Defining Configurable Virtual Reality Templates for End Users

VALENTINO ARTIZZU, Dept. of Mathematics and Computer Science, University of Cagliari, Italy
GIANMARCO CHERCHI, Dept. of Mathematics and Computer Science, University of Cagliari, Italy
DAVIDE FARA, Dept. of Mathematics and Computer Science, University of Cagliari, Italy
VITTORIA FRAU, Dept. of Mathematics and Computer Science, University of Cagliari, Italy
RICCARDO MACIS, Dept. of Mathematics and Computer Science, University of Cagliari, Italy
LUCA PITZALIS, Dept. of Mathematics and Computer Science, University of Cagliari, Italy
ALESSANDRO TOLA, Dept. of Mathematics and Computer Science, University of Cagliari, Italy
IVAN BLEČIĆ, Dept. of Civil Engineering and Architecture, University of Cagliari, Italy
LUCIO DAVIDE SPANO, Dept. of Mathematics and Computer Science, University of Cagliari, Italy

This paper proposes a solution for supporting end users in configuring Virtual Reality environments by exploiting reusable templates created by experts. We identify the roles participating in the environment development and the means for delegating part of the behaviour definition to the end users. We focus in particular on enabling end users to define the environment behaviour. The solution exploits a taxonomy defining common virtual objects having high-level actions for specifying event-condition-action rules readable as natural language sentences. End users exploit such actions to define the environment behaviour. We report on a proof-of-concept implementation of the proposed approach, on its validation through two different case studies (virtual shop and museum), and on evaluating the approach with expert users.

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

Additional Key Words and Phrases: virtual reality, end-user development, configuration, toolkit, natural language, meta-design, rules, event-condition-action

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

Authors' addresses: Valentino Artizzu, Dept. of Mathematics and Computer Science, University of Cagliari, Via Ospedale 72, Cagliari, Italy, 09124; Gianmarco Cherchi, Dept. of Mathematics and Computer Science, University of Cagliari, Via Ospedale 72, Cagliari, Italy, 09124; Davide Fara, Dept. of Mathematics and Computer Science, University of Cagliari, Via Ospedale 72, Cagliari, Italy, 09124; Vittoria Frau, Dept. of Mathematics and Computer Science, University of Cagliari, Via Ospedale 72, Cagliari, Italy, 09124; Riccardo Macis, Dept. of Mathematics and Computer Science, University of Cagliari, Via Ospedale 72, Cagliari, Italy, 09124; Luca Pitzalis, Dept. of Mathematics and Computer Science, University of Cagliari, Via Ospedale 72, Cagliari, Italy, 09124; Alessandro Tola, Dept. of Mathematics and Computer Science, University of Cagliari, Via Ospedale 72, Cagliari, Italy, 09124; Ivan Blečić, Dept. of Civil Engineering and Architecture, University of Cagliari, Via Marengo 2, Cagliari, Italy, 09123; Lucio Davide Spano, davide.spano@unica.it, Dept. of Mathematics and Computer Science, University of Cagliari, Via Ospedale 72, Cagliari, Italy, 09124.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
2573-0142/2022/6-ART163 $15.00
https://doi.org/10.1145/3534517

1 INTRODUCTION

In the past few years, there has been an increase in the availability of consumer Virtual Reality (VR) devices, such as the Oculus Rift, Quest, Steam VR, etc. They provide immersive and engaging experiences, developed mainly for gaming entertainment. The maturity of these devices opened the path to a growing number of applications in different sectors, such as healthcare, aerospace, automotive, retail, and manufacturing. The VR impact on such markets is expected to grow in the near future, reaching the volume of 84.09 Billion US dollars by 2028 [46]. As happened in other fields such as the Internet of Things (IoT) or the publication of web content, the variegated needs of particular users or niches will push for opening the authoring of VR content to end users. We expect such a scenario when the user's needs change over time, which applies to VR content. For instance, we may use a VR environment for training people in performing potentially dangerous tasks (e.g., sanitize and environment), but the procedure requirements may change over time (e.g., new measures to counter the spread of Covid-19 pandemics). Currently, such modifications would require the intervention of a professional developer, which could not always be feasible.

However, opening VR content authoring to end users is still a tough challenge, even using limited and simplified approaches. Shifting the entire building process towards end users is not entirely feasible because building VR experiences requires different skills and involves a team including 3D modelling experts, developers, game designers, etc. In addition, they create environments closed to changes: we need a develop-build-distribute cycle to modify their behaviour. Currently, it is possible to configure an environment through scene builders or inspectors, which allow placing and orienting 3D models that constitute the virtual world in both immersive or desktop mode. We can find different solutions for this task, both in the literature and in commercial products [21, 47, 48]. The most challenging part is still the definition of the dynamic behaviour of an environment, i.e., responding to the interaction with a user or other objects. The tools supporting the articulation of dynamic behaviours are too complex for end users. They target people skilled in game engines such as Unity3D (e.g., Fungus [16]), or limit their support to animations (e.g., Ottifox [38]).

In this paper, we propose an End-User Development approach [29] for supporting the configuration of a VR environment by users without skills in programming and/or 3D modelling. The approach includes three roles inspired by meta-design [2, 11, 15], collaborating for transforming the template of a VR environment created by an expert into a peculiar VR experience configured by an end user. In particular, we detail the engineering of a solution dedicated to expert users for including configurable objects in a VR environment, whose behaviour is defined by end users through Event-Condition-Action rules expressed in natural language. The solution is adaptable to different VR engines. It supports i) the rule language specification, ii) the definition of configuration points inside a VR environment, and iii) the execution of the rules at run-time, without requiring further builds of the VR environment. We report on a proof-of-concept implementation of the solution in Unity3D, the design of two sample templates (a virtual shop and a virtual museum), and their configuration for supporting two different experiences each. Finally, we discuss the results of a user study assessing the utility and the usability of the proof-of-concept in defining a template.

We organised the paper as follows. First, we discuss the related work on EUD approaches for VR, and we summarise rules applications in EUD (Section 2). Then, we introduce the general concepts and roles in the EUD approach, detailing the components of the proposed solution (Section 3). After that, we describe a proof-of-concept implementation (Section 4), the development of two sample templates and their configuration through rules (Section 5). Then, we report on the results of a user study involving expert VR developers (Section 6), and we discuss the contributions and the limitations in our approach (Section 7). Finally, we conclude the paper by tracing the path for further research (Section 8).

2 RELATED WORK

End-User Development (EUD) [29] approaches focus on supporting users without programming skills in developing or adapting software applications. EUD can reduce the time and costs needed for customization and increase software quality [39]. In general, the main advantage for users consists in developing and adapting systems at a level of complexity that is adequate to their practices, background, and skills [39]. On the other hand, professional software developers can rely on a broader adoption, impact, and diffusion of their applications. In the following, we summarise the techniques enabling end users to create VR environments and the relevant applications of rules in EUD.

2.1 End-user definition of VR environments

Different commercial and research tools support end users in creating VR environments. Most of these tools are limited to the definition of static scenes or multimedia content overlay. For instance, Hubs by Mozilla [34] allows people to create shared rooms for VR communities and groups, while also being able to modify their configuration through multimedia content and appearance customisation. Its focus is on gathering people, so the environment's behaviour is a second-class concern. A specific tool for Hubs, called Spoke [35], supports the composition of 3D models for setting up a static representation of a scene through user-friendly tools. It exploits well mature techniques for end user VR modelling in a desktop setting. It contains tools for positioning, scaling, and rotating 3D models, an extensive catalogue of 3D models ready to be included in the scene and features for seamless result sharing on the web. Its main drawback is the lack of support for the definition of behaviours beyond the animation, the need for build and test iterations and the long time required for getting a good quality result.

The research literature focused for a long time on techniques for editing the VR environment while the user is immersed in the virtual world. Steed et al. [41] proposed the first environment targeting both the scene configuration and the behaviour definition in an immersive setting. It used a dataflow paradigm and aimed at speeding up the development process by technical people. Lee et al. [26, 27] evolved such a research thread defining the so-called "immersive authoring", where users created the scenes while immersed into them. Again, such an environment supported the behaviour definition but targeted developers. Takala [42] introduced a toolkit for simplifying the creation of VR, providing a set of building blocks that allows hobbyists and students to create VR experiences rapidly. While the tool lowers the barrier for editing VR, users require technical knowledge for assembling the building blocks through dedicated code. More recently, tools like FlowMatic [50] evolved such ideas adapting to the newly available devices. It is an immersive authoring tool that allows programmers to specify reactions to discrete events (e.g., user actions and system timers). All these tools have a purpose similar to our project, but their goal is to increase developers' efficiency, reducing the build-test-fix cycle. EntangleVR [8] proposes an innovative representation of the VR scene behaviour inspired by the entanglement phenomenon in quantum physics. The approach is interesting since it lowers the barrier for modelling such a phenomenon, but it also limits the definition of behaviour to a specific case.

We can also find VR tools designed explicitly for end-user development in the literature. For instance, XOOM [18] is a tool designed for non-ICT-specialists to create web-based immersive VR applications in the cultural heritage field. It contains a simplified behaviour model, defining a restricted set of experiences. The approach we propose in this work raises the ceiling of the possible interaction, also supporting the calibration of the behaviour definition on the domain needs. VR GREP [47] is a tool dedicated to end users for the design and development of VR applications. The tool runs in two different modes, one for authoring and one for running the

resulting VR environment. The authoring environment offers means for creating the navigation in the environment, inserting and manipulating 3D objects. While it provides standard interaction and manipulation techniques for statically defining the environment configuration, the support for behaviour definition is limited to navigation and reaction to button clicks.

There are also attempts to build commercial tools for end-user development. Ottifox [38] is a tool for creating generic VR environments. The tool supports an entirely visual development of the VR environment, and it exploits a simplified version of rules we use in this paper, without support for conditions and working only with animations. Other commercial or open-source tools offer support for end-user storytelling or game development. Fungus [16] is an open-source visual storytelling tool designed as a Unity3D extension. It allows creating visual novels through flowcharts, but the visualization is demanding in terms of screen space, limiting the end user's comprehension [22].

We can find examples of tools targeting specifically the end-user development of VR behaviour in serious game literature. For instance, Manestrina et al. [32] discuss the definition of an actor programming environment for creating and modifying the behaviour of non-playable characters in serious games. The aim is to shift the changes in the game behaviour with the evolving needs of the game. While we share some goals with this thread in the literature, we aim to cover a more generic representation of VR environments.

Relevant behaviour definition techniques are also available in the literature targeting the editing of interactive 360° videos. While we cannot consider them as full VR experiences, they share the immersion and the need to define how to react to the user's interactions. In this field, Blečić et al. [5, 14] propose the solution closest to the one we present in this paper. They created an authoring tool for designing point-and-click games limited to 360° videos, defining the behaviour through rules. We share the same rule-based approach with this work, but we extend it to more complex interactions in full VR environments. Torres et al. [43] propose a more limited tool, supporting the editing of 360° videos by adding interactive content panels containing information, quizzes, pictures, and 3D models. Mendes et al. [31] introduce a similar tool working in a different domain. They support teachers and instructors in adding multimedia content for learning purposes. It shares with the previous tool also the limitations in defining the behaviour. Adao et al. [1] introduce a more complex behaviour modelling but is still limited in terms of generic VR interactions. It supports both interaction and time-triggered reactions in the environment, controlling multimedia content and spatial sound. A tool providing support to specific aspects of the experience is Culture4All [33]. It focuses on supporting the accessibility of VR content, providing a platform explicitly designed for cultural heritage, allowing the content creator to use accessibility-related services while authoring the VR content.

Another relevant field for our work is the support for creating VR prototypes. On the one hand, such tools require simplifying the interaction definition and the environment modelling to support rapid design and test cycles. On the other hand, they target specifically designers, who have a higher knowledge of the VR environments' structure than end users. Nebeling and Madier [37] support design teams in creating rapid prototypes of both VR and AR. The fast-paced cycle of creating and editing the sketches requires avoiding complex content editing and behaviour programming. The solution proposed in the paper offers a smart combination of pictures and hand-drawn sketches, together with "Wizard of Oz" simulations of the environment behaviour. In a follow-up work called XRDirector [36], they expand the approach for supporting multiuser authoring using a filmmaking metaphor that assigns different roles to the participants (director, actor, camera). While the result is again focused on producing prototypes and is still a simulation of the object behaviour, it introduces an interesting idea for supporting collaboration between end users in producing VR and AR content. Tvori [44] is a VR and AR tool for prototyping environments and interfaces while being inside an immersive environment and with the possibility to collaborate on the same project in real-time.

2.2 Rule-based approaches in End-User Development

Rule-based environment configurations for end users are widely adopted in research work. There are various rule programming styles documented in the literature. One of the most adopted is the Trigger-Action Programming (TAP), in particular in the Internet of Things (IoT) field, both at the academic level [10, 13, 19] and in successful commercial tools [24, 45].

Trigger-Action programming addresses users without strong IT skills, so rules assume the following simple pattern: "*if* <a trigger occurs>, *then* <an action is executed>". The first part describes the event that fires the rule, and the second specifies the action reacting to the trigger. Events arise when an object in the environment changes its state (e.g., the light turns on) or there is a change in the context (e.g., the temperature is below 21°). The action describes a command sent to the same object that triggered the rule (e.g., turn the light off) or another device (e.g., close the door). The rule's action may trigger another rule, leading to a chaining effect. This is useful for creating complex behaviours, but it can also cause unexpected effects [9].

Because of their simplified format, TA rules lend themselves well to wizard-style visual interfaces, where the user can specify the desired trigger and action, resulting in the final rule. This metaphor causes problems when the environment has many devices or, in general, when the complexity of the ecosystem becomes high [3].

An interesting variant of Trigger Action rules includes an *else* block, which provides an alternative set of actions to execute if the event is not satisfied. Coutaz and Crowley [12] proposed it in a EUD environment designed to empower people with tools to control their home. Unfortunately, they only provide a preliminary test in their home environment, so the variant requires further research.

Event-Condition-Action (ECA) rules add a third, intermediate part between the event and the action part: a condition defining a guard preventing the rule from being executed if not verified. This part usually checks the state of the environment (e.g., "if the light is on", "if the timer elapsed more than 3 seconds"), or other context-related information. ECA rules distinguish the trigger and the condition by the adverb introducing the rule part: "*when* [something happens] *if* [condition] *then* [action]". Usually, such a format allows triggering more than one action in the *then* part. ECA rules are more expressive than simple TA rules since they may associate different actions to the same trigger event, and they support filtering through the condition part. The higher expressiveness may also lead to the increased complexity of the behaviour definition, raising the barrier for end users.

The study by Brackenbury et al. [6] shows that one of the most challenging issues for users in adding the condition part to an ECA or TAP rule consists in understanding the difference between events and conditions. One may exchange them in many situations, but sometimes there are subtle differences in the semantics that are difficult to grasp for end users. Instead, an erroneous understanding of the distinction between events and states could lead to inconsistencies, loops, and redundancies [9]. The research also focused on supporting fixing or detecting bugs in rules through model checking [7, 23, 28, 49]. Currently, the results are limited to enforcing different safety checks (for instance, the fridge's temperature should never exceed 5°C). The solution proposed in [3] introduces a hybrid ECA-TAP approach, where users can introduce rules through both *where* and *if* adverbs, depending on the result they want to achieve. However, some participants misunderstood the difference among them, showing that we require more than changing the syntax for solving this problem.

ECA rules support the configuration of the environment in different domains. For instance, Barricelli and Valtolina [4] exploit them in an interactive visual system for the collaborative management of IoT sensors for improving the quality of life and promoting wellness awareness. Blečić et al. [5, 14] created an authoring tool for point-and-click games, whose logic is defined

through a set of ECA rules following a natural language representation. The environment supports the editing through a set of drop-down lists, guiding the user in creating the rules.

We adopt the Event-Condition-Action rules format inside a VR environment in this work. The guidelines in [39] support this choice: trigger-action paradigm is not enough for managing complex scenarios and automation the end users need to define for configuring Virtual Reality environments.

3 AN END-USER DEVELOPMENT APPROACH FOR CONFIGURING VIRTUAL REALITY ENVIRONMENTS

In order to define the workflow for supporting end users in creating their own VR experiences, we applied the three levels identified in the *meta-design* approach [2, 11, 15] which envisions the participation of end users in a hybrid role of designers and consumers of software artifacts. It defines three hierarchical levels in the creation and evolution of the software, which exchange information among them, exploiting different languages and tools at each level. At *meta-design* level, software engineers, professional developers, or content creators define the core aspect of the software using languages characterized by a high computational power (e.g., Turing Machine equivalent). End users cannot manage such development tools, so we have the highest computational power at this level but the lower usability. At the *design level*, domain experts participate in the design of software, using languages having less computational power (e.g., permitting a limited set of operations), which are usable by non-technical people. The *use* level includes the final users performing the well-defined activities that the software must support. It basically exploits a domain-oriented language with the lowest computational power but the highest usability for users.

Applying the principles of meta-design, requesting an end user to build an entire VR environment is unfeasible. Instead, it is reasonable to apply a solution similar to Content Management Systems for publishing web content: end users download website templates, working out of the box but including dummy content. End users exploit specific languages and interactions for configuring it for their purposes. Therefore, we expect end users to start from predefined VR environments to configure and adapt them to their needs. In the proposed workflow, the environment prepared by experts represents a solution adaptable to different settings by configuring its behaviour and adding content in predefined points. They will not resemble a complete, final version of an interactive VR environment, but rather a *template*, which end users can tailor to their needs. Therefore, the same template may fit the need of many end users.

The main problem is defining the language at the *design level* for configuring such templates. As already pointed out in Section 2.1, while we already have usable solutions for moving the content around an existing VR environment, we do not have an equivalent for its behaviour. In this paper, we propose to borrow the knowledge developed in the last years of research in rule-based EUD, especially in the IoT domain (see Section 2.2). We use Event-Condition-Action (ECA) rules for defining 3D objects' behaviour in isolation and their interaction with other objects. The main contribution of this paper is the definition of components enabling rule-based configuration and the rule runtime support for VR.

In summary, our solution foresees three different roles:

- **Template Builders** (TB), which represents users with good skills in both 3D modelling and game programming. We assume that they are proficient with game programming platforms and can build complex VR environments. TBs create the *templates*, which are almost-complete VR environments, open to end-user configurations. They represent the experts of the *meta-design* level.
- **End-User Developers** (EUDevs), which represent users without skills in 3D modelling and game programming, but having an average familiarity with computer use (e.g., proficient in

using standard office programs) and with VR environments (e.g., they played 3D videogames or have VR experience as users). They represent people who may require creating VR content for their business or leisure (e.g., touristic promotion, content advertisement, etc.). They are supposed to have a limited budget for creating the content or, even if they have the budget, they need to modify the content by themselves (e.g., they update the VR content frequently). They download or buy templates to configure for getting the final version of the VR environment. They represent the hybrid designer/consumer at the *design* level.
- **Users**, which represent the final consumer of the VR contents, either created by professionals or by EUDevs. They represent the final user of the *use* level.

In Table 1, we introduce a sample configuration scenario for concretely explaining the concepts in our approach. It partially covers one of the case studies outlined in Section 5. It will ease the understanding of the general solution in the current section and the proof-of-concept implementation in Section 4.

We suppose having a template representing a virtual clothing shop, including products (e.g., t-shirts, hats, shoes etc.) and multimedia elements for displaying additional information (e.g., text panels, virtual monitors for videos etc.). The EUDev wants to configure the environment for displaying a text panel containing the price when the user picks a pair of shoes.

Table 1. A simple template configuration scenario.

3.1 The VR Object Taxonomy

For creating a usable configuration mechanism, EUDevs need a language for specifying the VR environment behaviour they can understand and manipulate. Even if we take inspiration from the rule-based configuration in IoT, there is a point that marks a substantial difference between the two domains. In an IoT environment, the capabilities of the physical devices define the actions they can perform. Two instances of the same device have the same associated actions. In VR, this is not true. The Entity Component System architecture [30], used by most VR engines, allows fine-grained control of the behaviour for each object instance in an environment. The component list defines what objects can do, not their physical appearance. So, we can have objects in the environment that look the same (e.g., they share the same 3D model) but behave differently, as typically happens in games.

Therefore, before introducing a rule language defining *when* objects should do something, a TB needs a way for specifying *what* an object in the scene can do. Such a definition must include actions understandable by EUDevs, so it must avoid technical concepts and jargon, and rely on a general naïve understanding of the world and knowledge of the particular domain the VR environment models. On the one hand, we expect different domains to require performing different actions. On the other hand, if TBs need to map all the necessary internal components to high-level actions, adopting the solution would be unpractical. Therefore, we opted for providing TBs with a taxonomy of reusable high-level types for building VR environments templates. The taxonomy does not claim to completely represent all the possible VR object categories, but it defines a reusable starting point. Its effectiveness depends on an appropriate coverage of common VR concepts, but also on domain-dependent extensions, which we must support at the implementation level (see Section 4.1). For creating the concept list, we analysed the literature on model-based approaches for VR development (e.g., [20]), and the categorisation used by 3D models and assets repository, such as SketchFab, TurboSquid, Unity Asset Store etc. We briefly introduce the main concepts in this section, and we provide a more detailed description in Appendix A.

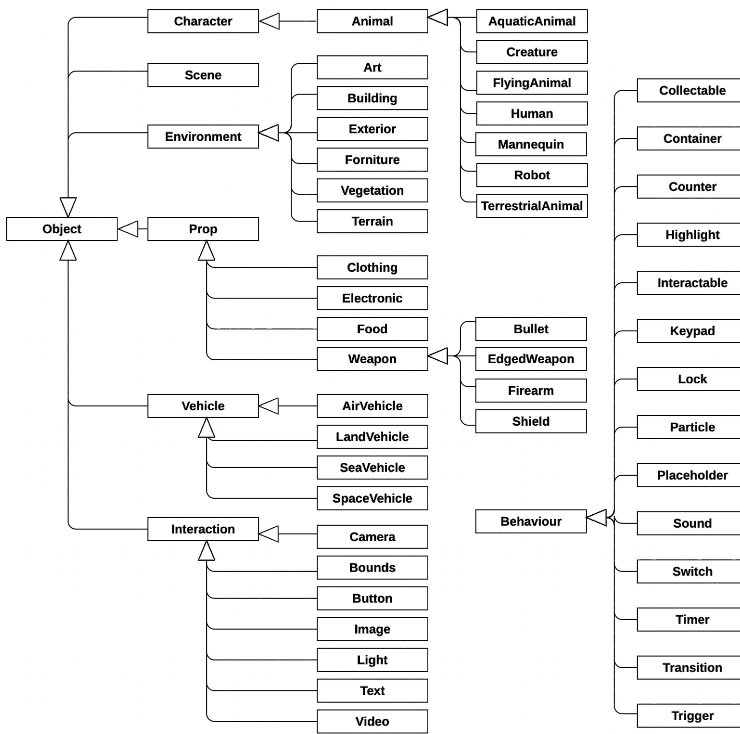

Fig. 1. Class diagram for the object categories in the VR templates.

Figure 1 shows a class diagram depicting the proposed taxonomy. Its types define a coarse representation of virtual objects categories. In our approach, TBs assign such categories to each virtual object instance, while the EUDev can exploit the actions associated with each category for configuring their behaviour. Such assignment provides the following pieces of information to both the EUDev and the runtime support: i) a semantic tag for the game object, specifying the category it belongs to; ii) the specification of a set of status variables, whose control allows changing the object's state dynamically; iii) the specification of a set of actions associated with the object, which are the building blocks the EUDev uses for programming the VR environment behaviour.

In our taxonomy, we make a distinction between *Objects* and *Behaviours*. The difference between the two categories is related to their physical appearance, which drives the assignment: the former are the things a user can perceive, while the latter are common behaviours we can identify across the different categories of objects. We categorised perceivable objects according to an ontological criterion: each object instance belongs to a single category, and we cannot assign it to multiple ones. For instance, a model whose shape resembles a chair belongs only to the *Furniture* category, and we cannot assign it, for instance, to *Food*. However, we can add the *Container* behaviour to the same chair, meaning that we can put other objects in it. If we consider a sandwich, we can assign it to the *Food* category, but we can consider it also as the *Container* of its ingredients. Therefore, *Food* and *Furniture* are *Objects*, the *Container* is a *Behaviour*.

The object categorisation is coarse. We limited the hierarchy's depth to high-level categories, stopping when we considered that going further would not add useful information for the EUDev.

For instance, we assign an object representing a cat to the *Terrestrial Animal* category. We did not create a specific sub-category for the concrete species. The reason is that a hypothetical *Cat* category would have the same status description and actions of a *Terrestrial Animal*. However, such simplification on the object taxonomy would cause usability issues for EUDev in creating the behaviour rules: they will struggle in reasoning about objects belonging to abstract categories. So we provided TBs with means for specifying EUDev-friendly aliases for a category name, which will appear on the rule representation in natural language (see Section 3.2). If a TB defines an alias, the EUDev can refer to the hypothetical *Cat Tom*.

The *Behaviour* category models typical interactive behaviours we can assign to different objects, independently from how they appear. It is possible to assign more than one *Behaviour* to a virtual object, composing the set of possible actions the EUDev can program. In general, a configurable virtual object is associated with one *Object* category and a list of *Behaviours*. For instance, a TB can assign a *Collectable* and *Sound* interaction to a piece of *Furniture*. In this way, the EUDev can define rules that, for example, collect the piece and play a sound when the main character bumps on it. The final set of actions that a EUDev can configure on a single object consists of the union of all the actions defined by the *Object* and the different *Behaviour* components.

We have categories representing dynamic attributes of the 3D scene (*Camera*, *Light*s, *Transitions*), simple techniques for defining environment manipulation and automation (*Bounds, Container, Collectable, Counter, Keypad, Switch, Lock, Timer, Trigger*) and simple representations of media content (*Image, Sound, Text, Video*). For instance, considering a *Container*, it holds the information about the capacity and the number of objects currently it contains. It has the actions for inserting and removing objects or for emptying it.

A behaviour requiring a particular mention is the *Placeholder*. We introduced it to support TB in explicitly defining extension points in the template where the EUDev can insert custom content. We include a sample using such a category in Section 5.

Considering the sample in Table 1, the environment contains at least a *Character* representing the shop visitor (i.e., the user of the VR environment), *Clothing* objects representing the shoes and all the other products, *Text* objects representing the information panels for showing the price. In addition, all products are associated with a *Placeholder* and an *Interactable* behaviour. The former allows the EUDev to set a custom 3D model for each product. The latter allows the users (i.e., the shop visitors) to pick and release the products.

3.2 The EUD Rule Language

The rules EUDevs use for configuring the VR environment behaviour have a structured definition, i.e., they follow a precise syntax scheme, but are readable as natural language (English) sentences for grasping their meaning.

The complete grammar for the rule language is available on in the GitHub repository[1], defined using AntLR[2]. In the following, we report some excepts using uppercase words for identifying terminal tokens (e.g., `DEFINE` for the token "define"). Listing 1 shows the grammar for a single rule, including two sample structures: a rule including an event and a single action, and one including an event, a condition and multiple actions.

```
// rule grammar
rule : WHEN action (IF condition)? THEN (action)* ;

// sample structure: no condition, single action
```

[1]https://github.com/cg3hci/ECARules4All
[2]https://www.antlr.org

```
when [action1] then [action2]
// sample structure: single condition, multiple actions
when [action2] if [condition] then [action 3] [action 4] ... [action n]
```

Listing 1. Declaration of a rule

A rule follows the event-condition-action (ECA) structure. The `when` keyword introduces the *event* part, described through an `action`. An action is associated with every method supported by classes in Figure 1, or domain-dependent extensions. So, a EUDev can specify rules that trigger whenever a managed object in the VR environment does something. The syntax representing the action that triggers a rule is the same for describing further actions the EUDev enters to define the environment response, in the `then` part. This eases the identification of actions as commands and event sources. The `then` part of a rule corresponds to the *action* part in the ECA schema. EUDevs can insert multiple actions in the `then` part, including multiple changes in response to a given event. All these actions, in turn, will raise the associated events, possibly triggering other rules. We report in Listing 2 six sentence schemes we identified, together with an example for each of them.

While the first five are straightforward, the last one requires some explanation. Passive actions are associated with the object type that undergoes the action, but they increase the available actions for the subject in the rule language. This happens when, considering a `[subject] [verb] [direct object]` structure, we have the implementation of the verb logic on the class representing the direct object. This sometimes makes sense for separating the concerns from an engineering point of view. We have an example of this type of action in the *wear* method of the *Clothing* object, which requires a *Character* parameter. By keeping its implementation in the *Clothing* class, we decouple the basic actions of a character from those involving other objects. Otherwise, given that users in the VR environment are *Character*s, it would become the class defining the large majority of all the possible actions. In contrast, from a EUDev perspective, the best description in natural language is “the character wears the clothing” and not “the clothing is worn by the character”. By using a passive action, the owner is the type of the direct object, while it uses as a parameter the subject of the sentence in natural language.

```
// schema 1: [subject] [verb]
THE LANDVEHICLE IDENTIFIER STARTS
the land vehicle car starts;

// schema 2: [subject] [verb] [direct object]
reference: THE object IDENTIFIER;
THE character IDENTIFIER LOOKS AT reference
the human visitor looks at the art painting;

// schema 3: [subject] [verb] [value]
THE character IDENTIFIER JUMPS ((TO position) | (ON path)) ;
the human visitor jumps to the position entrance;
the human visitor speaks "Hello world!";

// schema 4 (state change): [subject] changes [property] to [value]
THE ART IDENTIFIER CHANGES AUTHOR TO STRING_LITERAL;
the art painting changes author to "Picasso";

// schema 5 (number increase/decrease): [subject] increases [property] by [value]
THE character IDENTIFIER (INCREASES | DECREASES) LIFE (BY floatLiteral)?
the human player1 increases life by 2;
```

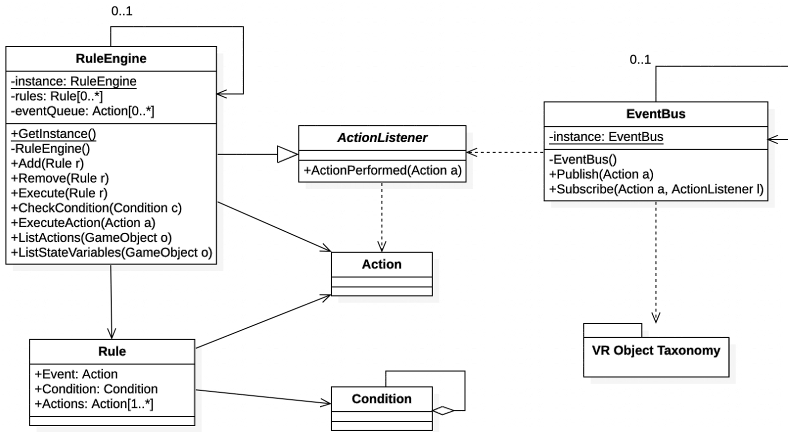

Fig. 2. Class diagram for the ECA rule engine.

```
// schema 6 (passive action): [subject] [verb] [direct object]
THE character IDENTIFIER WEARS THE CLOTHING IDENTIFIER;
the human player1 wears the clothing tShirt;
```

Listing 2. Different types of action specification

Each rule has an optional *condition* (the `if` part), which defines the conditions for executing the `then` part. End users usually avoid writing conditions, or they use a simple check on a single property [45]. Given this fact, many ECA languages for end users limit the condition to a single predicate. We support the definition of composite expressions in the language specification, but in the implementation of the rule-editing interface we support only one level of *and/or* on simple checks to avoid clutter. The solution resulted in a useful compromise between expressiveness and the interface usability already validated in other tools [5, 14].

Listing 3 shows the rule defining the custom behaviour required by the scenario in Table 1. The event triggering the rule is the *pick* action, whose subject is the character representing the user of the VR environment. Its direct object is *sneakers4* one of the *Clothing* objects representing the selling products. There is no need to specify further conditions for triggering the rule, so we omit the condition part. The reaction of the environment to the event is showing the information panel, called *infoPanel2*, which is a *Text* object.

```
when    the character visitor picks the clothing sneakers4
then    the text infoPanel2 shows
```

Listing 3. A rule specifying the behaviour required by the scenario in Table 1

3.3 The Rule Execution Support

The last component in our solution is the support for executing the rules at run-time. Since it is not feasible for EUDev to build the environment each time they change the rule definition, they need a specific run-time component inside the VR template, which we call *Rule Engine*. At a high level, it has two tasks: triggering and executing the rules. We provide here an abstract description (i.e., independent from the VR engine) of the classes we used for solving this problem, depicted in Figure 2.

We start from an object-oriented description of the rules, consisting of three classes: `Rule`, `Condition` and `Action` (see Figure 2). The class `RuleEngine` is responsible for managing and executing such a rule description. We have a single instance of such class in the system (singleton pattern [17]). The class has methods for adding and removing rules from the configuration set (`add` and `remove`).

A `Rule` contains the `event` field, specifying the action that triggers the rule. A `Condition` is an object describing either a simple predicate or a tree composing such predicates through boolean operators. The predicates need to be tested against the values contained in the state variables. Both action and the condition description contain references to the VR objects in the environment (either memory references or identifiers to lookup in the object repository). Executing a rule means first checking the condition on the current state of the VR objects and then invoking the method corresponding to each action, passing references to objects or the values specified in their definition.

Besides defining how to execute the rules, the *RuleEngine* also defines *when* to execute them. The triggers are the same actions the EUDevs use for specifying the behaviour. This means that each action in a rule definition can potentially trigger other rules. So, after finishing the execution of an action, the rule engine needs to check whether there are rules to trigger or not. The rule engine requires a notification each time an action completes. If such action triggers a rule, the engine enqueues it and continues the execution loop until the queue is empty. In Figure 2, we adopted the *publish-subscribe* pattern (a variant of Observer [17]) to avoid the proliferation of interfaces defining callback methods. The `EventBus` class represents a channel abstraction where it is possible to publish messages on different actions and to register for receiving them. The publishers are the classes implementing *Objects* and *Behaviours*, which will send a message on the `EventBus` each time an action they implement is completed. The `RuleEngine` class represent the subscriber, which registers for notifications from each action in the *when* clause contained in the environment rules. The reaction to the notification enqueues the rule corresponding to the message and starts the execution loop if needed.

It is worth pointing out that there are two main causes for completing an action. The first is the `executeAction` method in the `RuleEngine`, which occurs after triggering a rule as a consequence of the completion of another action. The entry point in the rule execution loop is the second cause, which is an interaction between the user and the environment or between two game objects. The responsible for raising such notifications is the implementation of the taxonomy types. It must define a mapping between the internal state of the VR objects in the engine and its high-level representation for the EUDev and vice-versa, and publish messages on the *EventBus* whenever the values change.

Considering the scenario in Table 1, the `RuleEngine` contains a single *Rule* object, including an *Action* instance (picking the shoes) in the event field and another instance (showing the text panel) in the actions list. The rule triggers as a consequence of the interaction between the user and the environment. Thus, the *Interactable* implementation is responsible for publishing a `pick` message on the `EventBus`. The `RuleEngine` receives the notification and enters the rule execution loop, it performs the action corresponding to showing the information panel, checks whether it triggers other rules and, supposing that there are no further rules to execute, it ends the loop.

4 PROOF-OF-CONCEPT IMPLEMENTATION

In this section, we discuss the main properties of the implementation we provided for the solution depicted in Section 3. We selected Unity3D[3] as VR engine and C# as development language. The discussion has two main objectives: the first is demonstrating that the approach is feasible,

[3] https://unity.com

while the second is showing concrete opportunities for customizing the approach for different needs through extensions. The plugin relies on the standard Unity XR Interaction Toolkit for supporting VR interactions, minimizing the requirements for external libraries. We developed the other functionalities leveraging objects directly provided by the game engine. The code is available on GitHub[4], together with a short demonstration video in the additional paper material.

4.1 The ECALibrary

The `ECALibrary` contains the implementation of the classes included in the VR taxonomy introduced in Section 3.1. A TB can assign a given `GameObject` (the generic VR object in Unity) to at most one category. It is not mandatory to assign each `GameObject` to a category. The TB assigns only those that the EUDev can further configure.

The root of the simplified object hierarchy we created for representing VR environments to EUDevs is the `ECAObject` class. It contains the variables and actions common to all categories. The same applies to the `ECABehaviour` component and its sub-classes, which model the interactive behaviours. They are basically a C# implementation of the taxonomy described in Section 3.1.

Both the `ECAObject` and the `ECABehaviour` are components in the Entity-Component-System architecture supported by Unity. They are sub-classes of `MonoBehaviour`, the abstract component in Unity. We do not use inheritance for expressing the category hierarchy, which would break the composition in the ECS architecture. However, we specify that the implementation of a component requires another one through the `RequireComponent` class-level attribute. We implemented the inheritance in the taxonomy through such an attribute.

An interesting point to discuss here is the effort we put into implementing the rule execution support that does not assume any object categorization, keeping its definition agnostic of the types in the `ECALibrary`. The only distinction we assume in the categorization is between objects and behaviours. The other pieces of information about the category or behaviour type name, the list of state variables, and the exposed actions are included as metadata in the component class implementation, declared as C# custom attributes[5]. Given such a structure in the implementation, the following is both the description of how we defined the taxonomy in Section 3.1 and of how third-parties can enhance the set of available categories.

The first step for including a component among those managed by the rule engine is specifying a type name that a EUDev can understand. For doing this, we use the `ECAType` class-level attribute, which marks it as a managed component and provides the runtime (see Section 4.2) with the information on the type name.

```
[ECAType("human")]
[RequireComponent(typeof(Animal))]
public class Human : MonoBehaviour { ... }
```

Listing 4. Annotating a Unity Component for managing it at runtime through ECA rules

Then, the rules run-time needs information about the variables that maintain the state and the methods implementing the actions. For defining the state variables, we provide another custom annotation (`ECAStateVariable`) marking public instance variables or properties of a C# class. A state variable has of the following base types: `Boolean, Color, Float, Integer, Position, Path, Identifier, Rotation, Text, Time`, having a straightforward meaning. Listing 5 shows two sample annotations defining two state variables in the `Character` object. The first annotation marks a flag for identifying the character controlled by the final user. The annotation defines the

[4] https://github.com/cg3hci/ECARules4All

[5] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/creating-custom-attributes

name of the object variable as presented to the EUDev and its type in the rule system. The second annotation defines the variable containing the life points for a character, stored as a float.

```
[ECAStateVariable("playing", ECABaseType.Boolean)]
public ECABoolean playing;

[ECAStateVariable("life", ECABaseType.Float)]
public float life;
```

Listing 5. Annotating the instance variables of a Unity Component for modifying them through rules

Finally, a third custom attribute (`ECAActionAttribute`) marks the actions that a given object or behaviour provides to EUDevs for defining the environment behaviour. It has a constructor for each action type we identified in Listing 2, defining the rule syntax. It specifies the allowed taxonomy type for each part of the action. Listing 6 shows three examples of recurrent action definition patterns. The first defines the *swims to* action for an aquatic animal, which moves it to the position specified in the parameter. The second allows any *ECAObject* to look at (i.e., turn to) another object. Finally, the last sample is a passive action. Differently from the previous ones, the action belongs to *Character* type in the taxonomy in the EUDev syntax, but the method is actually defined in the *ECAFood* class, which is the implementation of the *Food* type.

```
[ECAAction(typeof(ECAAquaticAnimal), "swims to", typeof(ECAPosition))]
public void Swims(ECAPosition p) { ... }

[ECAAction(typeof(ECAObject), "looks at", typeof(ECAObject))]
public void Looks(ECAObject o) { ... }

// the following method belongs to the class ECAFood
[ECAAction(typeof(ECACharacter), "eats", typeof(ECAFood))]
public void EatenBy(ECACharacter c) { ... }
```

Listing 6. Annotating a Unity Component instance variable for modifying it through rules

In order to implement the sample configuration scenario in Table 1, the player in the Unity scene is associated to the `ECACharacter` component, the virtual object representing the shoes contains both the `ECAClothing`, the `ECAPlaceholder` and the `ECAInteractable` components, while the virtual information panel contains the `ECAText` component. They define the actions and the status variables required for implementing the logic in Unity (e.g., the visibility boolean, the text in the information panel etc.).

4.2 The Rule Engine

The `ECARuleEngine` is the class responsible for executing the rules defined by the EUDev. When a template configuration starts in the VR experience mode, it loads the rules from a text file and builds their object-oriented description. The implementation of the engine does not assume any structure on the ECAObject types, but it dynamically obtains it reading the custom attributes. Thus, no change to its implementation is required if a TB extends the set of available objects or behaviours. This means that extending the taxonomy has no impact on the engine, and the new rules are ready to be used by EUDevs.

An `ECARule` is the C# representation of a rule. An `ECACondition` is an object describing either simple or composite predicates on managed objects. The action and the condition description contain references to `GameObjects` in the VR environment. For instance, if we consider the sample action “`the human visitor looks at the art painting`” (*[subject] [verb] [object]*), the object-oriented description will contain a `GameObject` reference having id *visitor* and containing the

component ECAHuman, a reference to its LooksAt instance method and a reference to a GameObject whose identifier is *painting*.

Supposing that all these pieces of information are available, the problem of writing a method in the ECARuleEngine class for executing a generic action is simple: we receive an ECAAction instance as a parameter, and we invoke the referenced method on the subject via reflection. We pass the direct object as a parameter and, in the more general case, all the other values or references requested by the six action types we identified (see Listing 2).

We assume that rules and Unity share the same virtual object identifiers and that the GameObjects contain the components from the ECALibrary, specified by TBs. In such a case, identifying the components and the VR object references is trivial. The last information we need is related to the methods that implement the actions and the instance variables that represent the state in a VR object. Having the custom attributes identifying both state variables and actions we discussed in Section 4.1, we can inspect a GameObject instance using C# reflection as shown in Listing 7. We loop over all the components associated with the object, and we look for those marked with the ECAType annotation, indicating a managed type. Then, we go down looking for action methods or, symmetrically, for state variables. In this second step we search for the ECAActionAttribute at the method level. For state variables we look for the ECAStateVariable on instance variables (or properties). Once we found one, we add the action or state variable to the ones supported by the current VR object instance.

```
foreach(Component c in obj.GetComponents())
{
    Type cType = c.GetType();
    if(Attribute.IsDefined(cType, typeof(ECAType)))
    {
        // we found a managed type
        foreach (MethodInfo m in cType.GetMethods())
        {
            ECAActionAttribute[] actions = (ActionAttribute[])
                m.GetCustomAttributes(typeof(ActionAttribute), true);
            foreach (ECAActionAttribute a in actions)
            {
                actionMethodList.Add(a, m);
            }
        }
    }
}
```

Listing 7. Finding all the actions associated with a GameObject instance. We follow a symmetric procedure state variables.

The object-oriented representation of the rule in Listing 3 in the proof-of-concept implementation is the one described in Section 3.3. The reflection-based implementation of the engine requires including into ECAAction references to game objects for the subject and the direct object, while the verb field corresponds to a method reference (i.e., an instance of the MethodInfo class). The event references the player as subject, the pick method as verb and the virtual shoes object as the direct object. The action references the text panel as subject and the shows method as verb. Such structure allows theRuleEngine to find the rule to execute by comparing the ECAAction instance received in the notification with the event field of each ECARule in the list. In addition, it allows executing the actions by invoking the verb method through reflection.

4.3 Adding Configuration Points to a Scene

This section briefly details the operations that a TB should perform to make a generic Unity Scene configurable by a EUDev, transforming it into a template.

The first step is loading the plugin into the Unity project. The plugin consists of a simple Unity prefab, called `ECAKit` that a TB can simply drag among their assets for empowering a scene with the ECALibrary solution. Prefabs in Unity are game objects persisted with all their components and configurations that may be loaded in different scenes.

The second step the TB needs to perform is assigning one of the Unity Components described in Section 4.1 to all VR objects he would like to open for EUDev configurations. Such assignment may also include the library extension with ad-hoc objects or behaviours.

Finally, the TB creates an executable version of the template, performing a standard build of the Unity project. TBs will share the result with the EUDevs for the configuration. No further build is required after creating the template.

4.4 Configuring a Template

This section briefly describes a proof-of-concept interface we implemented to provide EUDevs with support for configuring a template. After a TB creates the template, the changes specified by a EUDev are stored as external assets of the template. They are dynamically loaded at the environment startup or immediately after they are defined. This allows distributing a template as a simple Unity-based application without any build step for the configured environment. The resulting template executable application runs in two modes:

- In the **VR experience** mode, the template loads the EUDev configuration and supports experiencing the resulting VR environment, executing the specified set of rules in response to the user's interactions. The rules and the other external assets defined may differ according to the different configurations, changing the resulting experience. A given template configuration is a bundle of both the template executable and the configuration assets and rules.
- In the **authoring** mode, the EUDev exploits an immersive authoring interface for defining a new configuration or modifying an existing one. Such mode requires a "secret" action sequence for being activated, including a password, which TBs may generate for each template customer. The EUDev creates the rules in such a mode and associates the external assets with placeholder objects.

Launching a template application starts by default in the VR experience mode. To activate the immersive rule authoring interface, the EUDev clicks a button hidden under the wrist of the non-dominant hand. A numeric keypad asks for an access code to avoid unauthorized and unwanted activation of the authoring interface (Figure 3-A). Upon successful authentication of the EUDev, the template starts the authoring mode, and the application shows the interface in Figure 3-B. It contains four buttons: the first allows creating a new rule (Add Rule), the second put the template in the VR experience mode for testing the current rules set (Test Rules), the third shows a panel containing all the rules in the environment (Show All Rules), while the fourth allows selecting custom 3D models for placeholders.

The rule list is a simple panel containing the natural language representation of each rule in the environment. The EUDev can pick one of them for changing it, or s/he can create new ones. In both cases, the editing UI shows the rule detail visualization (Figure 3-C). Each line contains one of the natural language sentences belonging to the rule, according to the language specified in Section 2.2. The initial adverbs identify the rule parts, while the fields in each line allow editing the sub-parts of an action or a condition. We designed the panel trying to minimize possible syntax errors while preserving the readability of the rule.

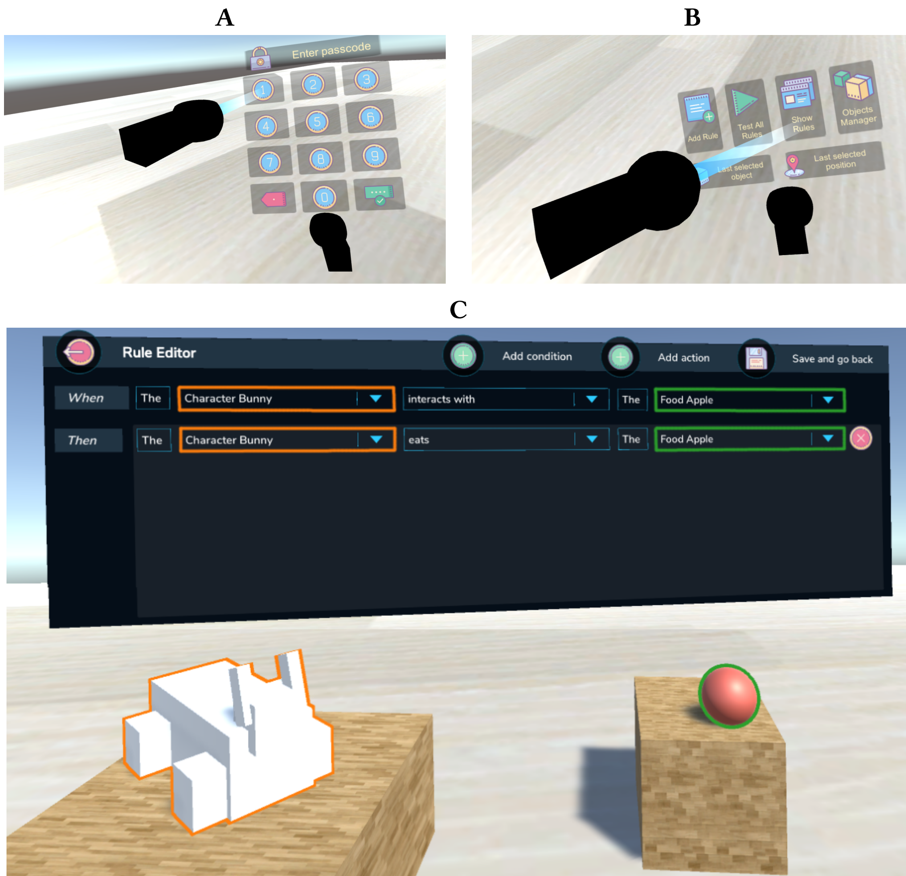

Fig. 3. Rule-editing interface for EUDevs. Part A shows the activation of the authoring mode, part B shows the non-dominant hand menu for activating the different authoring sections, while part C shows the rule editing panel and the highlighting of the involved VR objects.

When starting from an empty rule, the EUDev needs to insert at least an action for the *when* part and an action in the *then* part. So, it includes both the *when* and the *then* keywords with two fields inviting the EUDev to select the subjects of the two actions. Pointing to one of the highlighted fields, the EUDev can choose the subject of the action or the condition simply by pointing it in the virtual environment. Once selected, the object's type and name will appear inside the field. The editor assigns a highlight colour to the VR object, drawing a border both around the object and around the field in the panel (see Figure 3-C). The EUDev can cancel the selection by pressing the X button inside the text field. Once the EUDev has selected the subject, the editor calculates the set of verbs it supports. Then it shows this set as a standard list menu and, depending on the selected verb, the editor will support picking further objects or entering values through pointing or standard input techniques (e.g., virtual keyboards). It will not allow completing an action with objects or values that do not respect the language specification.

Since the possible values for each action or condition part depend on its predecessor, the editor enforces such dependencies when the EUDev changes one of the values s/he entered at a previous step by cleaning all the values that depend on it. For instance, if the user changes the subject after

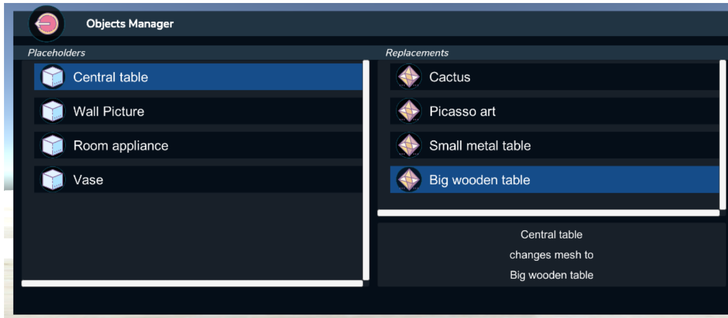

Fig. 4. The UI panel for managing placeholders.

selecting a verb, the editor clears the verb too. The same applies to the object or values, which depend on both the subject and the verb.

A EUDev can also insert additional actions in the *then* part by pressing the add action button in the main menu, creating a new slot below the *then* clause, and starting from the subject selection. Finally, s/he can insert one or more optional conditions by pressing the add condition button in the menu. The first time the EUDev presses this button for a rule, it will add the *if* part between *when* and *then*, and the condition editing will start from picking the subject. The second time will add a list menu for selecting the boolean operator (and/or).

We included a dedicated panel for managing `Placeholders`, which correspond to a virtual object whose appearance cannot be determined while creating the template. For setting the actual content, which means replacing the mesh of an exiting `GameObject`, the EUDev uses the interface in Figure 4. On the left-hand part, the UI shows the list of the existing placeholder in the virtual environment. On the right-hand part, we have a list of meshes, which contains the EUDev specific 3D models. Currently, they must be loaded into a specific sub-folder at the same level of the template executable, but we plan to support online repositories in future versions of the plugin. Assigning a replacement to a placeholder implicitly defines a rule containing an action that we execute at the environment startup without passing from the rule editor. This is a compromise for keeping low the barrier since startup events are hard to understand for non-programmers.

Finally, the UI allows saving a rule once its definition is correct from a syntax point of view. Rules are persisted in a text file, and they are added to those available for the `ECARuleEngine` (or replaced it if we are modifying existing ones), to support the run-time execution.

5 DEVELOPMENT CASE STUDIES

We demonstrate the support provided by the ECALibrary plugin reporting on the development of two sample templates: a virtual showcase and a virtual museum. We show their flexibility in accommodating two different usage scenarios for each template. We created the configuration ourselves for showcasing the plugin implementation.

5.1 Case Study 1: Virtual Shop

5.1.1 Template. The template consists of a large square room, including tables, clothes hangers and shelves where EUDevs can show their products. Figure 5 depicts how the template appears before an EUDev configuration. We placed four tables at the same distance from the corners of the room, with shelves positioned in front of them and mannequins in between. In addition, there are some additional elements supporting the visualization of multimedia content, represented as

Fig. 5. The virtual shop template.

a TV screen. We included a small screen on each table and a bigger one near the wall opposite to the initial position of the visitor. Nearby the bigger mannequins, we placed some buttons for supporting the EUDevs in defining some interaction or animation on their content.

More in detail, the template contains many `ECAObjects` the user can configure:

- All the furniture and light elements in the environment are available for EUDev configurations respectively through the `ECAFurniture` and `ECALight` categories.
- All the light sources in the environment are available for EUDev configurations through the `ECALight` object.
- The TV has a `ECAVideo` interaction.
- Each mannequin is a `ECAMannequin` object.
- There is an audio source in the shape of a radio, which is associated with a `ECASound` behaviour.
- All the areas of interest in the shop (for instance around a table, nearby the mannequins, etc.) have an `ECAInteractable` object for triggering proximity-based interaction.
- On top of each table there is a `ECAText` object that may be used for including additional information on the available items.
- All items on top of the tables, on the shelves, etc. are `ECAPlaceholders`.
- A `ECATimer` for time-based interaction.

5.1.2 Clothing Store. The first sample configuration we deliver for this virtual shop template represents a clothing store. In such a configuration, the placeholders throughout the room have been replaced with a custom clothes model. The large TV on the wall is hidden together with the placeholders on the shelves. The overall idea is to support the inspection of different clothes on top of the mannequins and show advertisements on the production process and quality. In the following, we summarise the supported interactions.

The shop includes a customized advertisement video that plays when the user approaches one of the tables and interacts with it (by pressing a button on the remotes). The environment sets an ambient blue light focused on the table and plays an advertisement on the small TV placed on top of it (see Figure 6-A).

```
when    the character visitor interacts with the furniture table1
then    the light mainLight turns off
        the light tableLight1 turns on
```

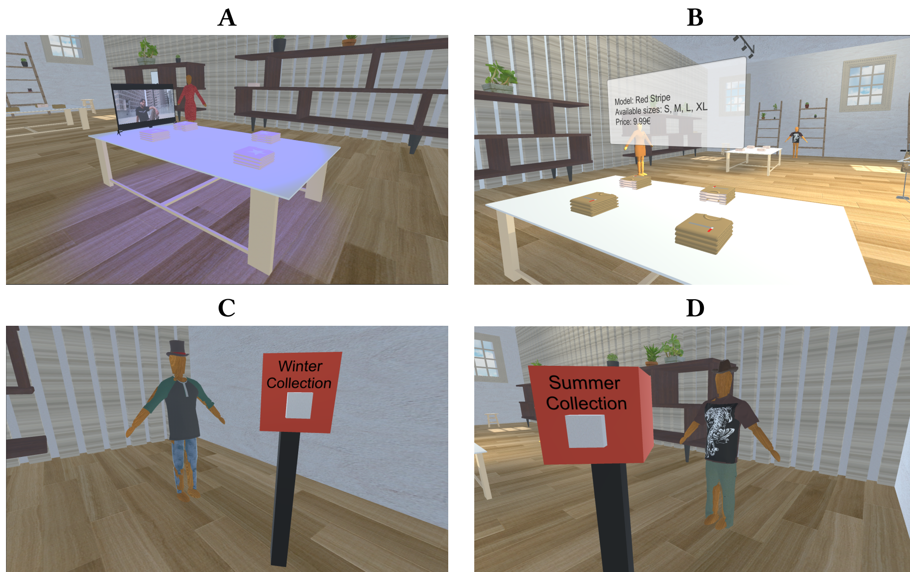

Fig. 6. Sample interactions in the clothes store configuration. A) When the visitor approaches the table, a blue light highlights the table, and the TV monitor plays an advertisement. B) The visitor approaches another table, and the environment shows some price tags. C and D) The mannequin wears a different outfit according to the button pressed.

```
        the video tableAdvertisement1 plays
```

The buttons near the bigger mannequin support the change between the winter and summer collection of clothes, changing the mannequin's dress. The interaction is depicted in Figure 6-C and D. The rule for setting the winter collection is the following:

```
when    the character visitor presses the button bigButton1
then    the mannequin bigMannequin wears the clothing hat
        the mannequin bigMannequin wears the clothing greentshirt
        the mannequin bigMannequin wears the clothing bluejeans
```

In addition, when the visitor approaches one of the clothes in the shop, a price tag appears on top of the nearby items. The resulting interaction is displayed in Figure 6-B. The tag is hidden after 5 seconds.

```
when    the visitor interacts with the table2
then    the text infoPanel2 shows
        the timer timeActions changes duration to 5
        the timer timeActions starts

when    the timer timeActions reaches 0
        then the text infoPanel2 hides
        ... // the other infoPanels hide too...
```

5.1.3 Shoes Store. We created a virtual shoe shop on top of the virtual shop template in the second configuration. The pieces of furniture specifically useful for clothes are hidden, e.g., the racks and the clothes hangers, the mannequins, etc. Shoe models replace the placeholders on the shelves and

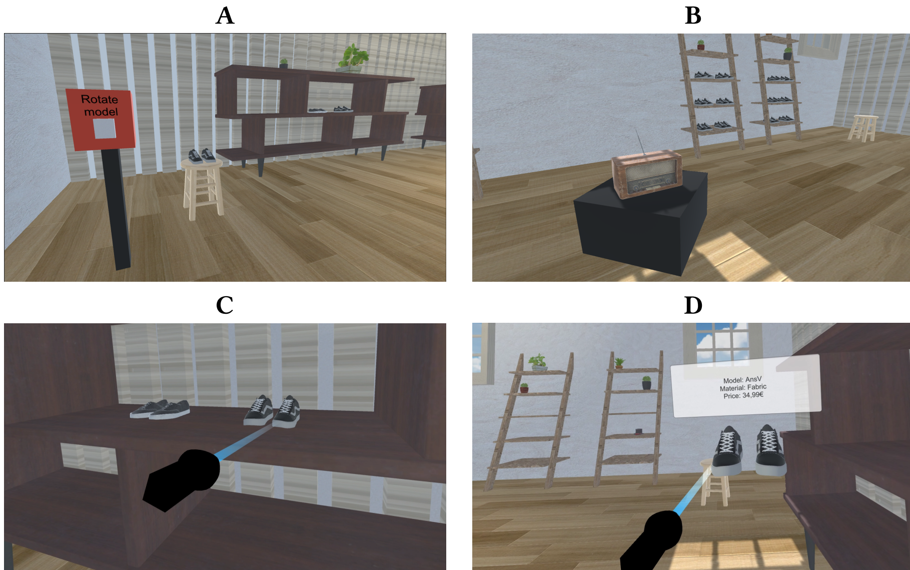

Fig. 7. Sample interactions in the shoe store configuration. A) Pressing the button, the stool will rotate by 30°. B) Interacting with the radio will turn on or off the background music. C and D) When the user manipulates the shoes, above it will appear an informative panel. It disappears as soon as the user releases them.

tables position. The small TVs on top of each table and the large TV on the wall are hidden since this configuration does not use multimedia promotional content. We replaced the placeholder on top of the stool with a custom 3D model of shoes. When the visitor presses the nearby button, the stool rotates together with the shoe model for facilitating the visitor inspection (see Figure 7-A). The associated rule is the following:

```
when    the character visitor presses the button bigButton1
then    the furniture stool rotates by 30 degrees around Y
```

We show the information about the shoe price when the user points them. The environment shows a panel including such pieces of information. The panel closes as soon as the user releases the shoes (Figure 7-C and D). We used the rule defining the first interaction as a sample for describing our approach in Sections 3 and 4.1 (see Listing 3). The rule for the second one is the similar, replacing the *picks* action with *releases* and *shows* with *hides*.

Finally, the visitor can decide whether or not to listen to some background music while visiting the shop. For stopping or playing the music, it is sufficient to approach the radio in the environment and interact with it (see Figure 7-B). Such interaction allows us to show a rule having a condition statement. A similar rule stops the music if it was playing.

```
when    the visitor interacts with the audio radio
if      the audio radio playing is true
then    the audio radio stops
```

5.2 Case Study 2: Virtual Museum

5.2.1 Template. The template scene consists of four square rooms connected by the main entrance having a large skylight that gives natural light to the room. The main room is connected with the other four through open doors. We included a dark parquet floor and white concrete walls

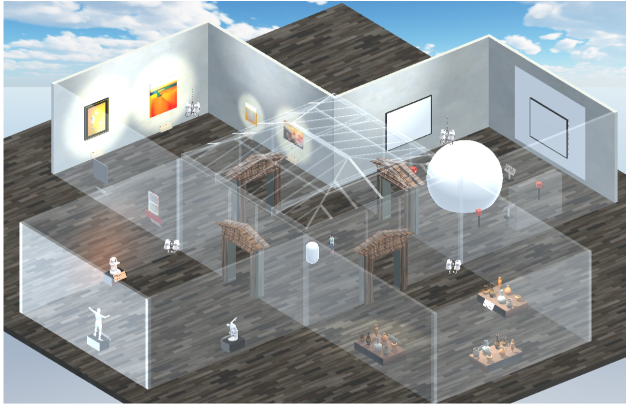

Fig. 8. The virtual museum template (isometric view - walls and ceilings are transparent for clarity).

creating a simple yet elegant environment for placing artworks, which are placeholder elements in the rooms besides the ceiling chandelier. The four rooms contain some sample artworks for providing EUDevs with easy starting points for different museum types: one contains paintings, one pottery, one a collection of sculptures, while the last contains a multimedia room. Figure 8 shows an overview of the template.

The `ECAObjects` included in the template are the following:

- All the rooms in the environment, associated to the `ECABuilding` category, for implementing interactions when the user enters or leaves the room.
- All the pieces of art are `ECAArtworks` in the environment. They are also associated to a `ECAPleaceholder` behaviour for changing their 3D model
- Environmental lights (`ECALight` objects) controlling the illumination settings
- A directional light (`ECALight`) for each artwork dedicated to its highlighting.
- A non-playable human character, playing the role of a virtual museum guide (a `ECAHuman`).
- A set of videos in the multimedia room (`ECAVideo` objects)
- A set of `ECAButtons` for selecting up to 4 different videos in the multimedia room.
- An `ECAAudio` for reproducing music or environmental sounds.
- A `ECAText` and an `ECAImage` for putting information about each artwork.
- A 360° video for showing immersive video contents.

As in the previous case study, we show that different results can be achieved using the same template providing two sample configurations.

5.2.2 Virtual Art Gallery. For the first scenario, we suppose having to promote an art show in the real world. People interested in learning more and visiting the real show can first experience a preview in the virtual environment. When the visitor approaches an artwork, a virtual museum guide will briefly introduce it by voice, an ambient light will fade out while projectors aimed at the artwork will light up.

```
when    the character visitor interacts with the building statueRoom
then    the human guide starts-animation "artwork1Presentation"
        the light globalLight changes intensity to 0
        the light spotLight1 changes intensity to 1
```

The visitor can touch the artwork to show a metal plate with the artwork's name, and by touching the plate, the environment shows a descriptive content represented as an image (see Figure 9).

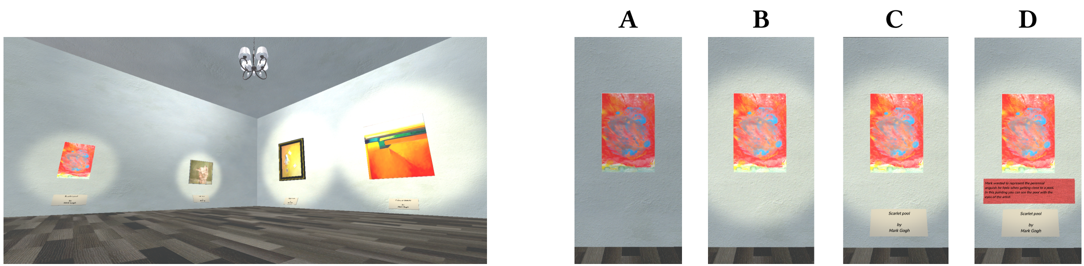

Fig. 9. Left: Each room in the museum has a specific soundtrack that plays when the player enters it, the music stops when the player leaves the room. Right: When the user approaches any artwork, the corresponding plate with information will appear.

Fig. 10. When the user steps at the centre of the room (into the spotlight on the floor), a 360° video starts playing around him. As the user steps out from the centre of the room, the video stops and disappear (the second image shows the outside of the video-globe, the user will be inside it, as shown in the image on the right).

```
when    the character visitor interacts with the art Artwork_1
then    the furniture Artwork_1_plate shows

when    the character visitor interacts with the furniture Artwork_1_plate
then    the image information changes hides
```

5.2.3 Virtual Geography Museum. In this configuration, instead of paintings, the museum shows big pictures of real places, like Niagara Falls or the Grand Canyon, creating a virtual experience for education purposes. When the visitor enters the room with the pictures (the paintings in the template), the environment plays an ambient sound from the place portrayed in the pictures.

```
when    the character visitor interacts with the building paintingRoom
then    the audio backSound plays
```

The visitor can touch the plate with the place's name to trigger a fade out of the lights and play a 360° video. When the video ends, the lights go back to normal and the visitor can continue exploring the museum.

```
when    the character visitor interacts with the furniture plate1
then    the light globalLighting changes intensity to 0
        the audio backSound stops
        the video video360 plays

when    the video video360 ends
then    the light globalLighting increases intensity
        the audio backSound plays
```

In addition, the user can select the 2D video playback in the multimedia room through two buttons. For instance, the button on the left triggers the following rule:

```
when    the character visitor pushes the button videoButton1
then    the video video1 changes source to "sea.mp4"
        the video video1 plays
```

6 DEVELOPER EVALUATION

Besides the validation through the development of the case-studies, we ran a lab study with developers for getting their feedback on the perceived *usefulness* of the overall approach and the *usability* of the Unity plugin we developed.

Procedure. The study consists of five different parts. In the first one, the participants read the study description statement and expressed their informed consent for participating in the study. After that, they filled out an anonymous demographic questionnaire, including questions on their experience level in VR development.

In the second part, we asked them to read a brief description of the roles in the envisioned EUD workflow (see Section 3), how to create a template using the Unity plugin, and how to extend the library. Afterwards, they filled out a questionnaire evaluating the perceived usefulness of the approach and the expected usage difficulty through a set of Likert items. This allowed us to measure the expectations of the developers *before* using the plugin.

In the third part, they executed three development tasks, starting from the scene of the Virtual Shop template. The tasks are the following:

- T1 The participant has to provide the EUDevs with the support for defining the environment's reaction when a push-button in the scene is pressed. This task represents the case in which it is sufficient to add an existing component to define the configuration point of the behaviour.
- T2 The participant has to find the 3D model of a stool into the environment and allow EUDevs to replace it with their own 3D model (e.g., of a chair). This task represents the case in which the TB knows that the EUDevs would like to put their own content instead of the model included in the scene (e.g., their products in a showcase).
- T3 The participant has to extend the set of objects provided by the plugin adding a *key* that can unlock (i.e., make visible) another object. The EUDev must define the object to unlock through a rule. This task represents the case in which the object set is not enough for the TB purposes and requires an extension.

At the end of each task, the participant answered the Subjective Mental Effort Questionnaire (SMEQ) [40] for measuring the effort, while we collected the completion time.

The fourth part consists of filling a post-test questionnaire, including the same Likert items from the second part and assessing differences in the perception *after* the usage. In addition, we asked developers to rate the plugin's *efficiency* and *effectiveness* together with their general *satisfaction.*

The fifth part consists of a debriefing phase, where we discussed with each participant the problems they encountered and the strong points they noticed in the approach.

Participants. Participation in the study was completely voluntary. We contacted former students who graduated from our University and work or study in game and/or VR development. We invited those who responded to the call in our lab for individual sessions. To thank them for their time, we offered a coffee break at the end of the session.

Nine people participated in the study, eight males and one female. Their age ranged from 22 to 37 years old ($\bar{x} = 26.7$, $\tilde{x} = 22$, $s = 4.95$). They had a good education level: 3 a Bachelor Degree, 5 a Master Degree and 1 a PhD. The development experience was also good: 5 participants had 4 to 6

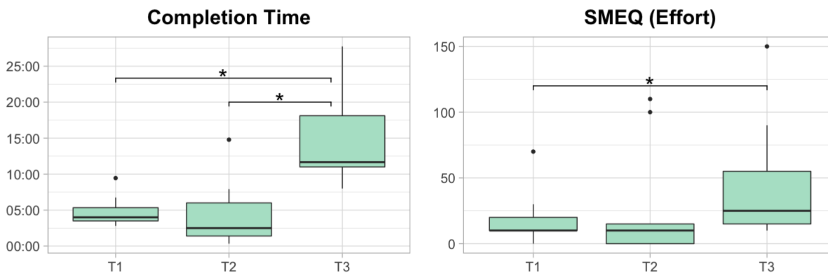

Fig. 11. Task completion time and perceived effort [40]. Highlighted pairs have a significant difference (Wilcoxon signed-rank test, $p < .05$. We used the Bonferroni p-valued adjustement for repeated comparisons).

years experience, 3 from 7 to 10 years and one more than 10 years. They had less experience in game and VR development, but it was adequate for the purpose of the study: 8 from 1 to 3 years and one more than 10 years. All participants were proficient in Unity, but they used also 3D and VR-related tools: 3 used Unreal Engine, 3 A-Frame, 2 Blender, 2 Maya, one 3D Studio Max one Steam VR. We asked them to self-assess their skill in different tasks related to VR development on a 1 to 7 Likert scale. They rated high their ability to define the environment behaviour ($\bar{x} = 5.3$, $\tilde{x} = 6$, $s = 1.94$) and the interaction ($\bar{x} = 4.9$, $\tilde{x} = 5$, $s = 1.83$). Instead they gave an average rating to their 3D object modelling abilities ($\bar{x} = 2.9$, $\tilde{x} = 2$, $s = 1.83$), their expertise in modelling environments ($\bar{x} = 3.0$, $\tilde{x} = 3$, $s = 1.58$) and their skills in composing them ($\bar{x} = 3.8$, $\tilde{x} = 4$, $s = 1.71$).

Results. All participants concluded all tasks, even if one of them required suggestions for properly defining the behaviour of the new object in T3. Figure 11 shows the differences in the completion time and perceived effort (measured through the SMEQ [40] questionnaire) between the tasks. We expected T3 to be the most difficult among the tasks since it requires the development of a dedicated class in C#, and this is clearly confirmed by the results. T1 and T2 took a comparable amount of time, while T3 required longer.

The effort reported by participants through the SMEQ [40] follows a similar trend. T1 and T2 received similar evaluations, and T3 required a higher effort as expected (see Figure 11, right part). According to the labelling of the difficult levels in [40], the median for both T1 and T2 sets this tasks between “Not very hard to do” and “A bit hard to do”, closer to the lower level. T3 is instead near the “A bit hard to do” level. The high standard deviation reflects some difficulties encountered in this task by some participants. We clarify better this point through the debriefing comments.

Such results show that exploiting existing components in the plugin for inserting configuration points requires little additional time and effort. Writing an extension is more difficult but less frequent. Nevertheless, the time and effort required seem reasonable for underlying task complexity.

Following the guidelines for evaluating HCI toolkit research [25], we included questions assessing the perceived usefulness of the approach and its usability in terms of templates development. We collected the participants’ opinions on different statements related to the utility and the difficulty of the approach before and after completing the tasks for highlighting differences between the expectation and the actual experience. Figure 12 (top part) lists the statements we included in the questionnaire and the rating pre (blue) and post-task (red). The two sets of answers are pretty similar. We did not register any significant difference in both the usefulness and the difficulty questions. This means that, overall, the plugin met the participants’ expectations. However, we notice that the post-task assessment of the statement “I will use the plugin for creating templates” follows a decreasing trend, while we notice an increased perception of the overall utility after the

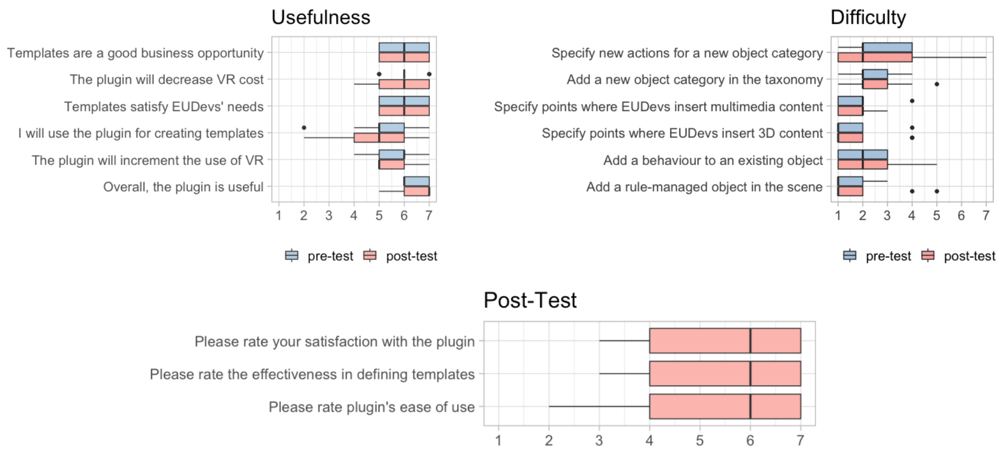

Fig. 12. Participants assessment of the usefulness, difficulty and overall usability of the approach. They assessed the usefulness and the difficulty before and after completing the task. Values are in a 1 to 7 Likert scale.

usage. This means that our participants appreciated the idea and its implementation, but some of them reconsidered a possible direct involvement in the envisioned meta-design process. Among the questions related to the expected and perceived difficulty, we notice a decreasing trend for the statement about the plugin extension. This is related to the difficulties encountered in T3 that we already mentioned, and we will discuss when reporting on the debriefing.

Finally, the items about the overall usability of the approach (see Figure 12, bottom part) received a very good assessment, showing an overall appreciation of the plugin support. They included statements about the ease of use, effectiveness and satisfaction.

Debriefing. We collected the participants' feedback at the end of each session, which is helpful for understanding strengths and weaknesses in our approach and for improving it in the next iterations.

We received both positive and negative comments about the project from all participants and suggestions for improving it. Generally speaking, the project received positive feedback concerning its ease of use, internal structure and taxonomy, usefulness, extensibility and reusability. One participant stated that the idea could be very useful for avoiding the redefinition of standard behaviours in an end-user development setting and even in VR environments created by experts. A second participant, who was already familiar with ECA rules, underlined the ease of use of the component-based approach for defining configuration points. In addition, he considers successful the representation of the rule in natural language for involving users without programming experience. Another participant highlighted that the TB requires skills spread over different roles in a typical game development organization, including game developers, level designers and 3D artists. We agree in general with this comment, and we would need further research for understanding whether the plugin is effective across these three roles, especially considering people acquainted only with 3D modelling.

Negative feedback comments expressed some confusion in the taxonomy structure and a perceivable initial learning curve to get used with the object categorization. Indeed, some taxonomy elements have names recalling similar concepts, and the difference between them needs some learning effort. For instance, a participant spent some time in grasping the difference between

the *Behaviour* (base class for all behaviours identifiable across the different categories of objects), *Interactable* (a *Behaviour* associated with something a character can interact with) and *Button* (a perceivable object that a character can push), since they looked similar to him (e.g., a button should be interactable and it must have a behaviour). In general, assigning names is not easy. But it is also true that we define similar configurations in different ways in many toolkits. In ours, something that logically corresponds to a button may be defined through the object in the taxonomy, but also assigning an *Interactable* behaviour to objects that belong to a different category (e.g., a chair). The difference is, as explained in Section 3.1, the physical appearance. If it resembles something that we can push, we consider it a *Button*, otherwise it belongs to a different object category, and we associate an *Interactable* behaviour. Both ways are correct depending on the situation.

Task 3 also highlighted difficulties in understanding the extension process. In particular, the information included in the code documentation and the descriptive material we created for the test was not enough for one participant. He expected some startup code, such as the skeleton of a dummy extension, to fill up with custom code. He found the material we provided particularly unhelpful, and he stated that reconstructing the process from such descriptions required too much effort. We agree that including additional information and startup code would ease the task for TBs, and we will complete the documentation in further versions of the plugin. However, the issue seems more related to the documentation provided than the EUD approach itself. Therefore, even if the difficulty is relevant, we do not consider it a barrier to using the plugin.

Some participants suggested to improve the current documentation with more examples for our future developments, even in the natural language rules part. This could help the TB to understand how a EUDev specifies rules and how s/he can use the plugin. One of them proposed creating a debug mode to test the rule set in play mode, reporting the TB when an action is triggered using a console in the game tab of Unity. We agree that this is useful, but it is currently beyond the scope of the proof-of-concept implementation. Another participant requested more help from the development environment while formalizing new actions for the EUDev to guarantee the expected syntax for the rules through an auto-completion feature. Again, it would be very nice to have it in a product, but it goes beyond our proof-of-concept.

7 DISCUSSION

In this section, we summarise the paper's contribution, the lesson learned and the limitations of the work. We introduced an end-user development approach for supporting the configuration of a VR environment by users without skills in programming and/or 3D modelling. The approach leverages on meta-design [2, 11, 15], introducing the hybrid role of the EUDev, that is, both a designer and a user. The EUDev configures the templates created by TB, providing customised VR experiences for users who consume the final VR content. Having defined the roles, we presented an engineering solution for supporting the EUD approach, based on natural language Event-Condition Action rules, defined by EUDevs. We describe the high-level actions on top of a taxonomy defining categories of virtual objects. We also provide a solution for executing rules at runtime without rebuilding the VR environment after the configuration by EUDevs.

Besides the overall approach, we discussed a proof-of-concept implementation in Unity3D. Its peculiarity relies on extensibility, which mitigates possible shortcomings in the taxonomy of 3D object modelling. By including some custom annotations, TBs can provide further actions and object categories to EUDevs.

To demonstrate our claims, we provided a validation of the approach. The plugin discussed in the paper is a toolkit enabling TB to create templates. Properly validating a toolkit in HCI research is a debated topic in the community. An excellent summary of the documented techniques in the literature is available in [25]. Following their categorization, we validated through demonstration

and a user study. The demonstration, discussed in Section 5, shows that the current proof-of-concept implementation of the approach can support the definition of templates for two different use cases, providing support for creating substantially different experiences through rules. The user study shows that potential TBs consider the plugin (and the overall approach) useful and usable, requiring a reasonable effort for defining configuration points. From the discussion with participants, we identified points where they require improvement, especially in the documentation of the extension mechanism.

The paper has, of course, limitations. First, we focused on the approach definition and the support provided to TBs. In the current state of our research, we cannot claim the usability of the rule language, the effort required by EUDevs for configuring a VR environment or the effectiveness of the immersive rule-authoring interface. While we are positive that the rule-based approach may have comparable effectiveness documented in other domains (e.g., for IoT, see Section 2.2), further research is required. In particular, we need to investigate the differences in rule authoring through desktop or immersive interfaces. In an immersive mode, EUDevs would avoid authoring and test cycles, requiring to put on and take off the HMD. In contrast, creating many rules in the immersive mode may be tedious, so there is space for hybrid solutions. However, no configurations are possible without the VR environment support, and this paper is the first step towards that goal.

A second point to discuss, which is somehow connected with the previous, is the limited scope of the object taxonomy. Our goal was not to provide complete modelling of all the possible categories of VR objects but only include commonly used definitions that may be reused across different templates, using our experience in creating VR environments and looking at existing categorizations. This allows TBs to start with a reasonable base. Otherwise, they would need to redefine the whole rule language. However, we expect different domains to have different needs in expressing rules. So, we included an extension mechanism that allows such adaptation. We are persuaded that searching for a complete taxonomy in all templates is not a realistic goal. We may find arguments in favour of this having a deeper look at the literature references discussed in Section 2.2, where it is clear that we have no shared definition even of the rule format, which are adapted in different variants for different domains.

Finally, the last limitation that requires discussion is the small number of participants in the user study. This is mainly due to the difficulty in recruiting people corresponding to the TB profile, who are experts busy in their own development tasks. In addition, for keeping short the time required for each evaluation session, we included very specific tasks and not, e.g., creating an entire template, again to make the recruitment feasible. We are aware that more in-depth evaluations would be required, even outside the lab environment, but, as discussed in [25], this is seldom possible in evaluating research toolkit prototypes.

8 CONCLUSION AND FUTURE WORK

This paper introduced an end-user development approach for fostering a VR environment's configuration by end users, exploiting rules in natural language and relying on templates created by experts. The approach advances state of the art going beyond the construction of static VR scenes for EUDevs. We provided insights on the configuration ceiling through two case studies and on the usability and utility as perceived by template developers. Future work will focus on defining an effective interface supporting EUDevs in creating rules, exploiting conversational interaction and a more usable graphical representation. In addition, we plan to assess the rule-language definition through further evaluations with end users.

ACKNOWLEDGMENTS

The work was partially supported by the ECARules4All project, which has received funding from the European Union's Horizon 2020 research and innovation programme through the XR4All H2020 project with grant agreement No 825545.

REFERENCES

[1] Telmo Adão, Luís Pádua, Miguel Fonseca, Luís Agrellos, Joaquim J. Sousa, Luís Magalhães, and Emanuel Peres. 2018. A rapid prototyping tool to produce 360° video-based immersive experiences enhanced with virtual/multimedia elements. *Procedia Computer Science* 138 (2018), 441–453. https://doi.org/10.1016/j.procs.2018.10.062 CENTERIS 2018 - International Conference on ENTERprise Information Systems / ProjMAN 2018 - International Conference on Project MANagement / HCist 2018 - International Conference on Health and Social Care Information Systems and Technologies, CENTERIS/ProjMAN/HCist 2018.

[2] Carmelo Ardito, Paolo Buono, Maria Francesca Costabile, Rosa Lanzilotti, Antonio Piccinno, and Li Zhu. 2015. On the transferability of a meta-design model supporting end-user development. *Universal Access in the Information Society* 14, 2 (01 Jun 2015), 169–186. https://doi.org/10.1007/s10209-013-0339-7

[3] Raffaele Ariano, Marco Manca, Fabio Paternò, and Carmen Santoro. 2022. Smartphone-based augmented reality for end-user creation of home automations. *Behaviour & Information Technology* 0, 0 (2022), 1–17. https://doi.org/10.1080/0144929X.2021.2017482

[4] Barbara Rita Barricelli and Stefano Valtolina. 2017. A visual language and interactive system for end-user development of internet of things ecosystems. *Journal of Visual Languages & Computing* 40 (2017), 1–19. https://doi.org/10.1016/j.jvlc.2017.01.004 Semiotics, Human-Computer Interaction and End-User Development.

[5] 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

[6] Will Brackenbury, Abhimanyu Deora, Jillian Ritchey, Jason Vallee, Weijia He, Guan Wang, Michael L. Littman, and Blase Ur. 2019. How Users Interpret Bugs in Trigger-Action Programming. 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, 1–12. https://doi.org/10.1145/3290605.3300782

[7] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 147–158. https://www.usenix.org/conference/atc18/presentation/celik

[8] Mengyu Chen, Marko Peljhan, and Misha Sra. 2021. EntangleVR: A Visual Programming Interface for Virtual Reality Interactive Scene Generation. In *Proceedings of the 27th ACM Symposium on Virtual Reality Software and Technology* (Osaka, Japan) *(VRST '21)*. Association for Computing Machinery, New York, NY, USA, Article 19, 6 pages. https://doi.org/10.1145/3489849.3489872

[9] 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

[10] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2019. A high-level semantic approach to End-User Development in the Internet of Things. *International Journal of Human-Computer Studies* 125 (2019), 41–54. https://doi.org/10.1016/j.ijhcs.2018.12.008

[11] M.F. Costabile, D. Fogli, P. Mussio, and A. Piccinno. 2005. A meta-design approach to end-user development. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. 308–310. https://doi.org/10.1109/VLHCC.2005.7

[12] Joëlle Coutaz and James L. Crowley. 2016. A First-Person Experience with End-User Development for Smart Homes. *IEEE Pervasive Computing* 15, 2 (2016), 26–39. https://doi.org/10.1109/MPRV.2016.24

[13] Giuseppe Desolda, Carmelo Ardito, and Maristella Matera. 2017. Empowering End Users to Customize Their Smart Environments: Model, Composition Paradigms, and Domain-Specific Tools. *ACM Trans. Comput.-Hum. Interact.* 24, 2, Article 12 (apr 2017), 52 pages. https://doi.org/10.1145/3057859

[14] 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.

[15] G. Fischer, E. Giaccardi, Y. Ye, A. G. Sutcliffe, and N. Mehandjiev. 2004. Meta-Design: A Manifesto for End-User Development. *Commun. ACM* 47, 9 (sep 2004), 33–37. https://doi.org/10.1145/1015864.1015884

[16] Fungus Games. 2020. *Fungus*. Retrieved January 3, 2021 from https://fungusgames.com
[17] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. 1995. Elements of Reusable Object-Oriented Software. *Design Patterns. massachusetts: Addison-Wesley Publishing Company* (1995).
[18] 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.
[19] Giuseppe Ghiani, Marco Manca, Fabio Paternò, and Carmen Santoro. 2017. Personalization of Context-Dependent Applications Through Trigger-Action Rules. *ACM Trans. Comput.-Hum. Interact.* 24, 2, Article 14 (apr 2017), 33 pages. https://doi.org/10.1145/3057861
[20] Juan Manuel González-Calleros, Jean Vanderdonckt, and Jaime Muñoz-Arteaga. 2009. A Structured Approach to Support 3D User Interface Development. In *2009 Second International Conferences on Advances in Computer-Human Interactions*. 75–81. https://doi.org/10.1109/ACHI.2009.14
[21] Google. 2017. *Google Poly*. Retrieved January 3, 2021 from https://poly.google.com/
[22] T.R.G. Green and M. Petre. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131–174. https://doi.org/10.1006/jvlc.1996.0009
[23] Kai-Hsiang Hsu, Yu-Hsi Chiang, and Hsu-Chun Hsiao. 2019. SafeChain: Securing Trigger-Action Programming From Attack Chains. *IEEE Transactions on Information Forensics and Security* 14, 10 (2019), 2607–2622. https://doi.org/10.1109/TIFS.2019.2899758
[24] Zapier Inc. 2022. Zapier. https://zapier.com [Online; accessed 17-February-2022].
[25] 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
[26] 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
[27] 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
[28] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F. Karlsson, Dongmei Zhang, and Feng Zhao. 2016. Systematically Debugging IoT Control System Correctness for Building Automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments* (Palo Alto, CA, USA) *(BuildSys '16)*. Association for Computing Machinery, New York, NY, USA, 133–142. https://doi.org/10.1145/2993422.2993426
[29] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. 2006. End-User Development: An Emerging Paradigm. In *End User Development*, Henry Lieberman, Fabio Paternò, and Volker Wulf (Eds.). Springer Netherlands, Dordrecht, 1–8. https://doi.org/10.1007/1-4020-5386-X_1
[30] Adam Martin. 2007. Entity Systems Are the Future of MMOG Development–Part 1. *T= Machine* (2007). http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/
[31] Paulo R. C. Mendes, Álan L. V. Guedes, Daniel de S. Moraes, Roberto G. A. Azevedo, and Sérgio Colcher. 2020. An Authoring Model for Interactive 360 Videos. In *2020 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*. 1–6. https://doi.org/10.1109/ICMEW46912.2020.9105958
[32] Zeno Menestrina and Antonella De Angeli. 2017. *End-User Development for Serious Games*. Springer International Publishing, Cham, 359–383. https://doi.org/10.1007/978-3-319-60291-2_14
[33] Mario Montagud, Pilar Orero, and Anna Matamala. 2020. Culture 4 all: accessibility-enabled cultural experiences through immersive VR360 content. *Personal and Ubiquitous Computing* 24, 6 (01 Dec 2020), 887–905. https://doi.org/10.1007/s00779-019-01357-3
[34] Mozilla. 2022. Mozilla Hubs. https://hubs.mozilla.com [Online; accessed 17-February-2022].
[35] Mozilla. 2022. Mozilla Spoke. https://hubs.mozilla.com/spoke [Online; accessed 17-February-2022].
[36] 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*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376637
[37] 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
[38] Ottifox. 2018. *Ottifox*. Retrieved January 3, 2021 from https://ottifox.com/index.html
[39] Fabio Paternò and Carmen Santoro. 2019. End-user development for personalizing applications, things, and robots. *International Journal of Human-Computer Studies* 131 (2019), 120–130. https://doi.org/10.1016/j.ijhcs.2019.06.002 50 years of the International Journal of Human-Computer Studies. Reflections on the past, present and future of human-centred technologies.

[40] Jeff Sauro and Joseph S. Dumas. 2009. Comparison of Three One-question, Post-task Usability Questionnaires. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, MA, USA) *(CHI '09)*. ACM, New York, NY, USA, 1599–1608. https://doi.org/10.1145/1518701.1518946

[41] 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

[42] Tuukka M. Takala. 2014. RUIS: A Toolkit for Developing Virtual Reality Applications with Spatial Interaction. In *Proceedings of the 2nd ACM Symposium on Spatial User Interaction* (Honolulu, Hawaii, USA) *(SUI '14)*. Association for Computing Machinery, New York, NY, USA, 94–103. https://doi.org/10.1145/2659766.2659774

[43] Andrei Torres, Christopher Carmichael, William Wang, Matthew Paraskevakos, Alvaro Uribe-Quevedo, Paul Giles, and Jamie Lee Yawney. 2020. A 360 Video Editor Framework for Interactive Training. In *2020 IEEE 8th International Conference on Serious Games and Applications for Health (SeGAH)*. 1–7. https://doi.org/10.1109/SeGAH49190.2020.9201707

[44] Company Tvori. 2020. Tvori. https://tvori.co/ [Online; accessed 17-February-2022].

[45] 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

[46] Vv.Aa. 2021. *Virtual Reality Market Research Report*. Technical Report. Fortune Business Insights.

[47] 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

[48] 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.

[49] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. 2019. AutoTap: Synthesizing and Repairing Trigger-Action Programs Using LTL Properties. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 281–291. https://doi.org/10.1109/ICSE.2019.00043

[50] Lei Zhang and Steve Oney. 2020. *FlowMatic: An Immersive Authoring Tool for Creating Interactive Scenes in Virtual Reality*. Association for Computing Machinery, New York, NY, USA, 342–353. https://doi.org/10.1145/3379337.3415824

A APPENDIX: VR OBJECT TAXONOMY

In this appendix, we describe the high-level types included in the VR object taxonomy for building environments templates. The appendix completes the general overview provided in Section 3.1 with a detailed description of the different categories included in the taxonomy. We describe the state of *Objects* and *Behaviours* using a simplified set of base types, summarised in Table 2.

Objects and *Behaviours* have a set of attributes depicting the state and a set of actions representing the operations they can perform (or receive, in case of passive actions). In the following, we list the state description and the actions for each one of the elements in the taxonomy.

An *Object* can be a *Character*, a *Scene*, an *Environment*, a *Prop*, a *Vehicle* or an *Interaction*. In general, each element belonging to the *Object* category has a *position* attribute, and two flags: one for controlling its visibility and the second for setting it as active or not. In this way the EUDev controls whether the object is visible or not and if someone or something can interact with it.

We use a *Character* object to represent an animal, a humanoid, a robot or a generic creature. A character can be autonomous or controlled by the user of the VR environment (the *playing* flag).

A Character, can be further subdivided in several sub-categories, according to the *Animal* or *not-Animal* classification. Figure 13 shows the actions and the state variables associated with the *Character* sub-categories.

Environment objects represent inanimated elements that configure the VR scene, like buildings, vegetation etc., or elements that decorate or furnish a room. Figure 14 shows the actions and the state variables associated with the Environment sub-categories.

In the *Prop* category, we represent generic objects that can be placed in a scene and manipulated by characters. We have several possible sub-categories in this case. However, in our definition, we decided to abstract the representation keeping the groups rather abstract, while delegating a more precise indication of the object type to type aliasing or extensions. Figure 15 depicts the categorization. We provided deeper modelling for the *Weapons* given their fundamental role in most video games, which we expect will also replicate in EUD VR environments.

It is worth pointing out that *Prop* objects contain actions that belong to them from a software point of view, while the EUDev would rather see them as performed by a character. We call such actions *passive*. For instance, please consider a *Clothing* object. Keeping the implementation of the

EUDev Type	Definition
Boolean	A boolean value. We support equivalent labels for true and false (e.g., yes/no, on/off etc.).
Color	A pair including the color name and its hex-code
Float	A number having decimal places
Integer	An integer number.
Position	Coordinates of a position in the 3D world.
Path	An ordered list of positions in the 3D world.
Identifier	A human-readable representation of a reference to an object in the scene. We will render it in natural language using the pair *<Type, Name>* (e.g., the Robot Daitarn-3).
Rotation	Euler angles in degrees.
Text	A string of text.
Time	A time interval (days, hours, minutes, seconds).

Table 2. EUDev data types for the object status variables

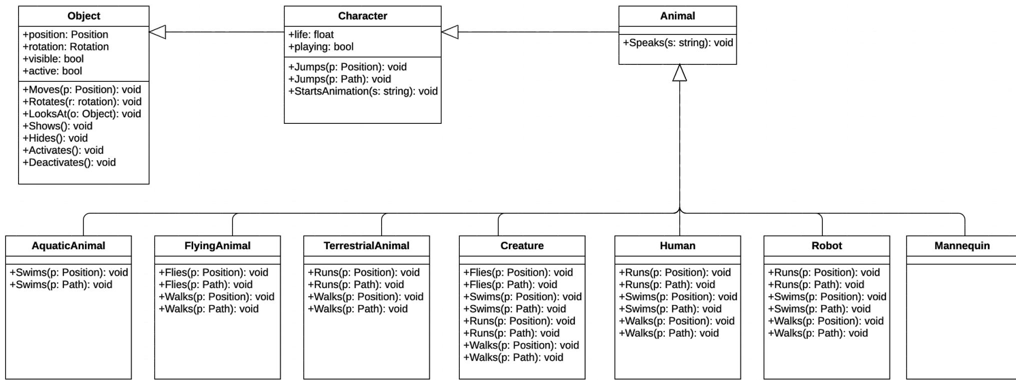

Fig. 13. Class diagram for the Character sub-category.

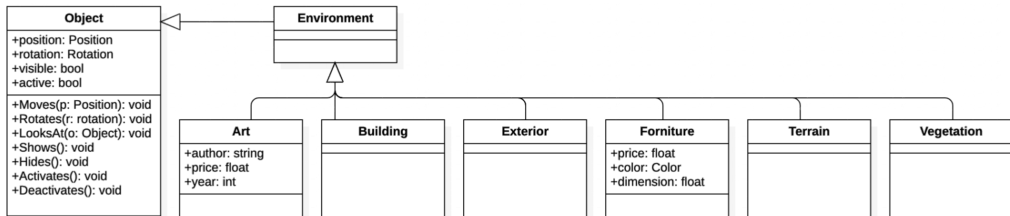

Fig. 14. Class diagram for the Environment sub-category.

wear action in the Clothing and not in the Character class, allows decoupling the basic actions of a character from those involving other objects. Otherwise, we would break the separation of concerns principle, and the character would become the class defining the large majority of the system. However, from the EUDev point of view, the best action description in natural language is "*the character wears the clothing*" and not "*the clothing is weared by the character*". So, passive actions in the class representation use the parameter as the subject of the sentence in natural language (see Section 3.2).

A *Scene* represents a setting in the VR environment a user can visit entirely in a continuous way, i.e., without teleportation. We can associate it to a videogame level: when the level changes, the scene changes too. It defines passive actions for entering and leaving the scene.

The *Vehicle* category represents, in an abstract way, all the vehicles TB can place in scene. We distinguish them according to the element they support travelling on (air, land, sea, space). Figure 16 depicts the categorization.

The *Interactions* category contains all the elements allowing some interaction with the scene and its objects. The main point that distinguishes them from the *Behaviour* elements is that a typical user would perceive them as physical entities of their own, which would exist independently from other objects. Instead, a behaviour adds the ability of e.g., counting, opening or highlighting *other* objects. For instance, we consider two multimedia items such as an advertisement video and a song played by a juke-box. We model the advertisement as a *Video* interactive element (i.e., belonging to the interaction category) since the user of the VR environment perceives the 2D plane projecting

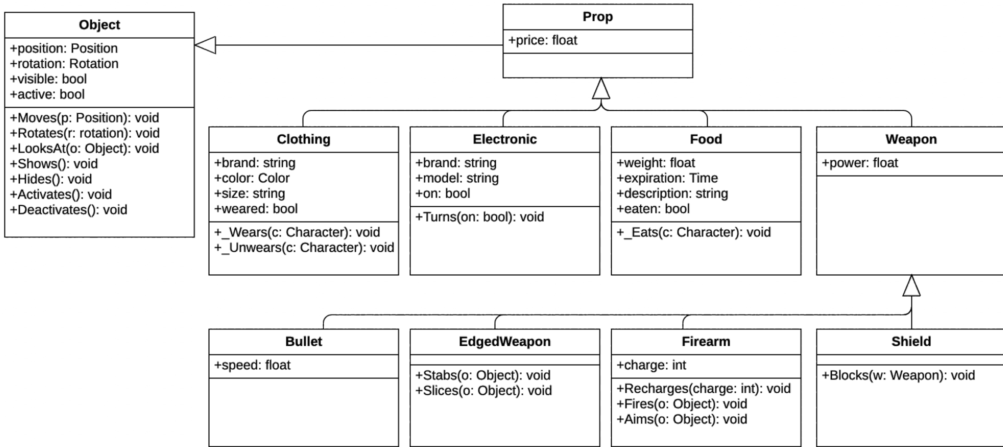

Fig. 15. Class diagram for the Prop sub-category. Actions beginning with an underscore are passive.

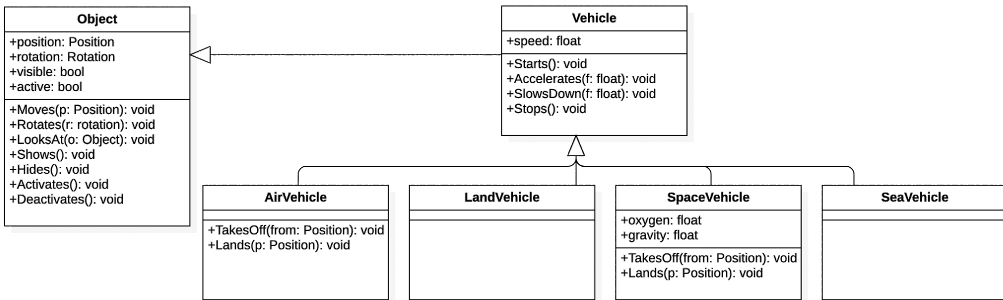

Fig. 16. Class diagram for the Vehicle sub-category.

the video. The song instead has no physical presence of its own, but the user associates it to the juke-box, which is its source and the object modified by the *Sound* behaviour. In this way, a TB can model objects having multiple behaviours, for instance, creating a juke-box that plays songs and highlights when the user interacts with it. Figure 17 lists all the interaction objects and the associated actions.

Given their lack of physical presence of their own, the *Behaviour* does not extend *Object*. Figure 18 depicts the list of available behaviours and the actions they add to the associated objects. It is worth pointing out that adding a behaviour to an object increases the number of actions that the instance supports in the description of the rules in natural language (see Section 3.2).

Received February 2022; accepted April 2022

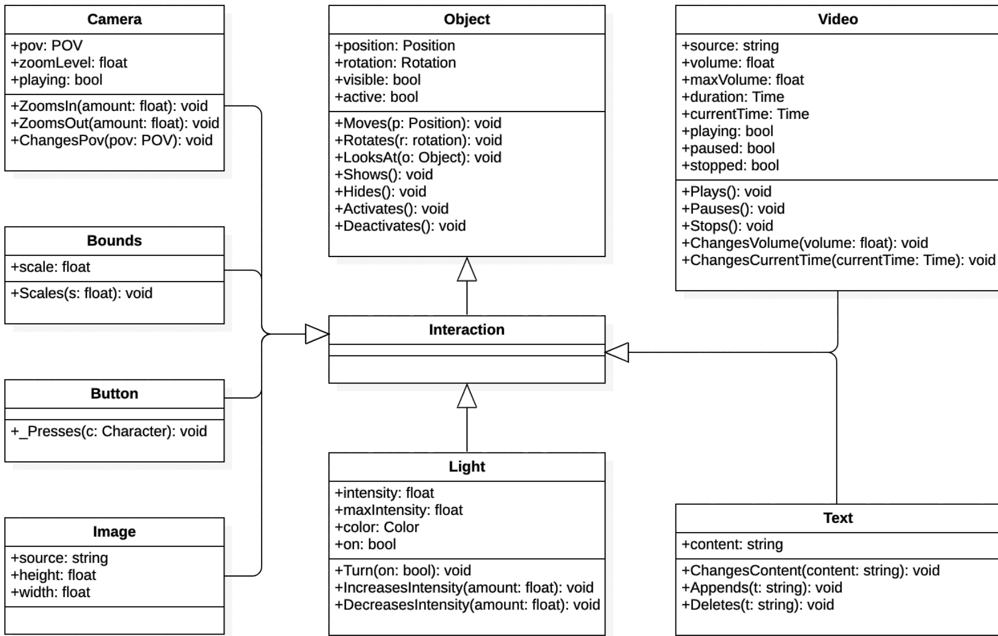

Fig. 17. Class diagram for the Interaction sub-category.

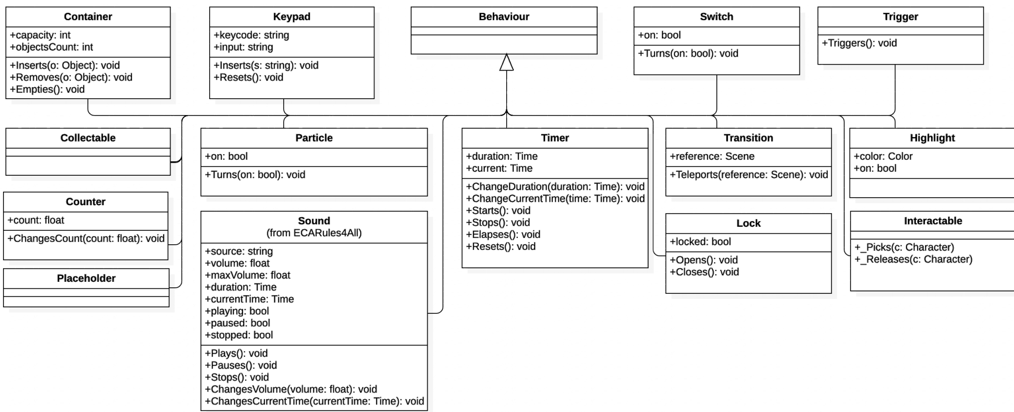

Fig. 18. Class diagram for the Behaviour category.