2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) | 978-1-6654-0296-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE43902.2021.00052

How Developers Optimize Virtual Reality Applications: A Study of Optimization Commits in Open Source Unity Projects

Fariha Nusrat
Department of Computer Science
University of Texas at San Antonio
fariha.nusrat@my.utsa.edu

Foyzul Hassan§
Department of Computer and Information Science
University of Michigan-Dearborn
foyzul@umich.edu

Hao Zhong§
Department of Computer Science and Engineering
Shanghai Jiao Tong University
zhonghao@sjtu.edu.cn

Xiaoyin Wang§
Department of Computer Science
University of Texas at San Antonio
xiaoyin.wang@utsa.edu

***Abstract*—Virtual Reality (VR) is an emerging technique that provides immersive experience for users. Due to the high computation cost of rendering real-time animation twice (for both eyes) and the resource limitation of wearable devices, VR applications often face performance bottlenecks and performance optimization plays an important role in VR software development. Performance optimizations of VR applications can be very different from those in traditional software as VR involves more elements such as graphics rendering and real-time animation. In this paper, we present the first empirical study on 183 real-world performance optimizations from 45 VR software projects. In particular, we manually categorized the optimizations into 11 categories, and applied static analysis to identify how they affect different life-cycle phases of VR applications. Furthermore, we studied the complexity and design / behavior effects of performance optimizations, and how optimizations are different between large organizational software projects and smaller personal software projects. Our major findings include: (1) graphics simplification (24.0%), rendering optimization (16.9%), language / API optimization (15.3%), heap avoidance (14.8%), and value caching (12.0%) are the most common categories of performance optimization in VR applications; (2) game logic updates (30.4%) and before-scene initialization (20.0%) are the most common life-cycle phases affected by performance issues; (3) 45.9% of the optimizations have behavior and design effects and 39.3% of the optimizations are systematic changes; (4) the distributions of optimization classes are very different between organizational VR projects and personal VR projects.**

***Index Terms*—Empirical Study, Virtual Reality, Performance Optimization**

I. Introduction

Virtual reality (VR) techniques [24] have provided revolutionary user experience in various application scenarios (e.g., training, education, product / architecture design, gaming, remote conference / tour). According to a report from Mordor Intelligence [13], the market of virtual reality is valued at 11.52 billion dollars in 2019, and is expected to grow by 48.7% per year. While the majority of the VR market is still on hardware, the market size of VR software is estimated to reach 1.9 billion dollars in 2019 [16], with thousands of apps being developed and uploaded to Google Play [9], Apple Store [1], and Oculus Market [14]. These apps having been downloaded by more than 171 million users from all over the world [22].

Compared with traditional GUI-based software applications, VR software often consumes much more computation resources due to their complicated real-time animations and the double rendering requirements (i.e., rendering twice for both eyes at the same time) [23]. Furthermore, VR devices often have relatively limited computation resources due to their wearable nature, and their performance downgrade often causes more severe consequences (e.g., dizziness and motion sickness caused by low frame rates and in-continuous animations) than traditional software applications [51]. As a result, performance optimization plays a very important role in VR software development, and performance defects are of high priority to be repaired as soon as possible [23], [27].

While there have been many empirical studies [30], [34], [39], [47], [57] on general / mobile performance bugs and optimizations to understand their common patterns and give suggestions to developers / researchers, there are few studies on VR performance optimizations (e.g., [36]), and thus the understanding of them is very limited. In particular, factors affecting VR performance can be very different from those affecting traditional software performance, because VR applications have rich real-time animations, extensive usage of GPU, and high impacts of asset / scene design (besides source code) on rendering costs.

To deepen the understanding on VR performance optimizations, in this paper, we present an exploratory study[1] on 183 performance optimizing changes in 45 actively maintained

§Corresponding Authors

[1]See our data set at: http://sites.google.com/view/vroptstudy2020/

978-1-6654-0296-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00052

open source VR projects from UnityList [21], the largest online repository of open-source VR software projects based on Unity. We believe this scope of subject selection is reasonable because (1) Unity dominates VR development with over 60% market share according to multiple sources [19], [20], and (2) Unity, itself open-source, nurtures a large VR community with thousands of open-source projects (i.e., UnityList).

In our study, we first applied a keyword search on the version history of all actively maintained software projects (in UnityList) which have at least 100 code commits. The keyword search found 609 code commits as potential performance optimizations. Two of authors manually inspected each of the 609 code commits independently to identify performance optimizations, and further analyzed the code changes, the type, and the root cause of each performance optimization. The following reconciliation among authors confirmed 183 performance optimizing changes. It should be noted that one code commit may contain multiple logic code changes (each of which can be either performance optimizing change or not), so one performance optimizing change may not cover all file revisions in its corresponding code commit, and multiple performance optimizing changes may come from the same commit. After that, we further applied static call-graph analysis and code-asset dependency analysis to the performance-optimizing changes to precisely answer our research questions.

Our study mainly tries to answer the following six research questions and we briefly summarize our major findings below as their answers given by the study.

- **RQ1:** *What are the major categories of performance optimizations in VR software projects?*
  Motivation. The answers will present an overview of VR performance optimizations.
  Answer. Our findings show that the top five categories of optimizations are: (1) graphics simplification (24.0%), (2) rendering optimization (16.9%), (3) language feature / API optimization (15.3%), (4) heap avoidance (14.8%), and (5) value caching (12.0%), among which (1), (2), and (4) are not reported in prior studies on general performance bugs and optimizations.
- **RQ2:** *Which life-cycle phases of VR software are more affected by performance issues?*
  Motivation. The answer is useful on allocating resources for VR performance optimizations.
  Answer. Our findings show that game-logic updates, before-scene initialization, and game-object initialization are mostly affected life-cycle phases of VR scenes, affected by 30.4%, 20.0%, and 10.4% of the 135 source-code-revising optimizations, respectively.
- **RQ3:** *How complex are performance optimizations in VR software projects?*
  Motivation. The answers reveal the complexity of handling VR performance optimizations.
  Answer. Our findings show that 70 (38.3%) of the 183 performance optimizations involve asset-file revisions, but 109 (59.6%) optimizations are simpler inner-class code changes. The average number of revised files is 5.6, and the average number of revised source-code lines is 19.9 (among source-code-revising optimizations).
- **RQ4:** *Do performance repairs have effect on other aspects of software quality?*
  Motivation. The answers reveal the side effects of VR performance optimizations.
  Answer. Our findings show that 84 (45.9%) performance optimizations have potential effects on software design and behavior. In particular, 52 of them have potential effect on program behavior (user interface) and the remaining 32 have potential effect on software design / coding style.
- **RQ5:** *How many performance repairs are systematic changes which apply the same or similar code revisions to multiple code locations?*
  Motivation. The answers reveal the repetitions of VR performance optimizations, and are important for the detection and repair of such issues.
  Answer. Our findings show that 72 out of 183 (39.3%) performance optimizations are systematic changes, and certain categories of optimizations (i.e., Heap Avoidance, API/Language optimization, and Graph Simplifications) contains large proportion of systematic changes.
- **RQ6:** *Do developers in organizational projects and personal projects behave differently in resolving VR performance optimizations?*
  Motivation. Based on the answers, we can understand the knowledge and habit difference between organizations and personal developers.
  Answer. Our findings show that the distributions of performance optimization categories are very different between organizational and personal projects. In particular, the top three categories of performance optimizations in organizational projects are heap avoidance (24 of 107), language features / API optimizations (23 of 107), and graphics simplifications (20 of 107), while the top three categories of performance optimizations in personal projects are graphics simplifications (20 of 76), rendering optimizations (18 of 76), and value caching (11 of 76).

To sum up, our study makes the following contributions:

- We construct a data set of VR performance optimizations that we used for our study. The data set also forms a foundation for future research in this area.
- We develop a taxonomy of performance optimizations in VR applications, and it extends the taxonomies of general performance optimizations in the prior studies.
- We answer six research questions on VR performance optimizations, including their impact on VR scene phases, their complexity, their behavior / design effects, whether they are systematical changes, and their different distribution in organizational and personal projects.

The remaining part of this paper is organized as follows. After presenting background knowledge about VR and Unity in Section II, we will describe our experiment methodology in Section III. Section IV presents the results of our study

and Section V presents discussion of lessons learned. Related works and Conclusion will be discussed in Section VI and Section VII, respectively.

II. Background

In this study, we selected VR applications from UnityList, and these applications are built upon Unity framework [18], which is the dominating framework in VR software development, and integrates with almost all existing VR / AR platforms, including Apple ARKit [2], Android Daydream / Cardboard [7], [8], Google ARCore [6], Steam VR [17], Windows Mixed Reality (Hololens) [12], etc. In the remaining of our paper, we refer to a series of VR concepts using Unity-specific terms. Please note that these concepts are general and exist in all VR development frameworks as evidenced by the seamless integration of Unity with a large variety of VR platforms. We introduce these terms as follows.

Scenes. A VR scene refers to a space where a user immerses in and interacts with when using a VR application. For examples, a scene can be a virtual meeting room in a VR remote conference application. A VR application usually consists of multiple scenes linked with each other through events triggered by users.

Game objects. Game objects are core components of scenes, and a game object represents a virtual object in a scene space. For examples, game objects can be tables / chairs in a virtual meeting room. Game objects have very rich attributes to specify their appearance (e.g., color, surface texture, transparency), physical properties (e.g., mass, speed, collision types), run-time behaviors (defined in a c-sharp script attached to the object), etc. Game objects can be compound hierarchically. For example, a game object can combine a sword with a scabbard. In this sense, game objects are similar with views / controls in traditional GUI-based apps, but largely enriched for run-time animations.

Prefabs. Since a VR application often contains multiple game objects of the same type (e.g., multiple chairs of the same type in one room and across multiple rooms). In such cases, the definition of the set of similar game objects can be abstracted to so-called **Prefabs.** The relationship between game objects and prefabs are analogous to that between objects and classes in object-oriented programming languages.

Scripts and assets. In a VR application, scripts (typically in c sharp) are source code files, and they are attached to game objects to define their logic behaviors. Besides scripts, there are various asset files defining graphics behaviors (e.g., `.3ds` and `.fbx` files for 3D models, `.shader` files for lighting effects, `.mat` files for material textures, and `.unity` files for scene settings). In our study, for simplicity, we consider script files as *source code files*, and all other files as *asset files*.

The life cycle model of game objects. In the life cycle of a scene, the Unity framework will trigger a series of life-cycle callback methods on each of the game objects (as implemented in the scripts attached to the objects) in the scene. Figure 1 presents a simplified life-cycle model of game objects. In particular, the Before-Scene phase methods

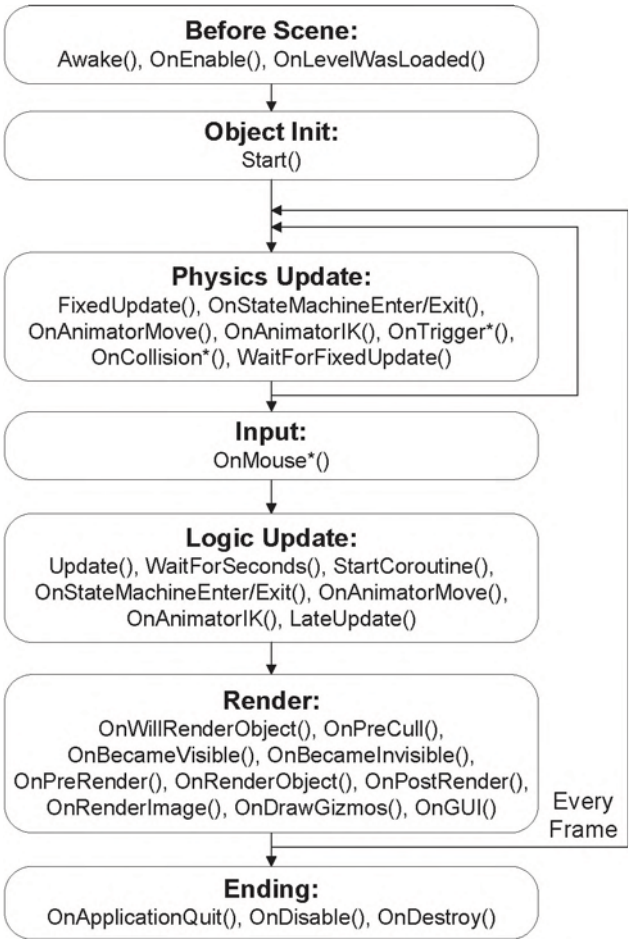

Fig. 1: The Life-cycle Model of Game Objects

are executed before a scene is started; the Object-Init phase methods are executed before the first frame update of the scene or before the frame update immediately after a game object is added to the scene; the methods in Physics-Update, Input, Logic-Update, and Render phases are executed sequentially for each frame; and the Physics-Update phase may be executed multiple times within a frame if the frame rate is low. Finally, the methods in the Ending phase are executed after the last frame of the scene or after a game object is removed.

III. Methodology

Our study has the following steps. First, we collected 609 candidate performance-related commits from the git repositories of the 100 most recently updated VR projects. Second, we manually inspected each of the code commits to identify performance optimizations, and labelled them with category tags, systematic-change tags, and behavior / design (**RQ1**, **RQ4**, **RQ5**, and **RQ6**). Third, we applied static analysis on the performance optimizations to understand they relations to other parts of the code (**RQ2** and **RQ3**).

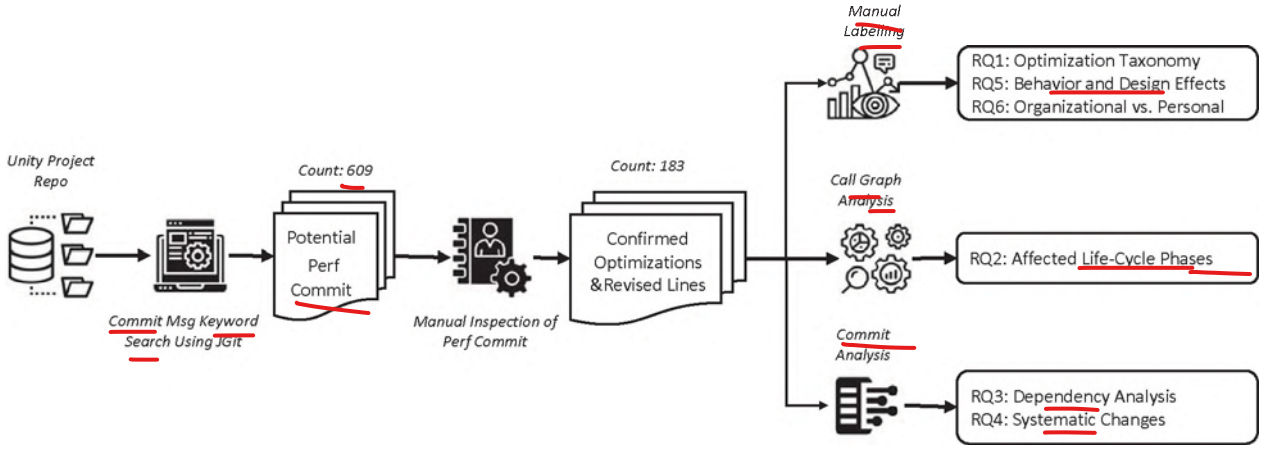

Fig. 2: The Overview of Our Performance Optimization Study

A. Collection of Candidate Commits

From UnityList, we selected 100 most recently updated VR software projects that have at least 100 historical commits. We chose the recently updated projects and used the commit threshold to ensure that the projects we considered are actively maintained. We implement a tool upon JGit [10]. Within these 100 projects, our tool searches commit messages of all their historical commits for performance-related commits, and the returned results are our candidate commits. For the search, we used **performance**, **speed up**, **accelerate**, **fast**, **slow**, **latency**, **contention**, **optimize**, and **efficient** as our search keywords, in that a prior study [26] used these keywords to collect performance related bug fixes. We further used four keywords **fps**, **framerate**, **frame rate**, and **frame per second**, which are specific to VR/AR software projects. With this search strategy, we identified 609 candidate code commits.

B. Manual Inspection and Reconciliation

Two of the authors independently inspected each of the 609 candidate code commits. For each code commit, each inspector read its commit message and the revised files and reported the following information: (1) revised lines in the commit that represent one or multiple performance optimizations, if there are any, (2) a category tag of the identified performance optimization (to answer RQ1 and RQ6), (4) whether the performance optimization has behavior / design effect to the application such as affecting its functional features and design quality (to answer RQ5).

In particular, for (1) identifying performance optimizations, we consider only revisions directly related to performance optimization. For example, in method m, when a local object instantiation ins is replaced by a field reference (to avoid heap allocations), some of m's parameters required by ins become unnecessary and thus can be removed. All invocations of m also need to be updated to remove the corresponding arguments. In this example, the replacement of ins is directly related to the performance optimizations, so it is reported, while the removal of parameters in m, and updates of m's invocations do not need to be manually reported and we rely on static analysis in Section III-C to further identify these related revisions. For (2) category tagging, we refer to existing taxonomies in prior studies [34], [39], [47]. If a performance optimization falls into categories in existing taxonomies (e.g., API misuse, redundant checking), we add the corresponding category tag on it. If an inspector cannot put the optimization in an existing category, the inspector needs to invent a new term for the new category. Terms invented by different inspectors are merged in the later reconciliation phase. For (3) behavior and design effects, we identify behavior effects on user interface from developers' comments in commit messages, and design effects by manually checking whether the code revision violates common design principles.

After the inspection, we performed reconciliation to merge inspection results. For (1) identifying performance optimizations, we went through each identified optimization to confirm / reject inspection results. We used the following rule to count performance optimizations. Within one commit, if there are multiple performance optimizations but they are unrelated to each other, we report them as different performance optimizations. Meanwhile, even if they are unrelated, we count them as one and label it as a systematic change, in the case that they are instances of the same systematic change. For example, when developers find that API method $m1$ is more efficient than the API method $m2$ they are using, they may replace all invocations of $m2$ to $m1$. Although these replacements may not be related to each other they are instances of a same systematic change, so we count all such replacements as one performance optimization. For (2) category tagging, we first merged the category tags we invented to form a combined taxonomy and mapped category tags of performance optimizations given by each inspector to the combined taxonomy. After the mapping, if a performance optimization has incon-

sistent category tags, we discuss to decide its final category tag. For (3) design and behavior effects, similar to category tagging, we first merged the types of effects identified by inspectors, and then identified performance optimizations with inconsistent results after mapping. After that, we discussed to make a final decision on the design and behavior effects of those optimizations. We calculated the Cohen's-Kappa value between two coders. The score for optimization determination is 0.840. For categorization, after labeling, we match the newly added categories to have unified names for categories and then Cohen's-Kappa value on categorization is 0.713. Both values show substantial agreement (above 0.6).

C. Analysis of Performance Optimizations

To answer RQ2 through RQ4, we need to identify code related to performance optimizations and relations between revisions in a commit. In particular, to answer RQ2, we applied a call-graph analysis to identify the life-cycle methods (as listed in Figure 1) that transitively invoked the revised methods within a performance optimization. To answer RQ3, we applied a code dependency analysis and code-asset analysis to identify other revised lines that are related to the performance optimization. To answer RQ4, we applied code clone detection to identify potential systematic changes and manually confirmed the detected instances.

Before we can apply static analysis to revised lines, we precisely identified the revised code elements by comparing Abstract Syntax Trees (ASTs). It should be noted that due to the challenge of automatically building the projects [32], [33], we chose to apply static analysis to the source code and perform partial-code analysis [60]. In particular, we used srcML [15] to parse the C-sharp source code and extracted AST-level changes with the state-of-the-art diff tool GumTree [29]. Then, we performed the call-graph analysis [28], code-dependency analysis [5], and code clone detection [4].

To identify dependencies between scripts and asset files, we compared meta IDs of scripts and asset files. In particular, all scripts and asset files in Unity have a corresponding .meta file which contains a unique identifier of the script / asset. By tracking the IDs in the definition of game objects (in .prefab files and .unity files), we can check whether a script and an asset are attached to the same game object. If so, we consider the revised lines in the script and the revised lines in the asset file are related to each other.

IV. Study Results

A. RQ1: Categorization of VR Performance Optimizations

In total, our study identified 11 categories of performance optimizations from our candidate commits. The distribution of optimizations over 11 categories is presented in Figure 3. For the type of optimizations that can be matched to those in earlier studies of traditional software optimizations [34], [47], we present the mapping to corresponding literature in Table I. In Column 1 of the table, we present our optimization types that can be mapped to earlier literature. In Column 2 and 3 of the table, we present their corresponding name in literature

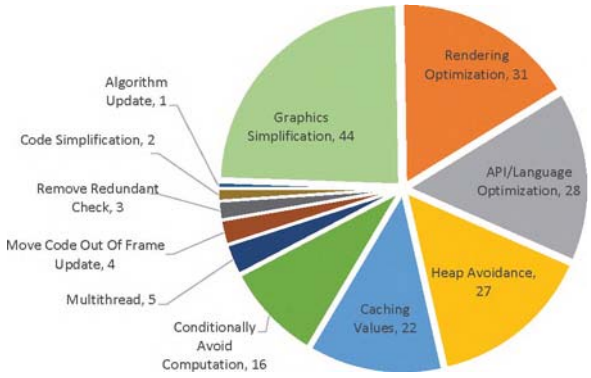

Fig. 3: Categorization of VR Performance Optimizations

[47] and [34], respectively. We use "-" to indicate that there is no matching category in the corresponding literature. Note that we use different names for our categories because we are describing optimization types instead of root-cause types of performance issues and we need category names to be short enough to fit into chart legends.

Figure 3 shows that the most common types of optimizations are graphics simplifications, rendering optimizations, API / language optimizations, heap avoidance, caching values, and conditionally avoid computations. The six categories cover 168 of our total 183 studied optimizations. Among the six categories, three of them (graphics simplifications, rendering optimizations, heap avoidance) are new categories of optimizations, and the prior studies did not observe such optimizations in general software projects. The three categories cover 102 (55.7%) optimizations.

> **Finding 1:** *We find three new categories of optimizations which are graphics simplifications, rendering optimizations, and heap avoidance. They are not observed in prior studies, and account for more than half of studied optimizations, showing that the landscape of optimizations in VR software is very different from that of traditional software.*

To better understand the characteristics and common patterns of new major categories of VR optimizations, we present some representative examples of optimizations below.

C1: Graphics Simplification. Developers simplify graphics to achieve better performance. This type of optimizations is similar to workarounds [49], since simplifications sacrifice the quality of graphics display. We found 44 graphics simplifications from the 183 studied optimizations. The most common subcategories of graphics simplifications are shader simplifications (19 out of 44) which removes or simplifies runtime lighting effects of game objects, and 3D model simplifications (14 out of 44) which replace the 3D model of a game object with a simpler 3D model with less polygons. Other

TABLE I: Mapping of Optimization Types

Optimization Type	Root Cause Type [47]	Root Cause Type [34]
API/Language Optimization	Inefficient API Usage	API Misunderstanding
Caching Values	Repeated Execution of the Same Operations	-
Conditionally Avoid Computation	A Computation Can Be Simplified or Avoided in Special Cases	Skippable Functions
Multi-Thread	-	Synchronization Issues
Remove Redundant Check	Repeated Check of Same Condition	-

graphics simplifications include downgrade resolutions (6 out of 44) and particle systems (provided by Unity framework to handle particle effects such as fog and fire) simplification (4 out of 44), and directly removing game objects from a scene (1 out of 44). Most of these optimizations affect only the visual experience of users, but some extreme cases may have effect on software features. For example, in a shader simplification (a85ab3ab6a0) from the Moon Motion project (MoonMotionProject/MoonMotion.git), the developers largely reduced the visible lighted area to reduce rendering costs.

Finding 2: *Unlike in traditional software where performance optimizations usually do not affect external software behavior, in VR software, users' visual experience and even some features can be sacrificed / adapted for better performance.*

Revision-wise, graphics simplifications are typically related to revise the references pointing to model files or shader files, as illustrated in example below.

```
cameraSkybox.mat
- m_Shader: {fileID: 103, guid: 000...000, type: 0}
+ m_Shader: {fileID: 10700, guid: 000...000, type: 0}
```

Listing 1: An Example of Shader Replacement (jdknox/vrDemo, 87a97d0)

C2: Rendering Optimizations. As VR frameworks like Unity typically provides many optimization opportunities for its off-stream software applications, developers shall properly design their scenes and configure their projects to take full advantage of such opportunities. For example, turning on static / dynamic occlusion culling can avoid the rendering of game objects that are occluded by other game object and turning on light-baking can pre-calculate light effects during compilation time. Also, setting game objects as static can avoid invoking many life-cycle methods, and bundling objects together can reduce the number of draw calls because when bundled multiple objects can be drawn together with one draw call.

We identified 31 rendering optimizations in total. The most common rendering optimizations are (1) draw-call batching (10 out of 31) through game-object / mesh combinations, (2) rendering setting changes (10 out of 31) such as turning on occlusion culling and light baking, and (3) disabling objects (8 out of 31) to avoid invoking unnecessary life-cycle methods on static game objects. The remaining 3 rendering optimizations change settings of colliders on game objects to avoid simulation of certain types of collisions.

```
ChemView Main Scene.unity
- GameObject:
- m_PrefabParentObject: {fileID: 100178, guid: 130...823, type: 3}
- m_PrefabInternal: {fileID: 371302497}
CombinedMeshes_ibuprofenDecimated.prefab
+ m_Materials:
+ - {fileID: 2100002, guid: 130...823, type: 3}
```

Listing 2: An Example of Mesh Combination (CallumHoughton18/ ChemViewAR, 54ebcaf3)

For draw-call batching and disabling game objects, developers often need to re-organize game objects. List 2 shows an example of mesh combinations. In the revision, meshes are moved from the scene (.unity file above) to a newly created combined mesh file (.prefab file below). Rendering setting changes are not straightforward because developers need to balance the positive and negative effects of certain settings. For example, turning on occlusion culling may reduce the rendering cost of occluded objects but will also cause extra cost of calculating the occlusion relations.

Finding 3: *The second largest category of optimizations are rendering optimizations (31 out of 183). Unlike optimizing traditional software where often only source code are modified, rendering optimizations involve various rendering configurations, so VR developers shall have comprehensive knowledge on configuring the framework and tuning scene designs (e.g., the combinations of objects and meshes).*

C3: Heap Avoidance. Although heap allocations can also potentially cause performance issues in traditional software, their effect is mainly on the memory side and are rare as reported in the prior studies [35], [47]. However, avoiding heap allocations is the fourth largest category of VR optimizations and accounts for 27 out of 183 (14.8%) studied optimizations. In some commit messages, developers mentioned that "to improve performance one of the goals is to reduce heap allocations to a minimum" (ExtendRealityLtd/Zinnia.Unity, 324b52b). VR developers try to remove as many heap allocations as possible because the garbage collection process is unpredictable and costly. Although the garbage collection seldom causes sensible lags in Internet and GUI software, its short-term performance overhead may largely affect animation fluency. Furthermore, many methods of VR software are invoked in each frame, so if local objects are allocated in them, garbage will accumulate

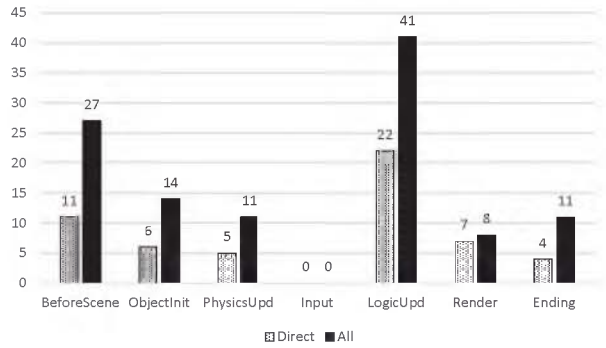

Fig. 4: Life-Cycle Phases Affected by Optimizations

very fast and the garbage collection will become very frequent, making the performance overhead even more severe.

PatronBorder.cs

```
+ private Vector3 weighDirection; ...
- Vector3 weighDireciton = new Vector3(Mathf.Abs(Alignment.x),
-         Mathf.Abs(Alignment.y), Mathf.Abs(Alignment.z));
+ weighDirection.x = Mathf.Abs(Alignment.x);
+ weighDirection.y = Mathf.Abs(Alignment.y);
+ weighDirection.y = Mathf.Abs(Alignment.y);
```

Listing 3: An Example of Heap Avoidance (microsoft/MixedReality-Toolkit -Unity, 5c26eb54)

In our study, we observed that the most common way (14 out of 27) to remove heap allocations is using a field reference to replace object initialization, because a field of a class can be initialized only once and reused in different methods. Other ways to avoid heap allocations include avoiding the usage of language features (e.g., LINQ [11]) that lead to allocation of temporary objects (8 out of 27), and simplifying boxing / object data types to primitive types (5 out of 27). One interesting observation is that developers may use references to fields or singletons to replace local objects, which may downgrade the application's design quality. One example of such replacement is presented in List 3.

> **Finding 4:** *VR developers try to reduce heap allocations to a minimal level, even if the avoidance of heap allocations may cause design quality downgrade and requires understanding of temporary objects in programming language implementation.*

Other Categories. The remaining categories are reported in prior studies. In particular, API / Language optimizations replace API methods or language features with more efficient ones. The prior studies [35], [47] report that this type of optimizations is the most common type of performance optimizations in other software. We find that API / Language optimizations are also a major type of optimizations in VR software. For example, we observed that developers replaced

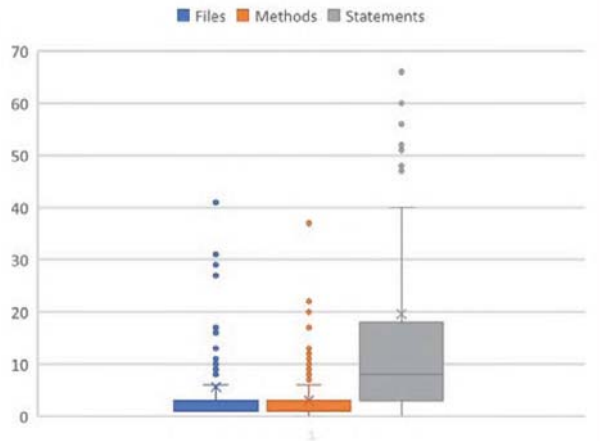

Fig. 5: Size Distribution of Optimizations

their implemented matrix multiplications with API calls, `UnityObjectToClipPos()`, to more efficiently translate positions between coordinate systems. As another example, we observed that developers replaced `foreach` with `for`, since `foreach` increases the overhead [3].

Caching values is another major type of optimizations in both VR and other software. It caches computation results in memory, so that when the same computation is required, the values stored in memory can be reused. The prior studies [39], [47] observed caching values in other software, but not quite common. Caching values is especially important in VR software because many of its computations are performed per frame, so caching the results will save computation resources.

In the category of conditionally avoid computations, developers add condition checks to skip certain computations, and in the category of removing redundant checks, developers remove the predicates whose values are always true or false. Both categories are commonly observed in other software. In the category of moving code out of frame updates, developers move code from `Update()` methods to `Start()`, if the code only need to executed only once.

Finally, multi-thread-related optimizations (adding parallelism), code simplification (removing dead code), and algorithm updates (using more efficient algorithms) are also observed in other software, and they are rare performance optimizations in both VR and other software.

B. RQ2: Affected Life-Cycle Phases

To answer RQ2, for each optimization, we use a call-graph analysis to determine the influenced phase (see Figure 1) of the optimization. This analysis identifies the impacts of the optimizations with source code changes, but cannot identify the impacts of the optimizations with only asset changes. After our manual inspection, we determine that the optimizations on only asset changes mainly affect the rendering phase.

The results are presented in Figure 4. In this figure, the "Direct" bars denote that the revisions are directly made in life-

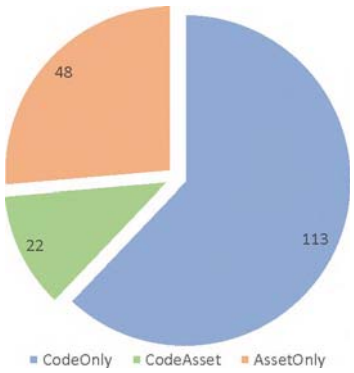

Fig. 6: Distribution of Source Code and Asset Optimizations

cycle phase methods, while the remaining are detected through call-graph analysis. We notice that a half of the optimizations are direct revisions of life-cycle methods. The distributions in Figure 4 show that the logic update phase is the most affected phase (41 out of 135 source-code-revising optimizations), and the second affected phase is the before scene phase (27 out of 135 source-code-revising optimizations). The observations lead to a finding:

> **Finding 5:** *Among all life-cycle phases, the logic-update and before-scene phases are mostly affected by performance optimizations, indicating that developers shall pay more attention to the two phases, when they debug performance issues.*

C. RQ3: Size and Complexity of Optimizations

To answer RQ3, we conduct a code dependency analysis to obtain modified files of optimizations. For each optimization, we measure its size with the number of modified files, methods and statements, and we measure its complexity by whether it involves both source files and asset files. If an optimization involves only source files, we further measure its complexity by whether it involves only one class (i.e., inner-class). Here, if an optimization is a systematic change, we consider it as inner-class, if each instance of the change is inner-class.

The distributions of modified files, methods and statements are presented in Figure 5 (some extremely large outliers are omitted). For methods and statements, we consider their changes on optimizations with source file changes, in that asset files do not have methods or statements. As many asset files are automatically generated and updated, even a simple modification can introduce many modified lines. For example, when developers replace a game object in a scene, this simple change modified many lines of code. To calculate the manual effort accurately, we do not count modified lines in asset files.

Figure 5 shows that most performance optimizations do not involve many files. The medium number of modified files and methods are both one, and the medium number of modified statements is eight. Furthermore, except a small number of outliers, most optimizations modify fewer than six files, six methods, and 40 statements. We further inspected the outliers. Among the six optimizations involving more than 20 file revisions, three are systematic changes which replace graphics models / shaders, two are systematic changes to avoid the usage of heaps, and the remaining one is a rendering optimization combining multiple game objects. Among the seven optimizations involving more than 60 statement revisions, five are systematic changes to avoid the usage of heaps or API replacements to more effective ones; one is an algorithm enhancement; and the remaining one is replacing the developers own implemented code with an API call.

We further studied the proportion of optimizations that involve only source code revisions, only asset file revisions, and both source code and asset revisions. The result is presented in Figure 6. From Figure 6, we can see that the majority of optimizations involve only source code revisions. While 70 (38.3%) optimizations also involve asset file changes, only 22 (12.0%) optimizations involve both source code and asset revisions. Furthermore, within the 113 optimizations with only source-code revisions, 101 of them are inner-class revisions.

The above observations lead to a finding:

> **Finding 6:** *The majority of VR optimizations need only small modifications on local files. Even when some optimizations involve more files, they mostly consist of repetitive smaller changes.*

D. RQ4: Systematic Changes

In our study, we find that many optimizations are systematic changes where a similar revision is made at different locations. Figure 7 shows the proportion of systematic changes within each category of optimizations. From the figure, we can see that different categories of optimizations have different likelihood to be systematic changes. Systematic changes are mostly common in heap avoidance optimizations and API / Language optimizations, where more than half of all optimizations are systematic changes. Systematic changes are also common in graphics simplifications because developers often need to simply multiple game objects of the same category. In total, 72 (39.3%) out of the 183 optimizations are system changes.

> **Finding 7:** *Systematic changes are common among VR performance optimizations, indicating that code pattern detection tools and code-clone detection tools can be useful for developers to find all locations to perform revisions.*

E. RQ5: Design and Behavior Effects

Performance optimizations may affect software design quality. However, a prior study [47] shows that effects on software design are actually not common in Javascript optimizations. In VR software development, due to the more severe performance

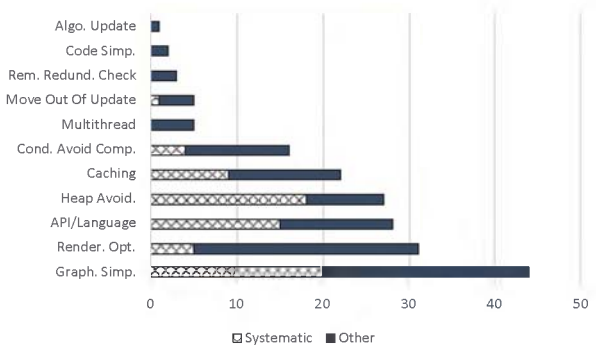

Fig. 7: Systematic Changes in Optimization Categories

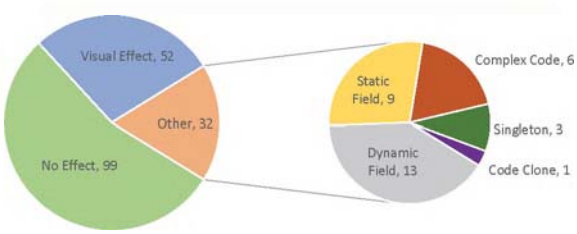

Fig. 8: Behavior/Design Effects of Performance Optimizations

bottleneck, developers may be more aggressive to optimize code. To understand to what degree such optimizations affect VR software, we analyzed the design and behavior effects of VR optimizations, and the results are presented in Figure 8. In total 84 (45.9%) out of 183 optimizations have design and behavior effects. In particular, we identified 52 optimizations with behavior effects, including lower resolutions (6), game object simplifications (19), and animation simplifications (27). Meanwhile, we identified 32 optimizations with design effects, including replacing local variables with private dynamic field access (dynamic field, 13), replacing local variables with public static field access (static field, 9), using singletons to avoid object allocation (singleton, 3), resulting in more complicated code (complex code, 6), and code clone (1). Note that the complex-code category mainly includes the change from `foreach` to `for` which requires more code to iterate over the collection. For dynamic fields, accessing private dynamic fields is less negative than accessing public static fields (like global variables), but it still causes side effects of methods and additional dependencies (which are added only for better performance) among methods.

Finding 8: *Unlike the results on traditional software from prior studies, a large portion of VR performance optimizations may have effects on program behavior and software design, showing that VR developers are more aggressive in sacrificing other aspects of software quality for performance.*

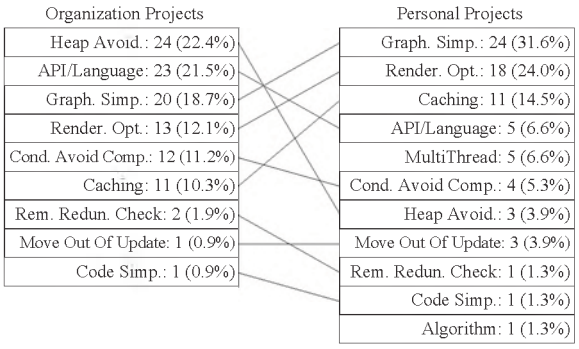

Fig. 9: Optimization Categories in Organizational/Personal Projects

F. RQ6: Organizational vs. Personal Projects

In our study, we found that VR software projects fall into two very separated groups: organizational projects which are driven by large companies and organizations like Microsoft, Unity, and Extended Reality and personal projects which are developed by individual developers or very small teams. We believe that this is partly because VR is an emerging technique, so large companies just start to move in while small start-ups have not grown up yet. We use the criterion of 3 developers and 50 stars (we set a star threshold because we find one of the studied projects is a course project with more than 10 student developers) to separate projects. We identified 11 organizational projects with 107 optimizations, and 34 personal projects with 76 optimizations.

It is interesting to see whether performance optimizations are different between these two groups of projects. We compare the distribution of different optimization categories between two groups of projects in Figure 9. Another conjecture is developers from large organizations may care more about design / behavior effects of optimizations, so we further compare the distribution of optimizations with design / behavior effects between two groups of projects in Figure 10.

From Figure 9, we find that the distribution of optimization categories are different between the two groups of projects. In particular, heap avoidance is most common in organization projects, but rarely found in personal projects. Here, we double checked to find that it appears in 8 out of 11 organizational projects, showing that it is not the case that one project dominates the result. API / Language optimizations also have similar distribution. There may be two reasons behind this major difference. First, personal project developers may be inexperienced and not aware of the performance issues caused by heap allocations and some inefficient API methods. Second, since personal projects are typically of smaller scale, the performance issues related to these two optimization categories may be not exposed or simply not important.

Graphics simplifications and rendering optimizations are common in both groups of projects, but more common in personal projects. This is possibly because organizational developers are less flexible in downgrading graphics quality which may affect user experience. Also, some personal project

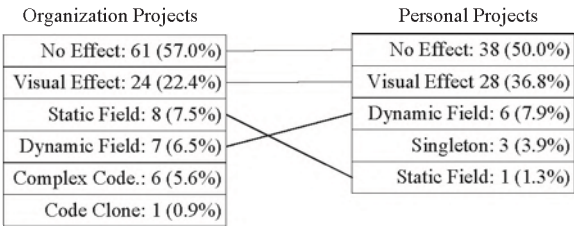

Fig. 10: Design/Behavior Effects in Organizational/Personal Projects

developers may be unfamiliar with rendering optimization settings so they need to fix them while organizational developers already set them properly at the beginning. Finally, the caching category has similar distribution in two groups of software.

> **Finding 9:** *Distributions of optimizations categories are very different between organizational and personal software projects. Heap avoidance and API/Language optimization are most common in organizational projects but rarely seen in personal projects, while there are more graphics simplification and rendering optimizations in personal projects.*

From Figure 10, we can see that optimizations in organizational projects and personal projects have similar likelihood to have design and behavior effect (43.0% vs. 50.0%). In particular, optimizations in organizational projects cause more design / behavior effects on static field access, and optimizations personal projects cause more design / behavior effects on visual effect and singleton. We believe this is mainly related to the difference between optimization-category distributions (on heap avoidance and graphics simplification).

G. Threats to Validity

The major threat to the internal validity of our study is the correctness of our manual inspection of performance optimization commits. To reduce this threat, we have two inspectors to independently inspect the commits, and perform reconciliation of the inspection results afterward. Furthermore, we use static analysis including call-graph analysis and static dependency analysis to support our manual inspection to reduce potential negligence of code portions. The major threat to the external validity of our study is that our findings may be specific to our set of projects and commits, or Unity and C# projects with meta files. To reduce this threat, we collected performance commits from 45 different projects including 11 organizational projects from Unity, Microsoft, ExtendedReality, etc., and 34 projects from small teams / individual developers. While our findings may be specific to UnityList / Unity / C# projects, since UnityList is the largest VR open source repository, Unity is a dominating VR framework, and C# is the default programming language for Unity, such findings are still significant for VR developers and researchers on VR performance and VR software engineering.

V. Lessons Learned

A. For VR Developers

For novices or experienced developers without VR experience, the findings in this paper lead to the following concrete suggestions:

- VR developers should try to *understand computation cost of 3D models and shaders* before using them. Testing them when adding them will be helpful because once a lot of models and shaders are added it can be difficult to tell which one(s) cause the performance downgrade (Note that profiling of individual game objects is still not possible in current GPU).
- There are a lot of *rendering optimization opportunities* (e.g., occlusion culling, static light baking, mesh combination) provided by the VR framework so novice developers should learn about them and try to adapt their scene design to make best use of them.
- Heap objects can cause frequent garbage collections and thus affect the fluency of animation. Organizational project developers tend to *reduce heap allocations as much as possible*. So, developers may try to avoid using too many heap allocations in methods invoked every frame, and be aware that garbage collection may be the reason for sudden downgrade of frame-per-second. Also, it should be noted that some language features (e.g., Language Integrated Query) will also lead to temporary heap objects after compilation.

B. For Software Engineering Researchers

Our study opens up many potential research opportunities for software engineering researchers towards developing better framework and IDE for VR developers.

Cost Estimation of Game Objects. Since GPU does not allow profiling of individual game objects and developers often need to select 3D models and shaders with proper cost, the cost estimation of game objects is highly desirable. Combining static analysis and machine learning (maybe supplemented with delta debugging [58]), it is possible to precisely estimate cost of individual game objects.

Estimate User Sensibility of Graphics Downgrades. Although more of a topic of human-computer interaction and machine learning, estimating whether graphics quality downgrades are sensible to users can largely benefit VR developers when they try to sacrifice rendering quality for performance.

Recommendation of Rendering Settings and Optimizations. VR frameworks have very complicated settings to configure the optimizations in rendering. Making proper decisions on settings often requires in-depth understanding of the optimization mechanism in the back end (e.g., the difference between static and dynamic occlusion culling, in what condition different meshes can be batched together), and the scenes in the VR applications. An expert system to recommend rendering setting can largely reduce developers' effort on searching the parameter space for best configuration.

Detection and Automatic Refactoring of Heap Allocations. Detecting and refactoring heap allocations (including latent allocations in certain language features) invoked every frame can help developers to locate performance bottlenecks, which can be done with static analysis and program transformation. Such removal often cause downgrade of design quality (e.g., singletons, static fields), so some compilation-time optimization technique may help to keep the benefit of both ends.

Pattern Detection Techniques for Systematic Changes. Our study finds that systematic changes are common in VR optimizations, showing that pattern detection techniques may help to detect certain optimizing opportunities. Although systematic editing tools [40] already exist, the extension of such techniques to asset files (e.g., finding and maintaining co-changing asset files) worth further study.

Empirical Studies on VR/AR Software Optimizations. Our exploratory study acquired a number of interesting findings, which may need further validation with a larger dataset and more advanced statistical analysis. These findings may serve as candidate hypothesis for future empirical studies on VR/AR software optimizations.

VI. Related Work

A. Studies on VR and Game Development

The authors are not aware of many efforts in the area of VR software development, but there exist studies on game development. Murphy-Hill et al. [42] performed a study on video game developers to understand the challenges in video game development and how they are different from traditional software development. Washburn et al. [53] studied failed game projects to find out the major pitfalls in game development. Lin et al. [38] studied the common updates in steam platform to understand the priority of game updates. Rodriguez and Wang. [46] performed an empirical study on open source virtual reality software projects to understand their popularity and common structures. Pascarella et al. [44] studied open source video game projects to understand their characteristics and the difference between game and non-game development. Zhang et al. [59] studied possible solutions to detect potential privacy leaks based on graphics user interface analysis [45], [52], [54] of mobile augmented reality apps. Compared with these research efforts, our study focuses on VR software projects and concrete performance optimizing code commits.

B. Studies on Performance Bugs and Optimizations

Since performance is a critical non-functional requirement of software systems, various studies analyzed the performance optimizations and bugs of software systems. Zaman et al. [56] performed an bug report analysis [50] of Firefox project. Their study focused on unique security and performance bugs of Firefox web browser. Their followup work [57] analyzed the bug reports of Firefox and Chrome to differentiate characteristics performance and non-performance bugs. To identify performance bug code patterns, Jin et al. [35] analyzed the root causes of 109 performance bug collected from five projects. Nistor et al. [43] did a study on performance and non-performance bugs from three popular codebases: Eclipse JDT, Eclipse SWT, and Mozilla. Liu et al. [39] studied the characteristics of performance bugs in Android apps and proposed pattern-based approach to detect such bugs. Selakovic and Pradel [47] studied optimizations in JavaScript applications to find out the major categories of JavaScript performance optimizations and how they are different from optimizations of other software. Han and Yu [30] performed an empirical study of performance bugs in highly-configurable systems to find out the relations between performance optimizations and configuration spaces. Mostafa et al. [41] developed a testing approach to identify performance regressions more efficiently. In a recent effort, Chen et al. [26] studied 700 performance bug fixing commits across 13 popular open-source projects characterizes the relative frequency of performance bug types as well as their complexity. Based on the results of performance bug/optimization studies, there are also research efforts on pattern-based performance bug detection. Chen et al. [25] discussed performance anti-patterns for mobile applications. Several other techniques [37], [48], [55] adopted static pattern detection to detect performance bugs. Han et al. [31] utilize machine learning to generate test frames to guide actual performance test case generation. The prior studies on performance bugs and optimizations focused on generic software or mobile software. In our study, we focus on performance optimizations in VR software, which we found to be very different from those in other software projects.

VII. Conclusion

In this study, we conducted an empirical analysis on 183 VR performance optimizations from 45 open source VR projects from UnityList. We manually inspected the optimizations and developed a taxonomy of VR performance optimizations. In addition, we applied various analyses on the code revisions to understand the life-cycle phases they affect, their complexity, their design / behavior effects, and the repetitiveness of revisions. We also compared the optimizations between large organizations and small teams. To the best of our knowledge, this is the first empirical study on VR performance optimizations and we summarized nine useful findings for VR developers and software engineering researchers to better understand VR optimization. In the future, we plan to perform larger scale studies on more projects to further validate the findings of our study. Moreover, based on our findings, we plan to develop code analyzers to detect performance-related bugs in VR software, and to build a tool for providing suggestions on VR performance optimization.

Acknowledgments

The UTSA authors are supported in part by NSF Awards NSF-1846467 and NSF-2007718. Hao Zhong is sponsored by the National Key R&D Program of China No. 2018YFC083050.

REFERENCES

[1] Apple app store. https://www.apple.com/ios/app-store/, 2020. Accessed: 2020-06-30.

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

[3] Avoid foreach. https://aeflash.com/2014-11/avoid-foreach.html, 2020. Accessed: 2020-06-30.

[4] C sharp clone detection. https://archive.codeplex.com/?p=clonedetectivevs, 2020. Accessed: 2020-06-30.

[5] Code dependency analysis. https://github.com/topics/dependency-analysis?l=c\%23, 2020. Accessed: 2020-06-30.

[6] Google arcore. https://developers.google.com/ar, 2020. Accessed: 2020-12-30.

[7] Google cardboard. https://arvr.google.com/cardboard/, 2020. Accessed: 2020-12-30.

[8] Google daydream. https://arvr.google.com/daydream/, 2020. Accessed: 2020-12-30.

[9] Google play. https://play.google.com/store, 2020. Accessed: 2020-06-30.

[10] Jgit - java implementation of the git version control system. https://www.eclipse.org/jgit/, 2020. Accessed: 2020-06-30.

[11] Language integrated query. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/, 2020. Accessed: 2020-06-30.

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

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

[14] Oculus app store. https://www.oculus.com/experiences/quest/, 2020. Accessed: 2020-06-30.

[15] srcml - an infrastructure for the exploration, analysis, and manipulation of source code. https://www.srcml.org/, 2020. Accessed: 2020-06-30.

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

[17] Steam vr. https://store.steampowered.com/steamvr, 2020. Accessed: 2020-12-30.

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

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

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

[21] Unitylist. https://unitylist.com/, 2020. Accessed: 2020-06-30.

[22] Vr user statistics. https://techjury.net/blog/virtual-reality-statistics/#gref, 2020. Accessed: 2020-06-30.

[23] R. Albert, A. Patney, D. Luebke, and J. Kim. Latency requirements for foveated rendering in virtual reality. *ACM Transactions on Applied Perception (TAP)*, 14(4):1–13, 2017.

[24] L. P. Berg and J. M. Vance. Industry use of virtual reality in product design and manufacturing: a survey. *Virtual reality*, 21(1):1–17, 2017.

[25] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1001–1012, New York, NY, USA, 2014. Association for Computing Machinery.

[26] Y. Chen, S. Winter, and N. Suri. Inferring performance bug patterns from developer commits. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 70–81, 2019.

[27] O. Ciftcioglu and M. S. Bittermann. Solution diversity in multi-objective optimization: A study in virtual reality. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 1019–1026. IEEE, 2008.

[28] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[29] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.

[30] X. Han and T. Yu. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2016.

[31] X. Han, T. Yu, and D. Lo. Perflearner: Learning from bug reports to understand and generate performance test frames. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 17–28, New York, NY, USA, 2018. Association for Computing Machinery.

[32] F. Hassan, S. Mostafa, E. S. Lam, and X. Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 38–47. IEEE, 2017.

[33] F. Hassan and X. Wang. Mining readme files to support automatic building of java projects in software repositories. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 277–279. IEEE, 2017.

[34] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.

[35] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. *SIGPLAN Not.*, 47(6):77–88, June 2012.

[36] C. D. Just. Performance analysis of a virtual reality development environment: Measuring and tooling performance of vr juggler. Master's thesis, Citeseer, 2000.

[37] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 17–26, New York, NY, USA, 2010. Association for Computing Machinery.

[38] D. Lin, C.-P. Bezemer, and A. E. Hassan. Studying the urgent updates of popular games on the steam platform. *Empirical Software Engineering*, 22(4):2095–2126, 2017.

[39] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering*, pages 1013–1024, 2014.

[40] N. Meng, M. Kim, and K. S. McKinley. Sydit: creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 440–443, 2011.

[41] 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*, pages 23–34, 2017.

[42] 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*, pages 1–11, 2014.

[43] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 237–246. IEEE Press, 2013.

[44] 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)*, pages 392–402. IEEE, 2018.

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

[46] 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)*, pages 474–475. IEEE, 2017.

[47] M. Selakovic and M. Pradel. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, pages 61–72, 2016.

[48] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 167–177, 2012.

[49] D. Song, H. Zhong, and L. Jia. The symptom, cause and repair of workaround. In *Proc. ASE*, page to appear, 2020.

[50] 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*, volume 2, pages 315–316. IEEE, 2010.

[51] B. Wang and P.-L. P. Rau. Effect of vibrotactile feedback on simulator sickness, performance, and user satisfaction with virtual reality glasses. In *International Conference on Human-Computer Interaction*, pages 291–302. Springer, 2019.

[52] 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*, pages 37–47, 2018.

[53] 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*, ICSE '16, page 280–289, 2016.

[54] X. Xiao, X. Wang, Z. Cao, H. Wang, and P. Gao. Iconintent: automatic identification of sensitive ui widgets based on icon classification for android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 257–268. IEEE, 2019.

[55] D. Yan, G. Xu, and A. Rountev. Uncovering performance problems in java applications with reference propagation profiling. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 134–144. IEEE Press, 2012.

[56] S. Zaman, B. Adams, and A. E. Hassan. Security versus performance bugs: A case study on firefox. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 93–102, New York, NY, USA, 2011. Association for Computing Machinery.

[57] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 199–208. IEEE Press, 2012.

[58] A. Zeller. Yesterday, my program worked. today, it does not. why? *ACM SIGSOFT Software engineering notes*, 24(6):253–267, 1999.

[59] 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)*, pages 114–1141. IEEE, 2019.

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