Entertainment Computing 18 (2017) 79–92

Contents lists available at ScienceDirect

Entertainment Computing

journal homepage: ees.elsevier.com/entcom

An approach to automated videogame beta testing

Jennifer Hernández Bécares, Luis Costero Valero, Pedro Pablo Gómez Martín *

Facultad de Informática, Universidad Complutense de Madrid, C/ Prof. José García Santesmases, 9, 28040 Madrid, Spain

ARTICLE INFO

Article history:
Received 22 January 2016
Revised 21 July 2016
Accepted 13 August 2016
Available online 22 August 2016

Keywords:
Gameplay testing
Testing
Automatisation

ABSTRACT

Videogames developed in the 1970s and 1980s were modest programs created in a couple of months by a single person, who played the roles of designer, artist and programmer. Since then, videogames have evolved to become a multi-million dollar industry. Today, AAA game development involves hundreds of people working together over several years. Management and engineering requirements have changed at the same pace. Although many of the processes have been adapted over time, this is not quite true for quality assurance tasks, which are still done mainly manually by human beta testers due to the specific peculiarities of videogames. This paper presents an approach to automate this beta testing.

1. Introduction

Today, videogames constitute an industry with a revenue comparable with the one in the motion picture industry. As an example, *Grand Theft Auto V*, from Rockstar Games, generated $800 million worldwide during its first 24 h on sale in 2013 [1].

Such a success does not come cheap. Videogame development involves a huge amount of high-skilled people with very different roles, from programmers to designers, artists and composers to name just a few. They must work together in long productions that can last several years. Again, the development of *Grand Theft Auto V* required 5 years and up to 1000 people working in seven different locations [2].

Quality assurance in such big software artifacts is a huge challenge. Although classic techniques such as *unit testing* can still be used, videogames have some peculiarities that usually require manual testing, which increase the already high costs. For example, videogames push *hardware* to the limits, but the user experience must be adequate in mid-range PCs and, at the same time, make high-level gaming PCs worthy. *Compatibility tests* become a nightmare due to hidden *hardware* relationships that become more and more complex over time.

On top of that, videogames are not just software. A big percentage of the development budget is devoted to the so-called *assets*, which define how the game look (3D models, textures, music, etc.) and the general gameplay (maps, missions and puzzles). Each time a level is changed or finetuned, gameplay errors might be introduced, preventing players from completing it. Although these problems cannot be considered *bugs* (at least not *software* bugs), they ruin the game, so they must be detected and solved.

In this paper we propose a way of automatising *videogame beta testing*, useful for testing the game not only after making changes in the source code but also for proving that the playability of a game and the global gameplay are still correct after introducing *level changes*. Next section covers some related work, and Section 3 describes component-based architectures, which have become the standard for videogames in the last decade. Section 4 introduces videogame testing and its limitations, whilst Section 5 talks about our proposal of carrying out automatic beta tests for videogames. Sections 6 and 7 introduce Petri Nets and how to model a game and run tests using them. After that, Section 8 puts into practice the ideas explained in previous sections in a small videogame, and Section 9 describes some implementation details. Finally, this paper ends with conclusions and future work.

2. Related work

With systems growth in size and complexity, tests are more difficult to design and develop. Testing all the functions of a program becomes a challenging task. One of the clearest examples of this is the development of online multiplayer games [3]. The massive number of players make it impossible to predict and detect all the bugs. Online games are also difficult to debug because of the non-determinism and multi-process. Errors are hard to reproduce, so automated testing is a strong tool which increases the chance of finding errors and also improves developers efficiency.

Monkey testing is a black-box testing aimed at applications with graphical user interfaces that has become popular due to its

* Corresponding author.
E-mail addresses: jennhern@ucm.es (J. Hernández Bécares), lcostero@ucm.es (L. Costero Valero), pedrop@fdi.ucm.es (P.P. Gómez Martín).

http://dx.doi.org/10.1016/j.entcom.2016.08.002
1875-9521/

inclusion in the Android Development Kit.[1] It is based on the theoretical idea that a monkey randomly using a typewriter would eventually type out all of the Shakespeare's writings. When applied to testing, it consists on a random stream of input events that are injected into the application in order to make it crash. Even though this testing technique blindly executes the game without any particular goals, it is useful for detecting hidden bugs. This technique can be improved if logs are analysed after each *monkey test* and an evolutionary algorithm is fed with the conclusions in order to make the test more and more destructive [4].

Due to the enormous market segmentation, again specially in the Android market but more and more also in the iOS ecosystem, automated tests are essential in order to check the application in many different physical devices. In the cloud era, this has become a service provided by companies devoted to offer cloud-based development environments. For applications with graphical user interfaces, this testing based on *test cases* requires specific frameworks that check whether the GUI meets its specifications. Examples of such a software are Selenium (for web applications) and Appium (for Android and iOS). They must cope with GUI changes whilst maintaining the original *test cases* still valid, one of the problems that we address in this paper for the videogames field.

In any case, unfortunately, all those testing approaches are aimed at *software*, ignoring the fact that games are also maps, levels and puzzles. Application-level tests usually need to be adapted (or even completely recreated), even when levels suffer small changes. We are not aware of any approach to carry out automatic beta testing that pursues solving this issue as we describe in this paper.

On the other hand, researchers have been trying for a long time to create intelligent systems that can learn by demonstration how to play a videogame using traces generated by expert players. This is specially useful for those domains where creating a plausible artificial intelligence is complex, as in real-time strategy games.

Once the traces have been analysed offline, those systems play the game replicating or modifying the expert movements. Sometimes, an initial step is required, where experts annotate the traces to incorporate extra useful domain knowledge.

A good introduction to this subject is provided in [5], where efforts are described to design a generic recording system with the purpose of annotating this traces later easily. Ontañón et al. [6] shows how to adapt traces recorded in order to be able to replay them afterwards. It also shows a description of how to detect goals and subgoals of the traces recorded and how to use Petri nets for modifying the traces.

Previous research in this *learning from demonstration* field was our source of inspiration for the automatic beta testing presented in this paper.

Next section introduces both *hierarchy architectures* and *component-based architectures*, and also their differences and the basics of *message passing*, which highlights why using the second type is better and useful for us in our purpose of running automated tests.

3. Game architecture

Videogame development constitutes a big challenge today. Broadly speaking, there are two different aspects that must be covered. The first one refers to the *technological requirements*, including graphics, sound, physics simulation or network communication, to name just a few. The second one is related to the *game* itself, the playful characteristics that the software must provide in order to be enjoyable. This is usually known as *gameplay* or *game mechanics* and it is built with *interactive elements*.

Many of the main runtime components of a modern videogame belong to the *technological requirements* and provide the basic infrastructure needed for creating the game interactive simulation. Only a small part is designed with a particular game in mind.

Apart from those runtime software elements, videogames need *resources* such as 3D models, textures or sounds for providing the interactive experience. Early videogames had all those resources hard-coded, but today no one conceives resources in that form and they are virtually always provided using external files. They are collectively known as *assets*.

The separation between source code and *assets* constitutes the key point of the *data-driven architectures* that allow software reusability: the game aspect can be completely changed without involving programmers.

But the videogame would continue to be exactly the same if the logic or game rules were hard-coded into the source code, preventing software from being reused to create *different* games. This can be solved if the game mechanics become assets themselves, or if they are correctly isolated in the source code to be easily replaceable. The term *game engine* is used to refer to "software that is extensible and can be used as the foundation for many different games without major modifications" [7]. Ideally, all the gameplay would be specified throughout assets using external files, and the game engine would be completely independent from the game itself. Usually, game engines (such as Unity3D or Unreal Engine) include *tools* used to create all the assets that the engine will load and run.

What distinguishes one game from another are the *game mechanics*, which emerge from game interactive elements such as the player avatar abilities, the non-player characters (NPCs) or any other element of the game logic with a certain behaviour. Those interactive elements are known as *entities* or *game objects* and they are responsible for creating the game experience. Essentially, a game engine is an *entities manager*, and those entities invoke the underlying subsystems to indicate the way they must be drawn, when and how they should generate a sound, or how they should react to specific events.

The classical way of programming the *entities* is using inheritance. An abstract class is used as the base class for all the other entity classes, and the game engine stores a list of instances of this base class. In the main game loop, all the entities are updated (using an abstract method) so they have the opportunity to react to events and modify their internal state. Afterwards they are drawn using a second abstract method. Entities' behaviour depends on the concrete implementation of those abstract methods.

Depending on the game, the hierarchy can have multiple levels, with intermediate classes that provide auxiliary methods used from different subclasses. As an example, the old 1998 game, *Half-Life* [8], had an entity hierarchy composed of 9 abstract classes and 10 concrete classes shown in Fig. 1.

Although they were extensively used during the nineties and early noughties, today entity hierarchies have fallen into disuse. With videogames getting bigger and bigger, more features were added to entities and many scalability problems arose. Specifically, programmers had to decide how to split a class at each level, but entities have different facets, depending, for example, on their game logic, graphics representations or physics behaviour. Class hierarchies are static, and making the wrong choice on how to split a class in different levels can be fatal if designers suggest to add new entities later. This situation can lead to problems such as *multiple inheritance*, which is not supported in many object-oriented languages. Even if it was, we would most likely suffer from the *diamond problem*.

[1] http://developer.android.com/tools/help/monkey.html.