Automatic Extraction of Code Dependency in Virtual Reality Software

Jacinto Molina[1], Xue Qin[2], and Xiaoyin Wang[1]
[1]*Department of Computer Science, University of Texas at San Antonio*
[2]*Department of Computing Sciences, Villanova University*
jacinto.molina@my.utsa.edu, xue.qin@villanova.edu, xiaoyin.wang@utsa.edu

***Abstract*—Virtual Reality (VR) is an emerging technique with various applications in education, navigation, remote communication, and gaming. In virtual reality software, VR objects, which are graphics units linking with script files, play an important role and they often need to be updated together with correlated source code in the script files. However, traditional code dependency analysis tools detect only direct source code dependencies and can hardly detect code dependency across VR objects. In this paper, we propose a novel technique, VRDepend, to extract code-asset dependency within VR software projects. In particular, we extract (1) the direct association between VR objects and script files, (2) the compositional relations between VR objects, (3) event triggering relations between scripts and VR objects. We evaluated VRDepend based on script dependency extracted from file co-revisions. Our evaluation results show that VRDepend can cover 64% script file dependencies in historical commits of our subject VR software projects.**

I. Introduction

The Virtual Reality (VR) market was valued at USD 17.25 billion in 2020 and is expected to reach USD 184.66 billion by 2026, at an average growth rate of 48.7% over the forecast period 2021 - 2026 [1]. Companies, such as Google, Microsoft, Oculus, HTC, and Sony, have created a range of hardware products that consumers can use for an immersive experience, and various software applications on education, remote assistance, navigation, and gaming have been developed. A recent study [2] shows that the number of open source VR software projects are increasing with a steady growth rate of 17%, and a lot of developers are moving toward VR software development.

However, VR software development is very different from traditional UI / server software development in that the source code is scattered on various VR objects in VR scenes and they usually interact with each other through the properties of objects they are attached to. This makes it very difficult for developers especially novices to trace code dependency and understand code interactions for analyzing bugs [3] and addressing security and privacy concerns [4], [5]. In this paper, to help trace and understand code dependency in VR projects, we propose a novel technique called VRDepend. It should be noted that we designed VRDepend based on the VR development framework Unity [6]. We chose Unity because it is dominating in VR software development and accounts for more than 60% of the market share. Furthermore, Unity provides a unified programming interface for developers to write applications for almost all main-stream VR devices (e.g., Apple ARKit [7], Google ARCore [8], Steam VR [9], DayDream / Cardboard [10], [11], Oculus [12], HTC Vive [13], and Hololens [14]), so we believe that the programming mechanisms in Unity (and our corresponding technique design) can be largely generalized to various software / hardware environment. In particular, besides direct code dependency through method calls and variable references, VRDepend further extracts the following types of dependency.

- **Script VR-Object Association.** In VR projects, source code are written in script files, which are then associated with VR objects in VR scenes. When a script file F is attached to a VR object O, methods in F can refer to all properties of O as `gameObject.*`. Since multiple scripts can be attached to one VR object, all methods in these scripts can refer to the same set of properties and thus cause latent code dependency among scripts associated to the same VR object.
- **VR-Object Composition.** VR objects can be combined to form composite objects. For example, a table and a tablecloth can be combined so that they are moved together, but the tablecloth color (or the whole tablecloth object) can be replaced at runtime. Scripts associated to a composite object and its components can refer to their properties through `parent.*` and `children[index].*`, and thus cause latent code dependency among scripts associated to components of a same composite VR object.
- **VR-Object Events.** Besides direct method calls, methods in a script S with certain names (e.g., `Update`, `Start`) will be called at certain life cycle events of the VR object O that S is associated to. For example, when O is instantiated or destroyed, corresponding methods in all scripts associated to O will be called. Therefore, when method in script A instantiated a VR object B, it indirectly called all the corresponding methods in scripts associated to B, and thus cause latent code dependency between A and B's associated scripts.

To detect the above additional code dependency in VR software projects, the basic idea of VRDepend is to further trace dependency among VR objects, and then transitively concatenate code-object dependency, object-object dependency, and object-code dependency. For the evaluation of VRDepends, we considered 20 top open source VR software projects from

UnityList[1] (based on the number of stars in their corresponding Github repository), and collected their version history. From the version history, we extracted 200 most recent code commits that contain two to five script file changes. Our initial evaluation results show that VRDepend can cover 64% dependencies between co-revised source files, compared with 31% of the baseline code-analysis-based technique. To sum up, we make the following contributions in this paper.

- We identified a number of major additional code dependency in VR software projects.
- We developed a novel technique VRDepend to detect additional code dependency in VR software projects.
- We carried out an initial evaluation of VRDepend based on the code dependency between co-revised files in version history of 20 top open source VR projects, and the results show that VRDepend largely outperforms the baseline approach relying on direct code dependency.

II. Background

In this section, we introduce some background knowledge about VR software development and explain terms we use in our paper.

VR Objects. In VR software development, a VR object is a basic element for visual experience and user interaction. An exemplar VR object is a table in a virtual room.

VR Scenes. A VR scene is a combination of VR objects in a virtual space to perform certain user task. It can be analogous to a window / frame in a traditional GUI software, or an activity in an Android app. The major differences are that VR scenes are three-dimensional and the VR objects have much richer animation behaviors than UI controls / views in GUI windows and Android activities.

III. Approach

The overview of our approach is presented in Figure 1. From the figure, we can see that VRDepend first performs three analyses (Event Trigger Analysis, Extraction of VR-Object-Script Association, and Hierarchical Analysis of VR Objects) to extract the additional types of dependency between script files and VR objects. Then, by gathering the property references of VR objects in script files, VRDepend chains dependency between scripts and VR objects to acquire additional code dependency among scripts. We next introduce the details of each component in VRDepend.

A. *Extracting VR-Object-Script Association*

The first step towards detecting indirect code dependency through VR objects is to extract the association between VR objects and scripts. Such association can be detected from the corresponding meta files of the VR object and the script file.

In Unity framework, all VR Objects in a VR scene are defined in the `.unity` file which is the meta file of the a VR scene. Example 1 shows a segment of `.unity` file which defines a VR object (called GameObject in Unity), the ID of the VR object is `1756271139` at the beginning of the definition, and its name is `worldmap`. The `MonoBehaviour` component of a VR object defines its run-time behavior, so the association with script files is also declared there as highlighted.

Example 1 Definition of a VR-Object and its Script Association

```
--- u1 \&1756271139
GameObject:
...
m\_Name: worldmap
...
--- u114 &1756271142
MonoBehaviour:
...
m\_Script: {fileID: 11500000,
guid: 1239b8acfd6fc154ca4bcccfb2bd7a54 , type: 3}
...
--- u1 \&2006036288
GameObject:
...
```

The highlighted ID is a unique identifier of a script file, which is automatically given by the framework in the meta file of the script file (i.e., `.cs.meta`), as shown in Example 2. To acquire all associations between VR objects and scripts, VRDepend first extracts all VR objects from the `.unity` files and then link them with scripts based on the IDs in their `MonoBehavior` component and meta files of scripts.

Example 2 The Corresponding Script Meta File

```
fileFormatVersion: 2
guid: 1239b8acfd6fc154ca4bcccfb2bd7a54
...
```

B. *Hierarchy Analysis of Composite VR Objects*

The hierarchical relationship among VR objects are also defined in scene meta files (`.unity` files). The snippet in Example 3 shows the definition of a VR Object named `Player` and the ID of one of its child VR object (highlighted).

Example 3 Parent Object Meta Information

```
--- u1 &1625246633
GameObject:
...
m_Name: Player
..
--- u4 &1625246634
Transform:
...
m_Children:
- {fileID: 1486385204 }
...
```

On the child VR object side, the link is less straightforward. The child object ID referred in the parent object is actually the component ID of the child object's transform component

[1]UnityList is a list of open source projects based on Unity framework

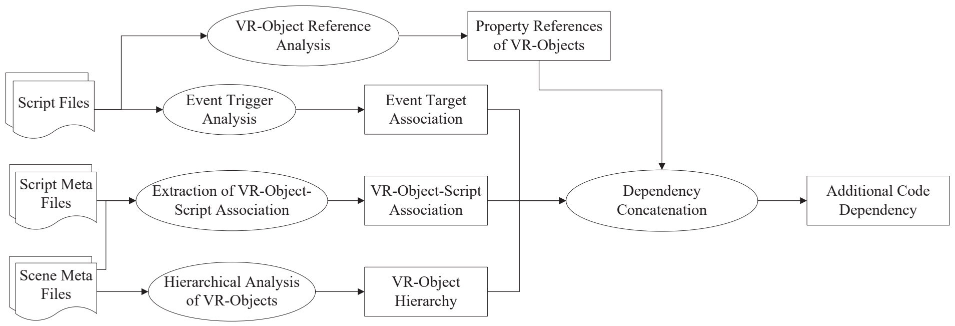
VR-Object Reference Analysis
Property References of VR-Objects
Script Files
Event Trigger Analysis
Event Target Association
Script Meta Files
Extraction of VR-Object-Script Association
VR-Object-Script Association
Dependency Concatenation
Additional Code Dependency
Scene Meta Files
Hierarchical Analysis of VR-Objects
VR-Object Hierarchy

Fig. 1: Overview of VRDepend

(highlighted in Example 4). The reason is that Unity combines transforms (i.e., the location and bounding box of a VR object) instead of VR objects themselves. By linking the transform component IDs and child object IDS of VR objects, VRDepend is able to construct the hierarchical relationship among VR objects.

Example 4 Child Object Meta Information

```
--- u1 &1486385203
GameObject:
...
m_Component:
- component: {fileID: 1486385204}
...
m_Name: MainCamera
...
--- u224 & 1486385204
RectTransform:
...
```

C. *Event Trigger Analysis*

The last additional type of dependency is between object events triggered in scripts and the life-cycle methods of the object. In Example 5, we show a code snippet where a VR object named `paper` is initiated in the code. This will trigger the `Start()` method on all scripts associated to object `paper`. In the event trigger analysis, VRDepend first detects all invocation of event methods such as `Instantiate` and `Destroy`, and then we trace back to the definition of the affected `GameObject` variable (i.e., `paperToSpawn` in the code) to find all possible VR objects the variable may refer to. Thus we are able to gather all event-target associations which links event-triggering code with VR objects the events are triggered on.

D. *Dependency Concatenation*

In the VR-Object reference analysis, VRDepend extracts all references to VR objects in script files (e.g., through `gameObject`, `parent`, `children[index]`). After that, all such references are mapped to VR object names based on the results of VR-Object-Script associations and VR-Object hierarchy. It should be noted that we resolve hierarchical relations in a conservative way. For example, if VR object A has multiple parents (being combined in multiple composite VR objects) denoted as a set S, then, the `parent` reference in any script associated to A will be mapped all elements in S. Also, since children objects are indexed in the `.unity` file and our extracted hierarchy, we can precisely handle children references if the index value is a constant. However, if the index value is a variable, we will conservatively map the reference to all children objects. Moreover, we link event-target associations with the VR-Object-Script associates from the target VR objects to construct additional code dependency due to indirect calls.

Example 5 Code to Trigger a Instantiation Event

```
public void SpawnPaper(){
      ...
      GameObject paperToSpawn = GameObject.Find("
          paper");
      Instantiate(paperToSpawn, position.transform.
          position, position.transform.rotation);
      ...
}
```

IV. Preliminary Evaluation

A. *Evaluation Setup*

It is difficult to directly evaluate the quality of code dependency constructed by VRDepend. Therefore, we use script file dependency extracted among co-revised files in software version history to evaluate our results. In particular, we collected 20 top VR software projects and their version history from UnityList. For each project, we extracted the most recent 10 commits that have 2-5 revised script files (i.e., `.cs` files) to form a dataset[2] of 200 code commits. We chose commits with 2-5 revised scripts because we need at least 2 script file revisions to form script file dependency, and commits with too many file revisions are more likely to be systematical changes and even addition of a whole feature so the revised files may not be dependent on one another. By inspecting the co-revised files in 200 commits, we confirmed 688 pairs of file dependencies (i.e., the co-revisions were caused by dependencies between files). Note that if a file

[2]Our dateset is at https://sites.google.com/view/vrdepend

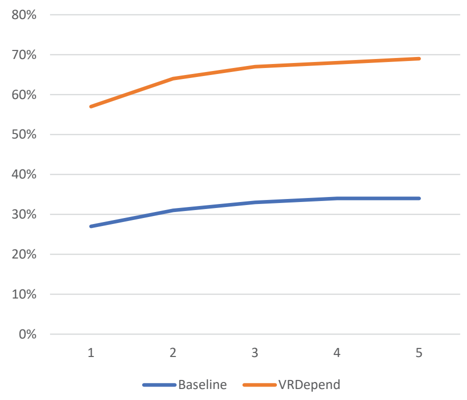

Fig. 2: Coverage of File Dependencies among Co-revised Files

revision r is irrelevant to other file revisions $s_1, s_2, ...$ in the commit, we will not consider there is dependency between r and s_i.

B. Evaluation Results

We measure the effectiveness of VRDepend by calculating what proportion of confirmed file dependency pairs can be covered by VRDepend results, and compare it with a baseline approach which considers only direct code dependency (i.e., method invocation and global field accesses). The results are shown in Figure 2. The X-axis of the figure shows the number of transitive dependency hops we consider. So if the results of VRDepend or the baseline state that A depends on B and B depends on C, but not A depends on C, the dependency (A, C) will be considered to be covered at dependency hop 2. The results show that VRDepend largely outperforms the baseline on coverage (57%, 64% vs. 27% and 31% at hops 1 and 2), indicating that VRDepend can effectively detect code dependency missed by the baseline.

Although VRDepend has better coverage, it may be argued that a naïve baseline that considers all file pairs to be dependent on each other will achieve even better coverage. Therefore, we further check whether VRDepend reports too many script dependency pairs. We evaluate this by comparing the average co-revision rate (the proportion of co-revised files among all dependent files of a when a is revised) of VRDepend reported dependency pairs with those reported by the baseline. This rate is expected to be low because dependency does not necessarily result in co-revision. The results are shown in Figure 3. From the figure we can see that the co-revision rate of VRDepend is comparable with the baseline at hops 1 and 2, showing that VRDepend does not detect too many script dependency pairs. When the number of hops increases, the co-revision rate of both approach drops because more hops means less close dependency and less likelihood of co-revision. However, VRDepend drops more sharply because VRDepend considers more types of dependency and thus the dependency closure extends further as hops increase. By contrast, the baseline drops more slowly because dependency

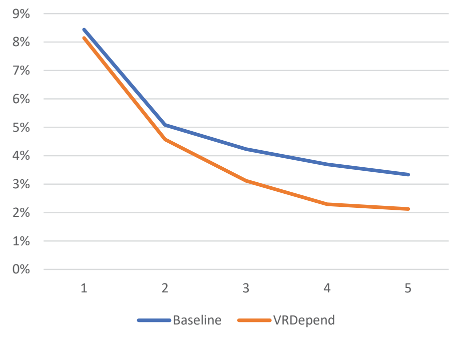

Fig. 3: Co-revision Rate of Files with Dependency

closure is reached earlier due to the missing of dependency across VR objects.

V. Related Work

The authors are not aware of research efforts in VR software dependency analysis, but there exist some empirical studies. Murphy-Hill et al. [15] performed a study on game developers to understand the challenges in video game development and how they are different from traditional software development. Washburn et al. [16] studied failed game projects to find out the major pitfalls in game development. Lin et al. [17] studied the common updates in steam platform to understand the priority of game updates. Rodriguez and Wang [2] performed an empirical study on open source VR software projects to understand their popularity and trends. Pascarella et al. [18] studied open source video game projects to understand their characteristics and the difference between game and non-game development. Zhang et al. [19] studied the privacy issues of VR applications. Code dependency analysis has a long research history dates back to program slicing [20]. Various analyses have been proposed to enhance precision of code analysis [21], [22], [23] and generalize it to more software artifacts [24], [25] or more scenarios [26], [27], [28]. There have also been research efforts on predicting file co-changes [29], [30]. Compared with all these works, our work first identified major types of latent code dependency in VR software projects and proposed VRDepend extract such latent code dependency.

VI. Conclusion and Future Work

In this paper, we present a novel technique VRDepend to detect code dependency in VR software projects. The evaluation results show that VRDepend detects twice as many useful code dependency pairs as detected by the baseline with comparable precision. In the future, we will further extend our approach by investigating the missing confirmed pairs of script dependency, and evaluating VRDepend on a larger dataset with more commits.

Acknowledgments

This work is supported in part by NSF Awards NSF-1736209, NSF-1846467, and NSF-2007718.

References

[1] https://www.mordorintelligence.com/industry-reports/virtual-reality-market, Recently visited on 01/13/2021.

[2] 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.

[3] Y. Song, X. Wang, T. Xie, L. Zhang, and H. Mei, "Jdf: detecting duplicate bug reports in jazz," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2. IEEE, 2010, pp. 315–316.

[4] X. Wang, X. Qin, M. B. Hosseini, R. Slavin, T. D. Breaux, and J. Niu, "Guileak: Tracing privacy policy claims on user input data for android applications," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 37–47.

[5] X. Zhang, X. Wang, R. Slavin, T. Breaux, and J. Niu, "How does misconfiguration of analytic services compromise mobile privacy?" in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1572–1583.

[6] "Unity documentation - 2d or 3d projects," https://docs.unity3d.com/, 2020, accessed: 2020-06-30.

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

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

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

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

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

[12] "Oculus," https://www.oculus.com/, 2020, accessed: 2020-12-30.

[13] "Htc vive," https://www.vive.com/us/, 2020, accessed: 2020-12-30.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[21] T. Reps, "Undecidability of context-sensitive data-dependence analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 22, no. 1, pp. 162–186, 2000.

[22] J. Ossher, S. Bajracharya, and C. Lopes, "Automated dependency resolution for open source software," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 130–140.

[23] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei, "Summary-based context-sensitive data-dependence analysis in presence of callbacks," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015, pp. 83–95.

[24] 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.

[25] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu, "Matching dependence-related queries in the system dependence graph," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 457–466.

[26] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Locating need-to-translate constant strings for software internationalization," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 353–363.

[27] ——, "Transtrl: An automatic need-to-translate string locator for software internationalization," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 555–558.

[28] S. Mostafa, X. Wang, and T. Xie, "Perfranker: Prioritization of performance regression tests for collection-intensive software," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 23–34.

[29] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[30] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, 2004.