XRINTTEST: An Automated Framework for User Interaction Testing in Extended Reality Applications

Ruizhen Gu
School of Computer Science
University of Sheffield
Sheffield, UK
rgu10@sheffield.ac.uk

José Miguel Rojas
School of Computer Science
University of Sheffield
Sheffield, UK
j.rojas@sheffield.ac.uk

Donghwan Shin
School of Computer Science
University of Sheffield
Sheffield, UK
d.shin@sheffield.ac.uk

***Abstract*—Extended Reality (XR) technologies offer immersive user experiences across diverse application domains, presenting unique testing challenges due to their spatial interaction paradigms. While existing works test XR applications through scene navigation and interaction triggering, they fail to synthesise realistic spatial input via specialised XR devices, such as 6 degrees of freedom controller gestures, that are essential for modern XR user experiences. To address this gap, we present XRINTTEST, an automated testing framework for Unity-based XR applications. XRINTTEST starts by constructing an XR User Interaction Graph that models interaction targets and required events. Leveraging this graph, it then automatically explores the XR scene under test and generates user interactions. We evaluated XRINTTEST on XRBENCH3D, a novel benchmark comprising seven XR scenes containing 367 distinct 3D user interactions. XRINTTEST shows great effectiveness, achieving 97% coverage of trigger and grab interactions across all scenes, 9x more effective and 5x more efficient than random exploration, while detecting runtime exceptions and functional defects. We open-sourced our tool and dataset at https://github.com/ruizhengu/XRintTest and https://github.com/ruizhengu/XRBench3D, respectively. A video demo is available on YouTube at https://youtu.be/K0Q6waE47Us.**

***Index Terms*—Extended Reality, Software Testing, 3D Interaction, Model-based Testing**

I. INTRODUCTION

Extended Reality (XR) encompasses Augmented, Mixed, and Virtual Reality (AR, MR, and VR, respectively). XR applications (XR apps) present immersive digital environments with interactive content. In recent years, XR has gained significant traction in various domains, e.g., engineering [1].

Functional testing XR apps prioritises *functionality* and *user interaction* above other non-functional test targets such as cybersickness [2, 3]. Current approaches, however, face critical limitations in testing XR user interactions: (i) inability to synthesise user spatial interactions from XR input devices, (ii) lack of systematic methods to model user interactions, and (iii) exclusive focus on simplified interaction types (i.e., 2D interactions like mouse clicks) [4, 5]. These gaps are particularly problematic given modern XR applications' reliance on 6 degrees of freedom (DoF) controllers and gesture-based interactions, which enable realistic manipulation of virtual objects in 3D space (e.g., grab and move a virtual ball).

To address the challenges, this tool paper presents XRINTTEST, the first automated framework for testing user interactions in XR apps. It first constructs *XR User Interaction Graph* (*XUI graph*), an abstraction of interactions within an XR scene. The graph represents interactive objects as nodes (e.g., a ball), the flow of interaction execution as directed edges, and interaction events (e.g., grab) as edge labels. It then automatically explores the scenes to *activate* interactions. Interaction activations are defined as successfully initiating inputs (e.g., pressing the grab button on controllers) on interactable objects, triggering their required interaction logic, and fully exercising their intended functionality. XRINTTEST enables developers to perform end-to-end testing to validate specific XR user interactions, reducing the manual testing burden associated to relying on physical devices like HMDs.

Unlike existing tools, XRINTTEST targets 3D interactions in XR apps, specifically the fundamental interaction types of *trigger* and *grab*, as well as their combinations. To evaluate our approach, we construct XRBENCH3D, a novel benchmark dataset comprising seven XR scenes with 367 3D interactions from six open-source apps. XRINTTEST demonstrates remarkable effectiveness and efficiency in covering trigger and grab interactions, achieving 97% across all subject scenes and reaching about 100% coverage within two minutes for most scenes. Compared to a random testing baseline, XRINTTEST yields approximately 9x higher effectiveness and 5x higher efficiency. Additionally, XRINTTEST can successfully detect both runtime exceptions and non-exception interaction issues.

To summarise, XRINTTEST is the first automated testing framework for comprehensive spatial user interactions in XR apps. By open-sourcing our tool and dataset, we hope that more researchers and practitioners can join and contribute to the emerging domain of XR app testing.

II. BACKGROUND AND RELATED WORK

A. XR Development with Unity

Unity XR projects are organised by means of *scenes*, each containing fundamental entities named *GameObjects* (GOs for short). Developers can extend GOs' functionality by configuring *components* and attaching *custom scripts* to define GO's specific behaviours and interaction patterns.

The XR Interaction Toolkit[1] (XRI) is Unity's development framework for XR experiences. It encompasses three primary features: 3D Interaction, UI Interaction, and Locomotion. The 3D interaction system, specifically, is essential for spatial user interaction. It comprises two script-based components attached to GOs: (i) *Interactors* handle user input and initiate interactions with interactable objects, and (ii) *Interactables* respond to interactors, defining specific behaviours for interactions. For instance, a controller interactor can grab a ball object with XRI's `XR Grab Interactable` component.

B. XR Scene Testing

Scene testing is a type of testing that focuses on exploring XR scenes through two primary tasks: scene navigation and interaction event triggering [3]. Wang et al. pioneered VR testing with VRTEST [4] and VRGUIDE [5], the first tools for automated VR scene testing. These tools employ techniques like pure random and greedy-based exploration. However, these approaches are designed and evaluated based on simplified VR projects that exclusively rely on basic interaction mechanisms such as triggering UI events. XRINTTEST addresses this limitation by synthesising spatial interactions through specialised XR input methods, such as hand-held controllers that enable realistic 3D interactions.

III. DATASET

We constructed the XRBENCH3D dataset using open-source XR projects built with XRI, the arguably most widely adopted framework for implementing XR interactions in Unity. We collected 13 candidate projects from two representative sources: GitHub and Unity Asset Store[2] by querying with XR-related keywords: "XR", "VR", "AR", and "MR". Upon manual inspection, we identified seven distinct XR scenes spanning six free and self-contained XR projects, with licenses that make them suitable as open-source benchmarks. Since the current implementation of XRINTTEST focuses on 3D interactions, we ensured the included projects all demonstrate XR-specific 3D interaction mechanics, such as manipulating virtual objects through hand-held controllers and gestures. Additionally, we exclude locomotion and UI interaction components through careful manual curation to make XRBENCH3D dedicated to 3D interaction testing.

The resulting dataset contains 367 distinct interactions in total. We categorised them into three groups: *trigger* (49 interactions, 13%), *grab* (230, 63%), and *others* (88, 24%). The *trigger*, *grab* interactions align with the *trigger* and *grab* buttons on XR controllers (II-A). The interactions in the *others* groups are primarily customised by developers to meet specific requirements or interaction patterns within the projects.

XRBENCH3D is carefully adapted with minimal modifications to ensure compatibility with the latest versions of Unity (6000.0.45f1) and XRI (3.1.1). The dataset is open-sourced at https://github.com/ruizhengu/XRBench3D.

[1]https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@3.1
[2]https://assetstore.unity.com/

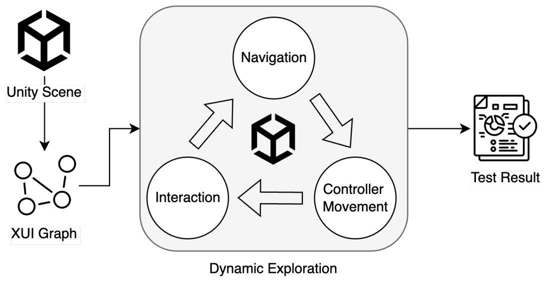


Fig. 1: XR User Interaction Testing with XRINTTEST

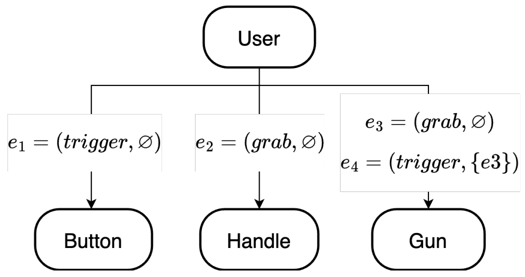

User
$e_1 = (trigger, \varnothing)$
$e_2 = (grab, \varnothing)$
$e_3 = (grab, \varnothing)$
$e_4 = (trigger, \{e3\})$
Button
Handle
Gun

Fig. 2: XR User Interaction Graph Example

IV. FRAMEWORK

Fig. 1 shows the workflow of XRINTTEST. It consists of three major steps: XUI graph construction, dynamic scene exploration, and test result checking. The framework is provided with detailed instructions and automated scripts to check and install the required dependencies. The framework is open-sourced at https://github.com/ruizhengu/XRintTest.

A. XR User Interaction Graph Construction

This step builds the XUI graph from the scene under test (ScUT). Fig. 2 shows an XUI graph example, depicting the XUI events within a scene, containing one interactor *User* and three interactables *Button*, *Handle*, and *Gun*.

The XUI graph construction is implemented as a `EditorWindow` script that enables users to generate graphs automatically within a Unity Editor tab. It traverses all the GOs within the ScUT and identifies those with interaction scripts attached, mapping these scripts to their corresponding interaction types. For example, an `XRGrabInteractable` script is mapped to the *grab* interaction type.

To support various developer-implemented custom interaction scripts, our framework provides extensibility through a JSON configuration file. It allows users to specify custom interaction types and their associated scripts. For instance, a custom interaction script `XRKnob` can be configured to be supported by the *grab* interaction.

To prevent misidentification of interactions, this step addresses an issue akin to Android testing's "ambiguous GUI" problem [6]. We resolve multiple GOs that share identical names by generating unique identifiers while preserving their functional properties. This ensures the XUI graph contains unique GO identifiers, eliminating ambiguity in interactables during dynamic exploration.

B. Automated Scene Exploration

This step automates the traversal of the XUI graph (e.g., Fig. 2) to activate all identified interactions. The exploration employs a finite-state machine with three sequential states: navigation, controller movement, and interaction. State transitions occur when specific conditions are met. For instance, when the virtual user–a simulated agent representing the user in the XR environment– reaches sufficient proximity to the target interactable during navigation, it switches to the controller movement states.

1) Navigation: Navigation, moving the virtual user between locations, is one key task of XR scene testing. XRINTTEST facilitates comprehensive scene exploration by leveraging information from the serialised XUI graph (JSON file) to locate corresponding GOs based on their unique identifiers. The exploration adopts a greedy policy–at the beginning of each navigation state, it guides the virtual user towards the nearest unvisited interactables from their current position [4].

2) Controller Movement: This state is activated when the virtual user is close enough to the current target interactable. It manipulates the position of simulated input devices to engage with interactables. XRINTTEST uses Unity's XR Interaction Simulator[3] to generate user interaction events from simulated XR controllers. This simulator cannot directly manipulate camera or controller positions; instead, it drives them indirectly through simulated keyboard and mouse input events (e.g., pressing the W key to move the controller forward). XRINTTEST automatically synthesises the required keyboard input events based on the relative position between the controller and target interactable, enabling continuous and directed movement towards the target.

3) Interaction: This state serves the key task of activating interaction events. Once the controller has contacted the target interactable, this state is activated to engage with the target using identified interaction types. Similar to controller positioning, we simulate specific key events to execute the trigger and grab actions on the controller. The compositional patterns involve sequential events of holding specific keys, pressing additional keys while holding, and releasing keys.

XRINTTEST currently supports the fundamental interactions of *grab* and *trigger* and *grab-and-fire* composition. Fig. 3a and 3b demonstrate the *grab* and *grab-and-fire* patterns, respectively. Since the simulator uses simulated input events for event execution (§ IV-B2), the implementations require keyboard inputs rather than controllers. Fig. 4 illustrates the interaction sequence executed by the implementation shown in Fig. 3b.

C. Test Result Checking

This step verifies successful interaction executions, addressing cases where controller inputs initiate but fail to activate target interactable behaviours. XRINTTEST register callback listeners to each interactable using XRI's built-in interactable event system[4]. After executing an event on an interactable, XRINTTEST validates success by confirming that the activated event data matches the target interactable's information.

```
var kboard = InputSystem.GetDevice<Keyboard>();
InputSystem.QueueStateEvent(kboard, new
↪  KeyboardState(Key.G));
```

(a) Executing *grab* interaction

```
var kboard = InputSystem.GetDevice<Keyboard>();
// Hold the grab key
InputSystem.QueueStateEvent(kboard, new
↪  KeyboardState(Key.G));
// Wait time/hold duration
yield return new WaitForSeconds(0.1f);
// Press the trigger key while grabbing
InputSystem.QueueStateEvent(kboard, new
↪  KeyboardState(Key.T, Key.G));
yield return new WaitForSeconds(0.1f);
// Releasing both keys
InputSystem.QueueStateEvent(kboard, new
↪  KeyboardState());
```

(b) Executing *grab-and-fire* compositional interaction

Fig. 3: Examples of direct *grab* and compositional *grab-and-fire* interactions

XRINTTEST continuously monitors runtime exceptions along the testing process, recording them with contextual information about the attempted interaction. Additionally, it supports integration of mechanisms that detect non-exception bugs, such as unresponsive interaction bugs where users cannot interact with certain interactables despite correct controller positioning and expected event execution. We incorporated a detection mechanism for unresponsive interaction bugs by leveraging GO's collider component. The framework can also be extended to integrate additional validation mechanisms for different testing aspects or test oracles, such as spatial UI interaction and locomotion.

V. EVALUATION[5]

We evaluate XRINTTEST using *XR User Interaction Coverage* (XUI coverage), which quantifies the proportion of successfully activated interaction events in a scene. XUI coverage is calculated as the number of covered edges divided by the total number of edges in the XUI graph of the ScUT.

We compare XRINTTEST's performance on trigger and grab interactions against a random testing baseline. The baseline randomly selects and executes user movements and primitive interaction types. It includes random spawn positioning for navigation and position reset functionality to prevent the virtual user and controllers from drifting beyond relevant testing areas. We assess both approaches in terms of effectiveness, efficiency, and bug detection capability.

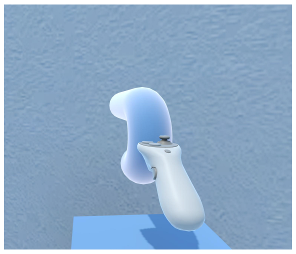
The evaluation is conducted on a MacBook Pro with Apple M3 Pro Chip, 36 GB of RAM, macOS 15.4.1, Unity 6000.0.45f1, and XRI v3.1.1. To account for Unity's nondeterministic processes, we averaged results from five independent

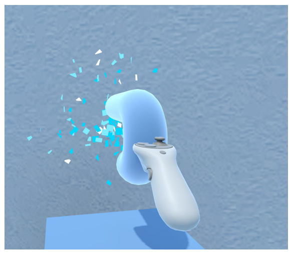
[3]https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@3.1/manual/xr-interaction-simulator-overview.html

[4]https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@3.1/manual/interactable-events.html

[5]Complete evaluation results available in the research track submission.

(a) Before interaction


(b) Grab action activated


(c) Trigger action activated

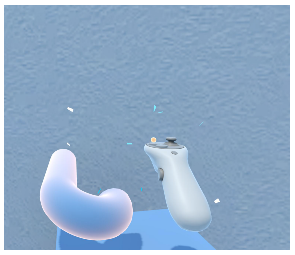

(d) Interaction released

Fig. 4: Examples of Compositional Interaction Sequences

TABLE I: Effectiveness (XR User Interaction Coverage)

Scene	**Random Baseline**	**XRINTTEST**
Sample Scene	3.6/9 (40%)	9/9 (100%)
Demo Scene	2.6/8 (33%)	8/8 (100%)
XRI Starter Kit	0.6/38 (2%)	37/38 (97%)
Examples_Main	1/52 (2%)	52/52 (100%)
Game Scene	6.6/24 (28%)	24/24 (100%)
Prototype Scene	5/17 (29%)	17/17 (100%)
Escape Room	10.2/131 (8%)	124.2/131 (95%)
Average	29.6/279 (11%)	271.2/279 (97%)

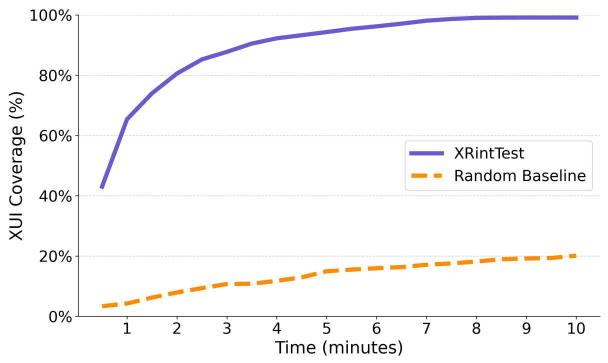


Fig. 5: Efficiency (achieving XR User Interaction Coverage)

executions for each scene. All comparisons used consistent environmental parameters, including a doubled simulation speed and a fixed execution budget of ten simulated minutes.

Table I compares the effectiveness of the random baseline and XRINTTEST in covering the grab and trigger interactions within the seven subject scenes. Across 279 *trigger* and *grab* user interactions, XRINTTEST shows great effectiveness with 97% coverage, vastly outperforming the random baseline, which achieved only 11% coverage overall.

Fig. 5 illustrates the efficiency of XRINTTEST and the random baseline in achieving XUI coverage averaged from seven subject scenes. The results show that XRINTTEST achieves high coverage within 2 minutes in most scenes, reaching near full coverage within the allocated time, consistently outperforming the random baseline.

Regarding bug detection capabilities, both XRINTTEST and the random baseline successfully detected one runtime exception, but only XRINTTEST, thanks to the test result checking component's mechanism to detect unresponsive interaction bugs, managed to identify two *unresponsive interaction* bugs attributable to misconfigured object properties.

VI. Conclusion and Future Work

In this paper, we introduced an automated testing framework targeting the user interactions in extended reality applications. Our evaluation demonstrates it can significantly outperform a random baseline approach. We plan to extend it to support custom interaction patterns and broader scopes of spatial interaction using compositional interaction patterns generated at runtime using learning-based approaches like reinforcement/imitation learning [7].

References

[1] S. Tadeja, P. Seshadri, and P. Kristensson, "Aerovr: An immersive visualisation system for aerospace design and digital twinning in virtual reality," *The Aeronautical Journal*, 2020.

[2] S. Li, C. Gao, J. Zhang, Y. Zhang, Y. Liu, J. Gu, Y. Peng, and M. R. Lyu, "Less cybersickness, please: Demystifying and detecting stereoscopic visual inconsistencies in virtual reality apps," *Proc. ACM Softw. Eng.*, no. FSE, 2024.

[3] R. Gu, J. M. Rojas, and D. Shin, "Software testing for extended reality applications: A systematic mapping study," *Automated Software Engineering*, 2025.

[4] X. Wang, "VRTest: An Extensible Framework for Automatic Testing of Virtual Reality Scenes," in *IEEE/ACM 44th Intl. Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022.

[5] X. Wang, T. Rafi, and N. Meng, "VRGuide: Efficient Testing of Virtual Reality Scenes via Dynamic Cut Coverage," in *38th IEEE/ACM Intl. Conference on Automated Software Engineering (ASE)*, 2023.

[6] T. Fulcini, R. Coppola, M. Torchiano, and L. Ardito, "An analysis of widget layout attributes to support android gui-based testing," in *IEEE Intl. Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2023, pp. 117–125.

[7] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *34th IEEE/ACM Intl. Conference on Automated Software Engineering (ASE)*, 2019.