Educating Novice Developers in Video Game Projects - An Experience Report

Samia Capistrano Gomes, Caio Machado, Victor Silva,
Matheus Silva, and Eduardo Santana de Almeida
Computer Science Department
Federal University of Bahia, UFBA
Salvador, Brazil
samia.capistrano@ufba.br, caiopinheiro@protonmail.com,
{vpinheiro.aguiar, matheuscardimdasilva}@gmail.com, esa@dcc.ufba.br

***Abstract*—Over the last decade, the video games industry has become the most lucrative industry of entertainment. The number of people playing games continues to grow every year. Games are not only being used as entertainment, but as tools in many other fields, such as education, medicine and fitness. Games are considered a rich and complex domain due to their multidisciplinary nature and distinct development process when compared to traditional software.**

Thus, learning how to develop games is a big challenge for both newcomers and experienced developers that come from typical software development projects. In this paper, we present a report on the experience of learning game development through the reimplementation of existing games by students. We present a description of the development process, what went right and wrong, the lessons learned and SE techniques that were viable to apply for each game. Finally, we provide a discussion about the benefits for the learning curve from reimplementing existing games.

***Keywords*-game reimplementation; learning newcomers; good practices software; software engineering;**

I. Introduction

In the last decade, the video game industry has become the most lucrative industry of entertainment. In 2018, more than 2.3 billion people played games, with a forecast to reach the revenue of US $ 137.9 billion [1]. Games are also becoming more and more diverse, being applied not only focused on entertainment, but also on education, business and even medical care. [2].

Due to the multitude of applications in multiple fields, games are considered a rich multidisciplinary domain. It often combines audio, graphics, control systems, artificial intelligence (AI) and human factors into a single product. It is an intrinsic peculiarity of game development, which distinguishes it from traditional software. [2], [3].

The complexity of video games make them interesting tools to support learning and knowledge. Game-based learning is a practical learning experience that facilitates teaching certain skills and concepts. Its application is attractive and motivating for students [4], [5]. Nevertheless, even with the large amount of information about how to develop games, the high number of engines and frameworks which can be used may confuse newcomers.

Consequently, they may face several difficulties at the beginning of their game development journey [6]. Newcomers should use software engineering good practices and reuse techniques to help minimize the difficulties. In typical software development, those practices have been used for years. However, when it comes to games, there are very few studies proving their effectiveness in this field. In 2014, Murphy et al. [7] identified that despite the large complexity, richness and impact of video games in software industry, they rarely are studied by the software engineering community. Three years later, Scacchi [3] continued to verify that software engineering for games is often neglected in curricula.

Therefore, how can we understand the learning curve of beginners in game development and guide them based on good development practices? How to facilitate this learning process? In this paper, we aim to answer those questions by understanding the relationship between good development practices and game complexity. For this, it was necessary to experience the newcomer's learning process in a practical way through the game development, so that those practices and challenges could be identified. We built this experience by having students developing two distinct games with different degrees of complexity, and documented it on this report. Instead of making brand new projects, we chose to reimplement existing games, keeping our focus mainly on the software development aspect. The selected games were *Pong* and *Super Mario Bros.*.

Our study provides the following contributions:

1) Clarify whether reimplementing games is a valid process for teaching game development;
2) Investigate how implementing existent games works as a way to teach a development process that follows good Software Engineering practices; and
3) Report how reimplementing games of different complexities worked for our students.

This experience report may serve as a starting point for beginners who just started making their first game and wonder what their next step might be. It is also helpful for educators who wish to follow a similar process in classrooms.

The remainder of this paper is organized as follows. Section 2 discusses related work. In section 3, we describe our experience report in detail and how the reimplementations were conducted. In Section 4, we present the lessons learned and final results. In Section 5, threats to validity are discussed. Finally, in Section 6 we present the concluding remarks and directions for future work.

II. Related work

Researchers have investigated games, their peculiarities, and ways of applying the consolidated software engineering practices used in traditional development to game development. They also researched about how to facilitate learning and how to apply software engineering practices through games development.

A. Game development versus Traditional projects

In order to analyze the difference between traditional software development and games, Murphy-Hill et al. [7] conducted an empirical study at industry comparing these domains. They interviewed software engineers working in game development from different companies and next surveyed Microsoft developers. Through this process, they identified the multidisciplinary nature of the development process, such as the need for sound, arts, artificial intelligence (AI) and many different characteristics. In order to get those connected, an interdisciplinary team is required, which also means that developers must communicate with non-engineers more than non-game developers do. Lastly, the innovative aspects, the need for quick responses and inflexible deadlines of games require a fast and flexible development approach. Thus, Agile methods are the main approach adopted.

In 2018, Pascarella et al. [8] conducted an exploratory investigation aimed at reproducing the findings of Murphy-Hill et al. [7] on game development by shifting their focus to free and libre open source software (FLOSS). They mined the repositories of 30 FLOSS games and 30 traditional FLOSS systems. They analyzed how developers committed versioned resources and the diversity in malfunctions. Then, they validated their findings through a survey involving 81 respondents among the most productive developers of the chosen projects. They identified that developing games involves activities on more diverse areas than developing non-game systems. Only in games they identified the presence of more diversified specialists confirming the previous findings that consider the development of games different from traditional software development.

B. Learning game development

Using game development as a learning tool provides students with more opportunities to learn software engineering techniques and practices in a more engaging and fun way [4], [9], [5]. It has several benefits such as increased learning effectiveness, motivation and interest. It also reduces training time, allowing students to try and see the consequences in practice by learning from their own mistakes and successes [10], [11].

Garris et al. [10] presented an analysis of the benefits of digital and non-digital games used in SE education. The study identified positive results regarding satisfaction and confidence in the use of games for teaching, and also showed its contribution as a way to increase motivation from students.

Yampolasky and Scacchi [9] conducted a qualitative analysis of a case study using game play testing as a starting point for learning mainstream issues and challenges found in modern software engineering projects and practices. Their study analyzed students with no experience in software development creating a game from scratch. They discussed issues in requirements, design, prototyping, testing and user experience assessment. After the experiment, the students described their learning experiences as both fun and constructive, as well as transformative for some of them. This work can be seen as the closest one to our investigation.

In our work, the students did not develop a game from scratch. Instead, we chose to reimplement two distinct games, *Pong* and *Super Mario Bros.*, because we would like to identify if there are some difference in developing two games with distinct complexities.

III. The Experience Report

Experience reports offer the opportunity to share a hands-on experience. In this section, we describe the execution of the experiment with the purpose of sharing and consolidating the knowledge about game development acquired by novices, while identifying difficulties and good practices that support the development. We gather the data through observation and non-structured interviews with the developers.

A. First steps

A team of developers was made by choosing students who were interested in studying software engineering in games. They were exposed to game development with a practical approach on RiSE Labs, a group researching reuse in Software Engineering. Two classic games, *Pong* and *Super Mario Bros.*, were selected to be implemented from scratch.

Pong is a game where you play a match of paddle tennis against the computer or another person. Each paddle is usually represented by a rectangle. The original version of Pong does not feature realistic graphics, mainly due to the time it was released. Pong has relatively few features and does not deal with complex logic or animations.

Super Mario Bros. is a platform game originally released for the NES. *Platform games* are games where the player makes progress by jumping platforms and walking forward.

This game, in comparison to Pong, includes multiple characters and scenarios. It is considerably more complex, featuring multiple courses, animations and platforming physics, highlighting the 13 years age difference with Pong.

Those games were chosen due to their historical relevance to video games and relative easiness of implementation for beginners when using modern game development tools. In this article, the terms ”made from scratch” or ”implemented from scratch” are referring to source code. Graphics, audio effects, songs and any other assets were either obtained from the Internet, or taken directly from the original games. This will be described with more details on following sections. Reimplementing games allows students to focus only on the programming and Software Engineering aspects of game development, since all assets (such as sound effects, songs and graphics) have already been made. Moreover, a methodology, a schedule and a set of tools were defined for each project.

B. The team

The development team was composed by 2 undergraduate students and one master's student. One of the undergraduates had previous experience working with *GameMaker: Studio* as a hobby, and the other one created *Minecraft* mods when he was a teenager. The master's student previously worked at multiple software development companies, but she did not have any previous experience with game development.

Thus, the group had different levels of experience with game development, with no one having ever worked with the game industry.

C. Project planning

At the beginning of each project, the students defined a schedule with all the tasks and how long they would take to be done. Every week, the team had meetings to discuss the progress and decide which tasks they would work on for the next week.

Features were prioritized based on how much value they would bring to their version of each game. The students also considered whether having a feature or not would impact the quality of the software or improve their learning experiences. Based on that, features like the multiplayer mode of *Super Mario Bros.* were ruled out due to not being relevant neither as a gameplay mechanic or as a development challenge. However, having multiple enemies was deemed useful both for maintaining the integrity of the game design and for encouraging code reuse between each enemy.

Since the games already existed, the team decided that a Game Design Document would be redundant. All design questions could be answered in a timely manner by playing the originals. These games already being available made it easy for the students to investigate mechanics (for example, observing how fast an enemy moves).

D. Tools

Game development can be done through *game engines*. They are software development environments that implement common functionality that most games use (graphic rendering, audio playback, input handling, and others) and provide an API for programmers. They often include their own purpose-built IDE (like *Unity*, *Game Maker: Studio* and *Construct 2*).

Unity is a popular game engine among both professional and amateur developers [12]. It is widely used in industry, has very extensive documentation and the basic version can be used at no cost. For these reasons, it was decided to use Unity for the projects.

Unity facilitates, for example, the usage of animations, physics, input and sound effects. Unity's *Animation State Machines* provide a graphical way to make patterns for animations and actions that a character may have. Unity encourages software reuse through *prefabs*. *Prefabs* are in-game objects composed of many engine components, such as code and graphics. Using them allows elements such as enemies or blocks to be seamless reused through the game, saving development time. *Prefabs* can be placed freely in different Unity Scenes. Unity's Scenes are virtual rooms where the developers place elements that they want to appear on that part of the game. For example, in a platform game, they may make a scene for each course, and another one for a menu.

As the development team performed tasks concurrently, a version control system was a mandatory tool. *Git* was chosen due to its popularity, free availability and support to parallel development among project members. The students used Git through the GitHub service[1].

E. Reimplementation approach

The first project was a reimplementation of *Pong* using Unity. Pong is one of the first video games ever made, originally released in 1972. It features 2 paddles and a ball moving around the screen, meaning to simulate a match of ping-pong. Pong can be seen as kind of a *”Hello World”* for game development. [13]

The key factors to choose Pong as the first project were its simplicity, ease of finding freely available assets and availability of tutorials about how to implement this specific game.

The second project was a reimplementation of *Super Mario Bros.*, the classic platform game for the Nintendo Entertainment System, released on 1983. We chose this game due to its greater complexity when compared to Pong, featuring multiple enemies, levels, sounds and basic physics. It was extremely successful when it launched and got re-released for multiple consoles. The students version was

[1]https://github.com/GamesRiSEUFBA

directly inspired by the 1993 version released for the Super NES.

On both projects, our team decided to customize the folder structure. In the documentation, Unity encourages its users to put all assets inside the ”Assets” folder. Our development team chose to organize the project files by making different folders such as Scenes, Sprites, Scripts, Animations and *Prefabs*. This practice facilitated code reuse in the project.

F. Reimplementing Pong

Initially, each student was given the task of making a reimplementation of Pong by themselves using Unity. The goal was to get everyone more familiarized with Unity before working in a larger project as a team. They could freely choose assets or tutorials to follow, and even add different features if they wanted to. After three weeks, each student had developed a Pong game with similar behavior and features.

Four weeks later, they started a collaborative project to reimplement the game as a team. This time, they started by making a list of features to be included. Besides playing against the computer or another player, it was decided that it would have an option to customize the in-game appearance of the paddles. It would also have a simple menu where the player could choose what game mode he would play. They could also go to the customization screen from there. Figure 1 shows the menu screen.

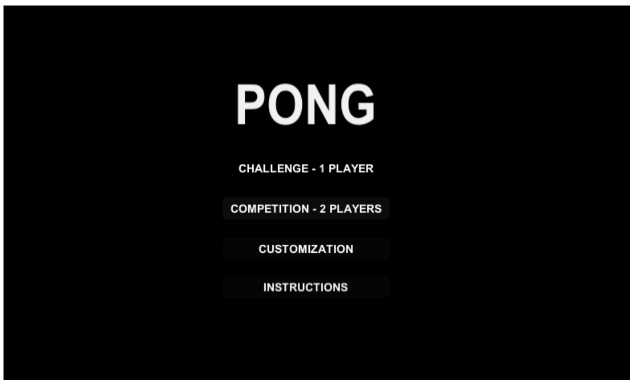

Figure 1. Pong Reimplementation Menu

The students kept progressing in their reimplementation, with a new build being tested and discussed every week until the final release, one month later.

1) What went right: The students learned more about how to work as a team. Most of their early efforts went towards learning how to use Git together with Unity. They also had to learn how to deal with version conflicts and avoid them when different people work on the same file.

They found that using the agile *method Scrum* [14] and meeting weekly to review the most recent build of the project allowed them to quickly grasp what could be improved and accordingly make changes to the game.

The software created collaboratively was technically superior to their individual works. This can be seen as a direct result of both their previous experience, and also due to direct collaboration and discussion between team members.

2) What went wrong: Unity uses hundreds of non-source code files that are very hard to read or edit outside the IDE. This made it considerably difficult to see what certain commits were doing. A modification to those non-code files (for example, moving one the paddles slightly to one direction inside Unity’s IDE) would be extremely hard to visualize by just looking at the commit. This made it even more important to describe what exactly each commit was.

Editing the same scene usually led to version conflicts. It was concluded that it would be easier and more efficient to develop collaboratively if everyone worked on different Scenes. This is not very feasible with Pong, as it has very few Scenes overall (compared, for example, with a game that has multiple ones).

G. Reimplementing Mario

Following the first project, we had the students working on a trimmed-down reimplementation of *Super Mario Bros.* . They were given the task of recreating the game’s ”first world”. In this context, a ”world” is a group of four courses, with the last one always being inside a castle with a ”boss” (a stronger enemy which the player needs to defeat to keep progressing). The original *Super Mario Bros.* had 8 worlds. The courses should be made so players that played the original game could recognize them. Figure 2 shows Mario in the beginning of the last course.

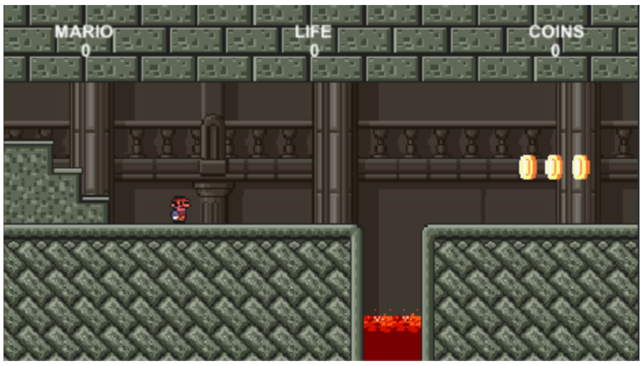

Figure 2. Super Mario Bros. Reimplementation

Each undergraduate student became responsible to work on 2 courses. The game would also feature a title screen and a game over screen, and all students worked on them interchangeably. Due to time constraints, the work was limited to only three courses. For comparison, the original game had 32 courses. This would not affect the learning process, as most of those other courses shared elements with the ones they worked and, from a software developer point of view, were very similar to the first courses. That would mean less time writing software and more time pasting elements using Unity's scene editor.

The students started with gathering graphic and audio assets from the Internet. Those can be found on dedicated websites[2]. We believe that this use is considered fair use, as this project was purely academic, with no commercial interest.

One of the first mechanics that was implemented was Mario's basic movements. That means allowing the player to control Mario and make him walk, run and jump. When manually testing the game, the development team learned that certain mechanics could look correct, but not feel like the original game. For example, Mario's maximum jump height was inconsistent with the original *Super Mario Bros.*. For the average player, this could go unnoticed, but it became evident when comparing both versions side-by-side. They had to tweak values like speed, friction and gravity to make the gameplay closer to the original.

Each week, they would discuss which features would be implemented next. Those tasks were distributed between members and had the next meeting as the deadline. If the students did not finish a task in time, they could either spend one more week working on it, or they could make it a multi-person effort.

Most courses were not fully implemented until the last weeks. This is due to the way the students planned to implement them: each course was essentially composed of common game elements (like blocks or enemies) plus the graphics that appeared on it. As such, they preferred to build the courses later on, when most or all of those elements were in a working state.

To test if a new feature was implemented correctly, they would play the original game and compare it to their own version. This was done to make the gameplay as close as the original as possible. Testing was done manually as they looked for bugs or elements that did not match their original behavior. If the bug was considered not trivial, an issue would be made in GitHub describing how to reproduce it.

1) What went right: One of the biggest motivations for this project was to get the students more experienced with Unity, and learn how to develop games with more complexity (physics, multiple courses, sound effects, Unity's state machines, etc.). We believe this objective was achieved with success.

Sometimes, they would implement a functionality and later find out that it was not working correctly, despite everything being right in theory. This happened with Mario's movement physics. The initial implementation worked, but after just a bit of testing, it was clear that the player movement felt heavy and too different from the original game. It was found that just having a requirement being implemented and passing its tests does not guarantee it is working as intended in video games, and human testing is vital to keep the game look and feel as planned.

Using Agile method Scrum and giving partial deliveries were both very important to the group. It helped them keep focused and better visualize important milestones, such as having sound effects added. This showed that having constant deliveries help visualizing progress in game development.

They managed to save time by reusing code for nearly all enemies that appear in the game. The first enemy to be implemented was the *Goomba*, shown in Figure 3. Goomba is an enemy that behaves without much intelligence, walking from one side to another, damaging Mario if it touches him from the sides, and getting killed if Mario jumps on top of him. Due to its simple behavior, they chose it as a basis to get all other enemies working. This was possible due to the behavior being relatively similar for the *Koopa Troopa* and *Bowser*, two enemies that share similar movement pattern (but with more states). *Bowser* also had a jump mechanic (it would jump from time to time in order to make it more difficult for the player to walk around him) that shared most of its code with Mario's jump.

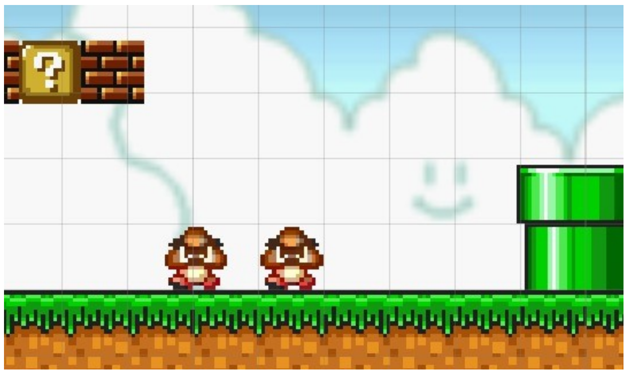

Figure 3. Enemy Goomba

2) What went wrong: Initially, the students settled to release four playable levels within 4 months of development. They underestimated the schedule and the time they would need to spend learning Unity features. Although they had not seen anything that could not eventually be figured out, the time spent looking for guides, reading Unity documentation or searching questions in forums could be better used implementing features and fixing bugs. In the end, they realized that they would not be able to recreate all four courses before the deadline.

Just like *Pong*, Unity generated a large number of temporary files when someone was working with it. When pushing to the project repository, the developers had to be extra careful to not push any of the generated temp files.

[2]https://www.spriters-resource.com/, https://www.mfgg.net/

IV. Discussion

The students developed functional reimplementations of two classical games with different degrees of complexity. They went from no experience at all with Unity to be able to develop relatively complex games over the period of 5 months. They did so with industry standard tools that are used for professional game development.

A. Results

From this experience, we provide the following findings:

1) Reimplementing games (rather than implementing them from scratch) to teach Software Engineering is a useful process and produces good results;
2) For games, manual testing provides results that automatic tests may fail to catch due to its subjective nature;
3) Just like with traditional software projects, constant deliveries help the team visualize their progress over time;
4) It is viable and easy to teach code reuse in game development due to how common it is in games. *Game engines* may further encourage reuse in later projects built using the same engine; and
5) Game development tools used in industry may be used in educational environments with success.

B. Quick feedback and quality control

The students used the Agile method Scrum to reimplement both Pong and Super Mario Bros. Tests were done before partial deliveries to guarantee their quality. If a task was not completed as planned within a sprint, they managed to do it in the next one.

Testing was extremely important to plan new tasks and also to keep both games similar to the original ones.

For *Pong*, a test session consisted of playing one to three matches against another player or against the computer. Then, the students would observe the behavior of all game elements (paddles, ball and score) and see if they matched the expected results. One problem that got noticed by manual testing was that the ball, when colliding with a paddle, would not change its vertical direction.

Super Mario Bros. proved to be more difficult to test. There were many more elements and their behavior was comparatively more complex than anything in *Pong*. A testing session was done by having the students making a full play through of the newest build. Then, they would take notes of any bugs they found, or any unexpected behavior (for example, if Mario's movement felt too different from the original *Super Mario Bros.*). In addition to weekly meetings, the students tested their changes by playing the particular level they were working on. On the very early stages, before they had any courses done, Mario's movement was tested in a development scene that was scrapped from the project later on.

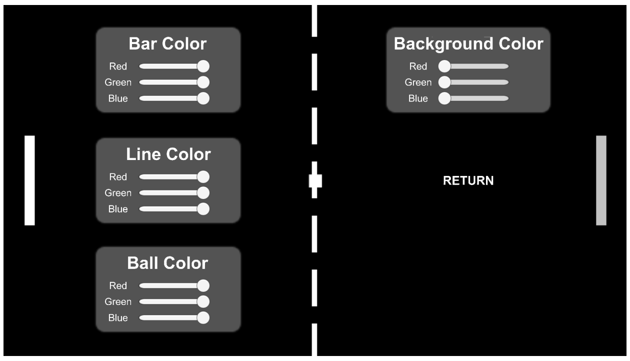

Figure 4. Pong's customization menu

C. Starting with Pong: the right choice?

In retrospect, starting with Pong was definitely an good choice to start learning a game engine. It is simple enough that a developer can get it done in a short amount of time. A developer just has to work with the very basic elements that Unity provides to build all components of a Pong game: the 2 paddles, a ball and the score points. That makes it a good first step in learning Unity, and perhaps any game engine.

The students managed to finish the project ahead of the schedule. So they used the remaining free time to add a feature that allows the player to further customize certain elements of the game, such as paddles and the ball color. Figure 4 shows the Pong customization menu.

D. Super Mario Bros. as the second project

Reimplementing *Super Mario Bros.* was a very different situation. Not only the game was much more complex as stated before, but the students could not find as many support (tutorials) as they did for Pong. Since most guides only explained how to make generic platform games, the students had to figure out on their own how to create any elements that were exclusive to *Super Mario Bros.*. An Unity tool they needed to learn was the Animation State Machine as stated before. This one contributed to make the development process last longer than expected.

For this project, the number of graphic and audio assets was considerably larger. The students decided to reuse them from the Super NES version of *Super Mario Bros.*.

They underestimated how much time was needed to develop this project. This was mainly due to one of the students having previous experience developing platform games using other game engines, and they believed this experience would be useful. This ended up not being the case, as Unity's workflow was proven to be way too different.

E. From Pong to Super Mario Bros.

As stated previously, *Pong* is a much simpler game than *Super Mario Bros.*. It had very few components to implement, and the development was generally quicker and easier. All difficulties were related to learning basic Unity features and C#, the scripting language that Unity uses.

On the other hand, *Super Mario Bros.* featured physics, multiple enemies and courses, sound effects, power ups, and other elements that made the game considerably more complex to implement. While it proved to be a valuable experience for all members, perhaps a better approach would be to introduce a third game among these two, with a middle ground in complexity. We believe this would allow students to have a smoother development process when dealing with these new elements. However, this is an experience to be tested in a future work with other participants.

V. Threats to Validity

During the design of this study, potential threats to validity have been addressed. The first one is the number of participants in the study. Only three students participated on the team. However, even if this could be considered a small quantity, the participants had a very distinct knowledge about software development and no one had experience with the Unity engine.

Another threat is the methodology used to develop the projects. We can not affirm that, when remaking a game that already exists, Scrum offers significant advantages over the *Waterfall method* [14]. However, the students did perceive using Scrum as something useful - it was possible to use sprints in order to make constant builds and separate tasks.

Moreover, there were some difficulties to learn how to deal with Git conflicts. Unity only launched its own versioning tool after the students had finished both projects. On the other side, when they learned to commit only useful files and ignore the non-source ones, this process became easy for them. This could be seen as a validity threat, since other students following this process are unlikely to face the same issues with version control.

Finally, two students had previous experience with game development, but as said before, they did not have any experience with Unity.

VI. Conclusion

In this paper, we presented an experience report about educating novices developers in game development. A group of students implemented two classic games with different degrees of complexity in order to get familiar with Unity and take their first steps in the field of game development. They started having no experience at all, and ended with a fair knowledge about the chosen game engine.

We find that it is valid to use reimplementation projects as a way to introduce students to game development. These projects can also be used in classroom to teach general concepts such as version control and team collaboration. Games also proved to be useful for teaching software reuse, but due to their often unpredictable behavior, developing automated tests proved to be quite challenging. Game engines made for industry use can also fit well in university classes, and dedicated software may not be a mandatory requirement. As future work, we intend to test these concepts with more students and compare their progress when developing games other than *Super Mario Bros.* or *Pong*.

References

[1] T. Wijmanr. (2018) Newzoos 2018 report: Insights into the $137.9 billion global games market. [Online]. Available: https://newzoo.com/insights/articles/newzoos-2018-report-insights-into-the-137-9-billion-global-games-market/

[2] S. Aleem, L. F. Capretz, and F. Ahmed, "Game development software engineering process life cycle: a systematic review," *Journal of Software Engineering Research and Development*, vol. 4, no. 1, p. 6, 2016.

[3] W. Scacchi, "Practices and technologies in computer game software engineering," *IEEE Software*, vol. 34, no. 1, pp. 110–116, 2017.

[4] J. Pieper, "Learning software engineering processes through playing games: suggestions for next generation of simulations and digital learning games," in *Proceedings of the Second International Workshop on Games and Software Engineering: Realizing User Engagement with Game Engineering Techniques*. IEEE Press, 2012, pp. 1–4.

[5] M. R. Souza, L. Veado, R. T. Moreira, E. Figueiredo, and H. Costa, "Games for learning: bridging game-related education methods to software enginering knowledge areas," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track*. IEEE Press, 2017, pp. 170–179.

[6] I. Steinmacher, I. Wiese, T. U. Conte, and M. A. Gerosa, "Increasing the self-efficacy of newcomers to open source software projects," in *Software Engineering (SBES), 2015 29th Brazilian Symposium on*. IEEE, 2015, pp. 160–169.

[7] E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?" in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1–11.

[8] L. Pascarella, F. Palomba, M. Di Penta, and A. Bacchelli, "How is video game development different from software development in open source?" in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 392–402.

[9] M. Yampolsky and W. Scacchi, "Learning game design and software engineering through a game prototyping experience: a case study," in *Proceedings of the 5th International Workshop on Games and Software Engineering*. ACM, 2016, pp. 15–21.

[10] R. Garris, R. Ahlers, and J. E. Driskell, "Games, motivation, and learning: A research and practice model," *Simulation & gaming*, vol. 33, no. 4, pp. 441–467, 2002.

[11] G. Petri, C. G. von Wangenheim, and A. F. Borgatto, “Quality of games for teaching software engineering: an analysis of empirical evidences of digital and non-digital games,” in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track*. IEEE Press, 2017, pp. 150–159.

[12] P. E. Dickson, J. E. Block, G. N. Echevarria, and K. C. Keenan, “An experience-based comparison of unity and unreal for a stand-alone 3d game development course,” in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 2017, pp. 70–75.

[13] H. Lowood, “Videogames in computer space: The complex history of pong,” *IEEE Annals of the History of Computing*, vol. 31, no. 3, 2009.

[14] R. Pressman and B. Maxim, *Engenharia de Software-8^a Edição*. McGraw Hill Brasil, 2016.