Measuring Quality of an Indie Game Developed Using Unity Framework

Mateo Bošnjak and Tihomir Orehovački
Juraj Dobrila University of Pula, Faculty of Informatics, Pula, Croatia
{mateo.bosnjak, tihomir.orehovacki}@unipu.hr

***Abstract* - This paper provides an insight into the entire process of developing an indie game by means of Unity framework. Special attention will be devoted to core game mechanics, such as colliders, physics components, and ray casting. With an aim to determine to what extent introduced indie game has met particular dimensions of quality, an empirical study was carried out. Data was gathered from the representative sample of gamers using post-use questionnaire. Outcomes of data analysis are presented and discussed.**

Keywords - Unity Engine; Indie Game Development; Quality Evaluation; Post-Use Questionnaire; Empirical Study

I. Introduction

Indie games, indie being short for independent, are games developed by independent developers. The opposite of indie games are triple-A games where AAA stands for A lot of time, A lot of resources and A lot of money. The main difference between these two types of games is that indie developers do not have a publisher (such as Activision, Electronic Arts, Konami, Nintendo or Ubisoft) for their game. Besides the lack of a financially strong publisher, indie developers lack the most important resource - people. While big triple-A game studios have teams of 100 people, indie developers are for the most part alone, or in a small team composed of few people.

The first adopted example of a game machine was introduced in 1940 by doctor Edward Uhler Condon, at the world fair in New York [1]. His machine was based on the ancient mathematical game called Nim. Over 50000 people played the game at the world fair. The machine managed to beat 90% of those 50000 players [2].

The first game system that was designed commercially was the “Brown Box”. It was introduced in 1967 by Ralph Baer and his team [3]. The Brown Box was actually a vacuum tube that needed to be connected to a television in order to play the game. The game consisted of controlling two cubes and trying to catch each other. It was later licensed by Magnavox that released the system under the name “Magnavox Odyssey” in 1972 [4].

Atari was the first bigger game company that tried to step up the game development industry. Their first and biggest game release was “Pong” [5] which represented a kick-start to arcades and gaming industry in general. They encouraged over 15 other companies to start developing games of their own.

This paper discusses the process of developing an Indie game using Unity game engine and examines particular dimensions of quality with respect to the developed game. The name of the game introduced in this paper is Indiana Ford [6]. It is a platformer which uses jumping puzzles in its level design. Game engines are software tools for game development. They offer a lot of tools for the developers to customize their game to every little detail. Already completed game engines are helpful in a way that the developer does not have to make every little thing on his own, but rather has them already implemented into the engine. For example, Unity offers components for physics, sprite rendering, sound and other. It also provides an asset store where developers can find various paid or free premade tools to help them develop their game.

Unity was chosen because it is free and beginner-friendly compared to some other commercial counterparts. Some other free popular game engines worth of mentioning are Unreal Engine, Cocos2d, and GameMaker.

The remainder of the paper is structured as follows. Related work is briefly described in the next section. Unity framework is explained in the third section. Core game elements are discussed in the fourth section. Pilot study findings are discussed in the fifth section. Conclusions are drawn in the last section.

II. Related Work

Literature in the field offers a number of papers devoted to indie games development. Nadav Lipkin [7] discussed the term "indie", the meaning the word has had through its existence and its effects on the indie developer community. The whole process of making an indie survival game was explained by David Michael [8]. He described the ways to design, plan, schedule, develop, and test a game. The same author also provided an insight into technologies that are out there and how to choose the best one for developer's particular needs. Finally, he offered some advices on how to correctly place developed game on the market.

There is also a lot of papers covering Unity engine itself. Some of them discuss developing projects with Unity that are not games. An example would be the work of Craighead et al. [9] which discussed the implementation of a robot simulator in Unity. They compared other popular platforms with Unity and explained how the various components of the Unity API and environment editor were used in their project. A paper about developing a game using Unity has been published by Polančec and Mekterović [10]. They described the process of implementing a working online MOBA game.

There has not been much work on indie game quality evaluation using post-use questionnaire, but there are some similarly themed papers. One of them is a work of Pinelle et al. [11] that discussed heuristic game usability evaluation. They analyzed PC game reviews and developed a heuristic for identifying usability problems in both early and functional game prototypes.

III. Unity Framework

Unity is today one of the most popular frameworks used for designing video games. It offers tools for designing 3D and 2D games. Although in the beginning the 2D capabilities were only basic, they decided to expand the tools due to users' demand. Unity supports three programming languages for scripts: C#, UnityScript (which is Unity's version of JavaScript) and Boo. C# is the most popular one while Boo is the least popular one.

The main element in Unity is a scene [12] that can be perceived as a level. Everything that a level needs, from players to various other elements, is placed inside a scene. The elements that are placed inside a scene are called game objects [13]. A game object can represent a player, an enemy, terrain or a pickup. Default game objects that are added when a new scene is created are a camera and a light source. Components are used to control the game objects in a scene [14]. They can be used to give game objects physical properties or animate them. Figure 1 illustrates how the Unity editor looks like.

Figure 1. Unity Editor

The scene view is located in the middle of the editor. On the left side is the scene hierarchy which displays all of the game objects that are currently in the scene. On the right side is the inspector that displays which components the currently selected game object has. On the bottom part is a hierarchy of the project. It shows all of the resources inside the project. The bottom right represents the view from the player side.

IV. Game Introduction

The platform video game developed for the purpose of this paper called Indiana Ford [6]. The story follows a hero on his path to find an ancient artefact inside an old Mayan forest. The main game mechanic is jumping. The player has to do jumping puzzles to try and get the key to unlock each level while trying to avoid deadly traps and enemies. At the end, the player has to defeat the boss in order to win the game. Some of the core game and Unity elements used in creating Indiana Ford will be described in following subsections.

A. Scenes

Scenes are main elements in Unity and can be for 2D or 3D space. Apart from representing different levels of a game, they can also represent a menu (for instance, the main menu). The default scene that is created by Unity contains two game objects: a camera and a light source. Every other game object that the developer needs is placed inside a scene.

B. Game Objects

Game objects are elements which exist within a scene. Developers can add game objects to scenes. They can represent the player, weapons, enemies or buttons in a menu. At least one camera game object is needed to be able to show the scene to the player.

C. Components

Game objects need components to do something because without them they are pretty much useless. The only component that every game object has is the transform component. It dictates where the game object is located. Rigid bodies are components that give the object physics properties [15] such as weight and gravity.

Colliders are components that allow objects to collide or interact with each other [16]. Objects without colliders would simply fall through the ground. An example of colliders is presented in Figure 2. The light green rectangles are the colliders. The player object and the terrain both have colliders. If for example the terrain did not have a collider, the player object would just fall through the terrain as soon as the level starts.

Figure 2. Interaction between two colliders

Colliders can also be triggers. They let the engine know when another collider enters. The trigger can also tell when a collider leaves or stays inside of it.

Animators are components responsible for the animation part of an object [17]. They contain different states of animations such as an idle animation, running animation, jumping animation, etc. These states are not predetermined but are made by the developer.

Sprite renderers hold the images that are being shown through various animation states [18]. They can be used in 2D and 3D scenes.

Scripts are components that contain code that is written by the developer. They are the main way of controlling game objects using various functions. All the scripts in the project presented in this paper are written in C#.

Figure 3 shows one of the many functions used in the project. This function is from the Player script and is used to control the player's movement. In almost every script, there is an Update function that is being called once per frame and inside that Update function is a call for the function. The first line of code gets the value of the input axis and stores it in a float named “horizontal”. The value ranges from -1 to 1, depending on the way the player wants to move. The second line takes that value and creates a new 2D vector which has two parameters: “x” and “y”. In this case, the “y” parameter remains the same and the “x” takes the value of the product of values “horizontal” and “speed”. The variable “speed” is predefined in the script. As a follow up, the variable rb2d (Rigid Body 2D component) updates its velocity with the newly created vector.

```
protected void HandleMovement() {
    if (input == false)
        return;

    float horizontal = Input.GetAxisRaw("Horizontal");
    rb2d.velocity = new Vector2(horizontal * Speed, rb2d.velocity.y);
}
```

Figure 3. Function “HandleMovement” from the Player script

Figure 4 presents the function “HandleJumping” that is responsible for the player's jumping. The first if statement is checking if the button for jumping is pressed and if the player is not sliding. The “IsSliding” variable tells if the player is wall sliding. There is a separate function for jumping while sliding on a wall. After that, the script checks if the player is grounded. If yes, the “rb2d.velocity” is updated with a new vector, much like in the previous function which handles player movement and the variable “canDoubleJump” is set to true. If the player is not grounded, the next if statement checks if the player is able to double jump. If the statement returns true, the “canDoubleJump” variable is set to false and the “rb2d.velocity” is updated with a new vector. The two variables “JumpPower” and “SecondJumpPower” are predefined.

```
protected void HandleJumping() {
    if (Input.GetButtonDown("Jump") && !IsSliding)
    {
        if (IsGrounded)
        {
            rb2d.velocity = new Vector2(rb2d.velocity.x, JumpPower);
            canDoubleJump = true;
            FindObjectOfType<AudioManager>().Play("Jump");
        }
        else
        {
            if (canDoubleJump)
            {
                canDoubleJump = false;
                rb2d.velocity = new Vector2(rb2d.velocity.x, 0);
                rb2d.velocity = new Vector2(rb2d.velocity.x, JumpPower * SecondJumpPower);
                FindObjectOfType<AudioManager>().Play("Jump");
            }
        }
    }
}
```

Figure 4. Function “HandleJumping” from the Player script

Figure 5 shows the function “HandleSliding” that handles jumping while wall sliding. This function is called in the “Update” function only when the player is not grounded and when he is close to a wall. The first line sets negative value to the “rb2d.velocity” which enables the player slowly slide down the wall. Afterwards, the variable “IsSliding” is set to true. The if statement checks if the jump button is pressed and if the horizontal input axis is equal to zero. That means that the player needs to jump before using the arrow keys to choose a direction. The next line updates the “rb2d.velocity”. The “wallDir” variable contains the direction the player is facing, so the minus ensures that the velocity direction is opposite to the wall. The “wallJumpx” and “wallJumpy” are predefined variables which contain the power of the jump. The “transform.localScale” command changes the direction the player is facing. Finally, the “FacingRight” variable is set to be the opposite of what it is now.

```
void HandleWallSliding() {
    rb2d.velocity = new Vector2(rb2d.velocity.x, -0.2f);

    IsSliding = true;

    if (Input.GetButtonDown("Jump") && Input.GetAxisRaw("Horizontal") == 0)
    {
        rb2d.velocity = new Vector2(-wallDir * wallJump.x, wallJump.y);
        transform.localScale = new Vector3(-wallDir, 1, 1);
        FacingRight = !FacingRight;

        FindObjectOfType<AudioManager>().Play("Jump");
    }
}
```

Figure 5. Function “HandleSliding” from the Player script

Raycasting is one of the core game mechanics that is being widely used in today's game development [19]. It is mostly used to check if one object can see another one, that is, to check if there is something between them. It works in a manner that a ray is being cast from point a to point b to see if there is something in the way. An example of ray casting is provided in Figure 6. The lines usually cannot be seen but here they are enabled through the debug console [20]. In the figure, the ray is being cast from point a (the player object) to point b (two enemy turrets) which are not in sight. Here the ray cast is being used to determine if the enemy turrets can see the player object. Various filters can be used to check if there is maybe a wall between the two points, or an enemy, and so on.

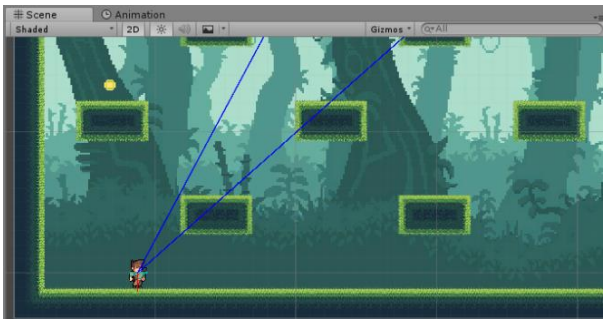

Figure 6. Raycasting

Figure 7 presents the function “RangeCheck” that is responsible for checking if the player is in range of the enemy turrets.

```
void RangeCheck()
{
    Debug.DrawLine(CastStart.position, target.position, Color.blue);

    bool test = Physics2D.Linecast(CastStart.position, target.position, 1
    << LayerMask.NameToLayer("Wall"));

    if (!test && isAwake)
        Attack();
}
```

Figure 7. Function “RangeCheck”

If the player is in range, then the turret fires a projectile at the player. The "Debug.Drawline" command draws a line between two points. This line is shown in Figure 6. The next command uses raycasting to see if there is a wall between the start and the end of the ray. This is ensured by using the third argument when calling the "Physics2D.Linecast". In this case, the third argument is "1 << LayerMask.NameToLayer("Wall")". The function now checks if any of the objects that the ray passes through are walls. The function returns the result and is being stored in the variable "test". The if statement checks if any walls are in the way and if the variable "isAwake" is true. If it is, the function "Attack" is being called. The "isAwake" variable is true if the turret is off cooldown.

V. Pilot Study

A. Procedure and Structure

The study consisted of playing the game and completing a post-use questionnaire that consisted of 3 items related to demographic characteristics, 2 items to determine how long and how often the participants play video games and 18 items for evaluating the quality of the played indie game. The items, besides the demographic and gaming habits ones, were modulated on a five-point Likert scale (1 – strongly agree, 5 – strongly disagree).

B. Participants

A total of 96 respondents ranging in age from 18 to 33 years took part in the study. The sample was composed of 81.2% male and 18.8% female participants. The majority (59.4%) of the participants confirmed they are playing video games for more than 10 years, 18.8% of the participants are playing video games for 6 to 10 years, while 10.4% do not play video games at all. When time spent on playing video games is considered, the majority (38.5%) of students are playing games less than 5 hours a week, 13.5% of them do not play video games while 14.6% confirmed they are playing games for more than 20 hours a week.

C. Findings

Responses to the post-use questionnaire items related to the quality of the introduced indie game are presented in Figure 8 (see Appendix). The majority (56.3%) of the respondents found the game hard while 38.5% enjoyed playing it. As much as 30.2% of students perceived their skills as average ones. Moreover, 61.4% of respondents believe that they improved from the first to the last level. Only 35.4% of study participants stated they would play the game again which could be affected by the difficulty of the video game and the fact that 64.6% respondents found the game frustrating.

The data gathered indicate that most of the respondents (52.1%) think that they did not need a lot of time to grasp the controls of the video game. It was also found that 78.2% of study participants believe that the objectives in each level were clear. The set forth indicates that the game was perceived as user-friendly.

Regarding level design, the majority (51%) of students perceived the graphics quality of levels as good. On the other hand, 46.9% of respondents believe the levels were complex while 81.2% of them found levels easy to understand. It could be therefore concluded that levels of Indiana Ford were well designed.

Responses to items related to game mechanics show that 39.6% of study participants expressed confirmative opinion about game mechanics fluidity while 46.8% of them think game mechanics are clear and simple. The set forth implies there is still a room for improvements in terms of game mechanics.

Findings presented in Figure 8 also indicate that sound effects were done well. More specifically, majority (64.6%) of students confirmed they were not distracted by the sounds integrated in the game, 32.3% of them found sound effects stimulative, while 69.8% believe sound effects were appropriate for the style of the video game.

Finally, regarding the atmosphere, the majority (77.1%) of the respondents think the video game theme was accordant to its style and mechanics whereas 63.6% of study participants believe that the sound effects together with the style and game mechanics make a good atmosphere.

VI. Conclusion

The Unity framework offers a lot of free resources, through YouTube tutorials and Unity's asset store which makes it a great choice for beginners in the field of game development. Although the learning curve can be quite steep, the abundant material provided by the community helps a lot. Even with all the resources provided by the community and Unity itself, the task of developing a game as an indie developer can take up a lot of time. It is nowadays very hard to be good at making art, sound effects and writing quality code in order to place a competitive product on the market. The introduced questionnaire helped us determine the main issues with the game so they can be fixed and a more whole and complete game can be offered to the market. Considering that study presented in this paper was the pilot one, in our future work we are planning to design more comprehensive measuring instrument that would enable assessment of all relevant quality dimensions in the context of video games. The main benefit of it would be that list of found issues could be more detailed and thus help the developer to improve the game more easily. Nevertheless, by applying introduced questionnaire to other indie games, the developers can also determine some general issues with their game and focus their effort on fixing them.

References

[1] C. Baker, "Nimrod, the world's first gaming computer", Culture, 2010, https://www.wired.com/2010/06/replay/

[2] L. Rougetet, "Combinatorial Games and Machines", In: R. Pisano (ed.) A Bridge between Conceptual Frameworks - Sciences, Society and Technology Studies, pp. 475-494, Springer, 2015.

[3] T. Donovan, "Replay: The History of Video Games", Yellow Ant, Lewes, 2010.

[4] M. J. P. Wolf, "Before the Crash: Early Video Game History", Wayne State University Press, Detroit, 2012.

[5] V. Burnham, “Supercade: A Visual History of the Videgame Age 1971 – 1984”, The MIT Press, Cambridge, 2001.

[6] M. Bošnjak, “Razvoj platformske igre u okruženju Unity”, Bachelor’s thesis, Juraj Dobrila University of Pula, Department of Information and Communication Technologies, September 18th, 2017.

[7] N. Lipkin, “Examinig Indie's Independence: The Meaning of “Indie” Games, the Politics of Production, and Mainstream Co-optation”, Loading… The Journal of the Canadian Game Studies Association, vol. 7, no. 11, pp. 8-24, 2013.

[8] D. Michael, “Indie Game Development Survival Guide”, Charles River Media, Boston, 2003.

[9] J. D. Craighead, J. Burke, and R. R. Murphy, “Using the Unity Game Engine to Develop SARGE: A Case Study”, 2007, https://www.researchgate.net/publication/265284198_Using_the_Unity_Game_Engine_to_Develop_SARGE_A_Case_Study

[10] D. Polančec and I. Mekterović, “Developing MOBA games using the Unity game engine”, In: Proceedings of the 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 1510-1515, IEEE, Opatija, 2017.

[11] D. Pinelle, N. Wong, and T. Stach, “Heuristic Evaluation for Games: Usability Principles for Video Game Design”, In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI), pp. 1453-1462, ACM, Florence, 2008.

[12] Scenes, https://docs.unity3d.com/Manual/CreatingScenes.html

[13] GameObjecs, https://docs.unity3d.com/Manual/GameObjects.html

[14] Using Components, https://docs.unity3d.com/Manual/UsingComponents.html

[15] Rigidbody, https://docs.unity3d.com/ScriptReference/Rigidbody.html

[16] Collider, https://docs.unity3d.com/ScriptReference/Collider.html

[17] Animator, https://docs.unity3d.com/ScriptReference/Animator.html

[18] SpriteRenderer, https://docs.unity3d.com/ScriptReference/SpriteRenderer.html

[19] Raycast, https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

[20] Debug, https://docs.unity3d.com/ScriptReference/Debug.html

APPENDIX

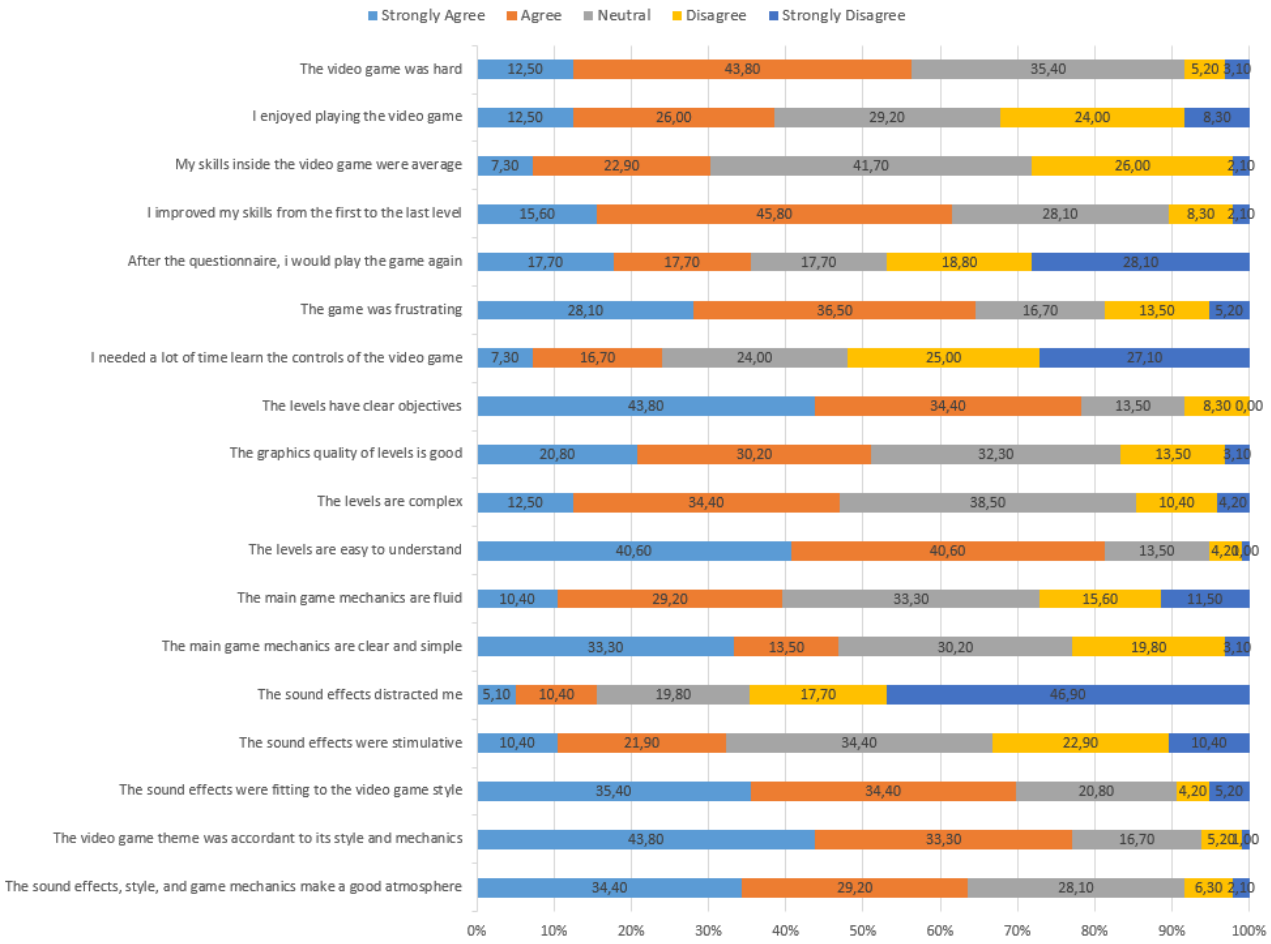

Figure 8. Responses to questionnaire items related to the perceived quality of Indiana Ford game