need for more fine-grained controls to achieve more complex or robust interactions. For example, many experts (P2, P3, P4, P6) wanted to implement constraints on interactions, (e.g., by using colliders to ensure the pan is dragged to a minimum distance away from the stove), which the current XRSpotlight implementation cannot identify, since requires a code analysis in addition to scene inspection, but they can be expressed in event-condition-action rules [2].

Finding similar interactions for debugging purposes. Regarding the feedback we received in the walkthrough demonstration, the feature for finding similar interactions (see Figure 3) was considered useful since Unity scenes are usually full of objects and interactions, and there is a lack of targeted locating technique. P2 and P4 suggested that they would use this feature for debugging purposes, seeing if they have forgotten something or checking the implementation. This was a very interesting point since we designed the feature for supporting exploration and not debugging.

Copy-paste increases the development efficiency. All the experts agreed that the copy-and-paste feature inside the same toolkit could be useful in their workflow. Some interesting observation came from P2: "*I can for instance copy from an object that has the interaction that I want without recreating it completely and I can be sure that the interaction weights are the same.*", and P3: "*I feel like this is a nice little kind of quality of life feature. I'm sure that Unity doesn't have a way to do that by default.*" The participants recognized the copy and paste between toolkit feature as helpful in two contexts: migrating to another toolkit and for novice users that want to understand the mechanism inside the scene. However, P6 thinks that "*this is really cool from a technical standpoint, I appreciate it, but I'm not sure about the use cases in a real workflow because if I'm working on a project that has SteamVR and then I have used MRTK before, I feel that there are so many differences in what I'm doing [... and it] would probably require more work than redoing it. I can see some cases where it could be useful but not in my real workflow.*". We agree with this comment since we designed the feature for novice developers, who may struggle in identifying the needed elements or even which toolkit they should use [3], and not for experts that may be fluent in more than one toolkit.

8.2 Study with Novice XR Developers

After the calibration with the experts, we evaluated XRSpotlight with novice XR developers, i.e., people having familiarity with programming languages but little experience in building XR environments and using game engines. The study has two different goals: i) understanding the benefits and limitations of using XRSpotlight compared to traditional means (using only the Unity Inspector) and ii) seeing to what extent novice developers shape a correct mental model so that they could correctly approach the development of XR interactions. The study group comprised eight participants, seven of them had a bachelor's degree, and one had a master's degree. Participants' ages ranged from 22 to 30; six were male, and two were female. All participants had at least one year of experience in XR programming, but none self-reported being experts. We recruited them by contacting students of XR development courses in our Universities.

8.2.1 Method. We first explained to the participants the goal of the study, the information we were going to collect and we asked them to sign the informed consent. After that, the study consisted

Table 5. Correctness criteria per task type.

Solution Type	Correct	Partially Correct	Incorrect
Mental Model	Answer contains a strategy for adding a correct interaction component *and* triggering the right event and effect.	Answer contains either a wrong interaction component or trigger.	Answer contains both a wrong interaction component and trigger.
Inspection	Answer contains the correct type of interaction and effect.	Answer is wrongly describes an interaction or effect.	Answer is missing an interactions or effect.
Implementation	Answer contains all the correct elements (i.e., collider, script, interaction component and event)	Answer is missing the collider required for detecting the interaction.	Answer is missing a required interaction component or the event is not exact.

of three parts: the inspection, the implementation and the mental model assessment in a specific scenario (we provide a summary in Table 4).

Inspection. In the Inspection tasks, they were asked to inspect the Unity scene called "Example Scene" represented in Figure 8-Example. They had to inspect and describe (thinking aloud) the user's interactions and effects for 1) interactions E1 and E3 and 2) interactions E2 and E4. Half of the participants started using Unity Inspector, and the other half used XRSpotlight. We introduced our tool through a short video walkthrough of the system. After each task, we asked the participants to rate the difficulty level on a 7-point Likert scale. At the end of all the inspections, we asked for feedback on comparing the task completion under both conditions. Before moving on to the second part of the study, participants watched a video showing the interactions and the effects for each example in Figure 8-Example since they would need them as starting points for the next tasks.

Implementation. The second part consisted of implementing K1 and K3 (see Figure 8), the same as the experts' study. Participants could use the example scene to transfer the needed interactions, but unlike the experts, they were asked to provide a working implementation in Unity. Again, half of the participants used Unity Inspector first and XRSpotlight after, with a reversed order for the other half. We asked for the difficulty level on a 7-point Likert scale after each task, and we asked them to reflect on the advantages and limitations of the two conditions at the end of the implementation part.

Mental Model Assessment. In the final part, they had to describe how they would obtain K4 (see Figure 8), the last interaction in the experts' study (dragging the pan), by using the think-aloud method and with a particular focus on MRTK. Ultimately, we asked the participants how confident they were about their solution on a 7-point Likert scale.

We tracked the time on task and counted the number of accesses to elements in Unity, providing irrelevant information for the current task to assess possible efficiency differences between the two conditions. In addition, we evaluated the correctness level of the tasks applying the criteria in Table 5, depending on the type (inspection, implementation and mental model assessment).

8.2.2 Results. **Inspection**. Regarding the inspection tasks (E1, E3, E2, E4), we registered a slightly higher completion rate for XRSpotlight (13 to 12 correct, 1 to 1 partially correct and 2 to 3 incorrect). Figure 9 (top) shows that XRSpotlight performed better on E3, while the Inspector has an advantage on E2. We registered the same trend in the time on task. This slightly higher completion rate reflects on reporting the perceived difficulty since XRSpotlight has been found slightly easier to use (mean, median and mode for the two conditions are the following: $\bar{x}_{xrs} = 1.25$, $Md_{xrs} = Mo_{xrs} = 1$ vs $\bar{x}_{ins} = Md_{ins} = Mo_{ins} = 2$). A clear advantage provided by XRSpotlight in this type of task is finding interaction-relevant information mostly at the first attempt. The Inspector instead requested one more attempt on average, especially in the first two tasks (E1 and E3).

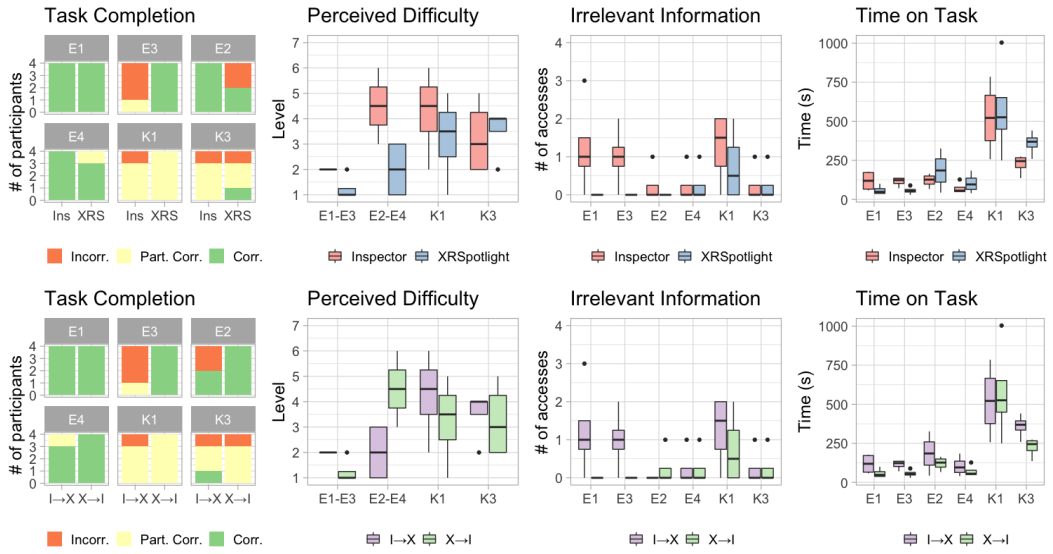


Fig. 9. Summary of the data collected during the novice study, including the correctness of the proposed solution to the task (see Table 5), the perceived difficulty, the number of accesses to irrelevant information during the task, and the time on task. We report the values per experimental condition (Inspector and XRSpotlight, top row) and per group defined by the condition ordering (bottom row): Inspector - XRSpotlight (I→X) or XRSpotlight - Inspector (X→I).

Figure 9 (bottom) provides an interesting perspective on the same data. It splits participants by starting condition, i.e., grouping those who started with the Inspector first and then used XRSpotlight (I→X) and those who followed the inverse order (X→I). In that case, the results show that participants starting from XRSpotlight completed all inspection tasks correctly and spent less time in the inspection. The perceived difficulty was lower when using XRSpotlight (E1 and E3), while it was higher for tasks completed through the Inspector (E2 and E4). We can provide a consistent explanation for these trends, which is confirmed by participants' feedback and observation (see Section 8.2.3): *XRSpotlight favours the transition from a guided to an autonomous inspection.* Indeed, after solving the first two inspection tasks through our tool, participants acquired the ability to identify the relevant elements in the Inspector for describing the interaction, and they made no further mistakes with the Inspector, completing the tasks in less time.

Implementation. Regarding task completion, using XRSpotlight, one participant completed the task correctly, one incorrectly (wrong event choice), and the rest partially correctly (forgetting about the collider). Using the Unity Inspector, two participants did the task incorrectly (wrong event choice), and six partially correctly (forgetting the collider). The perceived difficulty for the implementation tasks is comparable in both conditions. For XRSpotlight, part of it was caused by switching between two different views instead of using the same one from the beginning, which also explains the higher time on task for K3.

Mental Model Assessment. In the third part, almost all of the participants (7 out of 8) discussed a correct mental model for completing the presented scenario. The confidence level on the solution was high ($\bar{x} = 5.125$, $Md = 5$, $Mo = 5$). It is encouraging to notice that they could work on the same task as the experts and be almost successful as them (the mental model assessment task was the same across the two studies).

8.2.3 Feedback. **Simplicity**. The inspection tasks highlighted that participants appreciated the simplicity and the reduction of the viscosity [41] supported by XRSpotlight. Four participants out of eight affirmed that it was easy to understand the rules with XRSpotlight, even for non-experts (P2, P4, P6, P8). In addition, P7 said that "*you can easily understand what to do in a Unity scene*". They motivated this explicitly by referring to the list of the interactable objects and that "*the user can better focus on the relationship between the type of interaction and then the effect it's clear on the screen.*" (P3).

Transition out of the tool. The completion rate trend discussed in Section 8.2.2 shows that XRSpotlight helps transition from the rule-based simplified abstraction to a proficient understanding of the example organization in the Unity Inspector. P8 relied on the field identified through the XRSpotlight highlighting feature for inspecting the next examples using the Inspector. P5 mentioned that "*[...] it's easier to manage the rules with the XRSpotlight window and then set more parameters in the Inspector*". Commenting on the implementation tasks, three participants justified their confidence rating by saying that they have just understood how to do this interaction recalling the events and components to use in MRTK but depicting the solution in terms of the proposed rule-based description. This suggests that applying such representation for learning purposes may be a relevant future direction of the project, which we will cover in future works.. For instance, P7 stated that "*I should create a rule and . . .*", while P3 "*I would [select the correct game object] and add the selection interaction as the effect of shaking the pan*".

Copy-paste is the most useful feature. In the second part of the study (implementation tasks), seven participants out of eight reported the copy-and-paste feature as the main advantage of the system. P7 said that "*it's easier to copy rules to have them immediately, using the Inspector you have to create them*". All the participants used this feature while doing the task with XRSpotlight, even if two participants said that they would have preferred to create a rule from scratch through XRSpotlight. Most of the participants have chosen the source rule having the most similar type of interaction. Other participants preferred the effect similarity: they started from a rule having a similar effect compared to the one they wanted to create. In this case, some of them used the feature for changing the trigger to adapt it.

Novices want to edit rules in XRSpotlight. To obtain the correct rule when using XRSpotlight every participant had to use the Unity Inspector window to adapt the pasted rule according to their needs. As we expected, six participants stated that they would have liked to edit the effect of a rule directly from XRSpotlight, without changing their focus to the Inspector. This is clearly a fair limitation, considering that we took the explicit design decision of not making XRSpotlight a replacement for the Unity Inspector, avoiding duplicating its features.

Usability and missing features. We could distinguish the identified limitations into two categories: i) interface usability issues and ii) missing features. In the first category we have the vocabulary used for the triggers that were confusing for some participants, the phases without rules were considered unclear and P1 did not grasp the relationship between the action label and the method's name. We may address all these problems including further information (e.g., help) in the XRSpotlight panel. Among the missing features category, two participants requested to have more settings related to colliders and interaction components. Indeed, in E2, two participants misunderstood the effect of the cube interaction using XRSpotlight.

9 DISCUSSION

Simplifying the example inspection. In the complex landscape of existing toolkits for creating XR interactions [40], we introduced a tool empowering novice developers in example-driven development targeting the difficulties in selecting the right toolkit and example resources [3]. We consider as novices people with basic programming knowledge but without experience in XR or

game engines. We showed that the underlying abstraction captures the peculiarity of two different and widely-used toolkits, such as MRTK [35] and Steam VR [44]. Besides the technical validity of the modelling, we also demonstrate the abstraction matches the experts' mental model when planning the implementation of XR interactions (see the study with experts in Section 8.1). This has particular relevance for novices, whose target is acquiring and mastering the experts' mental model. Such abstraction eases the reasoning about how to create interactions provided by concepts explicitly designed for modelling such interactions without getting lost in the hurdles of the XR environment representation [20, 40], as shown in the novice study.

Transitioning out of XRSpotlight. The ceiling is one of the main problems affecting simplified XR development tools (see Section 2.1): when novices require going beyond the specific predefined tasks or platforms, they must face the complexity of professional game engines such as Unity [3]. XRSpotlight is the first tool proposing a simplified abstraction of XR interactions *inside* a professional game engine, aiming at guiding novices through its complexity rather than replacing it. In our view, linking the simplified view (XRSpotlight's rules) with the detailed definition in the Inspector allows for a guided transition *out* of its support. We found evidence of the desired effect in the higher completion rate of the group that started the novice evaluation using XRSpotlight.

Creating interactions through copy & paste. It is also relevant to analyse the differences in the opinions on the copy-paste feature between novices and experts. Such a feature provides semi-automatic support for transferring an example of a relevant interaction into the environment under development. When transferring from a toolkit documentation sample, the effect of the interaction usually needs adaptation by the developer, so the transferred information is the toolkit component and event the sample uses (i.e., the trigger). Such information is relevant for novices and less useful for experts, who can select a trigger without guidance. However, they found the feature useful in the case of repetitive interaction definitions in the same scene.

Limitations. There are some limitations in this work, which would require further research in the future. Avoiding the replication of the Unity Inspector editing features inside XRSpotlight was a design choice we carefully considered. The evaluation with novices highlighted that they would like to edit rules in the panel since switching between XRSpotlight and the Inspector requires effort. We think that such effort should not be completely removed, otherwise there will be no transition out of our abstraction and the novices will be locked into XRSpotlight. Further research is required to find the right balance between the editing effort and the transition support. The tool is implemented as a research prototype, and the evaluation highlighted some usability issues requiring some development work. Nevertheless, the provided features demonstrated that it eases novices in inspecting and the understardanding XR examples. The feedback and the performance in the novice study suggests that XRSpotlight may be useful for learning how to correctly implement XR interactions. However, more research is required for assessing its effectiveness and we plan to cover this aspect in future work. The rule representation may be improved by providing a summary of the actions' effects. This would require code analysis, but it would increase the novice's understanding. Through code analysis, we can also identify the interactions not listed in the Unity Inspector panel. Since the number of interactions defined only through code in real-world projects are relevant, this may result in an incomplete overview of the environment interaction capabilities. Finally, the tool requires extensions to further modalities, such as vocal or gestural interaction, and for supporting the identification of multimodal interactions.

10 CONCLUSION

XRSpotlight empowers novice developers in finding, navigating and understanding examples of relevant interactions for their development tasks. It helps them in transferring the examples in

their XR experiences. The underlying modelling and vocabulary support these tasks in a toolkit-agnostic way and ease the planning for creating interactions. In the future, we will investigate the effectiveness of XRSpotlight in describing more complex scenes, which contain interactions beyond examples. Since their definition focuses more on the resulting experience than on writing clean code for replication, we expect to find more complex structures. We are also interested in shifting the focus towards educational purposes, i.e., supporting people in learning how to create XR environments by gaining a simplified yet correct understanding of how XR interaction works. To this end, we would like also to include a non-computer science audience.

ACKNOWLEDGMENTS

We are grateful to the bachelor students who provided the kitchen environment modelling: Andrea Dore, Lorenzo Montesuelli and Daniele Pennisi.

REFERENCES

[1] Gleechi AB. 2022. *VirtualGrasp Documentation.* Retrieved April 25, 2023 from https://docs.virtualgrasp.com/index.0.15.0.html

[2] Valentino Artizzu, Gianmarco Cherchi, Davide Fara, Vittoria Frau, Riccardo Macis, Luca Pitzalis, Alessandro Tola, Ivan Blecic, and Lucio Davide Spano. 2022. Defining Configurable Virtual Reality Templates for End Users. *Proc. ACM Hum.-Comput. Interact.* 6, EICS, Article 163 (jun 2022), 35 pages. https://doi.org/10.1145/3534517

[3] Narges Ashtari, Andrea Bunt, Joanna McGrenere, Michael Nebeling, and Parmit K. Chilana. 2020. Creating Augmented and Virtual Reality Applications: Current Practices, Challenges, and Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376722

[4] Ivan Blečić, Sara Cuccu, Filippo Andrea Fanni, Vittoria Frau, Riccardo Macis, Valeria Saiu, Martina Senis, Lucio Davide Spano, and Alessandro Tola. 2021. First-Person Cinematographic Videogames: Game Model, Authoring Environment, and Potential for Creating Affection for Places. *J. Comput. Cult. Herit.* 14, 2, Article 18 (may 2021), 29 pages. https://doi.org/10.1145/3446977

[5] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-Centric Programming: Integrating Web Search into the Development Environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 513–522. https://doi.org/10.1145/1753326.1753402

[6] Joel Brandt, Vignan Pattamatta, William Choi, Ben Hsieh, and Scott Klemmer. 2010. Rehearse: Helping Programmers Adapt Examples by Visualizing Execution and Highlighting Related Code. (11 2010).

[7] William Buxton. 1990. A three-state model of graphical input. In *Human-Computer Interaction, INTERACT '90, Proceedings of the IFIP TC13 Third Interantional Conference on Human-Computer Interaction, Cambridge, UK, 27-31 August, 1990*, Dan Diaper, David J. Gilmore, Gilbert Cockton, and Brian Shackel (Eds.). North-Holland, 449–456.

[8] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2019. Empowering End Users in Debugging Trigger-Action Rules. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3290605.3300618

[9] CoSpace Edu. 2022. *CoSpaces: Make AR & VR in the Classroom.* Retrieved April 25, 2023 from https://cospaces.io/edu/

[10] Jonathan Edwards. 2004. Example Centric Programming. *SIGPLAN Not.* 39, 12 (dec 2004), 84–91. https://doi.org/10.1145/1052883.1052894

[11] Filippo Andrea Fanni, Martina Senis, Alessandro Tola, Fabio Murru, Marco Romoli, Lucio Davide Spano, Ivan Blečić, and Giuseppe Andrea Trunfio. 2019. PAC-PAC: End User Development of Immersive Point and Click Games. In *End-User Development*, Alessio Malizia, Stefano Valtolina, Anders Morch, Alan Serrano, and Andrew Stratton (Eds.). Springer International Publishing, Cham, 225–229.

[12] Sue Fitzgerald, Renée McCauley, Brian Hanks, Laurie Murphy, Beth Simon, and Carol Zander. 2010. Debugging From the Student Perspective. *IEEE Transactions on Education* 53, 3 (2010), 390–396. https://doi.org/10.1109/TE.2009.2025266

[13] Epic Games. 2022. *Blueprints Visual Scripting.* Retrieved April 25, 2023 from https://docs.unrealengine.com/5.0/en-US/blueprints-visual-scripting-in-unreal-engine/

[14] Fungus Games. 2020. *Fungus.* Retrieved April 25, 2023 from https://fungusgames.com

[15] Franca Garzotto, Mirko Gelsomini, Vito Matarazzo, Nicolò Messina, and Daniele Occhiuto. 2017. XOOM: An End-User Development Tool for Web-Based Wearable Immersive Virtual Tours. In *Web Engineering*, Jordi Cabot, Roberto De Virgilio, and Riccardo Torlone (Eds.). Springer International Publishing, Cham, 507–519.

[16] Danilo Gasques, Janet G. Johnson, Tommy Sharkey, and Nadir Weibel. 2019. PintAR: Sketching Spatial Experiences in Augmented Reality. In *Companion Publication of the 2019 on Designing Interactive Systems Conference 2019 Companion* (San Diego, CA, USA) *(DIS '19 Companion)*. Association for Computing Machinery, New York, NY, USA, 17–20. https://doi.org/10.1145/3301019.3325158

[17] Giuseppe Ghiani, Fabio Paternò, Lucio Davide Spano, and Giuliano Pintori. 2016. An environment for End-User Development of Web mashups. *International Journal of Human-Computer Studies* 87 (2016), 38–64. https://doi.org/10.1016/j.ijhcs.2015.10.008

[18] Google. 2022. *ARCore Documentation*. Retrieved April 25, 2023 from https://developers.google.com/ar

[19] Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R. Klemmer. 2007. Programming by a Sample: Rapidly Creating Web Applications with d.Mix. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* (Newport, Rhode Island, USA) *(UIST '07)*. Association for Computing Machinery, New York, NY, USA, 241–250. https://doi.org/10.1145/1294211.1294254

[20] Michelle Ichinco and Caitlin Kelleher. 2015. Exploring novice programmer example use. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 63–71. https://doi.org/10.1109/VLHCC.2015.7357199

[21] Apple Inc. 2022. *ARKit*. Retrieved April 25, 2023 from https://developer.apple.com/augmented-reality/arkit/

[22] GitHub Inc. 2022. *GitHub Copilot*. Retrieved April 25, 2023 from https://github.com/features/copilot

[23] Caitlin Kelleher and Randy Pausch. 2005. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Comput. Surv.* 37, 2 (jun 2005), 83–137. https://doi.org/10.1145/1089733.1089734

[24] Veronika Krauß, Alexander Boden, Leif Oppermann, and René Reiners. 2021. Current Practices, Challenges, and Design Implications for Collaborative AR/VR Application Development. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 454, 15 pages. https://doi.org/10.1145/3411764.3445335

[25] Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R. Klemmer, and Jerry O. Talton. 2013. Webzeitgeist: Design Mining the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Paris, France) *(CHI '13)*. Association for Computing Machinery, New York, NY, USA, 3083–3092. https://doi.org/10.1145/2470654.2466420

[26] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. 2018. *Evaluation Strategies for HCI Toolkit Research*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3173574.3173610

[27] Gun A. Lee and Gerard J. Kim. 2009. Immersive authoring of Tangible Augmented Reality content: A user study. *Journal of Visual Languages & Computing* 20, 2 (2009), 61–79. https://doi.org/10.1016/j.jvlc.2008.07.001

[28] Gun A. Lee, Gerard J. Kim, and Mark Billinghurst. 2005. Immersive Authoring: What You EXperience Is What You Get (WYXIWYG). *Commun. ACM* 48, 7 (jul 2005), 76–81. https://doi.org/10.1145/1070838.1070840

[29] Germán Leiva, Jens Emil Grønbæk, Clemens Nylandsted Klokmose, Cuong Nguyen, Rubaiat Habib Kazi, and Paul Asente. 2021. Rapido: Prototyping Interactive AR Experiences through Programming by Demonstration. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '21)*. Association for Computing Machinery, New York, NY, USA, 626–637. https://doi.org/10.1145/3472749.3474774

[30] Germán Leiva, Cuong Nguyen, Rubaiat Habib Kazi, and Paul Asente. 2020. Pronto: Rapid Augmented Reality Video Prototyping Using Sketches and Enaction. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376160

[31] Sarah Lim, Joshua Hibschman, Haoqi Zhang, and Eleanor O'Rourke. 2018. Ply: A Visual Web Inspector for Learning from Professional Webpages. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. Association for Computing Machinery, New York, NY, USA, 991–1002. https://doi.org/10.1145/3242587.3242660

[32] Extend Reality Ltd. 2022. *Virtual Reality Toolkit*. Retrieved April 25, 2023 from https://www.vrtk.io/

[33] Meta. 2022. *Oculus Integration SDK*. Retrieved September 10, 2022 from https://developer.oculus.com/downloads/package/unity-integration/

[34] Microsoft. 2022. *Hand interaction examples — MRTK2*. Retrieved April 25, 2023 from https://docs.microsoft.com/en-us/windows/mixed-reality/mrtk-unity/mrtk2/features/example-scenes/hand-interaction-examples?view=mrtkunity-2022-05

[35] Microsoft. 2022. Mixed Reality Toolkit Unity Plugin. https://github.com/microsoft/MixedRealityToolkit-Unity [Online; accessed 25-April-2023].

[36] Brad Myers, Scott E. Hudson, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.* 7, 1 (mar 2000), 3–28. https://doi.org/10.1145/344949.344959

[37] Michael Nebeling, Katy Lewis, Yu-Cheng Chang, Lihan Zhu, Michelle Chung, Piaoyang Wang, and Janet Nebeling. 2020. XRDirector: A Role-Based Collaborative Immersive Authoring System. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376637

[38] Michael Nebeling and Katy Madier. 2019. 360proto: Making Interactive Virtual Reality & Augmented Reality Prototypes from Paper. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3290605.3300826

[39] Michael Nebeling, Janet Nebeling, Ao Yu, and Rob Rumble. 2018. ProtoAR: Rapid Physical-Digital Prototyping of Mobile Augmented Reality Applications. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3173927

[40] Michael Nebeling and Maximilian Speicher. 2018. The Trouble with Augmented Reality/Virtual Reality Authoring Tools. In *2018 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*. 333–337. https://doi.org/10.1109/ISMAR-Adjunct.2018.00098

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

[42] Randy Pausch, Matthew Conway, and Robert Deline. 1992. Lessons Learned from SUIT, the Simple User Interface Toolkit. *ACM Trans. Inf. Syst.* 10, 4 (oct 1992), 320–344. https://doi.org/10.1145/146486.146489

[43] Mitchel Resnick, Brad Myers, Kumiyo Nakakoji, Ben Shneiderman, Randy Pausch, Ted Selker, and Mike Eisenberg. 2005. Design Principles for Tools to Support Creative Thinking. (1 2005). https://doi.org/10.1184/R1/6621917.v1

[44] Valve Software. 2022. SteamVR Unity Plugin. https://github.com/ValveSoftware/steamvr_unity_plugin [Online; accessed 25-April-2023].

[45] Vive Software. 2022. *Vive Input Utility*. Retrieved April 25, 2023 from https://github.com/ViveSoftware/ViveInputUtility-Unity/

[46] A. Steed and M. Slater. 1996. A dataflow representation for defining behaviours within virtual environments. In *Proceedings of the IEEE 1996 Virtual Reality Annual International Symposium*. 163–167. https://doi.org/10.1109/VRAIS.1996.490524

[47] Ryo Suzuki, Rubaiat Habib Kazi, Li-Yi Wei, Stephen DiVerdi, Wilmot Li, and Daniel Leithinger. 2020. RealitySketch: Embedding Responsive Graphics and Visualizations in AR with Dynamic Sketching. In *Adjunct Publication of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20 Adjunct)*. Association for Computing Machinery, New York, NY, USA, 135–138. https://doi.org/10.1145/3379350.3416155

[48] Unity Technologies. 2022. *XR Interaction Toolkit*. Retrieved April 25, 2023 from https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.0/manual/index.html

[49] Unity Techologies. 2022. *AR Foundation*. Retrieved April 25, 2023 from https://unity.com/unity/features/arfoundation

[50] Unity Techologies. 2022. *Unity Visual Scripting*. Retrieved April 25, 2023 from https://unity.com/features/unity-visual-scripting

[51] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L. Littman. 2016. Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. Association for Computing Machinery, New York, NY, USA, 3227–3231. https://doi.org/10.1145/2858036.2858556

[52] Valve. 2022. *QUickStart | SteamVR Unity Plugin*. Retrieved April 25, 2023 from https://valvesoftware.github.io/steamvr_unity_plugin/articles/Quickstart.html

[53] XRTK. 2022. *Mixed Reality Toolkit*. Retrieved April 25, 2023 from https://github.com/XRTK/com.xrtk.core

[54] Enes Yigitbas, Jonas Klauke, Sebastian Gottschalk, and Gregor Engels. 2021. VREUD - An End-User Development Tool to Simplify the Creation of Interactive VR Scenes. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–10. https://doi.org/10.1109/VL/HCC51201.2021.9576372

[55] Telmo Zarraonandia, Paloma Díaz, Ignacio Aedo, and Alvaro Montero. 2016. Inmersive End User Development for Virtual Reality. In *Proceedings of the International Working Conference on Advanced Visual Interfaces* (Bari, Italy) *(AVI '16)*. Association for Computing Machinery, New York, NY, USA, 346–347. https://doi.org/10.1145/2909132.2926067

[56] Telmo Zarraonandia, Paloma Díaz, Alvaro Montero, and Ignacio Aedo. 2016. Exploring the Benefits of Immersive End User Development for Virtual Reality. In *Ubiquitous Computing and Ambient Intelligence*, Carmelo R. García, Pino Caballero-Gil, Mike Burmester, and Alexis Quesada-Arencibia (Eds.). Springer International Publishing, Cham, 450–462.

[57] Lei Zhang and Steve Oney. 2020. FlowMatic: An Immersive Authoring Tool for Creating Interactive Scenes in Virtual Reality. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 342–353. https://doi.org/10.1145/3379337.

3415824

[58] Xiong Zhang and Philip J. Guo. 2018. Fusion: Opportunistic Web Prototyping with UI Mashups. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. Association for Computing Machinery, New York, NY, USA, 951–962. https://doi.org/10.1145/3242587.3242632

Received February 2023; accepted April 2023