2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00197

# VRGuide: Efficient Testing of Virtual Reality Scenes via Dynamic Cut Coverage

Xiaoyin Wang and Tahmid Rafi
*Department of Computer Science*
*University of Texas at San Antonio*
San Antonio, USA
xiaoyin.wang@utsa.edu, md.tahmidulislam.rafi@utsa.edu

Na Meng
*Department of Computer Science*
*Virginia Tech*
Blacksburg, USA
nm8247@vt.edu

***Abstract*—Virtual Reality (VR) is an emerging technique that has been applied to more and more areas such as gaming, remote conference, and education. Since VR user interface has very different characteristics compared with traditional graphic user interface (GUI), VR applications also require new testing techniques for quality assurance. Recently, some frameworks (e.g., VRTest) have been proposed to automate VR user interface testing by automatically controlling the player camera. However, their testing strategies are not able to address VR-specific testing challenges such as object occlusion and movement. In this paper, we propose a novel testing technique called VRGuide to explore VR scenes more efficiently. In particular, VRGuide adapts a computer geometry technique called *Cut Extension* to optimize the camera routes for covering all interact-able objects. We compared the testing strategy with VRTest on eight top VR software projects with scenes. The results show that VRGuide is able to achieve higher test coverage upon testing timeout in two of the projects, and achieve saturation coverage with averagely 31% less testing time than VRTest on the remaining six projects. Furthermore, VRGuide detected and reported four unknown bugs confirmed by developers, only one of which is also detected by VRTest.**

***Index Terms*—Software Testing, Virtual Reality, Scene Exploration**

## I. Introduction

Virtual Reality (VR) is a technique [1] to simulate user experience similar or completely different from the real world. Its applications include gaming, virtual exhibition and tour, training, education, product design, and remote communication. According to a recent market report [2], the VR market is emerging and its total market value has reached $11.52 billion in 2019, and is expected to grow at a high rate of 48.7% per year in the following five years. The pandemic of COVID-19 virus further triggered the requirement and accelerated the adoption of VR techniques. VR software is an indispensable part of VR techniques, and its market value has also reached $1.9 billion in 2019 [3]. In year 2020, thousands of apps were uploaded to Google Play [4], Apple Store [5], and Oculus Market [6], and these apps have been downloaded by more than 171 million users [7].

VR applications also need testing to validate their quality, but they raise new challenges for automatic testing techniques. Although VR scenes can still be considered as a type of Graphic User Interface (GUI), existing 2D GUI testing techniques can hardly be applied because users typically observe VR scenes through a player camera and only a small portion of the scenes can be observed and interacted with. Furthermore, the exploration of the scene and the whole user interface is mainly done by moving and rotating the camera rather than clicking buttons.

Most recently, researchers have developed novel techniques to test VR applications by automatically moving and rotating cameras. Autowalker [8] is a tool that automatically drives the player camera to randomly explore a VR scene. VRTest [9] further monitors the positions of all virtual objects and drives the player camera toward the nearest interactable objects that have not been triggered yet. However, although these techniques addressed challenges caused by new exploration patterns in VR scenes (camera movement vs. mouse clicks), their exploration strategies are not optimized because they did not consider the following major characteristics of VR scenes:

- **Object Occlusion.** In traditional 2D GUI, all GUI controls in the current window show up on the screen, and a test driver can directly trigger any interactable control given its coordinates. If a GUI control is occluded by another GUI control, it is typically considered a bug, and the user (or test driver) will not be able to trigger the occluded GUI control at all. However, in VR scenes, due to relative positions of objects and view angles of the camera, interactable objects are often occluded by other interactable or non-interactable objects and the camera needs to be moved and rotated properly to make the occluded objects become visible and interactable.
- **Object Location Affects Testing Efficiency.** Since accessing an interactable object requires camera actions and thus will take time, testing efficiency can be largely affected by the order (and route) of visiting different interactable objects. Theoretically, if the test driver can find a shortest route for the user camera driver to visit the objects to be interacted, the testing efficiency can be optimized. However, this is a well known NP-hard problem. The run-time creation as well as movement of objects and their ability to occlude each other make the problem even more complicated. This is a unique factor to be considered in determining camera movement / rotation strategy for testing VR scenes.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00197

- **Huge Input Space.** VR software provides an immersive experience for users, so typically a user is allowed to move in the scene and change watching angle freely (sometimes within a scope) within the virtual space. Therefore, each camera state corresponds to six float values (three dimensions of position coordinates and three dimensions of watching angle). These dimensions form a very large input space, making efficient strategy of moving / rotating the player camera very important.

In this paper, to efficiently test virtual reality scenes with consideration of the above three characteristics, we propose an automatic testing technique VRGuide. In particular, VRGuide takes into consideration the information of the entire VR scene (i.e., locations of all objects) to calculate interaction values of locations (measured by how many interactable objects can be triggered at the location), and move the camera towards the locations with highest interaction values. The basic intuition behind VRGuide strategy is that a user camera does not need to move to an object's exact location to access it, but it may access the object at a distance as long as the object is in its view and is not occluded by other objects. Therefore, there may exist some locations where multiple interactable objects are visible, and it is more efficient for the user camera to move to those locations if they are close. To guide the camera toward a more efficient route in a VR scene to be tested, we leverage a technique called *Cut Extension* from computer geometry, and adapt it as *Dynamic Cut Extension* to handle moving and dynamically created objects in VR scenes. We evaluated VRGuide on the four software projects used in VRTest [9] evaluation, and four other top Unity VR software projects (based on number of stars) from Github. Although our evaluation focuses on Unity-based software, we believe this scope of subject selection is reasonable because Unity dominates VR software development with over 60% market share according to multiple sources [10], [11], and Unity integrates with almost all existing VR / AR platforms, including Apple ARKit [12], Android Daydream / Cardboard [13], [14], Google ARCore [15], Steam VR [16], Windows Mixed Reality (Hololens) [17], etc., so the testing technique can be largely generalized. The evaluation results show that VRGuide achieved higher object coverage and higher method coverage compared with VRTest (VRGreed stategy) in two out of the eight projects upon testing timeout. For the remaining six projects, VRGuide uses 31% less time to reach coverage saturation compared with VRTest (VRGreed stategy). VRGuide also detected four previously unknown bugs confirmed by developers (only one of them is also detected by VRTest) in real-world projects.

To sum up, this paper makes the following contributions.

- We explore and summarize the major challenges in efficiently testing VR scenes.
- We propose a novel testing strategy called VRGuide, which performs dynamic cut extension based on information of objects in the scene collected at run time.
- We perform an evaluation to compare the efficiency of VRGuide with VRTest on eight top VR software projects, and the results show that VRGuide achieves higher testing efficiency in seven out of eight projects, and higher coverage upon testing timeout in two out of eight projects.

The remaining of this paper is organized as follows. In Section II, we will introduce some background knowledge about VR scenes. Then we will introduce cut theory as the preliminary knowledge in Section III. We will describe the details of our proposed testing technique in Section IV, and present the evaluation setup and results in Section V. After that, we will discuss some important issues in Section VI and related research efforts in Section VII. Before we conclude in Section IX, we point to several research directions where some future research efforts can be made.

## II. Background

In this section, we introduce some background knowledge of VR scenes and event triggers.

### A. VR Scenes

In VR software, all virtual objects are organized inside a virtual space called a VR scene. These objects can be either static (indicating that they are not moving during software execution) or dynamic (indicating that they may move during software execution). Typically, a user is initially placed at a location inside the VR scene, and a user camera is attached to the user so that the user can watch a portion of the virtual space based on the watching angle of the camera. A portion of these objects can be interactable, indicating that they can receive events triggered by the user. The remaining of these objects are not interactable, but they may limit the user's movement through collision and block the user camera's view. Therefore, intuitively exploration testing of a virtual software can be viewed as moving / rotating the user camera so that it can see all the interactable virtual objects (while avoiding all other virtual objects on its way and blocking its view) and interact with them.

### B. Event Triggers

Virtual objects can be interacted with if they have event triggers registered with them. The interaction types vary in different VR hardware devices. However, the most commonly used type of interaction is pointer click. Pointer clicks are based on a white pointer at the focus of a user's view. When a user turns her head to move the white pointer toward an interactable virtual object, the pointer will become a small circle. If the user keeps the pointer inside the range of the object for certain amount of time, the pointer click event will be triggered. For some VR devices where a clicker is provided, the user can simply click the button on the clicker when the pointer is changed to a circle, and then the pointer click will be triggered.

In Figure 1, we show an exemplar scene and a point click event being triggered. In particular, the virtual space simulates an apartment room, in which several objects (the door, the printer, the TV, etc.) are interactable. When a pointer click

Fig. 1. An Exemplar VR Scene and Pointer Click Event

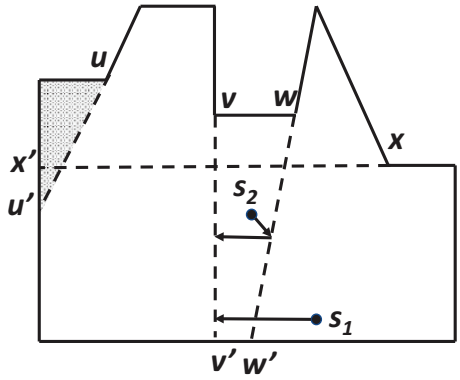


Fig. 2. Illustration of the cut theory

event is triggered on the printer, a virtual paper object will be created from the printer's location, and fall down to the ground.

While there are also some other types of interactions such as object grabbing, they are not supported commonly by existing VR devices so they are also rarely used in existing open source VR software projects. Based on these observations, our VRGuide framework focuses on pointer click events. It should be noted that adding a new type of interaction event to our VRGuide framework just requires additionally instrumentation of a new type of event handler, so it is straightforward in general.

## III. Preliminary

### A. The Watchman Route Problem

The cut coverage theory origins from the *art gallery problem* or *museum problem*, which is a well-known visibility problem in computational geometry. The problem asks how to select a set of fewest points $P$ in a simple polygon so that for each point $q$ in the polygon, there exists a point $p$ in $P$, and line segment $\overline{pq}$ is inside the polygon (i.e., point $q$ is visible from at least one point in set $P$). Its corresponding real-world problem is guarding an art gallery with the minimum number of watchmen who together can observe the whole gallery.

One of the problem generalized from the *art gallery problem* is called *watchman route problem*. In this problem, there exists only one watchman, but he will patrol from a starting point in the polygon. The goal is to find the shortest route the watchman needs to follow so that he can observe the whole polygon along the route. See Figure 2 for two exemplar watchman routes of a polygon from starting points $S_1$, and $S_2$, respectively. We can see that the goal to efficiently observe all the interactable objects in VR testing (if not considering the run-time creation and movement of objects) is actually a simplified version of the *watchman route problem*, where not the whole scene, but only the interactable objects need to be observed. The *watchman route problem* also has several well known variants such as the *zookeeper route problem* where the watchman needs to reach all the cages (nested polygons) without going into them.

### B. The Cut Theory

*The Cut Theory* is a basic technique to solve the watchman route problem [18]. In a $n$-sided polygon $P$, a vertex is called a *convex* if its internal angle is strictly larger than its external angle (i.e., its internal angle is strictly larger than 180 degree). For example, in Figure 2, vertexes $u$, $v$, $w$, and $x$ are the four convexes in the polygon. From each convex, we can extend any one of its two adjacent edges back to the polygon, until the extended line reaches an edge of the polygon. Clearly, the generated line segment will split the polygon into two pieces, so the line segment is referred as a *cut* of the polygon. For example, line segments $\overline{uu'}$, $\overline{vv'}$, $\overline{ww'}$, and $\overline{xx'}$ are four different cuts of the polygon.

In the two pieces of polygon split by a cut $C$, the piece that does not contain the original edge before extension is called the *essential piece* of the cut, denoted as $P(C)$. A more intuitive explanation of the essential piece is that, if the watchman is in the essential piece of a cut, then he must reach the cut to see the original edge. For example, the piece of polygon to the right of $\overline{vv'}$ is the essential piece of $\overline{vv'}$, because the original edge $\overline{ax}$ is not in the piece, and a watchman in this essential piece must reach $\overline{vv'}$ to see $\overline{ax}$ (or the non-essential piece of cut $\overline{vv'}$). A cut $C$ is called an *essential cut*, if its essential piece is not fully contained by the essential piece of any other cut $C'$. Otherwise, if there exists a cut $C'$ whose essential piece fully contains $C$'s essential piece, we say $C'$ *dominates* $C$. The intuition explanation is that, if the essential piece $P(C_1)$ of cut $C_1$ is fully contained by the essential piece $P(C_2)$ of cut $C_2$, a watchman in $P(C_1)$ must reach $C_2$ to observe the non-essential piece of $C_2$. In such a scenario, he will inevitably go across $C_1$ on his route to $C_2$ because $P(C_2)$ fully contains $P(C_1)$. So, $C_1$ is no longer important in determining the shortest route. Note that if the watchman is not in $P(C_1)$, then he automatically sees the non-essential piece of $C_1$, and does not need to reach $C_1$ at

all. As an example, $\overline{uu'}$ is not an essential cut, because its essential piece $P(\overline{uu'})$ (shadowed part) is fully contained by the essential piece of $\overline{ww'}$ (the whole area to the left of $\overline{ww'}$). If a watchman is in $P(\overline{uu'})$, he needs to reach $\overline{ww'}$ anyway and does not need to worry about the requirement of reaching $\overline{uu'}$. Otherwise, he does not need to reach $\overline{uu'}$ at all.

In the exemplar polygon, cuts $\overline{vv'}$, $\overline{ww'}$, and $\overline{xx'}$ are three essential cuts. Although an essential cut is not dominated by any other cut, it can still be dominated by a set of other essential cuts. For example, as shown in Figure 2, the essential pieces of cuts $\overline{vv'}$ and $\overline{ww'}$ contains the essential piece of $\overline{xx'}$ (actually they combined to form the whole polygon). So, a watchman no longer needs to reach $\overline{xx'}$ if he reached both $\overline{vv'}$ and $\overline{ww'}$. A subset of essential cuts that dominates the whole set of essential cuts is called a *watchman cut set* (e.g., $\{\overline{vv'}, \overline{ww'}\}$), and now the original problem of shortest watchman route is reduced to finding the shortest route to cover any of the *watchman cut sets*. To solve this problem, we can draw perpendicular lines from the starting point to all the cuts in the watchman cut sets, and concatenate the intersection points one by one (from the nearest to the farthest) to form a polygonal chain. Then we can shorten this line-segment sequence by sliding its intersection points with each cut along the cut. Some more efficient algorithms have been proposed recently by researchers.

For briefness, we will not introduce these algorithms because their goal is to find a globally optimal route for a static polygon, while in VR testing scenario we have to focus on dynamically optimizing the route to local cuts (i.e., the cuts close to the camera) at run time. Since the global layout of the scene is not static and is always changing, it does not make much sense to explore the globally optimized route for the camera in advance. Therefore, the basic notions of the cut theory will be sufficient to understand our approach.

## IV. Approach

In Section III, we show that the cut theory can be applied to find the shortest watchman route in a static VR scene (i.e., occluding objects can be viewed as holes in a polygon). However, since the objects can be created and moving in the scene, it would not be helpful to calculate a global shortest route at the starting point of the scene or at run time, because when the camera follows the route, the objects may already leave their original locations.

Therefore, our approach VRGuide does not consider the global and static coverage of cuts, but focuses on covering local cuts (i.e., cuts from interactable objects close to the player camera) and uses information of local cuts to guide the next step of the player camera. As an overview, VRGuide will calculate a distance value for each neighboring positions of the player camera, and guide the camera to the position with lowest distance value. The distance value of a specific position will be calculated by combining its distance to multiple closest cuts. Once the player camera reaches a new position, VRGuide will find out which objects are visible from the position, and rotate the camera to interact with them.

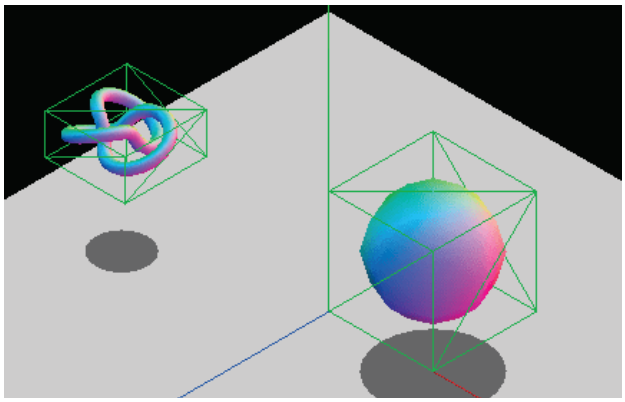

Fig. 3. Bounding Boxes of Virtual Objects from developer.mozilla.org

In the following subsections, we will introduce in more details how we calculate dynamic cuts for interactable objects, how we calculate distance value of a position given multiple dynamic cuts, and the VRGuide algorithm.

### A. Dynamic Cuts of interactable Objects

In the original watchman route problem, a cut is defined as the extension of a convex's adjacent edge, because a watchman needs to observe the whole area beyond the convex. However, this is not required in VR testing where the player camera needs to see only the interactable objects. Therefore, in VR guide, we define a dynamic cut of an interactable object as follows (we first define and illustrate concepts in 2D scenarios and then generalize them to 3D scenarios).

It should be noted that in VRGuide, all objects are approximated by their minimal enclosing boxes (see Figure 3), so they can all be viewed as rectangular cuboids. This approximation is safe for all obstacles because it enlarges their range in the space. However, it is not safe for the interactable object to be interacted with, because seeing part of the object's enclosing box may not guarantee that the object is actually visible from the camera. Therefore, to make sure the interactable object is visible, it is approximated as a point (i.e., the geometric center of the object).

*Definition 1:* In a 2D scenario, when concatenating the player camera C and an interactable object O with a line L, L may intersect with edges of multiple objects. The edge closest to O is defined as O's ***facing edge***, and the object the edge belongs to is defined as O's ***facing object***.

Figure 4 shows interactable objects in different 2D scenarios. In the top case, Edge $\overline{BD}$ is the facing edge, and in the bottom case, Edge $\overline{AB}$ is the facing edge. In both cases, the object $ABCD$ is the facing object.

*Definition 2:* Draw a line segment from an interactable object O to an end point of its facing edge and try to extend the line. If the extended line does not intersect with O's facing object, it is defined as a ***dynamic cut*** of object O.

To illustrate, we show dynamic cuts in both cases in figure 4. Generalizing the definition to 3D, the facing edge will become the facing surface, with four edges, and concatenating the

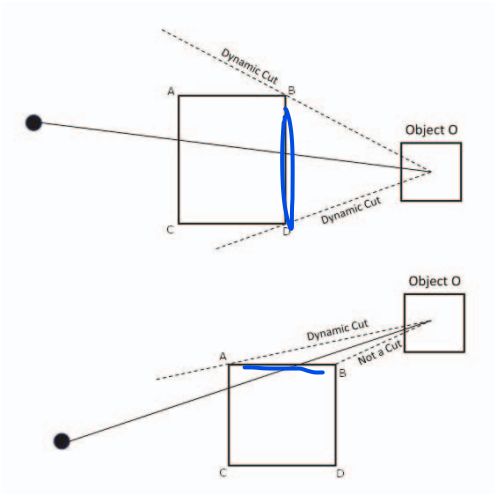


Fig. 4. Examples of Dynamic Cuts

interactable object to the four edges will form at most four cutting surfaces. The definitions are as follows.

*Definition 3:* In a 3D scenario, when concatenating the player camera C and an interactable object O with a line L. L may intersect with surfaces of multiple objects. The surface closest to O is defined as O's facing surface.

*Definition 4:* There exist a surface containing both the interactable object O and one edge of its facing surface. If the extended plane from the surface does not intersect with O's facing object, the surface is defined as a dynamic cut of object O.

### B. Distance Value Calculation

We assume the coordinates of the interactable object are $(x_0, y_0, z_0)$. An edge of the facing surface in parallel with $y$-axis can be represented as equations $\{x = x_1, z = z_1\}$. It should be noted that the edges of an object's enclosing box are always in parallel with one of the coordination axis. If the edge is in parallel with another axis, the equations will be in a similar form, just replacing $x$, and $z$ with $x$ and $y$ or $y$ and $z$. Then we can calculate the dynamic cut as a plane with equation below.

$$z - z_1 = \frac{(z_2 - z_1)(x - x_1)}{(x_2 - x_1)} \tag{1}$$

Once we acquired the equation of a dynamic cut (denoted as $Ax + By + Cz + D = 0$), we can calculate the distance from a specific position $(x_2, y_2, z_2)$ in the scene to the cut using the following formula.

$$Dist = \frac{|Ax_2 + By_2 + Cz_2 + D|}{\sqrt{A^2 + B^2 + C^2}} \tag{2}$$

### C. VRGuide

Once we are able to calculate the distance from the player camera's neighboring positions to the dynamic cuts, we can guide the player camera towards the neighboring position that has shortest distance to its closest dynamic cut. As illustrated in the simplified two-dimensional scene in Figure 5, this strategy can be more efficient than the existing strategy in VRTest [9] which is based on the distance between the player camera and the object to be interacted. When the user camera is at the solid black point, it needs to trigger pointer-click events on the two interactable objects on the left side and right side, respectively. Following VRTest's strategy, it will first move towards the object on the right and reach position $A$, and then go back to position $B$ to trigger the event on the object on the left. However, VRGuide will direct the player camera to position $C$ and then position $D$, which is a much shorter route.

We take advantage of the existing VRTest framework [9] to implement VRGuide. In particular, VRTest provides information (positions and sizes of enclosing boxes) about all objects in a VR scene, and table recording which objects have been interacted with. It also provides two interface procedures for implementing new testing strategies: `Move` and `Rotate`. The framework will execute procedures `Move` and `Rotate` in sequence for each testing step (by default 1 second or 30 frames under 30 fps). So, we insert our VRGuide algorithm as implementations of these two procedures.

Algorithm 1 shows the pseudo code of `Rotate` and `Move` procedures of VRGuide. The methods `getPossibleRotations()` and `getPossibleMoves()` return all the possible rotations and moves. In particular, if the action granularity is 1 meter for move, for the original position (0, 0, 0), all possible moves (without considering configuration) will be the set of (0, 0, 1), (0, 0 , -1), (0, 1, 0), (0, -1, 0), (1, 0, 0), and (-1, 0, 0). If the configuration sets the lower-bound of all dimensions to 0 and does not allow movement in $Z$-axis, then the only possible moves (returned by `getPossibleMoves()`) will be (1, 0, 0) and (0, 1, 0). Similarly, if the current rotation of the user camera is (90, 0, 0), then the only possible rotation returned by `getPossibleRotations()` will be (80, 0, 0). The reasons are (1) the upper-bound of $X$-axis rotation (up and down) is 90, so $X$-axis rotation cannot go to 100; (2) no $Z$-axis rotation is allowed, and (3) $Y$-axis rotation is meaningless if the $X$-axis rotation is at 90 degree (i.e., turning east and west does not make sense at the North Pole).

We have $Obj$ as a global variable to share information between `Rotate` and `Move`. $Obj$ stores all the interactable virtual objects visible at the current position. As long as $Obj$ is not empty (Line 2), the `Rotate` procedure will rotate towards the virtual object in $Obj$ whose direction is closest to the camera's current facing direction (Line 4). Otherwise, it will return $CurrentRotation$ indicating not to rotate and wait for `Move` procedure to find more visible objects.

The `Move` procedure first checks whether there are still currently visible objects have not been interacted using function `Reachable` provided by VRTest (Line 11). If so, the camera will stay at the current position (Line 17). Otherwise, it will fetch all neighbor positions that the player camera can move

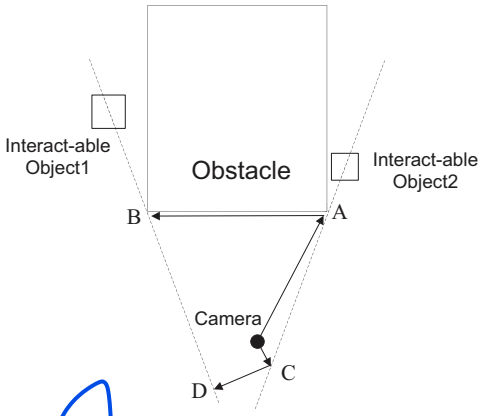


Fig. 5. A Simplified Two Dimensional Scene

to in the next step (Line 13), and find the position that has shortest distance to its closest dynamic cut (Line 14). Then, the `Move` procedure will return that position to guide the camera's following movement.

**Algorithm 1** VRGuide Algorithm

```
1:  procedure ROTATE()                    ▷ Objs is a global variable.
2:      if Objs is not Empty then
3:          Opts ← getPossibleRotations()
4:          Opt ← FetchClosest(Opts, Objs)
5:          Return Opt
6:      else
7:          Return CurrentRotation
8:      end if
9:  end procedure
10: procedure MOVE()
11:     Objs ←Reachable();
12:     if Objs is empty then
13:         Opts ← getPossibleMoves()
14:         Opt ← BestNeighbor(Opts)
15:         Return Opt
16:     else
17:         Return CurrentPosition
18:     end if
19: end procedure
```

## V. EVALUATION

To evaluate our approach, we compare VRGuide and VRTest on eight VR software projects on their object coverage, method coverage, as well as detected bugs. The implementation of VRGuide and the dataset used in our evaluation is available on our project website[1].

[1]https://sites.google.com/view/vrguide2023

### A. Evaluation Setup

*1) Research Questions:*

- **RQ1:** How VRGuide compares with existing approach on the efficiency to cover different interactable objects in a VR scene?
- **RQ2:** How VRGuide compares with existing approach on the efficiency to cover methods in VR software code?
- **RQ3:** How VRGuide compares with existing approach on the detection of bugs in VR software?

*2) The Compared Technique:* We compare VRGuide with VRTest [9], a state-of-the-art VR test framework. Since the implementation of VRGuide is based on VRTest framework, the difference is only on the testing strategy, so we believe the comparison with VRTest can fairly reveal the effectiveness of VRGuide. VRTest supports two testing strategies: VRMonkey, which is similar to Monkey in mobile testing and randomly move and rotate the player camera to interact with interactable objects, and VRGreed, which uses greed algorithm to approach the closest interactable objects one by one. Although VRGreed has been shown to be superior than VRMonkey in earlier studies [9], we still ran VRMonkey in our evaluation and include its results for reference. Note that we use the average results of five executions for VRMonkey due to the randomness in the technique.

*3) Evaluation Subjects:* In our evaluation, we reuse four out of five VR software projects from the original evaluation subject set of VRTest[2]: `UnityVR`, `UnityVREscapeRoom`, `Unity-vr-maze`, and `Unity-vr-cave-puzzle`. We did not use the remaining `VRND_Night_at_the_Museum` because it is no longer compatible with updated version of Unity.

Besides the four subjects from VRTest's original subject set, we further collected four more open source VR projects from Github. In particular, we searched for keywords "Unity" and "VR", and ranked the retrieved projects by the number of stars. We considered only VR software projects consisting of VR scenes, so we skipped the VR development libraries and tools such as `XRTK` and `Google VR Unity SDK`. Furthermore, we considered only projects with at least one virtual object with at least one event triggers.

The basic information of the eight subject projects in our evaluation is presented in Table I. In the table, we present the number of source files, the number of lines of code, and the number of static virtual objects / prefabs (dynamic virtual objects are typically created by cloning static virtual objects / prefabs), respectively.

*4) Testing Environment:* To perform the evaluation, we use Unity version 2021.3 LTS with Visual Studio 2017 (for compilation of C# source code) and run the experiment on a computer with Intel Core i7-6500U CPU, 8GB of memory, and Intel HD 520 Graphics card. We set a timeout of 300 seconds, which is the default timeout value of VRTest. Testing of most subjects saturate before 300 seconds.

[2]Downloaded from https://sites.google.com/view/vrtest2021

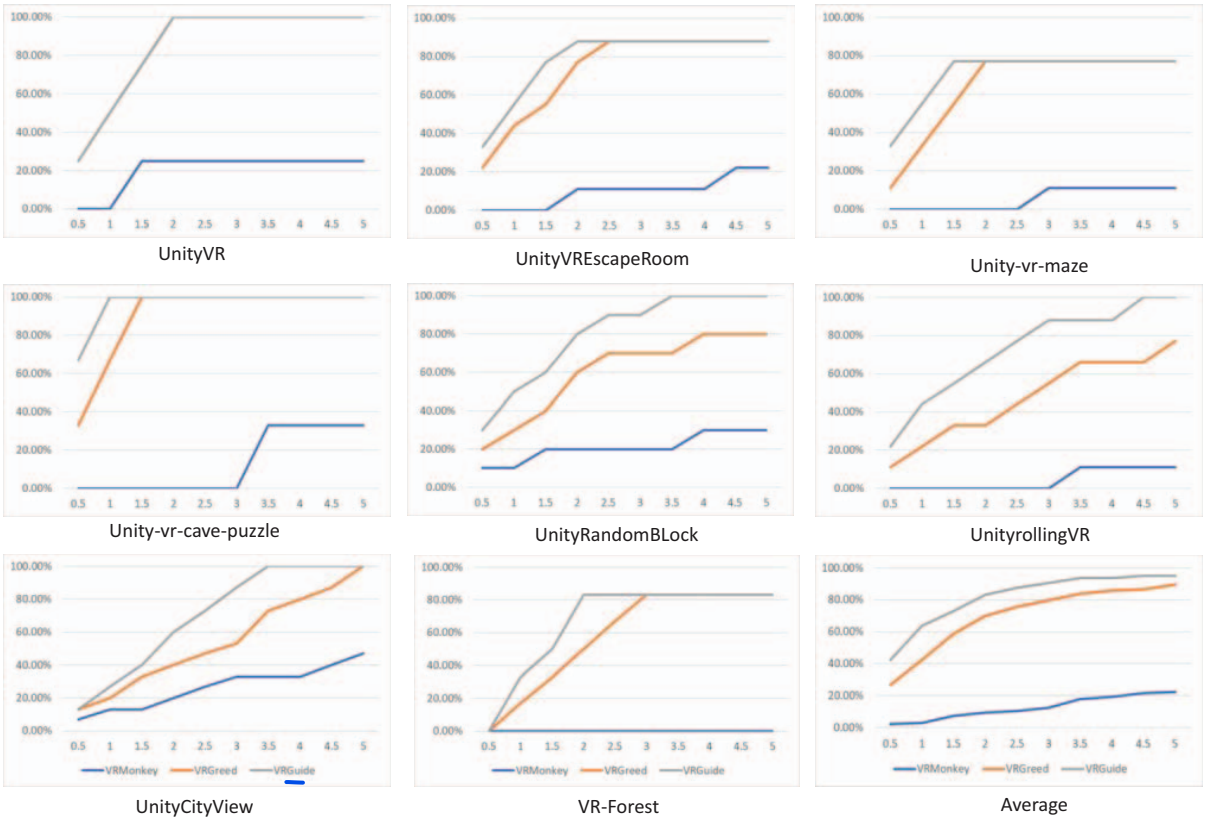


Fig. 6. interactable Objects Coverage of Different Testing Techniques

TABLE I
BASIC INFORMATION OF EVALUATION SUBJECTS

| Name | #Source Files | LOC | #Virtual Objects |
|---|---|---|---|
| UnityVR | 129 | 25.6K | 36 |
| UnityVREscapeRoom | 207 | 31.2K | 109 |
| unity-vr-maze | 7 | 503 | 26 |
| unity-vr-cave-puzzler | 7 | 8.0K | 27 |
| UnityRandomBlocks | 144 | 15.3k | 32 |
| UnityrollingVR | 136 | 12.2k | 22 |
| UnityCityView | 176 | 19.0k | 26 |
| VR-Forest | 422 | 41.1k | 43 |

*5) Testing Configuration:* When performing VR testing using VRTest, we need to configure three major parameters: the *rotation scope* that limits the user camera's watching angle, the *moving/rotating speed* defines the speed of moving and rotating the user camera, and the *Moving Granularity* that determines the maximal distance to be covered in one move action.

In our evaluation, we follow the default values of VRtest for all of the four parameters to make sure we have a fair comparison with VRTest. In particular, we set the rotation scope with $X$-axis rotation between -90 degree and 90 degree, $Y$-axis rotation between -180 degree and 180 degree, and no rotation for $Z$-axis. We use 1 meter per second (1 unit in Unity-based VR scene represents 1 meter) as the moving speed and 10 degree per second as the rotation speed. For action granularity, we use 1 meter and 10 degree as the elementary step of movement and rotation.

### B. Evaluation Results

In our evaluation, we measure the effectiveness of testing by the interactable object coverage and method coverage. For interactable object coverage, we count objects of the same type as one because they are attached with the same set of listeners and scripts. The evaluation results on interactable object coverage is presented in Figure 6. The figure consists of nine sub-charts. In each sub-chart, the x-axis denotes the amount of testing time passed (in minutes). The y-axis denotes the interactable object coverage. The first eight sub-charts present the results for eight subject projects, respectively, and the last sub-chart presents the average results of eight subject projects.

From Figure 6, we have the following major observations. First of all, both VRGreed and VRGuide achieved much higher interactable object coverage than VRMonkey, reaffirming that

purely random testing strategy does not work well in VR software (which is different from the case in Android testing). This is perhaps due the huge input space and sparsity of interactable objects in a VR scene. Second, VRGuide is more efficient than VRGreed in seven out of eight VR projects, and in the only remaining project `UnityVR` both strategies have the same efficiency. The major reason is that `UnityVR` does not have any obstacle and all virtual objects are visible at the beginning of the scene. Third, in projects `UnityrollingVR` and `UnityRandomBlocks`, VRGuide achieves higher coverage when the five minute time out is reached. The reason may be that these projects involve many moving virtual objects. For the other projects, VRGuide is able to achieve a coverage saturation using 31% less than time VRGreed.

The evaluation results on method coverage is presented in Figure 8. The figure is organized the same way as Figure 6. From the figure, we can observe a trend similar to that for interactable object coverage in Figure 6. The major difference is that, because VR software typically contains a lot of code in life-cycle methods (e.g., `start()`, `update()`) for animation rendering, such code will always be executed as long as the VR scene is initialized and executed. Therefore, even VRMonkey achieved a not-to-bad coverage between 35% and 45%. However, VRGuide still achieves higher testing efficiency in seven out of eight projects, and higher coverage upon testing timeout in two out of eight projects.

### C. Bug Detection

The ultimate goal of testing is to detect bugs in software. Therefore, we further investigate whether our technique is able to detect real bugs and how it compares with VRTest. During the testing process, we detected five bugs from three projects `UnityRandomBlocks`, `UnityrollingVR` and `UnityCityView`. *Four of the five bugs*[3] *have been confirmed and fixed by the developers.* It should be noted that without automatic oracle, unhandled exceptions are the only type of bugs we can automatically report. Among the five bugs, VRTest is able to detect the bug in `UnityCityView` (not confirmed yet), and one of the bugs in `UnityRandomBlocks` (confirmed and fixed), but missed the remaining three bugs because those bugs all require the interaction within time limit or with moving objects which will be destroyed after a while, and VRTest is not able to interact with the object in time.

Figure 7 shows the screenshot of one of the bugs we detected in `UnityRandomBlocks`. In particular, the blue ball is dynamically placed and moving in a scene with many obstacles. It is supposed to stop when caught by the pointer clicker. A rigid body is required to be attached to the ball when a force is applied to it. However, the developer forgot to attach a rigid body to the object, so an exception was thrown. *Our bug detection results show that although VRGuide performs just moderately better than VRTest on the final method / object coverage, it is more likely to detect real bugs. We believe that VRGuide's ability to achieve high coverage within shorter time allows it to detect time-sensitive bugs that are more difficult to detect with manual testing. And that is the reason why VRGuide is able to find more real-world bugs than VRTest.*

[3]https://github.com/hfzhg/UnityRandomBlocks/issues/1, https://github.com/hfzhg/UnityRandomBlocks/issues/2, https://github.com/spcover/UnityrollingVR/issues/9, https://github.com/spcover/UnityrollingVR/issues/10

Fig. 7. A Detected Bug in UnityRandomBlocks

### D. Threats to Validity

The major threat to our construction validity is whether our experiment setup is the same as the actual usage scenario of VR testing. Since VRGuide is fully automatic, the only potential issue is whether the configuration is reasonable. To reduce this threat, we followed VRTest with all their configurations. We believe the default configuration of VRTest should provide a fair environment for comparing testing strategies. The major internal threat to our evaluation is the potential bugs and errors in our implementation of VRGuide. To reduce this threat, we carefully reviewed the code of VRGuide, and tested it with multiple artificial testing projects. The major external threat to our evaluation is that our results may be specific to the subject projects we used, or Unity-based VR projects. To reduce this threat, we collected eight top subject projects with different features from UnityList and Github. Also, we believe that Unity-based projects are representative given that Unity is dominating the VR software development market. To further reduce this threat in the future, we plan to perform evaluation more subjects and projects based on a different framework.

## VI. Discussion

### A. More Event Types

Our VRGuide currently focuses on the pointer click event type as it is the most commonly supported event type. There are some other event types supported by certain devices, such as the grabbing event which allows a user to grab certain virtual object with their virtual hand, and the colliding event that allows a user to push or collect certain virtual objects when the user camera is at the same position or close to an existing virtual object. Since these events are mainly contact-based events (i.e., the user camera needs to be very close to the virtual object to trigger the event), they are less complicated

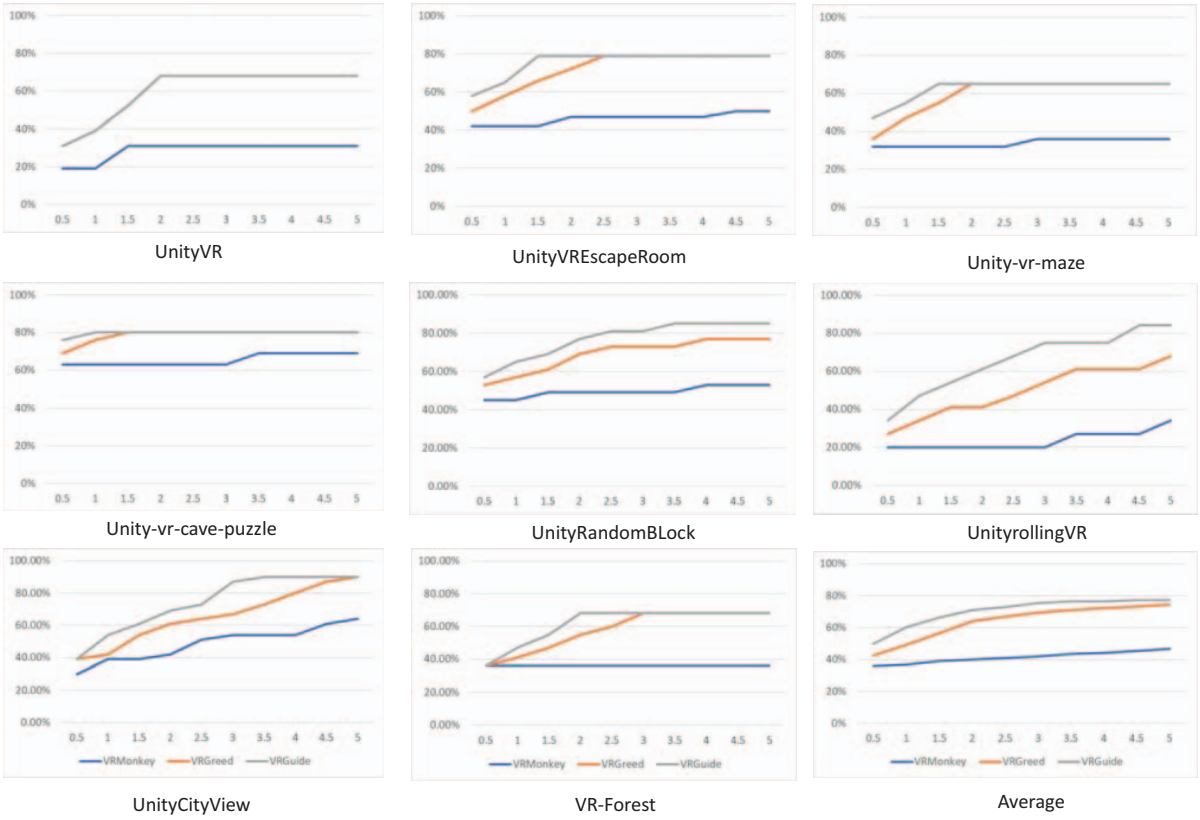


Fig. 8. Method Coverage of Different Testing Techniques

to trigger compared with pointer clicks as we do not need to consider scenarios such as object occluding. Meanwhile, when considering all types of events, we can measure the interaction value of a position inside the virtual scene by combining the number of visible interactable objects (receiving pointer click events) and the distance from the position to interactable objects which receive contact-based events.

### B. Event Constraints

Similar to GUI software where clicking a button may lead the user to a new window or make other controls available, there are also event constraints in VR software where triggering an event on a virtual object leads to the creation / destroy / movement of virtual objects and even scene switch. In VRTest, such dynamic objects are currently handled by periodically retrieving the states of the VR scene (through the VR Scene Monitor). A more complicated case is when the virtual objects must be interacted in certain order to lead to an outcome. None of our three testing techniques intentionally handle such interaction orders, so whether the outcome can be triggered may largely rely on repetitive triggering of events on interactable virtual objects when the methods associated with them are still not covered. In the future, we plan to use static analysis to identify the dependencies between event handlers. Based on the dependencies, VRGuide would be able to trigger events in more proper order to expose more software behaviors.

### C. Testing Metrics

In our paper, we use method coverage and interactable coverage to measure the test effectiveness of our framework and tools. It is arguable whether these testing metrics are effective for VR testing because VR software focus more on user experience. There exist many VR software projects which do not have much interactions and just have the user to view the VR scene. For such software, it may be more important for the testing process to explore the VR scene as much as possible instead of trying to trigger as many events as possible. So a different type of test coverage, such as scene coverage, which measures how much portion of the VR scene has been observed, may be also suitable for certain types of VR software projects.

## VII. Related Works

### A. Testing and Studies of VR Software

There have been some test frameworks that facilitate automating VR software testing such as VRTest [9], which we compare with in our paper, and AutoWalker [8], which is

similar to VRMonkey and randomly guides the player camera in the VR scene. Gil et al. [19] and Souza el al. [20] proposed approaches to model AR applications and cover the model nodes and edges using automatic test cases. However, their models are at higher level focusing on covering scenes and their transitions instead of the automatic exploration of VR scenes. Harms [21] proposed guidelines for the usability evaluation of AR applications and categorizes usability issues. Rafi et al. [22] proposed Predart, which automatically evaluates realisticness of VR object placement as test oracles and can be applied to automatic VR and AR testing. Very recently, Rzig et al. [23] studied the characteristics of unit tests in VR applications and found they were of lower quality than their counterparts in other applications. Compared with these efforts, VRGuide focuses on scene exploration, which is a core component of VR software testing and none of the existing works cover it.

There are also some works on game testing which is related to VR testing. Wuji [24] is a framework to automatically test games based on evolutionary algorithms and reinforcement learning. It explores the game spaces and branches as well as making progress by passing stages. Zhao et al. [25] proposed an approach to enhancing playing tactics in game testing by learning from player action sequences. Bergdahl et al. [26] proposed an approach to augment existing manually written test scripts with reinforcement learning. However, all of the above approaches mainly focus on game tactics and are designed for 2D games, so when applied to 3D software they still face the challenge of flexible camera movement/rotation and accessing out-of-view and occluded objects, which are the focuses of this paper.

There also have been some empirical studies on VR software and video game software. Murphy-Hill et al. [27] performed a study on video game developers to understand the challenges in video game development and how they are different from traditional software development. Washburn et al. [28] studied failed game projects to find out the major pitfalls in game development. Lin et al. [29] studied the common updates in steam platform to understand the priority of game updates. Rodriguez and Wang. [30] performed an empirical study on open source virtual reality software projects to understand their popularity and common structures. Pascarella et al. [31] studied open source video game projects to understand their characteristics and the difference between game and non-game development. Zhang et al. [32] studied possible solutions to detect potential privacy leaks in mobile augmented reality apps. Nusrat et al. [33] studied performance issues in VR applications from performance repair logs and identified the major reasons for performance downgrades in VR applications. Molina et al. [34] developed a novel technique to extract code dependencies [35] in VR applications and studied the types of dependencies. From these studies, we gain knowledge on the characteristics of VR software projects, which help us to understand VR-specific challenges when designing VRGuide.

### *B. GUI Testing*

Our VRGuide framework includes a lot of new designs to address special challenges in VR software testing, but in general, our research is also related to GUI testing and more advanced GUI testing strategies can be combined with VRGuide to better unleash its full power. GUI testing is an extensively studied research area. Some representative technical solutions include random techniques, model-based techniques, symbolic-execution-based techniques, search-based techniques, and learning-based techniques.

**Random techniques.** Monkey [36], DynoDroid [37], DroidFuzzer [38], are random-search-based approaches that randomly explore GUI windows. In particular, DynoDroid [37] instruments the Android framework and allows sending sequential and interleaving events. It further allows human testers to provide input for input boxes. DroidFuzzer [38] automatically generates random MIME messages on top of GUI events. Mimic [39] uses random strategy to detect incompatibilities [40] in GUI among different mobile devices.

**Model-based techniques.** Model-based techniques use static analysis or dynamic analysis to generate a GUI exploration model and then explore the GUI according to the model. Examples include GUIRipper [41] and its later extension MobiGUITAR [42]. A3E [43] and SwiftHand [44] also build finite state models for UI and generate events to explore states in the model systematically.

**Symbolic-execution-based techniques.** The symbolic-execution-based techniques use static or dynamic symbolic execution to generate test input that leads control flow to uncovered code. ACTEve [45] first applied concolic testing to the exploration of Android apps. It alleviates path explosion by detecting program executions that identify subsumption between different event sequences. JPF-Android [46], an extension of JPF (Java Path Finder) [47], uses static symbolic execution to find all feasible execution paths in an Android app and generate test inputs to cover them.

**Search-based techniques.** A representative tool for search-based testing for Android is EvoDroid [48], which boosts searching efficiency by considering the constraint of Android development framework. A more recent work, FuzzDroid [49], focuses on generating the execution environments (e.g., phone settings of country or language, other apps installed) by combining static and dynamic analyses through a search-based algorithm that steers the app toward a configurable target location. Sapienz [50] combines pre-defined GUI interaction patterns with a genetic algorithm to evolve from seed input sequences and search for the optimized exploration sequences containing short input sequences while maximizing test coverage and fault revelation. Stoat [51] is a UI test generation tool combining model-based testing and evolutionary testing. It first constructs a probabilistic state-transition model via dynamic exploration and optional static analysis, and then evolves the model to search for the optimized model with regard to comprehensive fitness scores involving Code coverage, model coverage, and test-suite diversity.

**Learning-based approaches.** Researchers have proposed testing techniques based on learning from the testing process. Most recently, He et al. [52] proposed a feedback-driven text input exerciser, which tries to meet the input constrains by analyzing the hint of the text fields. Pan et al. [53] proposed to use text similarity in NLP as a guidance to train a model which generates the test that leads to the state with highest difference. This approach requires large training set from open-source apps. Liu et al. [54] proposed to use training apps and human testers to train a model which can be used to automatically generate meaningful input for Android apps. Mariani et al. [55] developed Augusto, which provide high-level testing rules for three common app functions (i.e., log-in, create-read-update-delete, save) to guide the model exploration. Qin et al. [56] proposed to use event sequence mapping to migrate event sequences from iOS tests to the Android version of the same app. Behrang and Orso's recent work [57] further learns test oracles from existing tests.

## VIII. Future Works

In the future, we plan to work on the following directions. First of all, for the testing framework, we plan to extend it to support more types of events such as grabbing events and colliding events. Second, our VRGuide testing strategy, although considering more global information in the VR scene, is still a greedy-algorithm-based approach, so we plan to further enhance it by using AI-based or search-based techniques which have been shown effective in GUI testing to acquire a globally optimized route. Third, we plan to evaluate our framework with more subjects and software projects that are not based on Unity. Fourth, certain software behavior may be exposed only when events are triggered in certain order, so we plan to use static analysis to identify the dependencies between event handlers. Based on the dependencies, VRGuide would be able to trigger events in a certain order to expose more software behaviors.

## IX. Conclusion

In this paper, we propose a novel testing strategy called VRGuide to automatically test VR software. The VRGuide testing strategy is based on the cut theory from computer geometry and it takes advantage of the intuition that in VR testing, the player camera can typically interact with an object as long as the object is within the field of view. The VRGuide testing strategy involves three major steps: the calculation of dynamic cuts from interactable objects to their facing surfaces, the calculation of distances from the player camera's neighboring positions to the dynamic cuts, and guiding the player camera toward the neighboring point that has shortest distance to its nearest dynamic cut. We evaluated VRGuide on eight top VR projects from UnityList and Github, and the evaluation result shows that VRGuide is able to achieve higher test efficiency and coverage than existing approaches (i.e., VRTest), and detect unknown bugs in real world projects.

## Acknowledgment

The UTSA authors are supported in part by NSF Grants CNS-1736209, SHF-1846467, SHF-2007718, and CNS-2221843. The Virginia Tech author is supported in part by NSF Grants SHF-1845446 and SHF-2006278.

## References

[1] L. P. Berg and J. M. Vance, "Industry use of virtual reality in product design and manufacturing: a survey," *Virtual reality*, vol. 21, no. 1, pp. 1–17, 2017.

[2] "Mordor intelligence report on virtual reality market," https://www.mordorintelligence.com/industry-reports/virtual-reality-market, 2020, accessed: 2020-06-30.

[3] "Statistica report on virtual reality software market," https://www.statista.com/statistics/550474/virtual-reality-software-market-size-worldwide/, 2020, accessed: 2020-06-30.

[4] "Google play," https://play.google.com/store, 2020, accessed: 2020-06-30.

[5] "Apple app store," https://www.apple.com/ios/app-store/, 2020, accessed: 2020-06-30.

[6] "Oculus app store," https://www.oculus.com/experiences/quest/, 2020, accessed: 2020-06-30.

[7] "Vr user statistics," https://techjury.net/blog/virtual-reality-statistics/gref, 2020, accessed: 2020-06-30.

[8] (2022) Auto walk unity. https://github.com/onelei/auto-walk-unity.

[9] X. Wang, "Vrtest: An extensible framework for automatic testing of virtual reality scenes," in *Tool Demo, 2022 IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2022, pp. 392–397.

[10] "Unity engine: A unicorn powering the video game and vr/ar economy," https://digital.hbs.edu/platform-digit/submission/unity-engine-a-unicorn-powering-the-video-game-and-vr-ar-economy/, 2020, accessed: 2020-12-30.

[11] "Unity ipo aims to fuel growth across gaming and beyond," https://techcrunch.com/2020/09/10/how-unity-built-a-gaming-engine-for-the-future/, 2020, accessed: 2020-12-30.

[12] "Apple arkit," https://developer.apple.com/augmented-reality/, 2020, accessed: 2020-12-30.

[13] "Google daydream," https://arvr.google.com/daydream/, 2020, accessed: 2020-12-30.

[14] "Google cardboard," https://arvr.google.com/cardboard/, 2020, accessed: 2020-12-30.

[15] "Google arcore," https://developers.google.com/ar, 2020, accessed: 2020-12-30.

[16] "Steam vr," https://store.steampowered.com/steamvr, 2020, accessed: 2020-12-30.

[17] "Microsoft hololens," https://www.microsoft.com/en-us/hololens, 2020, accessed: 2020-12-30.

[18] P. Wang, R. Krishnamurti, and K. Gupta, "Generalized watchman route problem with discrete view cost," *International Journal of Computational Geometry & Applications*, vol. 20, no. 02, pp. 119–146, 2010.

[19] A. Gil, T. Figueira, E. Ribeiro, A. Costa, and P. Quiroga, "Automated test of vr applications," in *HCI International 2020–Late Breaking Posters: 22nd International Conference, HCII 2020, Copenhagen, Denmark, July 19–24, 2020, Proceedings, Part II 22*. Springer, 2020, pp. 145–149.

[20] A. C. Correa Souza, F. L. Nunes, and M. E. Delamaro, "An automated functional testing approach for virtual reality applications," *Software Testing, Verification and Reliability*, vol. 28, no. 8, p. e1690, 2018.

[21] P. Harms, "Automated usability evaluation of virtual reality applications," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 26, no. 3, pp. 1–36, 2019.

[22] T. Rafi, X. Zhang, and X. Wang, "Predart: Towards automatic oracle prediction of object placements in augmented reality testing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[23] D. E. Rzig, N. Iqbal, I. Attisano, X. Qin, and F. Hassan, "Virtual reality (vr) automated testing in the wild: A case study on unity-based vr applications," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1269–1281.

[24] 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 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 772–784.

[25] Y. Zhao, W. Zhang, E. Tang, H. Cai, X. Guo, and N. Meng, "A lightweight approach of human-like playtesting," *arXiv preprint arXiv:2102.13026*, 2021.

[26] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslén, "Augmenting automated game testing with deep reinforcement learning," in *2020 IEEE Conference on Games (CoG)*. IEEE, 2020, pp. 600–603.

[27] E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?" in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1–11.

[28] M. Washburn, P. Sathiyanarayanan, M. Nagappan, T. Zimmermann, and C. Bird, "What went right and what went wrong: An analysis of 155 postmortems from game development," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16, 2016, p. 280–289.

[29] D. Lin, C.-P. Bezemer, and A. E. Hassan, "Studying the urgent updates of popular games on the steam platform," *Empirical Software Engineering*, vol. 22, no. 4, pp. 2095–2126, 2017.

[30] I. Rodriguez and X. Wang, "An empirical study of open source virtual reality software projects," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 474–475.

[31] L. Pascarella, F. Palomba, M. Di Penta, and A. Bacchelli, "How is video game development different from software development in open source?" in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 392–402.

[32] X. Zhang, R. Slavin, X. Wang, and J. Niu, "Privacy assurance for android augmented reality apps," in *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2019, pp. 114–1141.

[33] F. Nusrat, F. Hassan, H. Zhong, and X. Wang, "How developers optimize virtual reality applications: A study of optimization commits in open source unity projects," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 473–485.

[34] J. Molina, X. Qin, and X. Wang, "Automatic extraction of code dependency in virtual reality software," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 381–385.

[35] H. Zhong and X. Wang, "Boosting complete-code tool for partial program," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 671–681.

[36] AOSP, "Android Monkey," https://developer.android.com/studio/test/monkey, 2007.

[37] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450

[38] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, ser. MoMM '13. New York, NY, USA: ACM, 2013, pp. 68:68–68:74. [Online]. Available: http://doi.acm.org/10.1145/2536853.2536881

[39] T. Ki, C. M. Park, K. Dantu, S. Y. Ko, and L. Ziarek, "Mimic: Ui compatibility testing system for android apps," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 246–256.

[40] L. Chen, F. Hassan, X. Wang, and L. Zhang, "Taming behavioral backward incompatibilities via cross-project testing and analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 112–124.

[41] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 258–261.

[42] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE software*, vol. 32, no. 5, pp. 53–59, 2015.

[43] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 641–660.

[44] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 623–640.

[45] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 59.

[46] H. van der Merwe, B. van der Merwe, and W. Visser, "Execution and property specifications for jpf-android," *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 1, pp. 1–5, Feb. 2014.

[47] "Jpf," http://javapathfinder.sourceforge.net/.

[48] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 599–609.

[49] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of android execution environments," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 300–311.

[50] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 94–105. [Online]. Available: https://doi.org/10.1145/2931037.2931054

[51] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 245–256. [Online]. Available: https://doi.org/10.1145/3106237.3106298

[52] Y. He, L. Zhang, Z. Yang, Y. Cao, K. Lian, S. Li, W. Yang, Z. Zhang, M. Yang, Y. Zhang, and H. Duan, "Textexerciser: Feedback-driven text input exercising for android applications," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1071–1087.

[53] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 153–164. [Online]. Available: https://doi.org/10.1145/3395363.3397354

[54] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 643–653. [Online]. Available: https://doi.org/10.1109/ICSE.2017.65

[55] L. Mariani, M. Pezzè, and D. Zuddas, "Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 280–290.

[56] X. Qin, H. Zhong, and X. Wang, "Testmig: Migrating gui test cases from ios to android," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, p. 284–295.

[57] F. Behrang and A. Orso, "Test migration between mobile apps with similar functionality," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 54–65.