An Investigation on Maintainability of Update Methods in Game Loops

Koyo KANAMORI  Shuji MORISAKI  Shuichiro YAMAMOTO
Graduate School of Informatics, Nagoya University
morikanayamamoto@gmail.com, morisaki@nagoya-u.jp, yamamotosui@icts.nagoya-u.ac.jp

Abstract  Previous researches revealed that games software has different contexts such as demand for early implementation in games software. We focus on the maintainability of update methods in game loops because maintainability is considered in games software. The investigation shows that update methods are implemented with either low or high maintainability. Following an explanatory sequential design, we investigate the implementation of update methods in popular repositories in Github and conduct semi-structured interviews with commercial games software developers to inquire about whether the results of the investigation are realistic in commercial games software. The investigation shows that update methods are implemented with conditional branches with state variables (lower maintainability) in 39 repositories and the State design pattern (higher maintainability) in 1 repository. In the interviews, 8 out of 11 commercial games software developers indicated that the results of the investigation are common in commercial games software.

Keyword  Games software development, Game loop, Update method

1. Introduction

Interest in computer games research has been increasing. The differences between software engineering for games and other types of software engineering have been studied [1][2][3][4][5][6][7][8]. Identified differences include software requirement engineering, need for early implementation, and the importance of runtime performance. Studies on software requirement engineering in games software include the following studies. Callele et al. identified pre-production and production phases, and investigated requirements elicitation in both phases in real games development [1]. Yannakakis and Hallam pointed out the importance of exploring enjoyment in requirement engineering [2]. Stacey and Nandakumar identified the play-test feedback loop in games development [3] and its requirement engineering [4].

Studies on early implementation in games software developments include the following studies. In a survey of seven games development organizations, Kasurinen et al. revealed that the requirements for games development and design tools for games include early prototyping [5]. Petrillo et al.’s survey on games development found that ensuring feasibility for graphics features is important [6]. Alves et al. noted a trade-off between creativity in games design and technical limitations in mobile games development and suggested early implementation to balance this trade-off [7].

Some studies have suggested that games developers should focus more on runtime performance [8][9]. Murphy-Hill et al. reported that games developers prioritize runtime performance by interviewing 14 developers and analyzing 364 responses in their survey [8]. Bishop et al. highlighted runtime performance as one of the three major design goals in developing NetImmerse, a game engine that they developed [9].

Blow observed that games development has changed from optimizations maximizing limited hardware resources to challenges for large-scale project management and domain-specific requirement engineering [10]. Games software engineering studies demonstrated the effectiveness of employing principles accumulated in general software engineering to deal with these challenges. Such studies often introduce design patterns. For example, Nguyen and Wong proposed design patterns for board games, including chess [11]. Wetzel defined a design pattern template for mobile mixed reality games and showed design patterns to determine game rules and solutions for technical constraints [12]. White et al. proposed the State-effect pattern, Restricted iteration pattern, and Concurrency pattern as solutions for issues among common action games [13].

Although applying design patterns increases maintainability and implementation productivity, it may reduce runtime performance, especially when developing games for limited-hardware resources. To empirically investigate this,
Zhang et al. re-implemented five existing role-playing games for cell phones by using runtime performance aware object-oriented programming and showed smaller lines of code and memory usage and higher runtime performance than the original implementation [14]. Ampatzoglou and Chatzigeorgiou re-implemented two existing open source games using the State pattern, and reported maintainability metric improvement (the method cohesion metric decreased) [15].

Whereas the empirical studies revealed that the runtime performance aware rules increased the maintainability of games software, such approach may disturb the other unique characteristics of games software developments such as a specific style of requirement engineering and early implementation. Furthermore, compared to other software developments, games software development usually increases the less-technical debt [16] due to the limited number of evolutions in games software. Hence, it is worthwhile to investigate the maintainability of games software as one of the investigations on employment of general software engineering principles in games software engineering.

This paper examines update methods in game loops, which are widely used as a common and fundamental implementation and architecture for action games to capture an aspect of the maintainability in games software. Game loops enable game objects to behave in parallel by invoking update methods sequentially. Each update method implements the corresponding game object behavior. Here, we selected update methods in game loops for three reasons. First, game loops and update methods are architectures that allow several implementation choices such as conditional branches with state variables (a runtime-performance-sensitive implementation) and the State design pattern (maintainability-focused implementation that enables other developers to change source code easily) [17]. Second, many action games, including classic action games such as Pac-Man and the latest action games for console game machines, use game loops and update methods [18]. Third, game loop implementations can be objectively identified and determined. Much of the latest games software is implemented using general game engines based on component-based architecture such as Unity and Unreal Engine. This study excluded implementations with the game engines for two reasons. First, they systematically enable game objects to be implemented with high maintainability. Second, maintainability considering safe parallel processing is more important among the game object implementation using the game engines. We employed an explanatory sequential design consisting of a quantitative investigation followed by a qualitative analysis [19] to cover open source and commercial games software while avoiding confidentiality issues. First, we investigated game loops and update methods implementations in popular GitHub source code repositories as the quantitative study. Then we conducted semi-structured interviews with 11 game developers in the Japanese games industry as the qualitative study.

This paper is organized as follows. Section 2 introduces related works. Section 3 defines game loops, update methods, and three update method implementation types. Section 4 describes the study settings, the quantitative study (Study 1), and the qualitative study (Study 2). Section 5 discusses the results. Section 6 concludes the paper.

2. Related research

2.1 Surveys comparing games software engineering to other domains

2.1.1 General

Murphy-Hill et al. interviewed 14 games software developers and conducted a survey to investigate the difference between games and other software developments. The results of the interviews and the survey with 364 respondents showed that run-time performance, reuse, and design effort differed between games and other types of software developments [8]. Specifically, run-time performance, including graphics, was more important in games software development, while reusing source code was less important. Additionally, the results indicated that spending less effort to design was preferred in games software development because there is a possibility that developers should re-design and re-implement if enjoyment and fun were judged to be insufficient for players.

Pascarella et al. analyzed open source games software repositories [20] to reveal the differences between games and other types of software. The results showed that games software repositories had resource files, including audio files and that the developers included artists. Bug reports contained graphics-related bugs. Architecture changes and source code reuses were less frequent than those in other software developments. Kultima and Alha regarded novelties in games as innovations [21]. They conducted interviews with 28 speakers at 3 game conferences to investigate how to produce innovations intentionally. Alves et al. pointed out that there was a trade-off between games design creativity and technical constraints, which must be developed by games software developers [7].

Kasurinen et al. conducted interviews at seven games software companies and analyzed the results by grounded theory to determine expectations of games development support tools [5]. The results of the analysis showed that the tools should provide reusable game engines as well as support artistic presentation authoring and early prototyping in both pre-production and production processes.

2.1.2 Process

Stacey and Nandhakumar conducted interviews with 20 stakeholders, including managing directors, project managers, and computer graphics programmers in a games studio company to investigate the games development process [3]. The results identified games development specific characteristics such as the play-test feedback loop. Callele et al. divided the games development process into the pre-production and production processes [1]. Then by observing real games development, they studied requirement elicitation and identification in the processes.

Some studies have focused on modding, which is a unique way to reuse and evolve in games software. Mod originated from the word modify. Scacchi explained modding as an open
innovation approach similar to open source software [22]. In
moding, developers usually in the games community add
new game rules or graphics to the original game. Scacchi
categorized modding into user interface customizations, game
conversions, machinima, and hacking closed game systems.
Some commercial games providers publish their software
including visual and audio resources and allow anyone to
modify. Poretski and Arazy reported that the sales for a
modded game were statistically significantly larger than that
of the original game [23].

2.4 Implementation

Patterns. They re-implemented publicly accessible game
The game graphics design pattern used the State and Bridge
logic design pattern used the Strategy and Observer patterns.
They also implemented game logic and graphics design patterns for action games [13].

White et al. proposed standards to port adventure and movie games developed in the
1980s and 90s to cell phone apps [27]. White et al. proposed a scripting language that
supported the State-effect pattern to easily add source code for
new game objects [33]. Passos et al. proposed the framework GCore to implement a game loop. GCore provided safe initialization of game objects using dependency injection to reduce initialization bugs [34].

3. Game loops and update methods

3.1 Preliminary definitions

3.1.1 Game objects

Game objects are objects that appear in games. A game
object can behave independently from other game objects
while satisfying a defined game rule (game logic) [35]. Game
objects include player characters, computer-controlled
characters, obstacles, and background images. They also
include abstract or invisible objects such as light sources and
event triggers. The behavior of each game object is defined by
source code statements in the corresponding update method. A
game object has its states that defines statuses such as
coordinate positions and invincible mode and determines its
behaviors such as rapid movement and a special command
support.

The active game objects $g_i$ are displayed in the game
screen or are located on the game map. The game screen is the
visible area of the game map. Game objects are categorized
into a game object type $T_i (1 \leq i \leq r)$. The update method $M_i$
behaves the behaviors of the corresponding game object
type $T_i$. Each game object of type $T_i$ is called $g_i (1 \leq j)$. If a
game is implemented in an object-oriented programming
language, game objects categorized into the same game object
type $T_i$ will be implemented as the same update method $M_i$
defined in the game object class $G_i$.

3.1.2 Frame

A frame is the minimum unit of time in a game. The
number of frames per second (FPS) usually corresponds to the
screen refresh rate. All processes in a game loop are designed
to be executed within one frame. Time wait processes are
implemented at the end of a game loop to absorb the
execution time difference because the execution time largely
depends on the game objects in the screen or on the map.
Hereinafter $n$-frame refers to the $n$-times iteration of the game
loop.

3.2 Game loop

A game loop is an event loop that decouples the progression
of game time from a user input and processor time. Game
loops are widely employed in action games [17] and are a
common application architecture of action games. A game
loop invokes each update method of the active game objects
gy within one frame. Invoking all update methods within one frame allows every game object to behave in parallel. The advantage of a game loop is that it can invoke update methods in pre-defined sequence compared to multi-threading, which executes each update method as an individual thread [36].

A game loop consists of the following four processes:
- \( L_1 \): Accept user input
- \( L_2 \): Update game objects (update method invocation)
- \( L_3 \): Wait time (time adjustment)
- \( L_4 \): Draw the screen

A game loop repeatedly executes processes \( L_1, \ldots, L_4 \). Figure 1 shows the execution sequence at the n-frame. Update methods are indicated as \( M_1, \ldots, M_i \), and in Fig. 1 (\( Z \) is the number of active game objects). Figure 2 shows an implementation example of the simple synchronized coupled model [37] written in C++. The function \( \text{getCurrentTime} \) in lines 12 and 26 in Fig. 2 obtains the current time at the execution. The constant value defined by the macro \( \text{MS\_PER\_FRAME} \) is the number of milliseconds required to execute one frame.

### 3.3 Update methods and variables

Game loops and update methods can be implemented in various types of programming languages. This subsection explains them in object-oriented programming language because the Observer pattern defined in the GoF design patterns [28] illustrates game loops and update methods well. The update method \( M_i \) and the invoker \( L_2 \) can be regarded as the observers and the subject in the Observer pattern. Figure 3 shows the class diagram of the Observer pattern. The Observer pattern requires that observers put themselves into the list of observers of the subject in advance and that observers are notified of a change in the subject. Figure 4 shows the class diagram of the update method \( M_i \) and the invoker \( L_2 \).

In Fig. 2, the update method for the game object \( g_j \) is implemented as a method in the game object class \( G_i \). The update method \( M_i \) is implemented as a virtual method defined in \( G_i \) to decouple the implementations of the game object class \( G_i \) and \( L_2 \). More specifically, the base class \( G_i \) defines the virtual method \( \text{Update} \). Each class \( G_i \) inherits the \( G_i \) and overrides the virtual method \( \text{Update} \) as its own update method \( M_i \). The update method invoker \( L_2 \) can invoke the virtual method \( \text{Update} \) for each game object \( g_j \). All the game object classes \( G_i \) inherit the base class \( G_i \). All the update methods override the virtual method \( \text{Update} \) defined in the \( G_i \). Thus, the update methods can be executed by invoking the \( \text{Update} \) method in each element of the array \( \text{gameObjects} \).

The scopes of variables used in the game object class \( G_i \) can be categorized as scope for temporary variables (local scope) and persistent variables (wider scope than local scope). The temporary variables are initialized and disposed within a frame (in one update method invocation). The persistent variables \( V_i \) are updated and referenced across frames (in two or more update method invocations) such as the counter and coordinate positions in the screen. The persistent variables are stored as member variables (instance variables) in the game object class \( G_i \). More specifically, the update method \( M_i \) executes the following processes, \( E(n) \), \( E(n) \), and \( E(n) \), in frame \( n \) to consecutively update the game object states among frames \( n-1, n, \) and \( n+1 \).

- \( E(n) \): Restore the game object states from the variables stored in frame \( n-1 \)
- \( E(n) \): Update the game object states for frame \( n \)
- \( E(n) \): Store the game object states into variables for frame \( n+1 \)

Updating the game object states \( E(n) \) requires that the game object states at frame \( n-1 \) are stored \( E(n-1) \) and the game object states at frame \( n \) are restored \( E(n) \). The values of the variables \( V_i = \{ v_{i1}, v_{i2}, \ldots \} \) are stored in \( E(n-1) \) and restored in \( E(n) \). Along with \( V_i \), variables \( W_i = \{ w_{i1}, w_{i2}, \ldots \} \) may be used. \( W_i \) are variables which should be able to be
11, which is located at \((x_0, y_0)\) at frame 0. First, it moves horizontally to \((x_0 + 10, y_0)\) at frame 100 at a pace of 1 pixel per 10 frames (Behavior 1). Then, it moves vertically to \((x_0 + 10, y_0 + 10)\) at frame 200 at a pace of 1 pixel per 10 frames (Behavior 2).

![Figure 5. Example of behaviors of the game object \(g_{11}\)](image)

3.4 Update method implementation types

Implementations of update methods are categorized into three types, which depend on the maintainability and the programming language [17]. Below each type \(U_i\) is described using the same example of game object behaviors.

The example consists of Behaviors 1 and 2. Figure 5 shows the behaviors of the game object \(g_{11}\), which is located at \((x_0, y_0)\) at frame 0. First, it moves horizontally to \((x_0 + 10, y_0)\) at frame 100 at a pace of 1 pixel per 10 frames (Behavior 1). Then, it moves vertically to \((x_0 + 10, y_0 + 10)\) at frame 200 at a pace of 1 pixel per 10 frames (Behavior 2).

3.4.1 Type \(U_1\) conditional branches

Game object states are stored in variables with a wider variable scope than local variables. Game objects’ behaviors are switched by conditional branches depending on variables \(v_i\), which store the game objects’ states. This approach can be used in various programming languages, including procedural and object-oriented programming languages. The maintainability of this approach is usually lower than that of approaches \(U_2\) and \(U_3\) because changing and evolving game objects’ behaviors need to add the entire statements implementing behaviors to the existing statements and conditional branches without encapsulation. The changes may cause side effects.

Figure 6 shows an example of source code written in C for the game objects’ behaviors defined in section 3.4. In Fig. 6, the update method (update function) is \(G_1\)\_update. The behaviors of the game objects are switched depending on state variables cnt1 and cnt2 by the if clauses in lines 15 and 20. Behavior 1 is implemented by the statements in lines 16–18. Behavior 2 is implemented by the statements in lines 21–23. If the number of behaviors increases, the number of branches and states increase, which affects the maintainability of the update method. Although Figure 6 shows an example source code written in C, the source code can be written in an object-oriented programming language by defining the update method as a virtual method in the base class and overriding a virtual method in an inherited class corresponding to each game object, as shown in Fig. 2.

3.4.2 Type \(U_2\) state pattern

The State pattern is defined in GoF design patterns [28]. This pattern enables an object to alter its behaviors according to the state changes. In the State pattern, the class \(C_g\) which corresponds to each state, is defined to implement the behaviors. The class \(C_g\) implements the behaviors corresponding to the states. The statements implementing each behavior are encapsulated in the class \(C_g\). Switching the class \(C_g\) changes the game objects’ behaviors.

Type \(U_2\) is categorized into Types \(U_{2,1}\) and \(U_{2,2}\). Update methods categorized into Type \(U_{2,1}\) are those use the object-oriented programming language support. Update methods categorized into Type \(U_{2,2}\) are those do not use it. Game object states are stored to variables in the instance variable scope, and their behaviors are changed by switching objects \((U_{2,1})\) or pointers to update functions (methods) \((U_{2,2})\) implementing each behavior in the update methods and functions. Each object \((U_{2,1})\) or pointer \((U_{2,2})\) corresponds to a game objects state and implements the overridden update methods \((U_{2,1})\) or functions \((U_{2,2})\).

Figure 7 shows an example of Type \(U_{2,2}\) source code written in C++ for the game objects’ behaviors defined in section 3.4. In Fig. 7, the class \(G\)\_base is the base class for classes implementing game objects’ states and behaviors. The class \(G_1\) is the class implementing the game objects \(g_{11}\) (game object type \(T_1\)). The update method is \(Update\) defined in the class \(G_1\). The behaviors of the game objects \(g_{11}\) are implemented in the Update method defined in the class \(G\)\_base and overridden in the class \(G_1\). The class \(State\) is the base class for the class \(C_g\). In the example behaviors, the state classes are \(HorizontalMove\) implementing Behavior 1 and \(VerticalMove\) implementing Behavior 2. The classes defined as the nested class of \(G_1\). The class \(G_1\) defines the variables \(V_1 = \{cnt1, cnt2\}\) and \(W_1 = \{x, y\}\). The variables \(x\) and \(y\) are the member variables in class \(G_1\). The variables \(cnt1\) and \(cnt2\) are member variables in the classes \(HorizontalMove\) and \(VerticalMove\), respectively. The classes encapsulate the statements implementing Behaviors 1
3.4.3 Type $U_3$ coroutine

The game object states are stored to local variables in the update methods. This type requires a coroutine of the programming language or its library support. In coroutines, suspending statements such as "yield" store the values of local variables and then exit the coroutine. When the coroutine is invoked again, the stored values are resumed and the subsequent statement of the suspended statement is executed. Both the statements implementing the game object's behaviors and the variables storing the game objects state can be encapsulated in the update method. The maintainability of this approach is higher than those of $U_1$ and $U_2$ because coroutines can encapsulate the variables storing the game objects state and do not require statements to store or resume the variables $V_i$. Note that the variables $W_i$ that may be stored to variables declared in a wider variable scope due to other reasons.

Figure 8 shows an example of source code written in C# for the game objects' behaviors defined in section 3.4. The suspending statement in C# is “yield return.” In Fig. 8, the class G_base is the base class for classes to implement game objects states and behaviors. The class G1 is the class implementing the game objects $g_1$. The statements in lines 13–16 implement Behavior 1. The statements in lines 18–21 implement Behavior 2. The variables $V_1 = \{cnt1, cnt2\}$ are declared as local variable in the update method Update. The variables $W_1 = \{x, y\}$ are declared as the member variables in class G1.

4. Empirical investigation

4.1 Objective

This paper investigates the extent that software engineering principles are employed in the source code for computer games in both open source and commercial software developments in terms of maintainability of update methods in game loops. We followed an explanatory sequential design [19], which is categorized as a mixed method research [38]. Explanatory sequential design conducts a quantitative study followed by a qualitative one using the results of the first study. This paper examines maintainability of open source computer games as a quantitative study (Study 1) and conducts semi-structured interviews [39] with commercial games software developers as a qualitative study (Study 2). The questions in the semi-structured interviews use the results of Study 1. We selected this research design because of its wide coverage for open source and proprietary games software while avoiding confidentiality issues.

4.2 Procedure

4.2.1 Study 1

Step 1.1 Select candidate repositories by a keyword search

This step selects candidates of computer games repositories that have game loops by searching GitHub repositories with the keyword “game loop.” The search results are repositories that include one or more files containing the keyword “game” and “loop” in this order.

Step 1.2 Select popular repositories from the candidates

This step narrows the number of repositories obtained in Step 1.1 based on the number of stars. The number of stars indicates the number of users who have marked the repository.
Marking a repository allows users to revisit it quickly and find similar repositories easily. We regard repositories with more stars as popular and representative repositories for two reasons. First, the GitHub help guide [40] states that a higher number of stars is an indicator of a highly evaluated repository. Second, a previous study revealed that repositories with more stars in GitHub tend to be more highly engineered software projects [41].

Step 1.3 Remove non-computer game repositories
This step manually removes non-computer game repositories such as game engines and libraries for games from the repositories obtained in Step 1.2. Game engines and libraries may contain sample source codes that include the keyword “game loop.”

Step 2.1 Identify update methods
This step manually identifies update methods in each repository obtained in Step 1.3. First, the game loop and the \( L_2 \) statements are found, and then candidates of update methods invoked by \( L_2 \) are determined. The identified candidates of the update methods are verified by comparing game object names and the names of the variables declared in the statements in the update methods. The game objects names are displayed in the player selection screen in the play time, and referred in game introduction guides and help manuals for players. This step does not exhaustively identify all of the update methods.

Step 2.2 Categorize them by implementation type
This step categorizes the identified update methods into one of three update method implementation types: \( U_1 \), \( U_2 \), and \( U_3 \). If the identified update methods cannot be categorized into one of these three, then a new type is defined.

### 4.2.2 Study 2

Study 2 conducts a semi-structured interview [39] with commercial games developers. The interview consists of the following free-answer questions based on the results of Study 1.

- Q1: Are the results of Study 1 common and realistic in commercial games software?
- Q2: What kinds of update methods implementations exist in commercial games software?

### 4.3 Results

#### 4.3.1 Study 1

Step 1.1 yielded over 150,000 repositories from GitHub as of April 2018. Step 1.2 extracted 346 as popular repositories, where popular means 364 or more (364+) stars. Excluding the 123,865 unstarred repositories, those with 364+ stars are approximately top 1.5% of those obtained in Step 1.1. Figure 9 shows the distributions of the number of repositories obtained by the keyword search and number of stars. The maximum number of stars was 40,702.

Table 1 shows the categories of the repositories with 364+ stars. Step 1.3 reduced the number of repositories to 54. The titles of the repositories are shown in Table A-1 in Appendix A. Table 2 shows the distribution of the update method implementation types and programming languages. Figure A-1 in Appendix A shows the distributions of the number of target repositories and the number of stars. One programming language was used in all repositories except one, which used M language and PL/I and is denoted as “M and PL/I” in Table 2. The percentages of the repositories categorized by implementation types were 72.2% (\( U_1 \)), 11.1% (\( U_2 \)), and 16.7% (others). No repository was categorized into type \( U_3 \).

The repositories categorized into others were as follows. Seven repositories such as quiz games and puzzle games had no update methods (no game objects). Two repositories were under development and update methods had yet to be implemented.

**Table 1. Selected popular repositories**

| Category        | Number of repositories |
|-----------------|------------------------|
| Games           | 54                     |
| Non-game        | 48                     |
| Libraries       | 48                     |
| Game engines    | 46                     |
| Sample codes    | 42                     |
| Frameworks      | 28                     |
| Others          | 128                    |

**Table 2. Distribution of implementation types and programming languages**

| Programming language | Implementation type |
|----------------------|---------------------|
|                      | \( U_1 \) | \( U_2 \) | \( U_3 \) | Other |
| C++                  | 11        | 0         | 0        | N/A   | 1     |
| JavaScript           | 9         | 0         | 0        | N/A   | 2     |
| C                    | 3         | N/A       | 5         | N/A   | 0     |
| Objective-C          | 3         | 0         | 0        | N/A   | 1     |
| Java                 | 3         | 0         | N/A       | N/A   | 0     |
| Python               | 2         | 0         | 0         | 0     | 1     |
| C#                   | 2         | 0         | 0         | 0     | 0     |
| Go                   | 2         | 0         | N/A       | N/A   | 0     |
| Dart                 | 0         | 1         | 0         | 0     | 0     |
| Kotlin               | 1         | 0         | 0         | N/A   | 0     |
| Lua                  | 0         | N/A       | 0         | 0     | 1     |
| M and PL/I           | 1         | N/A       | N/A       | N/A   | 0     |
| OCaml                | 1         | 0         | N/A       | N/A   | 0     |
| Ruby                 | 0         | 0         | 0         | 0     | 1     |
| Rust                 | 0         | N/A       | 0         | N/A   | 1     |
| Swift                | 0         | 0         | 0         | N/A   | 1     |
| Shell Script         | 1         | N/A       | N/A       | N/A   | 0     |
| Total                | 39        | 1         | 5         | 0     | 9     |
4.3.2 Study 2
We conducted semi-structured interviews with 11 games developers who worked at commercial games software companies. The interviews were conducted in August 2018 at a co-located event for developers with one of the biggest Japanese games conferences.

4.3.2.1 Q1: Distribution of implementation types
Table 3 shows the results for Q1. Eight interviewees answered that the distributions were similar and realistic to commercial games software developments. The interviewees gave the following reasons for their answers. Two indicated that game loops, including update methods, were reused without changes. One interviewee responded that elder developers were unfamiliar with the State pattern and coroutines. One interviewee answered that implementation type \( U_1 \) and \( U_{2.2} \) provided the largest run-time performance because types \( U_{2.1} \) and \( U_5 \) have run-time overheads. One interviewee denoted that the readability of type \( U_1 \) was higher than the others. One interviewee answered that maintainability was a lower priority compared to earlier implementation for increasing duration for play-test feedback during the development.

Three interviewees answered that the distributions in Study 1 differed from their commercial games software. One interviewee claimed that the repositories in Study 1 might be biased and suggested using another criterion to select appropriate repositories. All of the three interviewees indicated that there were other implementations and combinations. These other implementations and combinations are described in the next subsection.

4.3.2.2 Q2: Other implementations for update methods
Two interviewees gave specific implementations besides the four types in Study 1. One was behavior trees [42][43][44]. Behavior trees are mainly used to implement AI (Artificial Intelligence) like features in game objects’ behaviors. The other was implementing a finite state machine using the library provided by a programming language.

One interviewee answered that different implementation types were selected for different game objects in one game. For example, game objects with a smaller number of states and smaller lines of statements for their update methods are implemented as type \( U_1 \), while game objects with a larger number of states and lines of statements are implemented as type \( U_2 \) to increase maintainability.

| Answer | Number of interviewees |
|--------|------------------------|
| Agree  | 8                      |
| (similar to commercial games) | |
| Disagree | 3                   |
| (different from commercial games) | |

5. Discussion
5.1 Results of Studies 1 and 2
Study 1 revealed that over 70% of the repositories use type \( U_1 \). Although type \( U_1 \) is a typical implementation for procedural programming languages, 34 repositories categorized into type \( U_1 \) were written in object-oriented programming languages. In other words, 35 repositories written in object-oriented programming language could use type \( U_{2.1} \), only one repository was categorized as type \( U_{2.1} \), which were more robust source code against changes and evolutions. Additionally, 14 repositories were written in programming languages supporting coroutines, however, none are categorized as type \( U_3 \). Among the repositories in Study 1, programming languages supporting coroutines are C# [45], Dart [46], JavaScript [47], Lua [48], Python [49], and Ruby [50].

The results of Study 2 indicated that developers recognized low maintainability of type \( U_1 \) and selected type \( U_1 \) for a higher runtime performance and continuous improvements during the development by early implementation and feedback. These result were consistent with the previous studies reporting the importance of runtime performance [8][9] and play-test feedback by early implementation [5][6][7]. Contrary, the results implied that the studies investigating object-oriented design and programming with runtime-performance-aware rules such as [14] and [15] were not shared among the interviewees. Such approaches should be shared and studied among games software developers.

The results of Study 2 indicate both positive and negative opinions on readability and maintainability for type \( U_1 \). Some of developers familiar with type \( U_1 \) seem to think that readability and maintainability for type \( U_1 \) are higher than other types of implementation. On the other hand, other developers gave a lower priority to maintainability due to the limited opportunities as well as the limited duration of maintenance and evolution, as in common with other studies [8][20].

Two developers indicated that reusing existing games source codes make copies of the game loop and update methods. After the interviews, we checked their assertion, and found two sets of reused repositories in Study 1. The types of update methods are the same among the original and reused repositories. One of the sets is DOOM for DOS (original) and two copies (DOOM for iOS and DOOM for PCs). The other set is Keen Dreams on Steam Greenlight and Wolfenstein 3D. Both games reuse the original game Keen Dreams. Keen Dreams is publicly available but not included in GitHub. The update methods in the original and copies are implemented using type \( U_{2.2} \). Increasing the number of the originals that implement the update methods using type \( U_2 \) or \( U_3 \) should lead to a wider use of the types.

Deploying a game engine decouples a game loop implementation and implementations for game objects and improves the maintainability of implementations for game objects. In the investigation, we surveyed the repositories that implemented their own game loops. In a game engine that follows component-oriented architecture, each game object can be implemented as a component. The game engine defines the component including interface definition of game objects. The interface definition includes update methods. Many games software stored in GitHub are implemented with

Table 3. Results for Study 2
general game engines such as Unity and Unreal Engine\textsuperscript{1}. Although the implementation types of these games software are similar to type $U_2$ in terms of the decoupling and the maintainability, strict implementation types should be defined for games software implemented with general game engines. Investigating the maintainability considering parallel processing of such software is one of the important future works.

As shown in Table A-1 and Figure A-2, the number of committers varied from 0 to 965. The number of repositories with 50 or more committers was 11. Implementation types of the update methods in the repositories were Type $U_1$ (10 repositories) and Other (1 repository). The ten repositories needed more maintainability because many committers changed the repositories. On the other hand, repositories with a smaller number of committers did not need high maintainability because fewer committers are expected to reach consensus on a low-cost implementation even with lower maintainability easily. To investigate the relationships between lines of code per one contributor and other factors is one of future works.

The latest games, especially large-scale ones, require parallel processing. The game loops described in Section 3 and investigated in Study 1 did not assume that update methods were invoked parallely. Although one repository identified in Study 1 used Rust programming language [51], which supports parallel processing, the update methods in the repository were not implemented yet. Additionally, architectures supporting parallel processing such as State-Effect pattern [13] are used for the latest games. Such architectures and programming languages not only support parallel processing but also improve maintainability. Investigating the architectures and programming languages is an important future work to clarify the state of the art of games software.

5.2 Threats to validity

Although update method implementation types can differ among game objects in a single game software, Study 1 did not investigate game objects exhaustively. An interviewee also pointed out the combinations of the types in Study 2. However, Study 1 checked as many update methods for game object types as possible and prioritized major game objects, which may have many states and require higher maintainability. We believe that the coverage of the game objects in each repository is wide enough to capture potential combinations of different implementation types in one repository. Investigating update methods for game objects exhaustively and comparing them is one of the important future works.

The target repositories in Study 1 were selected by searching for the keyword “game loop” and the number of stars. Further studies are required to consider other keywords and other selection criteria such as the size of the software and game categories, the number of committers, and development process. Additionally, eighty percent of the target repositories were created from 2010 to 2019, comparing them with older repositories leads to clarifying changes among evolutions. Moreover the repositories in Study 1 are limited to GitHub. Although GitHub is one of the largest public repositories, investigations on other repositories including commercial games are important future works.

Interviewees in Study 2 were attendees of a developer event co-located at a games conference. The answers may be biased because interviewees may be more concerned with software engineering principles. However, in the interviews, we asked the interviewees to answer the questions from the viewpoints of not only highly concerned developers but also ordinary game developers in their game development companies.

6. Conclusions

This paper investigates the extent that software engineering principles are employed in computer games software with a focus on the maintainability of update methods implementing game objects’ behaviors. The investigation qualitatively captures software engineering principles employment in terms of maintainability because many games implementations have update methods and game loops. Game loops are fundamental and common architectures among computer games. Update methods implement game objects’ behaviors. Update methods vary from implementations using traditional conditional branches and state variables without encapsulation (low maintainability) to those decoupling from the game loop and encapsulating the state variable and statements for each game objects’ behaviors (high maintainability).

The investigation follows the explanatory sequential design, which consists of a quantitative study followed by a qualitative study that uses the results of the quantitative study. Study 1, as the quantitative study, surveyed update methods in popular repositories in GitHub. Study 2, as the qualitative study, conducted semi-structured interviews with commercial games developers to ask whether the results of Study 1 are common and realistic in commercial games software.

The results of Study 1 show that update methods are implemented with conditional branches with state variables (low maintainability) in 39 repositories and the State design pattern (high maintainability) in one repository. In Study 2, eight commercial games software developers answered that the results of Study 1 are similar to and realistic in commercial games software. As one of the reasons for selecting the low maintainability implementation type, three developers indicated that maintainability had a lower priority than early implementation for play-test feedback and higher runtime performance. Some developers pointed out that reusing game loops and update methods generate copies of the same implementation. This study not only confirms that the context of the games software differs from that of other software, but also reveals that studies including hardware-limitation-aware coding rules for object-oriented programming languages are not known to interviewees. More awareness of software engineering principles for games software developers and more discussions on software engineering principles and practices that consider the difference between the contexts are required.

\textsuperscript{1} We searched GitHub repositories with a keyword “ProjectSettings.asset”. The keyword is the file name used by Unity. The number of search results was over 60,000.
7. Acknowledgements
The authors would like to thank the interviewees in Study 2. This work is supported by the JSPS KAKENHI Grant 17K00102.

Bibliography
[1] Callele, D., Neufeld, E., Schneider, K. (2005) Requirements Engineering and the Creative Process in the Video Game Industry, In Proc. of 13th IEEE International Conference on Requirements Engineering, pp. 240-250, https://doi.org/10.1109/RE.2005.58
[2] Yanamakikis, N. G., Hallam, J. (2007) Capturing Player Enjoyment in Computer Games, Advanced Intelligent Paradigms in Computer Games, Studies in Computational Intelligence, vol. 71, pp. 175-201, https://doi.org/10.1007/978-3-540-72705-7_8
[3] Stacey, P., Nandhakumar, J. (2009) A Temporal Perspective of the Computer Game Development Process, Information Systems Journal, vol.19, no.5, pp. 479-497, https://doi.org/10.1111/j.1365-2657.2007.00273.x
[4] Stacey, P., Nandhakumar, J. (2008) Opening Up to Agile Games Development, Communications of the ACM, vol. 51, no. 12, pp. 143-146, https://doi.org/10.1145/1409360.1409387
[5] Kasurinen, J., Strandén, J., Smolander, K. (2013) What Do Game Developers Expect from Development and Design Tools?, In Proc. of the 17th International Conference on Evaluation and Assessment in Software Engineering, pp. 36-41, https://doi.org/10.1109/ISEE.2013.665399
[6] Petrillo, F., Pimenta, M., Trindade, F., Dietrich, C. (2009) What Went Wrong? A Survey of Problems in Game Development. Computer Entertainment vol. 7, no. 1-13, pp. 1-22, https://doi.org/10.1145/1486508.1486521
[7] Alves, C., Ramalho, G., Darnescano, A. (2007) Challenges in Requirements Engineering for Mobile Games Development: The Meantime Case Study, In Proc. of 15th IEEE International Requirements Engineering Conference, pp. 275-280, https://doi.org/10.1109/RE.2007.53
[8] Murphy-Hill, E., Zimmermann, T., Ngapapan, N. (2014) Cowboys, Ankle Sprains, and Keepers of Quality: How is Video Game Development Different from Software Development?, In Proc. of the 36th International Conference on Software Engineering, pp. 1-11, https://doi.org/10.1145/2568225.2568226
[9] Bishop, L., Eberly, D., Whitted, T., Finch, M., Shantz, M. (1998) Designing a PC Game Engine, IEEE Computer Graphics and Applications, pp. 46-53, https://doi.org/10.1109/38.637270
[10] Blow, J. (2004) Game Development: Harder Than You Think, ACM Queue vol. 1, no. 10, pp. 28-37, https://doi.org/10.1145/971564.971590
[11] Nguyen, D., Wong, B. S. (2002) Design Patterns for Games, In Proc. of the 33rd SIGCSE Technical Symposium on Computer Science Education, pp. 126-130, https://doi.org/10.1145/563340.563387
[12] Wetzel, R. (2013) A Case for Design Patterns Supporting the Development of Mobile Mixed Reality Games. Second Workshop on Design Patterns in Games, 8 pages
[13] White, W., Koch, C., Gehrke, J., Demers, A. (2008) Better Scripts, Better Games. ACM Queue vol. 6, no. 7, pp. 18-25, https://doi.org/10.1145/1458101.1483106
[14] Zhang, W., Han, D., Kunz, T., Hansen, M. K. (2007) Mobile Game Development: Object-Orientation or Not, In Proc. of 31st Annual International Computer Software and Applications Conference, pp. 601-608, https://doi.org/10.1109/COMPSAC.2007.151
[15] Aampatagloog, A., Chatzigeorgiou, A. (2007) Evaluation of Object-oriented Design Patterns in Game Development, Information and Software Technology, vol. 49, no. 5, pp. 445-454, https://doi.org/10.1016/j.infsof.2006.07.003
[16] Cunningham, W. (1992) The WyCash Portfolio Management System, In Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications, pp. 29-30, https://doi.org/10.1109/ICSE.1992.278715
[17] Nystrom, G. (2014) Game Programming Patterns, Genever Publishing
[18] Madhav, S. (2013) Game Programming Algorithms and Techniques: A Platform-Agnostic Approach, 1 edition. Addison-Wesley Professional
[19] Cresswell, W. J. (2014) A Concise Introduction to Mixed Methods Research, SAGE Publications, Inc.
[20] Pascarella, L., Palomba, F., Penta, D. M. (2018) A. Bacchelli, How Is Video Game Development Different from Software Development in Open Source?, In Proceedings of the 15th International Conference on Mining Software Repositories, pp. 392-402, https://doi.org/10.1109/MSR.2018.25
[21] Kultima, A., Alha, K. (2010) "Hopefully Everything I'm Doing Has to Do with Innovation": Games Industry Professionals on Innovation in 2009, the 2nd International IEEE Consumer Electronics Society's Games Innovations Conference, pp. 1-8, https://doi.org/10.1109/ICEGIC.2010.5716899
[22] Scacchi, W. (2011) Modding as an Open Source Approach to Extending Computer Game Systems, IFIP Advances in Information and Communication Technology, vol. 365, pp. 62-74, https://doi.org/10.1109/798-3-642-24418-6_5
[23] Poretski, L., Arazy, O. (2017) Placing Value on Community Co-creations: A Study of a Video Game 'Modding' Community. In Proc. of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing, pp. 480-491, https://doi.org/10.1145/2998181.2998301
[24] Lin, D., Bezemer, C., Hassan, A. E. (2017) Studying the Urgent Updates of Popular Games on the Steam Platform, Journal of Empirical Software Engineering, vol. 22, no. 4, pp. 2095-2126, https://doi.org/10.1007/s10539-016-0481-0
[25] Lewis, C., Whitehead, J., Wardrip-Fruin, N. (2010) What Went Wrong: A Taxonomy of Video Game Bugs. In Proc. of the Fifth International Conference on the Foundations of Digital Games, pp. 108-115, https://doi.org/10.1145/1822348.1822363
[26] Nascimento, M. L., Almeida, S. E., Meira, R. S., (2008) A Case Study in Software Product Lines - The Case of the Mobile Game Domain, In Proc. of 34th Euromicro Conference Software Engineering and Advanced Applications, pp. 43-50, https://doi.org/10.1109/SEAA.2008.14
[27] Farini, M. (2008) An Architecture to Easily Produce Adventure and Movie Games for the Mobile Scenario, Computer Entertainment, vol. 6, no. 2-19, pp. 1-16, https://doi.org/10.1145/1371216.1371222
[28] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994) Design Patterns: Elements of Reusable Object-Oriented Software 1st Edition, Addison-Wesley Professional
[29] Björk, S., Lundgren, S., Holopainen, J. (2003) Game Design Patterns, In Proc. of Digital Games Research Conference 2003, 1 edition
[30] Cho, H., Yang, J. (2008) Architecture Patterns for Mobile Games Product Lines, In Proc. of 2008 10th International Conference on Advanced Communication Technology, pp. 118-122, https://doi.org/10.1109/ICACT.2008.4493725
[31] Folmer, E. (2007) Component Based Game Development – A Solution to Escalating Costs and Expanding Deadlines?, In Proc. of
International Symposium on Component-Based Software Engineering, pp. 66-73

[32] Cutumisu, M., Onuczko, C., McNaughton, M., Roy, T., Schaeffer, J. Schumacher, A., Siegel, J., Szafrron, D., Waugh, K., Carbonaro, M., Duff, H., Gillis, S. (2007) ScriptEase: A Generative/adaptive Programming Paradigm for Game Scripting, Science of Computer Programming, vol. 67, no. 1, pp. 32-58, https://doi.org/10.1016/j.scico.2007.01.005

[33] White, W., Sowell, B., Gehrke, J., Demers, A. (2008) Declarative Programming for Computer Games. In Proc. of the 2008 ACM SIGGRAPH symposium on Video games pp. 23-30, https://doi.org/10.1145/1401843.1401847

[34] Passos, B. E., Sousa, S. W. J., Clua, G. W. E., Montenegro, A., Murta, L. (2010) Smart Composition of Game Objects Using Dependency Injection, Computer Entertainment, vol. 7, no. 4, article 53, 15 pages, https://doi.org/10.1145/1658866.1658872

[35] Ebner, M., Levine, J., Lucas, M. S., Schaul, T., Thompson, T., Togelius, J. (2013) Towards a Video Game Description Language, Dagstuhl Follow-Ups, 6, Artificial and Computational Intelligence in Games, pp.85-100, http://doi.org/10.4230/DFU.Vol6.12191.85

[36] Rucker, R. (2002) Software Engineering and Computer Games, 1 edition, Addison-Wesley

[37] Valente, L., Conci, A., Feijó, B. (2005) Real Time Game Loop Models for Single-Player Computer Games, Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment, pp.89-99

[38] Creswell, W. J., Plano Clark, L. V., Designing and Conducting Mixed Methods Research, SAGE Publications, Inc. (2006)

[39] Bloor, M., Wood, F. (2006) Keywords in Qualitative Methods, 1 edition, SAGE Publications Ltd. (accessed on 2018.12.7)

[40] GitHub Help "About stars - User Documentation", https://github.com/chalin/dart-spec-and-grammar/blob/master/doc/dartLangSpec.pdf (accessed on 2018.12.7)

[41] Nicolau, M., Perez-Liebana, D., O’Neill, M., Brabazon, A. (2017) Evolutionary Behavior Tree Approaches for Navigating Platform Games, in IEEE Transactions on Computational Intelligence and AI in Games, vol. 9, no. 3, pp. 227-238, https://doi.org/10.1109/TCIAIG.2016.2543661

[42] MDN web docs "yield – JavaScript | MDN", https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield (accessed on 2018.12.7)

[43] Python 3.7.1 documentation "3. Data model — Python 3.7.1 documentation", https://docs.python.org/3/reference/datamodel.html (accessed on 2018.12.7)

Table A-1. Repositories obtained

| Title              | Number of | Implementation | Programming | Created on | Released in | Number of commits | Number of contributors | Lines of code |
|--------------------|-----------|
| BrowswerQuest      | 109       | C++           | JavaScript  | 2011.09.30 | 153        | 4                | 68445           |
| OpenRCT2           | 5627      | C++           | C++        | 2014.04.01 | 15088      | 283               | 73626            |
| OpenSC2K           | 4185      | C++           | C++        | 2018.02.03 | 36         | 2                | 31842            |
| Doorn 3            | 3750      | C++           | C++        | 2011.11.22 | 40         | 4                | 9178             |
| Minecraft          | 3623      | C++           | Python     | 2012.01.30 | 59         | 11               | 69               |
| Untrusted          | 3446      | C++           | JavaScript | 2013.03.09 | 1105       | 86               | 218963           |
| Doorn 3: BFG Edition | 2994   | C++           | C++        | 2012.11.26 | 2          | 2                | 12273            |
| DOOM               | 2941      | C++           | C++        | 2012.01.31 | 4          | 1                | 630              |
| Quake III Arena    | 2539      | C++           | C++        | 2012.01.31 | 1          | 1                | 5576             |
| SpaceEngineers     | 2496      | C++           | C++        | 2015.05.13 | 402        | 35               | 483696           |
| Terminology        | 2311      | C++           | Java       | 2011.03.04 | 8661       | 231              | 281585           |
| agar.io Clone      | 2096      | C++           | JavaScript | 2015.05.22 | 710        | 55               | 1199             |
| Cataclysm: Dark Days Ahead | 1782 | C++ | JavaScript | 2012.09.26 | 56729      | 965              | 1134694          |
| Keen Dreams        | 1646      | C++           | C++        | 2014.09.16 | 8          | 1                | 439              |
| Carnabth           | 1554      | Objective-C   | C++        | 2010.12.30 | 4          | 1                | 12098            |
| SSHlron            | 1493      | C++           | C++        | 2016.03.06 | 112        | 9                | 451              |
| Super Mario Bros Level 1  | 1309   | C++           | Python     | 2014.01.05 | 151        | 3                | 14962            |
| AR Tetriss         | 1278      | C++           | Swift      | 2017.06.30 | 24         | 3                | 28               |
| 0 A.D.             | 1078      | C++           | C++        | 2010.10.23 | 21701      | 6                | 3502765          |

 Appendix

A. Repositories obtained

Table A-1 shows the list of the obtained repositories. The column “Created on” indicates the date when the repository was created. The column “Released in” is determined as the following. If the released year described in the official site is older than the column “Created on”, the column “Released in” is the released year. Otherwise, the column is left blank.
| Title                                      | Repository | Language     | Download | Stars | Forks |
|-------------------------------------------|------------|--------------|----------|-------|-------|
| Pocket Island                             | 1030       | JavaScript   | 2012.04.25 | 67    | 21388 |
| Hauberk                                   | 905        | Dart         | 2014.07.07 | 697   | 2554  |
| Wolfenstein 3D                            | 854        | C            | 2012.02.06 | 1992  | 2084  |
| Tiny Wings Remake                         | 831        | Objective-C  | 2011.06.08 | 40    | 317   |
| VCMI                                      | 787        | C++          | 2014.04.06 | 7145  | 35854 |
| Dungeon Crawl Stone Soup                  | 777        | C++          | 2014.08.17 | 56494 | 176483|
| StepMania                                 | 766        | Other        | 2013.06.24 | 36101 | 521315|
| Warzone 2100                              | 739        | C++          | 2010.11.16 | 14359 | 618216|
| term2048                                  | 726        | Other        | 2014.03.13 | 223   | 150   |
| Nothing To Hide                           | 722        | JavaScript   | 2013.11.27 | 288   | 83761 |
| Much Assembly Required                    | 655        | Java         | 2017.11.03 | 267   | 900   |
| Scalenation                               | 589        | JavaScript   | 2012.04.22 | 205   | 7024  |
| Wizard War                                | 563        | Objective-C  | 2013.05.17 | 590   | 89499 |
| Space Invaders in Go                      | 543        | Go           | 2018.02.01 | 5     | 4006  |
| piu-piu-SH                                | 528        | Shell script | 2016.11.17 | 241   | 210   |
| Javascript Pseudo 3D Racer                | 505        | JavaScript   | 2012.06.15 | 14    | 16380 |
| Angband                                   | 495        | C            | 2010.10.29 | 7821  | 63575 |
| DOOM Classic iOS                          | 494        | C            | 2012.01.31 | 2009  | 11664 |
| Lemma                                     | 459        | C#           | 2012.11.06 | 1090  | 1278255|
| Chocolate Doom                            | 450        | C            | 2009.06.18 | 4021  | 17196 |
| Pirates!                                  | 449        | JavaScript   | 2017.08.27 | 29    | 106   |
| Kick Ass - Destroy the Web                | 441        | Other        | 2010.04.11 | 114   | 454   |
| The git committer guessing game           | 441        | Ruby         | 2015.02.08 | 37    | 243   |
| Chocolate Duke Nukem 3D                   | 431        | C            | 2012.12.11 | 193   | 1746  |
| Pit Noon                                  | 428        | C++          | 2014.11.04 | 886   | 309983 |
| MurtiOCaml                                | 426        | OCaml        | 2015.11.02 | 187   | 6655  |
| iPokeMon                                  | 420        | Objective-C  | 2013.01.13 | 1240  | 30421 |
| Q+                                       | 415        | Rust         | 2013.04.01 | 395   | 17088 |
| Habitat                                   | 408        | M and PL/I   | 2016.07.06 | 1985  | 5201  |
| KotCity                                   | 402        | Kotlin       | 2018.03.05 | 617   | 9746  |
| termux                                    | 397        | Other        | 2014.09.23 | 108   | 742   |
| Multiplayer HTML5 Osmos                   | 392        | JavaScript   | 2011.08.02 | 50    | 4653  |
| GZDoom                                    | 385        | C++          | 2013.06.23 | 14787 | 72226 |
| DECEIVER                                  | 378        | C++          | 2015.07.23 | 1398  | 353435|
| opsu!                                     | 366        | Java         | 2014.06.30 | 944   | 127959|

Figure A-1. Histogram of the number of target repositories and the number of stars

Figure A-2. Histogram of the number of target repositories and the number of contributors