A Game Framework Supporting Automatic Functional Testing for Games

Woei-Kae Chen, Chien-Hung Liu, Ping-Hung Chen and Chia-Sheng Hsu

Abstract When developing a computer (mobile) game, testing the game is an important task and takes a large share of the development cost. So far, testing a game's functional features relies mainly on human testers, who personally plays the game, and is thus labor intensive. This paper proposes a method that automates game testing. Since games are usually built on top of game frameworks, the idea is to enhance a game framework with a testing layer, which can execute (playback) test scripts that perform user events and assert the correctness of the game. We design an HTML5 game framework with such a support. In addition, a case study is performed to compare the testing cost of three different methods: writing a test script directly, recording a test script, and testing the game directly by a human tester. The results showed that when repeated testings are necessary, an automatic testing (either writing or recording test scripts) can reduce human cost. Among these three testing methods, recording scripts was the most favored method.

1 Introduction

Software testing plays an important role in software development. Computer (mobile) games, like any other software applications, require thorough testings to ensure their quality, and the testings take a large share of development cost.

W.-K. Chen (✉) · C.-H. Liu · P.-H. Chen · C.-S. Hsu
Department of Computer Science and Information Engineering, National Taipei University of Technology, 1, Sec. 3, Zhongxiao E. Rd., Taipei 10608, Taiwan, Republic of China
e-mail: wkchen@ntut.edu.tw

C.-H. Liu
e-mail: cliu@ntut.edu.tw

P.-H. Chen
e-mail: t100599001@ntut.edu.tw

C.-S. Hsu
e-mail: t101598062@ntut.edu.tw

© Springer International Publishing Switzerland 2016

R. Lee (ed.), *Applied Computing & Information Technology*,
Studies in Computational Intelligence 619,
DOI 10.1007/978-3-319-26396-0_2

For applications with a graphical user interface (GUI), there are testing tools (e.g., [1–5]) that allows a tester to automate the testing of an application's functional features. However, so far, there are no such tools available for games.

Before addressing the proposed game-testing method, we will explain the automatic testing for GUI applications first. A GUI application is usually created by using standard widgets (e.g., Button, TextBox, ComboBox, etc.). Each window (or page) in an application consists of a number of widgets, which interact with the user and also serve as the display, showing important information to the user. These widgets can be made *accessible* (by their vendor) at runtime. That is, when an application is running, a widget on a screen (window) can be searched, queried, and controlled by another test program (or test script). Thus, a test script can perform testing by directing the target application's widgets to perform certain operations and assertions. For example, Microsoft designed Windows controls (widgets) to be accessible and allowed *coded user interface tests* [4]. Similarly, Google provides *uiautomator* [5], which allows testers to create automated test cases for the user interface of Android Apps.

By utilizing the accessibility features of standard widgets, many testing tools (e.g., Robot framework [1], Unified Functional Testing [2], and Selenium [3]) make automated functional testing more efficient. Basically, a tester uses a tool's IDE to create (develop) a *test script*, which is composed of sequences of actions [6, 7]. An action is either an event, which drives the GUI, or an assertion, which affirms the correctness of the application. The test script can be automatically executed by the tool to verify whether the application works correctly. To speedup the development of test scripts, some testing tools (e.g., [2, 3]) also support capture/replay capability, i.e., user interactions to the application can also be captured as test scripts and can be replayed by the tool.

Games are in a sense like GUI applications in that both of them provide interactive visual experience to the users. However, unlike GUI applications, when automating game testing, two major issues arise: (1) accessibility—games do not share a standard set of widgets (visual objects); and (2) timing—for games, timing can be so critical that a slight delay (e.g., 1 ms) may change every thing in the game. A visual object in a game is maybe a man, a monster, a brick, a map, etc., each with completely different shapes, sizes, and user operations (events). Since each game creates its own specialized visual objects, when running a game, there is not a standard way that can access these objects from the test script. Moreover, even if the visual objects can be made accessible, a test script must drive the objects (e.g., sending an event) in precisely the right time. This is difficult to achieve because a slight fluctuation in CPU utilization may alter the execution speed of both the game and test script. When timing is not exactly right, the test script may not always reproduce the same results, which is unacceptable. Therefore, creating a generic game testing tool that can support all different games is highly difficult. Consequently, there are no such tools available for game testing.

Without tools, so far, testing a game's functional features relies mainly on human testers, who personally plays the game and determine whether the game works correctly. This is labor intensive particularly when repetitive testing is

necessary. For example, when a developer modifies some source code, a testing that covers all levels (stages) of the game is usually necessary to ensure that there are no regressions.

How to resolve both the accessibility and timing issues? For accessibility, when a game is running, the visual objects of the game must be made accessible so that a test script can drive the game and perform testing. For timing, the test script must be able to perfectly control (synchronize with) the execution of the game so that every test run produces exactly the same results. These issues suggest that, the test script must be run tightly coupled with the game. Note that, instead of creating games from scratch, game developers usually build games on top of game frameworks or engines (e.g., Cocos2D [8] and Unity [9]). Therefore, the idea of this paper is to enhance a game framework with a testing layer, which cooperates with the test scripts to offer accessibility and precise timing control. This is possible because a game framework knows (handles) every visual objects in the game and controls game cycles (a cycle of updating and drawing game objects). Thus, providing such supports is feasible.

We designed an HTML5 game framework that supports automatic functional testing. A game developer can use this HTML5 game framework to develop any games. JavaScript is the programming language of use and the game runs on a browser supporting HTML5. A game developed based on this framework is automatically testable. A tester (sometimes a developer himself) can create a test script either by writing the script directly or by playing the game and capturing the game-play actions. The tester gets the following testing features:

- Test script: a tester can write a test script directly by using a text editor. The test script itself is also written in JavaScript.
- Event: a test script can perform any keyboard or mouse events, and can also request to execute any methods of any objects (e.g., jumping to a special game level or state).
- Assertion: a test script can retrieve any attribute from any objects and assert if the value of the attribute is as expected.
- Capture and replay: in capture mode, user actions can be captured and translated into a test script. A captured test script does not contain assertions. The tester should in general add additional assertions into the test script.
- Timing: timing control is based on game cycle. Each event or assertion is executed in exactly the specified game cycle and is perfectly in-sync with the game. The execution results will always be the same, even if the game runs on a different computer with a different speed.
- Unit testing: QUnit [10], a JavaScript unit testing framework, is incorporated to manage test execution. Each test script is run as a QUnit test case, thereby integrating both unit testing and functional testing in the same framework.

A case study is performed to compare the testing cost of three different methods: writing a test script directly, recording a test script, and testing the game directly by a human tester. The results showed that when repeated testings are necessary,

automatic testing (either writing or recording test scripts) can reduce human cost. Among these three testing methods, recording scripts was the most favored method.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents an HTML5 game framework that supports the proposed testing method. Section 4 reports a case study that compares automated and non-automated testing. A conclusion is given in Sect. 5.

2 Related Work

Software testing, an important part of software development, has been extensively studied in the literature. However, despite computer (mobile) games have gained a huge revenue and playing games becomes an important part of many people's daily life, there are only few researches that studied game testing. None of these researches addressed automatic functional testing. We will discuss each game testing research as follows.

Kasurinen and Smolander [11] analyzed how game developing organizations test their products. They interviewed seven game development teams from different companies and studied how they test their products. The results suggested that game developers focused on soft values such as game content or user experience, instead of more traditional objectives such as reliability or efficiency. Note that user experience testing is lot different from the functional testing studied in this paper. Both testings are important parts of creating a high quality game product. So far, both user experience and functional testing rely mainly on human testers. This paper proposes a way of automating functional testing so as to reduce the overall testing cost.

Zhao et al. [12] studied model-based testing for mobile games. They proposed a general rule model for mobile games by combining the features of the game software and the Wireless Application Protocol (WAP). They also built an adjusted model for generating test cases with the characteristics of mobile game software. A series of patterns were created to facilitate the generation of TTCN-3 test suite and test case. The limitation of this testing method is that it is suitable for games based on WAP only, and it can only test the communication between the game and WAP—functional testing is not supported.

Cho et al. [13, 14] proposed VENUS II system, which supported blackbox testing and scenario-based testing as well as simple load testing for game servers. Massive virtual clients could be created to generate packet loads and test the performance of game servers. Game grammar and game map were used to describe game logics. However, the focus of VENUS II system was the testing of game servers. In contrast, this paper addresses the testing of game clients.

Schaefer et al. [15] presented Crushinator framework, a game-independent framework that could test event-driven, client-server based game applications and automate processes by incorporating multiple testing methods such as load and performance testing, model-based testing, and exploratory testing. The Crushinator

allowed a tester to automate large numbers of virtual clients, utilizing different types of test methods, and to find server load limits, performance values, and functional defects. Like, VENUS II system, Crushinator was designed for the testing of game servers, not clients.

Diah et al. [16] studied usability testing for an educational computer game called Jelajah. They discussed an observation method designed for children. Five pre-school children aged between five and six years old were selected for their study. By collecting and analyzing data, the levels of effectiveness, efficiency and satisfaction were determined, which in turn gave an overall usability of Jelajah.

To automate functional testing, some game developers create test programs for their own games (e.g., [17]). This is feasible because, given a particular game, both the accessibility and timing issues (see Sect. 2) can be resolved by directly modifying the source code of the game. However, this is very expensive, game dependent, and the support cannot be extended to other games. In contrast, the testing support proposed in this paper is built-in into game frameworks. Therefore, game-independent functional testing can be achieved as long as the game is developed under the proposed game framework.

3 HTML5 Game Framework

This section describes the HTML5 Game Framework, called simply H5GF. H5GF was originally developed in 2013 as a game framework that supports the teaching of object-oriented programming [18]. In 2014, many students (a total of 7 different teams) have successfully used H5GF to create various types of games, indicating that H5GF is suitable for game development. In this paper, we enhance H5GF to support automatic functional testing. H5GF supports three different modes, namely *play* (playing the game), *replay* (running test scripts), and *record* (capturing the player's actions into test scripts) modes.

The architecture of H5GF is shown in Fig. 1. A game developer creates a game object (e.g., a race car) by subclassing `GameObject`. A game object typically uses either `Sprite`, `AnimationSprite`, or `Scene` to display the image of the object (note: an `AnimationSprite` or `Scene` is a composition of one or more

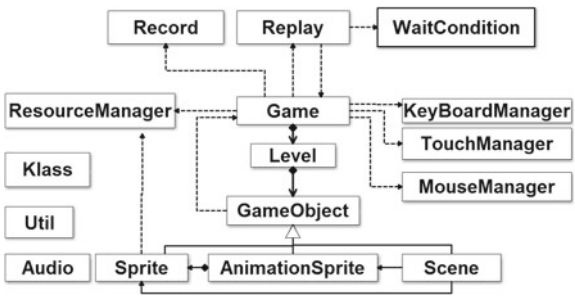


Fig. 1 H5GF class diagram

`Sprite` objects). A `Sprite` object can be rendered on the screen, after specifying its image location (file or URL) and screen position. A game typically has many levels (or stages) and each level uses a lot of different game objects. To create a level, the developer subclasses Level and stores its game objects inside the level object, and then develops the code that processes user events (e.g., keyboard and mouse events) and the code that updates the positions of game objects and draws them.

A game (or a level of a game) developed in H5GF has a life cycle shown in Fig. 2. First, the resources (images and audio clips) needed for showing the "loading progress" are loaded. Then, simultaneously, the game resources are loaded and the loading progress is shown. After that, the game objects in the game are initialized. Then the game enters the main game loop, periodically updating and drawing game objects. Finally, when the game ended, the resources are released (Teardown). A game is developed by providing the code (function) for each activity shown in Fig. 2. In particular, most game logic resides in Update and Draw functions, which control the behaviors of the game.

H5GF controls the transition from one activity to another. The game developer does not need to write any code for the transitions. In addition, as shown in Fig. 3, H5GF also accepts user events and controls *game cycles* (an update/draw cycle is call a game cycle). The game developer defines the game-cycle frequency by specifying a target FPS (frame per second). H5GF automatically calls the update and draw methods periodically, and calls the corresponding event-handling functions when events are triggered by the player.

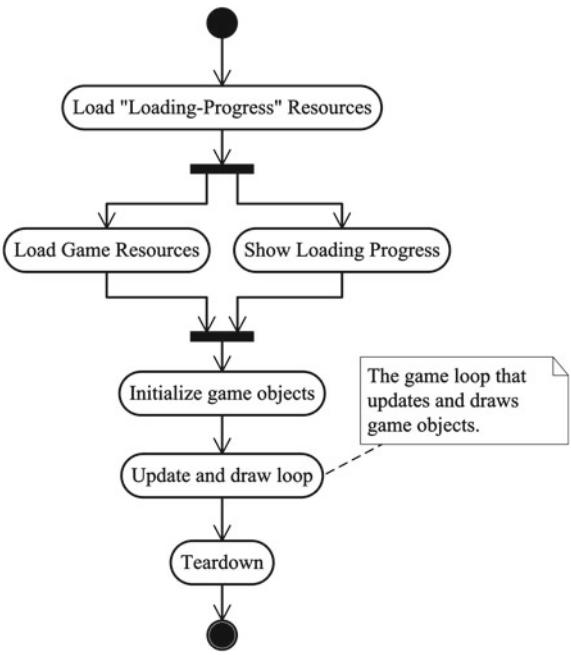


Fig. 2 The life cycle of a game (UML activity diagram)

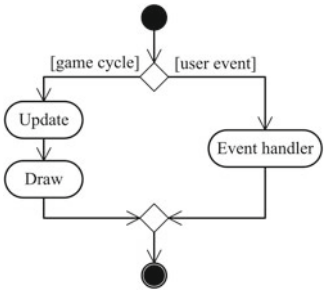


Fig. 3 Game cycle and event processing in play mode

We now describe the testing support of H5GF. The framework uses QUnit [10] to manage test cases and uses three classes, `Record`, `Replay`, and `WaitCondition` to support testing (Fig. 1). In replay (testing) mode, QUnit executes test cases and reports the results. Figure 4 shows a simple test script designed for a well-known game *Bomberman* (the goal of each game level is to

```
QUnit.module("Bomberman Test", {
  setup: function()
  {
    Framework.Replay.start();
  },
  teardown: function()
  {
    Framework.Replay.stop();
  }
});

var Walk = function(direction,times)
{
  for(var i=0;i<times;i++)
  {
    Framework.Replay.keyDown(direction);
    Framework.Replay.waitFor(
      new Framework.WaitCondition(
      "map.player1.isWalking",false));
    Framework.Replay.keyUp(direction);
  }
}

var PlaceBomb = function()
{
  Framework.Replay.keyDown("Space");
}

QUnit.asyncTest("Kill Monster", function( assert ) {
  Framework.Replay.goToLevel("level1");
  Walk('Down' 1);
  PlaceBomb();
  Walk('Up', 2);
  ...
  Framework.Replay.assertEqual(
      "map.monster[0].isdead", true);
  ...
});
```

Fig. 4 A simple test script

place bombs, destroy obstacles, and kill enemies). Lines 1–10 defines the setup and teardown functions, which starts and stops the execution of the game for each test case. Lines 29–38 defines a test case called "Kill monster." When the test case runs, the game jumps to level 1 (line 30), move the bomberman (line 31), place a bomb (line 32), etc. The functions Walk (line 12) and Placebomb (line 24) are reusable actions that can be used to create complicated testing scenarios.

We now explain how the timing issue (see Sect. 1) is resolved. The class `Replay` (see the Replay statements in Fig. 4) performs replay actions and controls timing. To achieve perfect synchronization, the test script and the game are in fact executed in the same thread. In other words, the game is completely frozen, when a test script statement is in execution (e.g., line 16). The only exception is a *waitFor* statement, which allows the game to continue running until a predefined condition is met (e.g., lines 17–19) or run for a specified number of game cycles (e.g., the statement Framework.Replay.waitFor(30) will allow the game to run for exactly 30 game cycles). Figure 5 shows how H5GF interleaves the execution of the test script and the game. In the beginning of each game cycle, H5GF takes an event (or statement) from the test script, and check if the event is a waitFor statement. If yes, H5GF allows the cycle to run until the waitFor condition ends. If not, the event handler is executed. Thus, the test script can precisely control the game without missing even a single cycle. This is crucial to game testing, since missing a cycle can change everything.

How is the accessibility issue resolved? Since both the game and the test script are developed under H5GF. Through H5GF, the test script can access all the game objects created by the game. For example, lines 35–36 in Fig. 4 assert that the first monster is dead by checking whether the attribute "isdead" of the object "map.-monster[0]" is true. Here, a distinctive JavaScript language feature, called *eval*, is

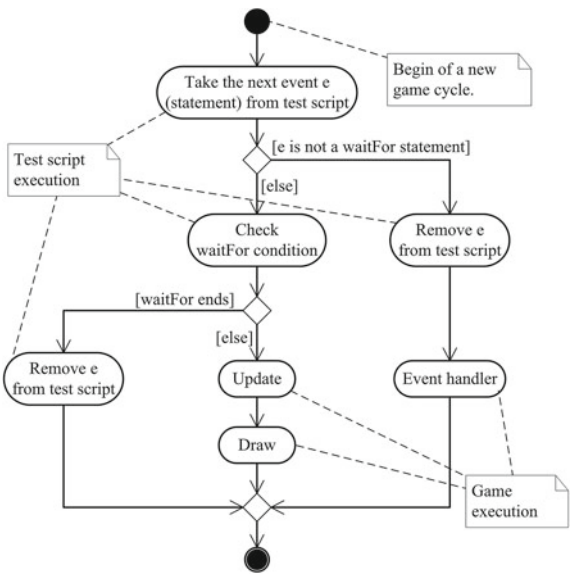


Fig. 5 Game cycle and event processing in replay mode

used to facilitate the access. An eval statement allows JavaScript to treat a string as a JavaScript statement and executes the statement. The assertEqual function (provided by H5GF) uses an eval statement to retrieve the value of "map.monster[0].isdead" and then performs the assertion.

Writing a test script directly can sometimes be cumbersome. Thus, H5GF also provides the ability of capturing (recording) a test script by playing the game. Figure 6 shows how user events are captured in record mode. When an event is triggered, the information of the event is stored and translated into a corresponding test script. The number of game cycles in between two consecutive events are also recorded and translated as a waitFor statement. Figure 7 shows a sample recorded

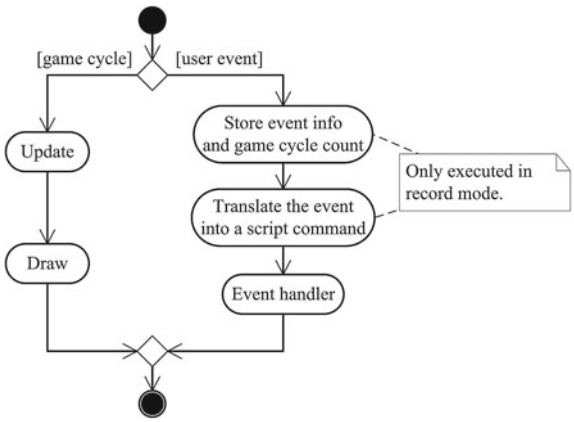


Fig. 6 Game cycle and event processing in record mode

```
QUnit.asyncTest("Recorded Script", function(assert) {
    Framework.Replay.waitFor(124);
    // Change Level :level1
    Framework.Replay.waitFor(21);
    Framework.Replay.mouseClick(457,273);
    Framework.Replay.waitFor(160);
    // Change Level :level2
    Framework.Replay.mouseClick(900,54);
    Framework.Replay.waitFor(282);
    // Change Level :level3
    Framework.Replay.waitFor(33);
    Framework.Replay.mouseClick(945,48);
    Framework.Replay.waitFor(375);
    Framework.Replay.mouseClick(237,524);
    Framework.Replay.waitFor(343);
    ...
    // The assertions are created by the tester
    Framework.Replay.assertEqual(
       'player[1].cash',24000,0);
    Framework.Replay.assertEqual(
       'player[1].point',0,0);
});
```

Fig. 7 A sample recorded test script



Fig. 8 Bomberman game

script in which mouse click events and waitFor statements are interleaved. Since a recorded script only drives the game, there is no testing at all. Thus, it is important to manually add assertions into the script so that the correctness of the game states can be ensured. Lines 18–21 are such examples. Note that, in comparison to the test script shown in Fig. 4, a recorded script is much more difficult to read. However, capturing a test script is much faster than writing a test script directly. Therefore, it is the tester's decision whether to use capturing function or not.

To verify that H5GF can be used to perform automatic functional testing, we developed a Bomberman game (Fig. 8) and created test scripts for Bomberman by both writing test scripts directly and by recording test scripts. We will report a case study based on Bomberman later in Sect. 4. In addition, we use H5GF to test three different games developed by 2014 students (a flying-shooting game, a monopoly game, and a block-removing game). The test scripts were created by recording player's actions and then enhanced manually with assertions. For all three games, H5GF were able to precisely replay the test script and perform assertions, indicating the support is game-independent.

4 Case Study

The proposed method supports automatic functional testing for games. A test script can be developed either by writing the test script directly or by capturing the test script and then adding assertions manually. Which one is better? In addition, to automate game testing, a tester needs to develop a test script first, which takes time. Therefore, in terms of testing cost (human time), automatic testings do not necessarily outperform human testers. This section reports a case study that compares the cost of the following three different testing methods, namely M1, M2, and M3:

M1 Test a game by writing a test script directly.
M2 Test a game by capturing a test script and then adding assertions manually.
M3 Test a game by a human tester who plays the game personally.

What about maintenance cost? When the contents of a game level is changed (e.g., the arrangement of a game map is changed), an existing test script that was designed for the level may become unusable and needs to be repaired. As described in Sect. 3, there is a trade-off between M1 and M2: using M2 is typically faster than M1, but an M1 script is typically easier to read than an M2 script. So, when taking maintenance (fixing a broken test script) cost into account, which method is better, M1 or M2? The answer to this question may depend on the scale of the change. For example a large-scale change (e.g., the entire game map is changed) may force the entire test script to be rewritten. In these case, it is reasonable to suspect that M2 is better. On the other hand, a small-scale change (e.g., only a small part of the map is changed) may favor M1 over M2, since only a small part of the test script needs modifications. Overall, considering the three testing methods and both their development and maintenance costs, the case study addresses the following research questions:

RQ1 How much cost (human time) do M1, M2, and M3 require?
RQ2 When the contents of a game level is changed, does the scale of the change influences the testing cost.
RQ3 For a large-scale content change in a game level, which method is the best?
RQ4 For a small-scale content change in a game level, which method is the best?
RQ5 Which method is the most favored method?

To answer the above research questions, we designed a case study based on Bomberman (Fig. 8), a game that the player places bombs, destroys obstacles, retrieve special (e.g., power-up) equipments, and kills enemies. Six graduate students were recruited to participate the case study and serve as testers.

Each participant is requested to perform two phases of testing. In the first phase, the participant is given a brand-new game level and requested to test the level thoroughly using all three different methods. The testing includes directing the bomberman to retrieve three different equipments and kill all six enemies to clear the level. For M1, the participant writes the test script step-by-step (moving bomberman, placing bombs, killing enemies, etc.) until the level is cleared. To ensure that the test script contains meaningful assertions, the participant is requested to add assertions that check whether an equipment is indeed retrieved, whether an enemy is indeed killed, and whether the final score is correct. For M2, the participant creates the test script by playing the game and recording the game-play actions, and then adds the same requested assertions into the script. For M3, the participant tests the game simply by playing it. The time that the participant spent on method is recorded.

In the second phase, the maintenance cost is measured. The participant is given a modified game level that is only slightly different from the level given in phase 1 (only one obstacle and one equipment are moved to different places). Again, the participant is requested to test the level thoroughly using all three different methods. Since the test script produced in phase 1 no longer works, for both M1 and M2, the participant is requested to revise (fix) the test script until the level is cleared. For

Table 1 The testing cost of three different methods (*unit* mm:ss)

	M1 time	M2 time	M3 time
Phase 1	51:36	30:52	01:43
Phase 2	16:08	15:07	01:34

M3, the participant re-tests the game simply by playing it. Again, the time that the participant spent on each method is recorded.

The resulting average testing cost (human time) is shown in Table 1. The answer to each of the research questions is listed as follows:

RQ1: How much cost (human time) do M1, M2, and M3 require? Answer: The cost of each method is given in Table 1. For both phases in the case study, M1 was slower than M2, and M2 was slower than M3. Based on the observation of the participants during the case study, the reason that M1 was significantly slower than M2 was because M1 needed to write test-script statements line by line, which was less efficient than recording user actions. The reason that M2 was significantly slower than M3 was because that adding assertions to M2 scripts needed a lot of time. Note that the results did not imply that M3 was the most efficient. If the testing was one-time-only, M3 was indeed the best. However, if the same testing was to be performed repeatedly, both M1 and M2 did not need any additional human time. Therefore, they could be better.

RQ2: When the contents of a game level is changed, does the scale of the change influences the testing cost. Answer: For M3, since a human tester had to re-test everything by hand, the testing costs of a brand-new level (phase 1) was not significantly different from a slightly changed level (phase 2). The story was however different for M1 and M2. Both the testing costs of M1 and M2 reduced significantly from phase 1 to phase 2, indicating that the participants were able to reuse parts of the test scripts developed in phase 1. Therefore, in terms of testing cost, a large-scale change in a game level (e.g., a brand-new level) was more costly than a small-scale change.

RQ3: For a large-scale content change in a game level, which method is the best? Answer: From Table 1, M2 was significantly better than M1 in phase 1. Therefore, we could focus only on the comparison of M2 and M3. In terms of human time, M2 cost was 18 times M3 cost ($30:52 \div 1:43 \approx 18$). Therefore, in case that the same testing was to be repeated for 18 times or more, M2 was the best. On the contrary, if the testing was not repeated for 18 times, M3 was the best.

RQ4: For a small-scale content change in a game level, which method is the best? Answer: From Table 1, M2 was better but only slightly better than M1 in phase 2. In terms of human time, M2 cost was 9.6 times M3 cost ($15:07 \div 1:34 \approx 9.6$). Therefore, if the same testing was to be repeated for 10 or more times, M2 was the best. Note that, for a small-scale change, automatic testings (both M1 and M2) could reuse some test scripts. Thus, for M2 to beat M3, the required number of repetitions was lessened. Moreover, the difference between M1 and M2 was not significant, which suggested that given a different setup (a different game, a different game level, a different difficulty level, etc.), the results could be different.

RQ5: Which method is the most favored method? Answer: After phase 2, the participants were requested to rank the three testing methods according to their own preferences (the most favored method received 1 point, the second 2 points, and the least favored 3 points). Overall, M1, M2, and M3 received 14, 9, and 13 points respectively, indicating that M2 was the most favored testing method.

5 Conclusion

This paper proposes a method of automating functional testing for computer games. An HTML5 game framework was created to address the accessibility and timing issues and offer the testing support. In addition, a case study is performed to compare the testing cost of three different methods: writing a test script directly, recording a test script, and testing the game directly by a human tester. The results showed that automatic functional testing for games is feasible. In addition, the participants of the case study consider recording test scripts (and then adding assertions) as the best testing method. Like testing any other applications, when repeated testing is necessary, automatic game testing can reduce the overall testing cost.

References

1. Robot Framework. http://robotframework.org/. Cited 1 Feb 2015
2. HP. Unified functional testing (UFT). http://www8.hp.com/us/en/software-solutions/unified-functional-automated-testing/. Cited 1 Feb 2015
3. Selenium. Selenium. http://www.seleniumhq.org/. Cited 1 Feb 2015
4. Microsoft. Verifying code by using UI automation. https://msdn.microsoft.com/en-us/library/dd286726.aspx. Cited 1 Feb 2015
5. Google. uiautomator. http://developer.android.com/tools/help/uiautomator/index.html. Cited 1 Feb 2015
6. Memon, A.M., Pollack, M.E., Soffa, M.L.: Hierarchical GUI test case generation using automated planning. IEEE Trans. Softw. Eng. **27**(2), 144–155 (2001)
7. Chen, W.-K., Shen, Z.-W., Tsai, T.-H.: Integration of specification-based and CR-based approaches for GUI testing. J. Inf. Sci. Eng. **24**(5), 1293–1307 (2008)
8. Cocos2D. Cocos2D. http://www.cocos2d-x.org/. Cited 1 Feb 2015
9. Unity. Unity. http://unity3d.com/. Cited 1 Feb 2015
10. QUnit. QUnit. http://qunitjs.com/. Cited 1 Feb 2015
11. Kasurinen, J., Smolander, K.: What do game developers test in their products? In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Ser. ESEM'14, pp. 1:1–1:10 (2014)
12. Zhao, H., Sun, J., Hu, G.: Study of methodology of testing mobile games based on TTCN-3. In: 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing (SNPD'09), pp. 579–584 (2009)

13. Cho, C.-S., Sohn, K.-M., Park, C.-J., Kang, J.-H.: Online game testing using scenario-based control of massive virtual users. In: The 12th International Conference on Advanced Communication Technology (ICACT 2010), vol. 2, pp. 1676–1680 (2010)
14. Cho, C.-S., Lee, D.-C., Sohn, K.-M., Park, C.-J., Kang, J.-H.: Scenario-based approach for blackbox load testing of online game servers. In: 2010 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, pp. 259–265 (2010)
15. Schaefer, C., Hyunsook, D., Slator, B.M.: Crushinator: a framework towards game-independent testing. In: IEEE/ACM 28th International Conference on Automated Software Engineering (ASE 2013), pp. 726–729 (2013)
16. Diah, N.M., Ismail, M., Ahmad, S., Dahari, M.K.M.: Usability testing for educational computer game using observation method. In: 2010 International Conference on Information Retrieval Knowledge Management, pp. 157–161 (2010)
17. HTML5-benchmark HTML5-benchmark. http://html5-benchmark.com/. Cited 1 Feb 2015
18. Chen, W.-K., Cheng, Y.C.: Teaching object-oriented programming laboratory with computer game programming. IEEE Trans. Educ. **50**(3), 197–203 (2007)



http://www.springer.com/978-3-319-26394-6

Applied Computing & Information Technology
Lee, R. (Ed.)
2016, XIII, 194 p. 111 illus., 41 illus. in color., Hardcover
ISBN: 978-3-319-26394-6