Virtual Reality (VR) Automated Testing in the Wild: a Case Study on Unity-Based VR Applications

Dhia Elhaq Rzig
University of Michigan - Dearborn
Dearborn, MI, USA
dhiarzig@umich.edu

Nafees Iqbal
University of Michigan - Dearborn
Dearborn, MI, USA
nafees@umich.edu

Isabella Attisano
Villanova University
Villanova, PA, USA
iattisan@villanova.edu

Xue Qin
Villanova University
Villanova, PA, USA
xue.qin@villanova.edu

Foyzul Hassan
University of Michigan - Dearborn
Dearborn, MI, USA
foyzul@umich.edu

Abstract

Virtual Reality (VR) is an emerging technique that provides a unique real-time experience for users. VR technologies have provided revolutionary user experiences in various scenarios (e.g., training, education, gaming, etc.). However, testing VR applications is challenging due to their nature which necessitates physical interactivity, and their reliance on specific hardware systems. Despite the recent advancements in VR technology and its usage scenarios, we still know little about VR application testing. To fill up this knowledge gap, we performed an empirical study on 314 open-source VR applications. Our analysis identified that 79% of the VR projects evaluated did not have any automatic tests, and for the VR projects that did, the median functional-method to test-method ratios were lower than those of other project types. Moreover, we uncovered tool support issues concerning the measurement of VR code coverage, and the assertion density results we were able to generate were relatively low, with an average of 17.63%. Finally, through a manual analysis of 370 test cases, we identified the different categories of test cases being used to validate VR application quality attributes. Furthermore, we extracted which of these categories are VR-attention, meaning that test writers need to pay special attention to VR characteristics when writing tests of these categories. We believe that our findings constitute a call to action for the VR development community to improve their automatic testing practices and provide directions for software engineering researchers to develop advanced techniques for automatic test case generation and test quality analysis for VR applications. Our replication package containing the dataset we used, software tools we developed, and the results we found, is accessible at https://doi.org/10.6084/m9.figshare.19678938.

CCS Concepts

• **Human-centered computing** → *Virtual reality*; • **Software and its engineering** → **Software testing and debugging**.

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.
Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Dhia Elhaq Rzig, Nafees Iqbal, Isabella Attisano, Xue Qin, and Foyzul Hassan. 2023. Virtual Reality (VR) Automated Testing in the Wild: a Case Study on Unity-Based VR Applications. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Virtual Reality (VR) applications provide an immersive experience to end-users with a computer-generated environment that includes scenes and objects that appear real in their surroundings. Although the term virtual reality was introduced three decades ago [1], its real surge started around 2016, with the release of VR devices such as Oculus Rift [2] and HTC Vive [3], and software support from the likes of Unity [4] and Unreal Engine [5]. Indeed, both industrial and personal usage have surged in recent years. According to a 2021 study [6], the global virtual reality market exhibited a significant growth of 42.3% in 2020 compared to the years 2017-2019, and this market is projected to reach over $80 billion within the next seven years. To accurately simulate the user experience and to support multiple areas high-quality VR software is essential. However, existing automated techniques' support for VR software development is still at an early stage with few available tools and frameworks, especially for VR testing. The importance of software testing and quality assurance has been widely confirmed in both academic and industrial communities. To address the challenges of software testing and design appropriate solutions, researchers have carried out many testing-practices studies. These studies employed different approaches ranging from developer-oriented interviews [7] to large-scale quantitative and qualitative empirical studies [8, 9]. Moreover, test-practice investigations have been reported for different application types: mobile applications [10, 11], Machine learning applications [12], and others [13–17]. However, similar studies have found that testing VR applications are a challenging task [18–20]. This is due to factors such as the complex structure of VR projects, their goal of providing a user-immersive experience, and issues related to inadequate tool support for VR development, debugging, and testing activities. Existing VR research activities have focused on development support, such as performance optimization [21], code dependency [22], and code smell detection [23]. In addition, there are several studies on Game applications, such as regression testing for Games [24], and a differentiation study between Game and

Non-Game applications [25]. Yet, none of the existing works studied characteristics of VR software testing in open-source software. To remedy this knowledge gap, we opted to perform a qualitative and quantitative analysis of the existing testing practices in VR applications.

We carried out an empirical study on 314 VR applications in this paper, ranging from small-scale projects to projects backed by large companies and organizations like Microsoft and Unity Technologies, where we analyzed the prevalence, quality, and effectiveness of existing VR tests. Then, we analyzed 370 VR tests in order to build a taxonomy based on their characteristics, and determined which of these categories are VR-attention.

Our main research questions are:

- **RQ1**: To what extent are test cases developed for VR applications?
  Motivation. This question allowed us to estimate the current effort that VR developers put into testing their projects and understand the potential need for VR testing support.
  Answer. We discovered test cases in only 61 out of the 314 VR projects we analyzed. Moreover, we found that the current ratio of code-to-test is very low using both class and method granularities. Indeed, across different VR categories, they were more than 14 times lower than those found by Vidacs et al. [26], who performed similar analyses on test-to-code ratios. We believe these results indicate an urgent need for improved testing practices and support.
- **RQ2**: How effective are the test cases developed in VR applications?
  Motivation. To develop a deeper understanding of the status quo of VR testing, we adopted the assertion density metric to evaluate the effectiveness of the existing tests.
  Answer. The assertion density values are lower than the recommended rates. The medians were lower than 17.46%, which itself is lower than the rate for software applications in general and even lower than the values associated with the comparatively novel mobile applications. In addition, the assertion density values we found are linked to higher bug rates within software projects [27]. This finding indicates that the testing practices of VR projects are less effective than other software project types.
- **RQ3**: What is the design quality of test cases developed for VR applications?
  Motivation. While assertion density reflects on the effectiveness of test methods, it does not inform us of their design quality. With this question, we wanted to evaluate the quality of existing VR Software tests, as we believe these findings can help guide future testing tool design.
  Answer. Using the work of De Bleser et al. [28] as a guide, we analyzed the tests for six smell types. We found that on average 38.43% of a project's tests have at least one smell type, and a project can have as much as 92% smelly tests. This indicates that test smells are common within most VR test methods, lowering their design quality.
- **RQ4**: What are the different categories of VR Test Cases and which categories require specific VR-attention?
  Motivation. With this question, we aimed to discover testing scenarios that reflect characteristics of VR applications. VR applications differ from other application types because of their hardware-specific support, new user experiences, and unique immersive design, among other characteristics.
  Answer. In our study, we manually analyzed 370 randomly-selected VR test methods. In total, we defined 13 main testing categories, such as *Physics Test*, *Animation Test*, *Graphics Test*, *Asset Test*, etc., which are detailed within Section 4.4. Furthermore, We found four main categories and two subcategories are VR-attention, meaning that test writers need to pay special attention to aspects specific to VR applications when writing tests of these categories.

The contributions of this paper are:

- The first quantitative and qualitative study on existing test practices of VR applications.
- The first tool for test effectiveness analysis and test smell identification for Unity-based projects.
- A detailed test case effectiveness analysis via the assertion density metric, and a detailed test-quality analysis through test-smell detection.
- A taxonomy containing 13 main test categories which reflect the characteristics of the VR applications, as well as the identification of VR-attention categories within this taxonomy.

This paper is organized as follows: Section 2 presents the background which defines terms we later use in our manual analysis; Section 3 contains details about the dataset and the methodology used within our automated and manual analysis; Section 4 describes the evaluation which answers all four research questions, followed by threats to validity in Section 5. Section 6 includes related work. Section 7 describes the implications of this work, and we conclude our research in Section 8.

2 Background

Automatic software testing allows developers to test application code in an automatic, rapid, and reliable way. Similar to traditional software applications, automated software testing can also be applied to VR applications. In this study, we analyzed the test characteristics of 314 Unity-based VR applications. We focused on Unity as it is one of the most popular frameworks for developing VR applications [29]. We found that VR tests mainly focus on the behavior of the different VR subsystems and class components. However, while discussing VR testing, familiarity with how Unity works and some technical terms is required. These details are presented in the following paragraphs:

Physics System: ensures that the virtual objects correctly respond to different forces such as collision and gravity. The Unity platform provides Rigidbody APIs which allow the usage of the physics engine to control the objects. Collision and Colliding are also important aspects of VR projects, as they define how virtual objects react to overlapping with or without physics effects, respectively.

Graphics System: enables developers to control the appearance of VR applications. It includes Rendering, Display, Camera, Lighting, etc. In 3-D graphic design, rendering is the process of adding shading, color, and lamination to a 2-D or 3-D wireframe in order to create life-like images on a screen. This process can be preloaded or occur in real-time when users interact with VR applications. Display is related to displaying the rendered objects within the VR scene, which users view through hardware such as monitors or head-mounted devices. Unlike non-VR applications, VR applications can create multiple cameras in the same scene, and the display

will update when the camera switches or the location changes. The camera represents the view angle from which the user sees the virtual world.

Animation System: allows developers to animate target objects via jumping, moving, stopping, rotating, etc. Animation design in VR applications is more complicated than its non-VR counterpart. With fixed angles in non-VR applications, animation design is a linear process that focuses on the representation from a locked direction or view. However, in VR applications, animating user surroundings is a parallelized process, as developers need to ensure the correctness of the representation from any possible angle.

Other terms: *GameObject:* the fundamental class for all the objects in a virtual world. By combining its different controls and features, developers can use it to enable custom functions such as moving objects. *Colliders* represent the invisible physical shapes of objects. The Physics system uses them to decide physical effects such as those that occur when objects overlap.

3 Research Approach

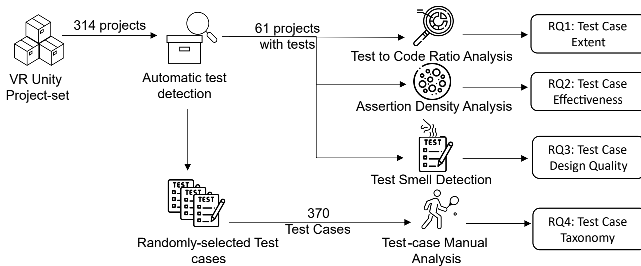

Figure 1: Overview of Research Approach

The research approach overview is illustrated within Figure 1. In this section, we will first introduce the studied dataset, then describe the AST-based automatic analyses used to measure the test prevalence, efficiency, and quality, and eventually discuss the manual analysis used to discover the taxonomy of VR tests as well as identify VR-attention test categories, where test writers need to pay special attention to the aspects of VR.

3.1 Dataset

Within this work, our goal was to study a group of Open Source Virtual Reality software projects that were built with Unity, as it is one of the most popular Game and VR development engines [29]. Initially, we started with the dataset of Nusrat et al. [21] as it contains 100 manually-verified VR projects that use Unity, three of which were unobtainable due to de-listing. We expanded our set with projects from GitHub. Multiple search queries for VR projects based on Unity were executed, using keywords such as *VR*, *Virtual Reality*, *VR Unity* etc., and applying the same conditions as those specified by Nusrat et al. [21] regarding the number of commits. The results of these queries were then merged and independently reviewed by two co-authors in order to extract projects that were Unity-based VR software. After discussion to resolve any differences, the authors added an additional 217 projects to the dataset. Out of the 314 projects within our set, 164 are *Independent*, generally owned and maintained by one contributor and with few secondary contributors, 67 are *Organizational* projects, generally owned by an Organization on Github and backed by companies and organizations like Microsoft, Vive, etc., and finally, 83 *Academic* projects, which are composed of class projects, assignments, and research projects maintained by students. All the VR projects within our set contain a minimum of 100 commits, with commit averages of 954.44, 365.47, 359.28 and commit medians of 345, 205, and 220 respectively for Organizational, Independent and Academic Projects.

Furthermore, they have averages of 119.20, 4.5, 7.37 and medians of 7, 3, 5 for the number of committers, and averages of 830, 655.17, 508 and medians of 641, 541, 337 for the number of days between their first and most recent commits, respectively for Organizational, Independent and Academic projects.

After performing the automatic analysis described within 3.2.2, we found that 61 projects within our set contained one or more tests, 21 of which were Academic, 24 of which were Independent, and 16 of which were Organizational.

3.2 Methodology of Automatic Analysis

3.2.1 Static Analysis of Unity Projects Since Unity makes use of the standard .Net Framework, alongside its internal frameworks and classes, and the C# programming language for its scripting [30], we needed to use an AST generator that supported these technologies. We opted to use SrcML [31] to generate the ASTs of the Unity C# code. SrcML is a research tool that allows the generation of ASTs for various programming languages, which facilitates the extension of our approach and tools to other sets of projects that use different programming languages. It supports C#, and does not rely on compilation to generate ASTs, thus allowing us to avoid any compilation-related issues. For each VR project, we analyzed its repository to extract the C# code files and then generated the AST of each file to perform the different analyses described in the following sections. It's important to note that the majority of tests we found were Unit and Integration tests. Although different testing approaches have been developed for Game projects such as Search-based, and Goal-directed testing, as discussed by Albaghajati et al. [32], and which by extension can be applied to VR Game projects, we have found no evidence of their usage within our set.

3.2.2 VR Test Cases Prevalence In order to evaluate the prevalence of test cases within our set of VR projects, we counted the number of test methods and classes, and functional methods and classes of each project. According to the Unity documentation [33] and via the manual analysis we performed, detailed in 3.3, we found that test cases are labeled with `[Test]` or `[UnityTest]` or `[MTest]`, and test classes are those that contain one or more test cases. Using these data points, we calculated the following metrics for each project:

$$TestToCodeMethodRatio = \frac{Count(TestMethods)}{Count(FuncMethods)}$$
$$TestToCodeClassRatio = \frac{Count(TestClasses)}{Count(FuncClasses)}$$

Based on the work of Klammer et al. [34] and Williams et al. [35], these metrics adequately represent the relative frequency of test code within our VR projects.

3.2.3 VR Test Cases Effectiveness A practical way of measuring the effectiveness of tests in a software project is to calculate the Code Coverage [36], which denotes the degree to which the functional

code is executed after a test suite finishes running. However, we faced a plethora of problems when attempting to generate these reports due to issues ranging from compilation problems to a lack of compatibility of code coverage measurement with some versions of Unity.

To circumvent these issues, opted to use the Assertion Density metric to evaluate the effectiveness of the test cases we collected. This metric is calculated via the ratio of the number of assertions to the length of test cases that contain them. The usage of this metric within previous works regarding testing supports its effectiveness [27, 37]. The equation for this metric is:

$$AssertionDensity = \frac{NbAssertions}{TestLOCs}$$

3.2.4 VR Test cases Design Quality Having collected information about the prevalence and effectiveness of tests within our project set, we wanted to learn more about their quality. In order to achieve this goal, we scanned these tests for test smells [38–40]. Test smells are similar to regular code smells in being symptomatic of technical debt and predicting future problems, but they are specific to test code. We considered six test smells from the work of De Bleser et al. [28], as they were found to be the most prevalent in a collection of previous works [41, 42]. These smells are:

Assertion Roulette (AR): This occurs when a test case contains more than one assertion, and more than one of which does not provide a message when detecting an issue. This makes it hard to diagnose which problems are present within the functional code being tested.

General Fixture (GF): A test fixture is too general if it initializes fields that are not used by one or more test methods, making it difficult to discern which fields are being shared by the different test methods within a test class.

Sensitive Equality (SE): If a test case has an assertion that compares the state of objects by comparing their representations as text, for example by using their ToString() methods, it makes itself susceptible to errors due to irrelevant textual representation details such as spaces.

Eager Test (ET): A test that evaluates more than one functional code method with the same fixture is *Eager*. This smell violates the principle that every test case should only test one method, and that one test failure should only signify issues within one specific method, not another irrelevant method.

Lazy Test(LT): A test case is lazy if it tests the same functional method using the same fixture as another test method. The problem this smell implies is that after modifying one function that is being tested, multiple test methods may need to be updated accordingly. Thus this smell affects the maintainability of the test cases.

Mystery Guest(MG): A test case has this smell if it uses external resources that are not managed by a fixture or are not Mock objects. This smell may cause issues since external resources might change over time or be unavailable during test-case execution. For example, a test method can fail if a specific database or file is not available during its execution.

3.2.5 Evaluation of smell-detection tool To evaluate the accuracy of our smell-detection tool and verify the correctness of our findings, we first applied our automated approach to four randomly selected VR projects which were manually determined to have tests within them. Then two co-authors manually evaluated the same projects separately by labeling the test methods with any corresponding smell types. Both co-authors found 220 test methods within these four projects, which is the same number found by the tool. An average Cohen Kappa [43] of 0.92 was found between the authors across the different smell types, signaling high agreement, and any differences were then resolved via discussion. Upon evaluating the automatically detected smells using the manual observations as a baseline, on average, an accuracy of 91.35%, a recall of 92.62%, and an F-1 score of 91.98% were found across the aforementioned test smell categories. As noted in Section 4.3, no instances of Lazy Test and Sensitive Equality were found via the manual or automatic analyses we performed. To verify the correctness of our tool for these smells, one co-author, who did not work on developing the test analysis tools, developed one test stub for the sensitive equality smell and two test stubs for the Lazy test smell and then verified that these smells were correctly detected.

3.3 Methodology of Manual Analysis

To explore the characteristics of VR test cases we carried out a manual analysis with a focus on discovering unique testing scenarios, exploring the test design patterns, and testing goals. Using our automatic results generated within 4.1, we constructed our initial set of tests, which was composed of 8723 tests. In order to construct a representative sample with 95% confidence and 5% error, we randomly selected 370 tests for manual analysis. We believe relying on a random selection process allows us to obtain a diverse set of tests that should contain the most prevalent test categories. These tests were from 36 different projects, out of a total of 61 VR projects with tests. Since there were no existing categories for authors to use as a reference when this study was conducted, we designed our manual approaches to minimize bias, follow a similar process to existing works which created taxonomies in software [44, 45] and to abide by the recommendations outlined by Usman et al. [46] concerning the construction of taxonomies. First, for the planning phase, three co-authors agreed on the area being VR testing, the goal being the identification of which aspects of VR projects the test cases are trying to validate, and the classification structure as a tree. Furthermore, it was determined that a qualitative approach would be optimal and that the tests would be selected from the existing set of VR projects with tests. Second, for the identification and extraction phase, two of the co-authors separately observed all the test methods and their related source code and also performed an exploration of Unity documentation and VR developers' forums to generate a comprehensive report for every test method. This report included details such as Unity API calls, observed tested target and environment behavior, test scenario description, text method code pattern, and corresponding tested functional code pattern. Then, the third co-author with previous VR experience was asked to rejoin this process. This was done in order to minimize the bias, as this author did not participate in deciding the categorization of the VR test code designs that were uncovered in the earlier step. These three co-authors then categorized all 370 test methods separately by reviewing the observed records. Eventually, voting within three rounds of consensus meetings was carried out to finalize the results and resolve any disagreements and remove any redundancies and inconsistencies. Before resolving the disagreements, Fleiss'

Kappa [47] coefficient was calculated, and was 0.54, indicating a moderate agreement between co-authors. Third, for the Design and construction phase, in order to construct the taxonomy composed of the different VR testing categories, a card-sorting process [48] was performed by the three authors, which grouped test methods of similar characteristics into Main categories, then where applicable, appropriate sub-categories. As noted within Section 4.3, a trend of one test case testing multiple functionalities was observed, hence, some tests may fall into more than one category within this taxonomy. Furthermore, specific examples along with more general definitions were provided in order to facilitate the usage and adoption of this taxonomy. Finally, for the Validation phase, a fourth co-author with VR development experience was able to verify the relevance of the different categories by connecting them to different aspects of VR software development and documentation of VR development within Unity. Similar to the results of the automatic analysis, the majority of tests we manually analyzed were Unit and Integration tests.

4 Empirical Evaluation

4.1 RQ1: Extent of Developing VR Test Cases

In order to identify the prevalence of test code in VR applications, we calculated the Method and Class ratios of test code in relation to functional code for VR projects with test cases. Recommended practices indicate that the optimal test code to functional lines of code ratio is 3:1, or in more general cases between 1:1 and 1:10 [49]. Indeed, it is recommended to add test code in parallel to functional code, where one test class should evaluate one functional class, and one test method should evaluate one functional method. This suggests that when considering the Method and Class granularity of ratios, the ideal values are close to 1. However, the results illustrated in Figure 2 show that test code only represents a small portion of VR projects' code and that current VR testing prevalence is far from the ideal scenario for all three project types. This is especially outlined through the median Method Count Ratios (MCR) of 0.04, 0.03, and 0.04 and Class Count Ratio (CCR) of 0.05, 0.06, and 0.02 for Independent, Organizational, and Academic Projects, respectively. Indeed, these are significantly worse in comparison to other categories of software projects, such as industrial Java projects, analyzed by Klammer et al. [34], where the equivalent LOC ratio was around 0.6 for the totality of the code-base, or C# projects by Microsoft [35], where the LOC ratios were between 0.35 and 0.89. Furthermore, an ANOVA analysis reveals that the P-values for MCR and CCR were respectively 0.26 and 0.06, revealing no significant statistical difference in the values of these metrics across project categories.

4.2 RQ2: Effectiveness of VR Test Cases

As discussed within Section 3.2.3 using Unity to generate code coverage reports for our project set has proven challenging, and no other tool allows the generation of these reports. This highlights the lacking tool support in Unity for testing-related activities and is the main reason why we opted to use Assertion Density to approximate our project-set tests' effectiveness. When considering the metric of Assertion density, illustrated within Figure 3 for projects that had test cases, it is clear that these values are not very

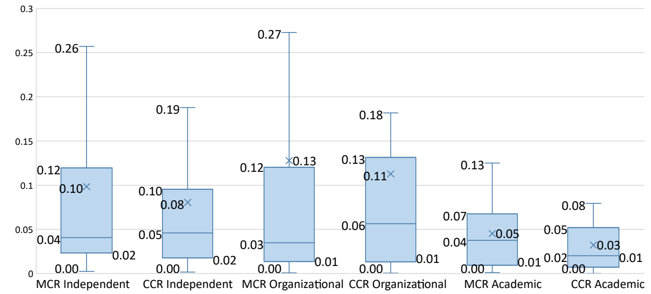

Figure 2: Method Count Ratio (MCR) and Class Count Ratio (CCR) per Project Category (outliers removed)

encouraging. Indeed, the median values are 14.57% for Independent projects. 17.46% for Organizational projects, and 14.74% for Academic projects. These values are even lower than those found within mobile applications [10], and they are linked with less effective testing practices [50]. In addition, the ANOVA analysis reveals that the P-value for this metric is 0.56, indicating no substantial difference across project categories. These results hint at a massive need for testing support for all the different types of VR projects.

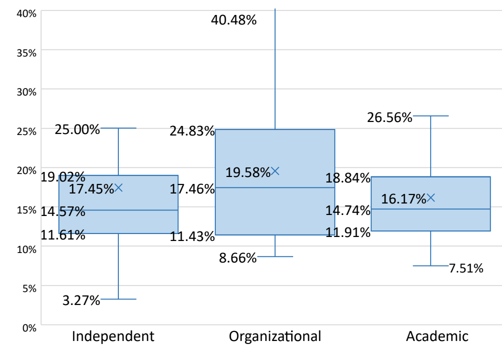

Figure 3: Assertion Density (outliers removed)

4.3 RQ3: Design Quality of VR Test Cases

After evaluating the extensiveness and effectiveness of test cases developed within our project set, we analyzed them to evaluate their quality using the method outlined in Section 3.2.4, and we obtained the following results:

Assertion Roulette(AR): It is clear within Figure 4 that the AR smell is quite common within our set of VR projects. Indeed, 78% of the projects across categories had assertion roulette in 20% or more of their test cases, 18 of which were Independent projects, 19 of which were Organizational projects, and the rest were Academic projects.

```
1 [UnityTest, Order(7)]
2 public IEnumerator TestListFeedback(){...
3     Assert.IsNotNull(listFeedbackRes.Feedback);
4     Assert.IsTrue(listFeedbackRes.Feedback.Count>0)
    ↪ ;...}
```

Listing 1: Assertion Roulette Smell from `unity-sdk`

Listing 1 is an example of an Assertion Roulette from the `watson-developer-cloud@unity-sdk` project, where the test is attempting to verify whether the feedback object meets the developer's expectations. It would be difficult to diagnose the exact cause of this

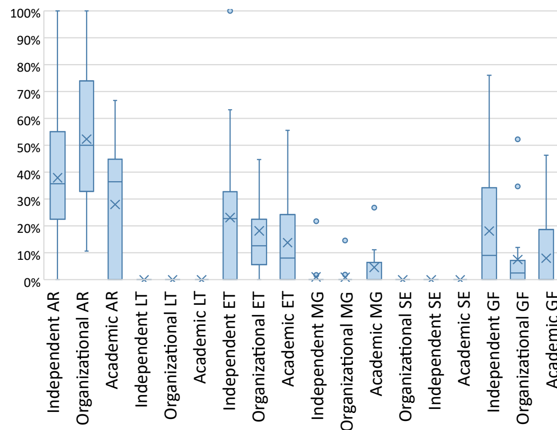

Figure 4: Summary of Detected Test Smells

test's failures. For example, whether the feedback response is null, or whether it's empty, as no messages are given in the assertion statements in lines 3 & 4.

Eager Test(ET): For the ET smell, represented within Figure 4, it is clear that this smell is also common within our VR project set. Indeed, we found that 23 projects have this smell within 20% or more of their test cases, 12 of which were independent, 6 of which were Organizational and the rest were Academic.

```
1 [Test]
2 public void CoordinateSystemsTest(){...
3     CubeUVCoordinates cubemap=cartesian;
4     Assert.IsTrue(Miscellaneous.approximately(cartesian
  ↪ .data,cubemap.reconverted_cartesian.data),"..."
  ↪ );
5     OctahedronUVCoordinates octahedron_map=cartesian;
6     Assert.IsTrue(Miscellaneous.approximately(cartesian
  ↪ .data,octahedron_map.reconverted_cartesian.data
  ↪ ),"...");}
```

Listing 2: Eager Test Smell from `Planetaria`

Listing 2 is an example of an Eager Test from `mattrdowney@Planetaria`, where the test is attempting to verify that the cubemap and octahedron maps are storing and converting coordinates correctly from their Cartesian form. However, it would be difficult to diagnose which of these coordinate conversion processes is failing, since the test would fail if either or both are problematic.

General Fixture(GF): The GF smell is quite widespread within our project set, as illustrated within Figure 4, and was found in 10% or more tests of 17 projects. Out of these, nine projects were Independent, four were Organizational, and the rest were Academic.

```
1 GameObject obj; VimeoRecorder recorder;
2 [SetUp]
3 public void _Before() {...
4     obj = new GameObject();
5     recorder = obj.AddComponent<VimeoRecorder>();}
6 [Test]
7 public void Init_LookingGlass(){...
8     asset = GameObject.Instantiate(...);
9     Assert.AreEqual(recorder.renderTextureTarget, null)
  ↪ ;}
```

Listing 3: General Fixture Smell from `vimeo-unity-sdk`

Listing 3 represents a General Fixture smell within `vimeo@vimeo-unity-sdk`, where the test is using the recorder object initialized by the `_Before()` fixture, but not the `obj` object. This causes a non-optimal memory consumption due to the initialization of an object that's not used by the test method. It may also make it more difficult to evolve the test cases due to ambiguity related to determining which objects are being used by the tests. For example, the test case described above initializes and uses a new game object instead of the one initialized by the game fixture.

Mystery Guest(MG): Unlike the aforementioned test smells, Mystery Guest is less common within our project set, as shown by the results within Figure 4. Indeed, seven projects had this smell within 3.5% or more of their test cases. One of these projects was Independent, two were Organizational and the rest were Academic.

Listing 4 represents a Mystery Guest smell within `Willychang21@MapboxARGame`. The issue this smell causes is that this test will fail if it is run within an environment where the database is not available, such as a CI environment or a different machine. This smell goes against the recommendation of using a mock object to programmatically represent an external resource during testing.

```
1 private SQLiteCache _cache;
2 [Test]
3 public void VerifyTilesFromConcurrentInsert(){
4     Assert.AreEqual(_tileIds.Count,_cache.TileCount(
  ↪ tilesetName),"...");}
```

Listing 4: Mystery Guest Smell from `MapboxARGame`

Lazy Test (LT) and Sensitive Equality(SE): We did not discover any examples of these smells through the use of our automatic smell-detecting tool or the manual inspection used to verify its accuracy. In the case of the SE smell, we found that most comparisons of objects within test cases were based on a specific object's property. Furthermore, based on the findings in Section 4.1, it's clear that the developers of the VR project-set write much less test code than functional code, thus making it less likely that they would write multiple tests for the same method, which in turn causes the LT smell. The results we found regarding the ET smell earlier within this section point to the opposite practice of testing multiple functional code methods within the same test being the more common practice.

We contextualized the overall test smell detection results from VR projects by comparing them with those from a similar test smell study of open-source Android applications by Peruma et al. [11] While 58.46% of files analyzed in Android exhibited the Assertion Roulette smell, an average of 41.43% of tests within VR projects we analyzed possessed this smell. A similar trend is noted for the other smells as well, where Eager Test was found in 38.68% of Android projects' tests; it was found on average within 19.52% of our VR projects' tests. This trend continues for General Fixture, which was found within 11.67% of Android projects' tests and within an average of 11.72% of VR projects' tests. For Lazy Test, Sensitive Equality, and Mystery Guest smells, the three means were close to 0% for VR projects, and respectively were 29.50%, 9.19%, and 11.65% within Android projects. Overall, both application types show a comparable and problematic frequency of the different test smells. However, the situation in VR applications is considerably worse, considering the lack of prevalence and effectiveness of its testing overall. Delving into the different VR project categories, the only test smell category for which we found a significant statistical

difference was Assertion roulette, with a P-value of 0.01. Indeed, through Figure 4, it's clear that Organizational and Independent projects display AR more severely than their Academic counterparts. For the ET, GF, and MG smells, the p-values were respectively 0.42, 0.12, and 0.11, indicating that there is no statistical difference across the various project categories for these smells. A similar conclusion can be reached through observing in Figure 4; however, it's important to note that Independent and Academic projects display more variance in the frequencies of these smells. Overall, test design quality is consistently problematic across different project categories.

4.4 RQ4: Categories of VR Test Cases

To categorize the test types that are present within VR applications, we followed the methodology discussed in Section 3.3 and carried out a manual analysis of 370 test methods from 36 VR projects. We divided all the test methods into 13 different main categories. Some of these methods fall under multiple categories. Figure 5 represents the taxonomy that we have generated based on a parent-child hierarchy. The numbers in each category indicate the number of test cases we discovered from independent projects, organization projects, academic projects, and the total number from all projects, respectively. In the rest of this section, we discuss each of the test categories as well as their sub-categories by giving detailed definitions. We also present code examples to help avoid the bias of definition descriptions, but these definitions can also be expanded for usage in other VR frameworks.

4.4.1 GameObject Property Tests As explained within Section 2, GameObject is the fundamental class for all entities defined in the Unity framework. It can be used to represent characters, props, environment, or other elements. This category focuses on assessing Game Object properties such as active, tag, and static. In total, we found 27 test cases in this category.

4.4.2 Audio/Video Tests validate audio and video playback functionality, including the ability to play, pause and stop audio and video clips. We found seven test methods that fall into this category.

4.4.3 Physics Tests verify the accuracy of three-dimensional designs of one or more objects are evaluated by the Physics Test [51]. For a single target, they assess the properties and controls of the physics simulation. For multiple targets, they examine the collision/colliding behaviors and the effects of different forces at specific locations and times. The observed subcategories are:

Rigidbody and Character Tests. Both Rigidbody and Character are control components used in VR physics simulation. Rigidbody enables control of objects through the Unity physics engine [52]. It allows interaction with physics-based movements, which include forces, gravity, mass, and momentum. Rigidbody Property Tests are concerned with physical behaviors such as movement and position. Character Tests, on the other hand, concentrate on validating the physical characteristics of a character, such as the names of their body's bones, skeleton size, etc. In total, we found two Rigidbody Tests and three Character Tests.

Colliding Tests. Colliding in VR simulates non-physical effects when multiple objects come into contact with each other. Colliding tests mainly verify the application design logic. In total, we found 14 tests.

```
1 [UnityTest]
2   public IEnumerator ShouldRelocatePowerupWhenColliding
      ↪ (){
3     Vector3 pos = powerup.transform.position;
4     MakePlayerCollideWith(powerup);
5     yield return new WaitForSeconds(TestConstants.
      ↪ WAIT_TIME);
6     Assert.AreNotEqual(powerup.transform.position, pos);
7   }
```

Listing 5: Colliding Test from `jet-dash-vr`

Listing 5 shows an example of a Colliding test from the project `iamtomhewitt@jet-dash-vr`. This test evaluates whether the powerup object will be relocated after colliding. The test first records the initial position to `pos` for powerup object in line 4. Then, it calls the function MakePlayerCollideWith() to make the player object contact with the powerup object in line 5. After waiting for a fixed time for the system update, an assertion is inserted at line 7 to check whether the new position is different from the initial position.

Collision Tests. Collision is very similar to colliding but with a realistic physical effect. The two terms are often used interchangeably in VR apps. We separate them because of their different API usage and consequences. Collision tests follow the three-stages-design: before, during, and after collision. We identified six collision tests.

4.4.4 GUI Tests evaluate the layouts, input fields, and text of the application's interface to ensure that they work correctly. We discovered that the scope of visual components in VR is beyond traditional GUI widgets. Tests can encompass anything from a simple toggle button to a fully immersive 3D environment. The test target includes traditional GUI elements and VR-customized immersive experiences. In total, we identified eight GUI tests.

```
1 [UnityTest]
2 public IEnumerator TestGazeCursorArticulated(){
3     Assert.IsTrue(inputSystem.GazeProvider.GazePointer.
      ↪ IsInteractionEnabled, "Gaze cursor should be
      ↪ visible at start");
4     Assert.IsFalse(inputSystem.GazeProvider.GazePointer.
      ↪ IsInteractionEnabled, "Gaze cursor should not
      ↪ be visible when one articulated hand is up");
5 }
```

Listing 6: GUI Test from `MixedRealityToolkit-Unity`

Listing 6 illustrates an example of a GUI test from the project `microsoft@MixedRealityToolkit-Unity` in the file FocusProviderTests.cs. The goal of this test method is to check if the gaze cursor behaves properly with articulated hand pointers. Line 4 checks if the gaze cursor is visible before the interaction begins. Then, in line 5, the assert statement is evaluating the invisibility of the gaze cursor.

4.4.5 Animation Tests Animation means objects are in action. Tests in this category verify the correctness of the movement functionalities and properties of test targets. For example, their location updates, properties of their movement, stopping, and rotation. The test cases we observed all follow a time-order-oriented design, where an assertion is inserted to compare the status of a test target before and after system updates. Some tests also verify the target's animation clips which refer to a sequence of animated movements. In total, we identified four Animation Tests.

4.4.6 Network Tests Networking and multiplayer support are two features in VR applications that require local or wide-area network

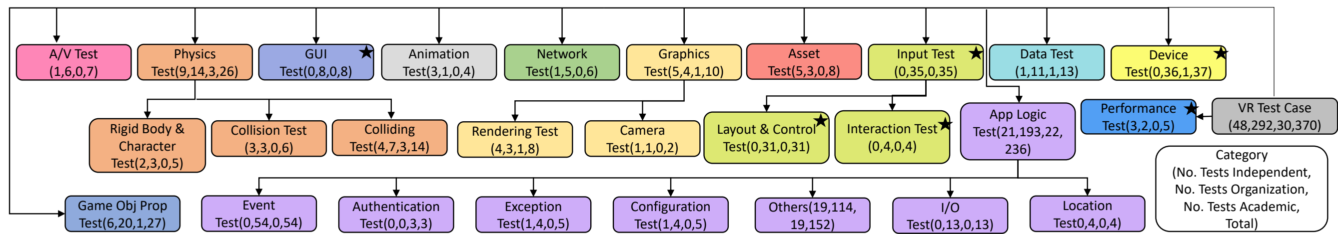

Figure 5: Taxonomy of VR Testing Categories (⋆ indicates a VR-attention category)

access. These tests check whether network calls and responses are correct, such as response count and response data. They set the connecting and receiving ports and ensure that the receiving end is configured correctly and that data is transmitted between the sender and the receiver without any issues. Six tests of this category were found.

4.4.7 Graphics Tests assess how visually appealing VR applications are. The observed subcategories are:

Camera Property Tests. In Virtual Reality (VR) applications, developers often incorporate one or more cameras into the virtual environment to offer users various perspectives of an immersive experience. These test methods verify the rotation of the camera on different axes within a given range, as well as the transitions between the camera's life cycle activities, such as when a camera should be active or inactive in various modes. We identified two camera tests in total.

Listing 7 shows an example of a Camera Property Test from project `stefaanvermassen@virtual-museum-app`. This test evaluates whether the camera can rotate correctly along with the test player. A first-person controller named `player` has been initialized in line 3. Then in line 5, it is vertically rotated 50 degrees. Eventually, an assertion in line 8 is evaluating whether the camera has rotated with `player` to the correct position.

```
1  [Test]
2  public void PlayerCharacter_Rotate_Rotated(){
3     FirstPersonController player = GameObject.
       ↪ FindObjectOfType<FirstPersonController> ();
4     . . .
5     player.RotateVertical (50);
6     . . .
7     cameraRot = camera.transform.localRotation.
       ↪ eulerAngles;
8     Assert.That(cameraRot.x, Is.EqualTo(50).Within
       ↪ (0.005), "Camera should reach target rotation
       ↪ of 50");
9     . . .}
```

Listing 7: Camera Property Test from `virtual-museum-app`

Rendering Tests. evaluate factors including texturing, lighting, image effects, and color correctness to determine the quality and accuracy of the visual representation of a Virtual Reality (VR) environment. We found eight tests of this category.

```
1  [Test]
2  public void EmissionColor(){
3      using (var tex = DisposableObject.New(wrapper.
       ↪ GenerateToonLitImage(setting)))
4      using (var main = DisposableObject.New(TestUtils.
       ↪ LoadUncompressedTexture("albedo_1024px_png.png"
       ↪ )))
5      Assert.Less(TestUtils.Difference(tex.Object,
       ↪ computed.Object), 1e-4);}
```

Listing 8: Rendering Test from `VRCQuestTools`

Listing 8 shows an example of a Rendering test from project `kurotu@VRCQuestTools` with the purpose of evaluating color emission. In line 3, an image is generated by adjusting the settings of the image. Then, in a main variable, a PNG image file is loaded. Finally, in line 5, the assert statement calculates the average difference between two rendered textures which returns the pixel difference between textures.

4.4.8 Asset Tests An asset is any item that the developer uses to create the VR application. Assets include visual, audio, or other elements like models and textures. An asset can either be external as a file or internal as data from the editor. Asset Tests verify the assets during the importing, loading, unloading, distributing, and other processes. Correctly managing a large number of assets is a demanding task for VR developers. We found eight tests of this category.

4.4.9 Input Tests The Input system is one of the largest components in VR applications. It serves as a bridge that connects the user's actions to the application's response. Input tests evaluate the accuracy and responsiveness of the VR application to user inputs, ensuring that user actions are correctly interpreted into designated responses. The two subcategories we observed are:

Layout and Control Tests. Besides the traditional user controls such as clicking and swiping, VR application provides users with more enhanced immersive input controls such as hands, eyes, and body tracking. To support different types of VR devices, developers design their app based on an internal control layout, which is later mapped to a particular user's device. Layout and Control tests validate the correctness of the layout mapping and the controls that bind to it. We found 31 tests in this subcategory.

```
1  [Test]
2  public void ButtonsArePackedByTheByte() {
3       runtime.ReportNewInputDevice(
        ↪ ButtonPackedXRDeviceState.
        ↪ CreateDeviceDescription().ToJson());
4       InputSystem.Update();
5       var layout = InputSystem.LoadLayout("XRInputV1::
        ↪ XRManufacturer::XRDevice");
6       Assert.That(layout, Is.Not.Null);
7       Assert.That(layout.controls.Count, Is.EqualTo(
        ↪ kNumBaseHMDControls + 8));
8       var currentControl = layout["Button1"];
9       Assert.That(currentControl.offset, Is.EqualTo(0));
10      Assert.That(currentControl.layout, Is.EqualTo(new
        ↪ InternedString("Button"))); }
```

Listing 9: Layout and Control Test `Unity-Technologies@InputSystem`

Listing 9 shows an example of a Layout and Control Test. This test case is checking if the button1 layout of a specific XRDevice from a particular manufacturer can be successfully loaded and parsed by the VR input system. In lines 3 and 4, the runtime system detects new devices and updates input system. The entire layout is loaded

in lines 6-8. The "Button1" from this layout is checked from line 10 to line 12.

Interaction Tests. evaluate the response of the whole system to a complicated user input, such as a sequence of interactions. We found four tests in this subcategory.

```
1 [UnityTest]
2 public IEnumerator TestEyeTrackingTargetMultipleTargets
    ↪ () {
3   . . .
4   TargetObject1.AddComponent<EyeTrackingTarget>();
5   TargetObject2.AddComponent<EyeTrackingTarget>();
6   . . .
7   inputSimulation.EyeGazeSimulation = EyeGaze.
    ↪ CameraForwardAxis;
8   . . .
9   Assert.True(EyeGazeProvider.GazeTarget ==
    ↪ TargetObject1);}
```

Listing 10: Interaction Test from `MixedRealityToolkit-Unity`

Listing 10 shows an example of an Interaction Test. It aims to check the correctness of the eye-tracking system's response to multiple target objects. From line 3 to line 6, this test creates `TargetObject1` and `TargetObject2` and adds them to the eye-tracking system. In line 7, eye gazing is simulated to be in the same direction as the camera. Based on the design logic, `TargetObject1` should be selected and passed to the assertion in line 9.

4.4.10 App Logic Tests are relevant to a specific VR application based on its business logic and functionalities. A total of 236 tests were found in this category. To provide a more complete understanding of the App Logic Test category, we divided it into seven different sub-categories. (1) Event Tests: evaluate the correctness of a response after triggering an event by using the event listener. For example, testing whether the motion controller was interactive by checking click events or examining the stage of the event lifecycle. (2) Authentication Tests: ensure a correct data verification using token error messages and token statuses. (3) Exception Tests: check if a test method behaves as intended during the execution of the program. (4) Configuration Tests: examine the environment variables, sessions, and configurable properties of different objects before evaluating the actual test method. (5) Others: tests that validate utility functions and domain-specific logic of VR projects. Utility functions often include common code and often-used methods. Domain logic is the underlying business logic of a specific VR project. (6) I/O Tests: Test interaction with the local filesystem. (7) Location Tests: verify the position values (e.g., longitude, latitude) in a three-dimensional world.

4.4.11 Data Tests involve dataset evaluation through caching systems, local file systems, or other local and remote databases. These tests confirm that a VR application is storing, retrieving, deleting, and updating data appropriately. For example, testing whether multiple objects can be added to a cache system concurrently. PlayerPrefs, a built-in Unity API that helps developers quickly access internal data between frames and across multiple VR scenes, is a commonly-tested structure within these tests. In total, we identified 13 Data Tests.

4.4.12 Performance Tests are found within traditional software [53] as well as VR software. They can be used to evaluate the VR applications' ability to handle complex interactions and high levels of visual details in a correct and timely manner. For example, a performance test can be validated by executing the same test on different gamepads by measuring the state of gamepads after a fixed interval. We observed five tests in total.

Listing 11 shows an example of a performance test from project `tterpi@VRSketchingGeometry`. This test aims to validate the performance of setting a large number of control points on a sketch object in VR world. The `Measure.Method` in line 4 is a measurement API in Unity Performance Testing Extension.

```
1 [Test, Performance]
2 public void SketchObject_SetControlPoints_Performance([
    ↪ Values(3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30,
    ↪  40, 50)]int length) {List<Vector3>
    ↪ controlPoints = GenerateControlPoints(length);
3 Measure.Method(() =>{this.LineSketchObject.
    ↪ SetControlPointsLocalSpace(controlPoints);}).
    ↪ Run();}
```

Listing 11: Performance Test from `tterpi@VRSketchingGeometry`

4.4.13 Device Tests can be used to assess the compatibility and functionality of VR hardware and software. This test is important due to the market's wide range of VR devices, such as head-mounted displays (HMDs), controllers, and mobile devices. We found 37 tests in total.

```
1 [Test][Category("Devices")]
2 public void Devices_CanGetDescriptorFromHID(){
3     var device = (HID)InputSystem.devices.First(x => x
      ↪ is HID);
4     Assert.That(device.hidDescriptor.productId, Is.
      ↪ EqualTo(1234));
5     Assert.That(device.hidDescriptor.vendorId, Is.
      ↪ EqualTo(5678));
6     Assert.That(device.hidDescriptor.elements.Length,
      ↪ Is.EqualTo(1));
7 }
```

Listing 12: Device Test from `Unity-Technologies@InputSystem`

Listing 12 shows an example of a device test from project `Unity-Technologies@InputSystem`. In line 5, the first device value from the list of devices is stored. Then in lines 6-8, assertion statements check the device's product id, vendor id, and the number of connected device are checked. Here, the device is GenericDesktop only, that's why `element.length` is equal to 1.

Overall, we generated a VR Testing taxonomy of 13 main categories and 14 subcategories. We believe that this generated taxonomy provides a comprehensive overview for VR developers and testers, enabling them to identify potential areas and aspects of testing that may have been overlooked in their projects. Additionally, this taxonomy expands the knowledge of VR researchers, providing them with insight into different research directions they can pursue. This taxonomy is especially important due to the apparent lack of established VR testing practices, as well as the several issues noted with testing prevalence, effectiveness, and quality, as found within Sections 4.1 to 4.3, which is concerning since modern VR headsets and their applications have been around since 2016 [54].

While building the final taxonomy, we primarily relied on the Unity documentation and our own Virtual Reality expertise. However, the Unity platform caters to both VR and non-VR applications, and many test designs and APIs overlap, particularly when it comes to a 3D world. Consequently, most of the categories in the taxonomy are relevant for testing in both VR and non-VR applications.

To assist VR developers in their test design, we have identified the test categories that require VR customization. We define a VR-attention test as a test that involves VR devices and VR experience, which are less likely to appear in non-VR applications. Categories that have one or more VR-attention tests are labeled as VR-attention categories.

Out of the 13 categories in Figure 5, we found four to be VR-attention categories: GUI Test, Input Test, Performance Test, and Device Test. While these categories are typical of traditional software and are not unique to VR applications, we recommend that VR developers consider VR-context adaptation when designing these tests. For instance, they may need to design an input event that simulates eye gazing or to evaluate their project's connection with an HMD device.

In GUI Test, were observed VR-attention test cases that focus on the VR experience. For example, one test method creates a sequence of hand movements to simulate actions like touch, and scrolling and evaluate the correctness of designated GUI effects. In the Input Test, the observed VR-attention test cases focus on both VR experience and VR devices. For example, one test method in Layout and Control Test focuses on validating the control layout load from the XR devices of certain manufacturers, and another test method in Interaction Test focuses on validating how eye-tracking system could capture the target object with other disturbing objects. In Performance Tests, the observed VR-attention test cases focus on the scalability of the VR experience. The tested app is a VR sketcher, which allows users to draw 3D lines and shapes. Users can freely twist and drag 3D lines by adding control points. The performance test evaluates the app's ability to handle a large number of control points. In Device Tests, the observed VR-attention test cases mainly focus on VR devices. For example, one test method validates whether a VR hardware device can be automatically configured by pre-defined identities.

5 Threats to Validity

On construct validity, the main threat is the soundness of the automatic analysis results. To measure the design quality of the test cases, we followed the work of De Bleser et al. [28] and selected six test smell types, including Assertion Roulette, General Fixture, Sensitive Equality, Eager Test, Lazy Test, and Mystery Guest. To further validate the automated reports, we followed a procedure detailed within 3.2.5 which resulted in a 91.98% F-1 score. Hence, we believe our automatic analysis results correctly represent the quality of VR test cases.

On internal validity, one threat is the usage of Assertion Density to estimate the effectiveness of VR tests, as there is no prior usage of this metric in the context of VR software. However, this metric was used in the context of Mobile Applications [10], C# software [27], and OSS Java projects [37], proving the versatility of this metric across several types of software as a proxy for test effectiveness, hence our adoption of it. Another threat is potential bias when answering RQ4. Since there is no existing taxonomy to use as a reference, three co-authors reviewed and categorized randomly-selected VR test methods separately. Then, the co-authors carried out voting and discussion to finalize the results. In addition, when defining the VR test taxonomy, this work uses a general language and adds code examples to reduce the gap between the theoretical concept and the observed practice. Furthermore, in order to further validate that none of the testing types described by Albaghajati et al. [32] were missed by the random selection process, two co-authors manually verified the documentations and source codes of the 61 VR projects with tests and found no examples of them. In addition, simple random selection was used to determine which tests were manually categorized by the co-authors, as no optimal stratification criteria emerged to perform stratified random sampling.

On external validity, the main concern is the representativeness of studied projects. Our empirical study consisted of 314 open-source Unity VR projects from Github, 97 of which were originally collected by Nusrat et al. [21]. These projects have been carefully selected with at least 100 commits per-project, and are of varying sizes, ages, and commit-frequencies. Furthermore, we considered both small-scale projects and large projects backed by companies. This allows us to uncover insight that may reflect the general characteristics of all VR applications. Moreover, the taxonomies defined in RQ4 do not depend on Unity alone and can be extended to more general scenarios.

6 Related Works

6.1 Study on Automatic Software Testing

Software testing is an essential but costly and effort-intensive activity of software development, which pushed the research community to study software testing practices. Greiler et al. [7] conducted a qualitative study where they interviewed 25 practitioners about how they test Eclipse plugins. They identified that unit testing plays a critical role, whereas integration problems are identified by the community. Kochhar et al. [8] performed a study on 20,000 non-trivial software projects and explored the correlation of test cases considering various factors. The study discovered that as projects grow in size, their ratio of test cases per LOC decreases. Pecorelli et al. [10] performed an empirical study targeting 1780 open-source Android apps and identified that the effectiveness of their test cases is low and that they suffer from quality issues. Several other studies [13–17] also performed empirical analyses of software testing practices and different aspects of testing adoption. Even though these works performed empirical analysis on general-purpose and Android app testing practices, none of the work performed an analysis on testing practices of VR applications.

6.2 Study on Game and VR Testing

As Game and Virtual Reality (VR) applications are becoming more popular and accessible, the research community has started studying the development practices of Game and VR projects. To identify common bugs in Game applications, Truelove et al. [55] performed empirical analysis on 12,122 bug fixes from 723 updates of 30 popular games. Politowski et al. [56] performed a survey to understand existing testing practices within Game development. Nusrat et al. [21] performed a study on 100 Unity VR applications and created a performance bugs taxonomy in the context of VR. Bierbaum et al. [57] examined existing testing techniques and their applicability to VR interfaces, and suggested new practices for their shortcomings. More recently, Harms [58] suggested an automatic

VR-usability evaluation method, and Wang [59] proposed a testing framework that automates VR-scene testing. In addition, Ashtari et al. [18] interviewed 21 AR/VR creators from different groups such as hobbyists, domain experts, and professional designers, and concluded pointed out that testing VR applications is a complicated task, and Vlahovic et al. [60] described different factors and dimensions that influence the Quality of Experience of VR applications, and presented options and recommendations for their future research.

Although there are several works related to VR and Game development, as well as a few works concerning VR software testing and evaluation, none of them analyzed VR-testing's adoption, practices, design issues and characteristics by analyzing existing code bases. In this work, we tried to fill this knowledge gap by analyzing existing projects and their testing practices.

7 Implications

Our analysis pointed out some critical factors and observations for VR application testing which can be beneficial for VR developers, tool builders, and researchers. The findings of this paper lead to the following implications.

For VR Developers: RQ1, RQ2, and RQ3 clearly point out that VR testing's effort and effectiveness are low and that it suffers from quality issues. These results illustrate the problematic state of testing within VR applications, and that VR developers and testers should put more effort into improving the quantity, coverage, and quality of their tests. In addition, RQ4 generates a VR test case taxonomy based on testing goals, which can be helpful for VR developers as a general guideline that directs the formulation of their test cases.

For VR Tool Builders: RQ2 points out the necessity of tool support for VR application code coverage analysis. Even though Unity provides tool support for code coverage measurement, the tool is unusable in a lot of cases due to compatibility issues. Similarly, based on findings from RQ3, tool developers should develop tool support for the automatic detection and repair of test smells for VR applications.

For Software Engineering Researchers: From RQ1 & RQ2, it is evident that VR application testing is not sufficient. Researchers can do further studies on barriers to VR application testing and formulate techniques to overcome them. Moreover, RQ3 clearly identifies some of the test smells in VR test code. These smell categories are derived from traditional software test smells. Since VR applications are different from traditional software, researchers can investigate the existence of VR-specific test smells such as rendering smells, GameObject configuration smells, etc. Finally, through RQ4, we identified some common categories of test cases developed for VR applications. Such categorization can be a basis for future research on pattern-based automatic test case generation for VR applications.

8 Conclusion

In this paper, we conducted the first quantitative and qualitative study on existing test practices of VR projects. We developed the first tool for test effectiveness analysis and test-smell identification for Unity-based projects. Moreover, we manually explored the characteristics of VR test cases and categorized them. Our automatic analysis shows low adoption of testing by VR applications, and that their test practices are less efficient and have a lower design quality. Furthermore, our manual analysis resulted in a VR test taxonomy composed of 13 main categories, which reflect the characteristics and specificities of VR applications, along with the identification of VR-attention test categories. We hope that our findings on testing practices in VR applications will allow future researchers to determine VR testing challenges and inspire future research on VR test automation.

Acknowledgments

The UofM-Dearborn authors are supported in part by UofM-Dearborn Research Support and NSF Award NSF-2152819.

References

[1] H. Rheingold, *Virtual reality / Howard Rheingold.* New York: Summit Books, 1991.
[2] "Oculus developer center," https://developer.oculus.com/, 2022, [Online; accessed 30-April-2022].
[3] "Vive developers," https://developer.vive.com/us/, 2022, [Online; accessed 30-April-2022].
[4] U. Technologies, "Unity real-time development platform | 3d, 2d vr & ar engine." [Online]. Available: https://unity.com/
[5] U. Engine, "Unreal engine | the most powerful real-time 3d creation tool." [Online]. Available: https://www.unrealengine.com/en-US
[6] "Fortune business insights market research report," https://www.fortunebusinessinsights.com/industry-reports/virtual-reality-market-101378, 2021, [Online; accessed 30-April-2022].
[7] M. Greiler, A. van Deursen, and M.-A. Storey, "Test confessions: A study of testing practices for plug-in systems," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 244–254.
[8] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *2013 13th International Conference on Quality Software*, 2013, pp. 103–112.
[9] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 233–242.
[10] F. Pecorelli, G. Catolino, F. Ferrucci, A. D. Lucia, and F. Palomba, "Testing of mobile applications in the wild: A large-scale empirical study on android apps," *Proceedings of the 28th International Conference on Program Comprehension*, 2020.
[11] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: An exploratory study," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '19. USA: IBM Corp., 2019, p. 193–202.
[12] S. Wang, N. Shrestha, A. K. Subburaman, J. Wang, M. Wei, and N. Nagappan, "Automatic unit test generation for machine learning libraries: How far are we?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1548–1560.
[13] D. Bowes, T. Hall, J. Petric, T. Shippey, and B. Turhan, "How good are my tests?" in *2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)*, 2017, pp. 9–14.
[14] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, pp. 1188–1221, 2017.
[15] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 1–12, 2017.
[16] M. Nejadgholi and J. Yang, "A study of oracle approximations in testing deep learning libraries," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 785–796.
[17] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1066–1071.
[18] N. Ashtari, A. Bunt, J. McGrenere, M. Nebeling, and P. K. Chilana, *Creating Augmented and Virtual Reality Applications: Current Practices, Challenges, and Opportunities.* New York, NY, USA: Association for Computing Machinery, 2020, p. 1–13. [Online]. Available: https://doi.org/10.1145/3313831.3376722

[19] Y. Zhang, H. Liu, S.-C. Kang, and M. Al-Hussein, "Virtual reality applications for the built environment: Research trends and opportunities," *Automation in Construction*, vol. 118, p. 103311, 2020.
[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] 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)*. Madrid, Spain: IEEE, May 2021, p. 473–485. [Online]. Available: https://ieeexplore.ieee.org/document/9402052/
[22] 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)*, 2021, pp. 381–385.
[23] A. Borrelli, V. Nardone, G. A. Di Lucca, G. Canfora, and M. Di Penta, *Detecting Video Game-Specific Bad Smells in Unity Projects*. ACM 2020, 2020, pp. 198–208. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3379597.3387454
[24] Y. Wu, Y. Chen, X. Xie, B. Yu, C. Fan, and L. Ma, "Regression testing of massively multiplayer online role-playing games," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 692–696.
[25] 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)*, 2018, pp. 392–402.
[26] L. Vidács and M. Pinzger, "Co-evolution analysis of production and test code by learning association rules of changes," in *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, 2018, pp. 31–36.
[27] G. Kudrjavets, N. Nagappan, and T. Ball, "Assessing the relationship between software assertions and faults: An empirical investigation," in *2006 17th International Symposium on Software Reliability Engineering*. Raleigh, NC, USA: IEEE, Nov 2006, p. 204–212. [Online]. Available: http://ieeexplore.ieee.org/document/4021986/
[28] J. De Bleser, D. Di Nucci, and C. De Roover, "Socrates: Scala radar for test smells," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, ser. Scala '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 22–26. [Online]. Available: https://doi.org/10.1145/3337932.3338815
[29] A. P. Doucet Lars, "Game engines on steam: The definitive breakdown," Sep 2021. [Online]. Available: https://www.gamedeveloper.com/business/game-engines-on-steam-the-definitive-breakdown
[30] U. Technologies, "Unity - manual: Unity architecture," 2022. [Online]. Available: https://docs.unity3d.com/Manual/unity-architecture.html
[31] "Srcml," 2022. [Online]. Available: https://www.srcml.org/
[32] A. M. Albaghajati and M. A. K. Ahmed, "Video game automated testing approaches: An assessment framework," *IEEE Transactions on Games*, p. 1–1, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9234724/
[33] [Online]. Available: https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html
[34] C. Klammer, G. Buchgeher, and A. Kern, "A retrospective of production and test code co-evolution in an industrial project," in *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*. Campobasso: IEEE, Mar 2018, p. 16–20. [Online]. Available: http://ieeexplore.ieee.org/document/8327151/
[35] L. Williams, G. Kudrjavets, and N. Nagappan, "On the effectiveness of unit test automation at microsoft," in *2009 20th International Symposium on Software Reliability Engineering*. Mysuru, Karnataka, India: IEEE, Nov 2009, p. 81–89. [Online]. Available: http://ieeexplore.ieee.org/document/5362086/
[36] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad India: ACM, May 2014, p. 72–82. [Online]. Available: https://dl.acm.org/doi/10.1145/2568225.2568278
[37] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "How the experience of development teams relates to assertion density of test classes," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 223–234.
[38] G. Meszaros, *xUnit test patterns: refactoring test code*, ser. The Addison-Wesley signature series. Upper Saddle River, NJ: Addison-Wesley, 2007.
[39] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001*, 2001, pp. 92–95.
[40] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *2016 Automated Software Engineering*. ACM, Aug 2016, p. 4–15. [Online]. Available: https://www.microsoft.com/en-us/research/publication/an-empirical-investigation-into-the-nature-of-test-smells/
[41] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. Trento, Italy: IEEE, Sep 2012, p. 56–65. [Online]. Available: http://ieeexplore.ieee.org/document/6405253/
[42] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, p. 1188–1221, Jun 2018. [Online]. Available: http://link.springer.com/10.1007/s10664-017-9535-z
[43] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, p. 37–46, Apr 1960. [Online]. Available: http://journals.sagepub.com/doi/10.1177/001316446002000104
[44] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, vol. 47, no. 6, pp. 77–87, Aug. 2012, copyright: Copyright 2012 Elsevier B.V., All rights reserved.
[45] S. Bayona-Oré, J. A. Calvo-Manzano, G. Cuevas, and T. San-Feliu, "Critical success factors taxonomy for software process deployment," *Software Quality Journal*, vol. 22, no. 1, p. 21–48, Mar 2014. [Online]. Available: http://link.springer.com/10.1007/s11219-012-9190-y
[46] M. Usman, R. Britto, J. Börstler, and E. Mendes, "Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method," *Information and Software Technology*, vol. 85, p. 43–59, May 2017. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584917300472
[47] J. L. Fleiss and J. Cohen, "The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability," *Educational and Psychological Measurement*, vol. 33, no. 3, pp. 613–619, 1973. [Online]. Available: https://doi.org/10.1177/001316447303300309
[48] S. Fincher and J. Tenenberg, "Making sense of card sorting data," *Expert Systems*, vol. 22, no. 3, p. 89–93, Jul 2005. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1111/j.1468-0394.2005.00299.x
[49] "Chapter 1. the goal of unit testing · unit testing principles, practices, and patterns," 2022. [Online]. Available: https://livebook.manning.com/book/unit-testing/chapter-1/
[50] B. Marick, J. Smith, and M. Jones, "How to misuse code coverage," in *Proceedings of the 16th Interational Conference on Testing Computer Software*, 1999, pp. 16–18.
[51] "Unity physics documentation," https://docs.unity3d.com/Manual/PhysicsSection.html, 2021, [Online; accessed 16-February-2023].
[52] "Rigidbody unity documentation," https://docs.unity3d.com/ScriptReference/Rigidbody.html, 2021, [Online; accessed 16-February-2023].
[53] E. Weyuker and F. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *IEEE Transactions on Software Engineering*, vol. 26, no. 12, p. 1147–1156, Dec 2000. [Online]. Available: http://ieeexplore.ieee.org/document/888628/
[54] Aug 2022, page Version ID: 1102053166. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Virtual_reality&oldid=1102053166
[55] A. Truelove, E. Santana de Almeida, and I. Ahmed, "We'll fix it in post: What do bug fixes in video game update notes tell us?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 736–747.
[56] C. Politowski, F. Petrillo, and Y.-G. Gu'eh'eneuc, "A survey of video game testing," *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pp. 90–99, 2021.
[57] A. Bierbaum, P. Hartling, and C. Cruz-Neira, "Automated testing of virtual reality application interfaces," in *Proceedings of the Workshop on Virtual Environments 2003*, ser. EGVE '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 107–114. [Online]. Available: https://doi.org/10.1145/769953.769966
[58] P. Harms, "Automated usability evaluation of virtual reality applications," *ACM Trans. Comput.-Hum. Interact.*, vol. 26, no. 3, apr 2019. [Online]. Available: https://doi.org/10.1145/3301423
[59] X. Wang, "Vrtest: An extensible framework for automatic testing of virtual reality scenes," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 232–236.
[60] S. Vlahovic, M. Suznjevic, and L. Skorin-Kapov, "A survey of challenges and methods for quality of experience assessment of interactive vr applications," *Journal on Multimodal User Interfaces*, vol. 16, no. 3, p. 257–291, Sep 2022. [Online]. Available: https://doi.org/10.1007/s12193-022-00388-0