Recreating Bomberman

Features, steps and challenges

Sofia Papadopoulou
Computer Games Engineering MSc,
Computer Sciences, Newcastle University
Newcastle Upon Tyne, UK
s.papadopoulou2@newcastle.ac.uk

ABSTRACT

Game developers can be considered as virtual architects. They should visualise the whole game they are requested to build, predefine every game aspect needed and develop it one by one. They must predict and prevent possible mistakes that may cost the game's success and satisfy the users' expectations. Developing a game from the beginning is not easy for any developer. However, experience comes with practice and thus developers discover the most effective method to achieve a satisfying result after several games. This project's goal is to demonstrate an indicative way of 're-creating Bomberman', or simply build a 'Bomberman-type game'. Every step taken to build this game is included and explained, in the order that the programmer followed.

KEYWORDS

Bomberman game, Features, Implementation, Challenges, Unity

1. Introduction

In the last decade, gaming has become a trend and pretty popular on a quite extended age group of 10 to 65 years old[1]. More and more people even decide to follow a gaming career path, by turning into streamers and having an income by playing video games. Other professionals from a totally different scientific background, have started showing interest in gamification, to make their work more pleasant. Many are those who have discovered the term 'serious games' and attempt to include it in their jobs. Thus, the game industry has gained a huge portion of consumers and the market has continuous needs for game developers. As a result, an increasing number of developers start their careers after graduation and are called to face challenges the same as all game programmers. In particular, they should produce good quality applications, in the shortest possible time and with the minimum cost and resources. Therefore, sometimes game engines come in handy, especially for developers without much of experience. Unity and Unreal Engine are always the top two on lists about the most popular[2] or the best game engines[3].

At the beginning of a game developer's career, programmers are usually asked to create a game like an already existing one, so that they have an image of what they are trying to create. Other times, they may be asked to build a separate feature from zero, with more details provided on the upcoming project. Either way, graduate game developers need some time to obtain the experience needed to be able to break any project in smaller tasks, sort them out in a convenient-to-build way and find the optimal way to implement their assigned tasks. The current dissertation describes the steps taken by a game developer to build a game like Bomberman in Unity. Decisions made in the process will be justified and explained, to provide an insight into the developer's thoughts.

Someone may ask 'Why Bomberman?'. The answer to this question is that Bomberman is one of the longest-running series in gaming, offering a great number of gameplay features and variations to implement and experiment on. It was first released in 1983 by Hudson Soft, a Japanese video game company. Its series consists of more than 80 different games, on various platforms. Bomberman is also known as Dynablaster (in Europe) and Atomic Punk (in North America). On the gameplay side, it is a strategy, maze-looking game, that usually has two modes, a normal and a battle one. Each level consists of static unbreakable (walls) and destructible objects (bricks, barrels etc.). In normal mode, the player's main goal is to kill all the enemies (monsters) on the map by dropping bombs that detonate after a few seconds. When all enemies are dead, the player must discover the exit, which is usually hidden behind one of the breakable objects. In battle mode, the player goes up against other players, that have to be killed, since the last standing player is the winner. The players can be destroyed either by bombs (even their own) or by the touch of the monsters. A feature that makes the game more interesting and attractive are the power-ups. They appear after breaking destructible objects, and they give some 'powers' to the player. For instance, they can extend the bombs' blast, unlock more bombs for the player to use at the same time or increase the player's movement speed. In the current project, the power-ups selected to be implemented are seven: the

[1] https://techjury.net/blog/video-game-demographics/
[2] https://techblogcorner.com/2020/02/06/most-popular-game-engines-for-game-development/
[3] https://gamedevacademy.org/best-game-engines/

'Extra Bombs' that unlocks one more bomb to use, the 'Extra Life' which gives one more life to the player, the 'Fire Range' that extends the bombs' blast, the 'Ghost Bomb' which makes the bombs penetrate breakable objects, the 'Throw Bomb' that allows the player to pick up and throw away a bomb, the 'Speed up' and the 'X-Ray Vision' which allows the player to locate power-ups that are still hidden.

The structure of the rest of the document is as follows:

- Section 2 examines and presents related work on game development and Bomberman
- Section 3 describes extensively the implementation process
- Section 4 presents the results of this project
- Section 5 contains the conclusions and future work.

2. Background and Related Work

To the best of my knowledge, existing literature on frameworks regarding the design and development of video games is limited. There are books with guidelines on game design and several publications on game design techniques based on various elements, such as the theory of visual attention or interactive gameplays. However, game development perspective seems to be less studied.

Rogers Scott [1] analyses the proper techniques to use in order to create a successful and addictive game. He shares his experience and suggests to always consider how users think and what they expect from a game. According to him, if a game is too hard to play or to understand, users will get tired or bored and the game will be a failure quite soon. He proposes strategies and tricks to make a game unique and engaging, for instance, by exploiting game controllers' actuators and giving rewards. He also includes suggestions on how to communicate properly with the user through the HUD, and how to organise and build a beat chart, to create gap-free gameplay. Multi-player games cover a whole chapter where Rogers [1] shares tips for implementation to keep the users satisfied. The significance of audio in games is described extensively and various recommended exercises for gamers and developers' wellbeing are also included in this book.

Rafael Prieto de Lope et al. [2] present and analyse a new methodology for building educational games that are based on interactive screenplays. They first assess software development methodologies such as the 5M (Method, Milieu, Manpower, Machine, Materials), the Padilla-Zea models, the Westera levels, the SUM and the Ontological methodology. Then, they make a list of limitations for each one of the assessed methodologies and they finally suggest a new one that consists of three preliminary and six main phases (Figure 1: Methodology proposed in [2]). Finally, they describe an example of their methodology application on a video game for comprehensive reading.

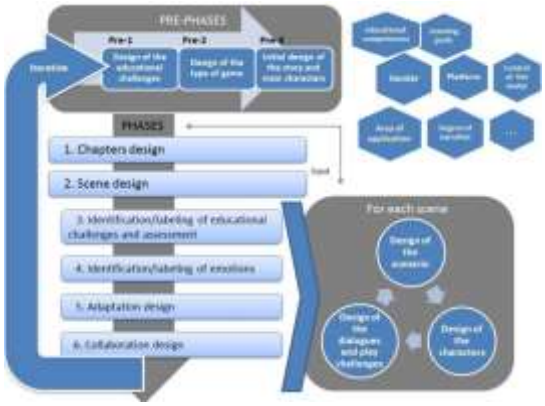

Figure 1: Methodology proposed in [2]

David Milam [3] emphasises the importance of interactivity with the player and how the player's perception and attention are associated with the game interactions. He investigates game designs, their implication with the user's experience and attempts to develop a user's perception-based guidelines on game tasks complexity. He summarises game design models and underlines the significance of game prototyping and playtesting during game design and development. After applying his proposed perception-based framework in six commercial games, he suggests three guidelines to control the cognitive load.

Regarding existing literature, the majority of research conducted is related to game design, instead of development. Besides the development-based methodologies are robust and as a result, quite generic. When game developers start building a game, apart from the order of game elements' implementation, they have to make a list of all the game objects needed, their behaviour and how they will engage the users in this particular game. It seems though that there is not enough research following a game developer's practical approach in implementing a game. Therefore, this dissertation aims to follow a game developer's path throughout a game's development. The implemented game is a Bomberman-type game.

The Bomberman game appears in several research papers, that are focused on artificial intelligence. For instance, Joseph Groot Kormelink et al. [4] examine and compare reinforcement learning methods as exploration systems for the best performance in Bomberman. They investigate the performance of two explorations strategies combined with connectionist Q-learning to learn how to play the game. The results presented demonstrate that methods combining explorations and exploitation actions achieve higher performance than the ones that choose only one of them.

Another research related to Bomberman is [6], where the author provides a platform called Bomberman as an Artificial Intelligence Platform (BAIP), to facilitate the implementation and evaluation of artificial intelligence(AI) methods. Furthermore, he conducts experiments developing two types of AI; primitive behaviour baseline agent and search-based agent and evaluates reinforcement

learning methods. Moreover in [7] and [8], Pommerman, a multi-agent environment is analysed and explained thoroughly, while the writers attempt to create various agents and evaluate their performance on this Bomberman-based environment.

In conclusion, existing literature work lacks in the implementation process of a Bomberman-type game from a game programmer's perspective. Game development in practice requires a good understanding of the game's concept and the ability to categorise implementation tasks and needed features. Besides, the developer needs to apply best practices and to be familiar with users' expectations.

3. Design and Implementation

In the following sections, the reader can follow a game developer's trail of thoughts and actions while building a Bomberman-type game.

3.1. Gameplay Features

The majority of attributes needed for a game are usually found in the specifications given. For instance, a possible description given to a developer who is called to create this game could be "*A 2D Bomberman-type game, with both single and multi-player mode. The maximum players will be four, but if less, players will be able to fill in the empty positions with bots. The camera will be tilted and following the player on single-player mode, but it will be top-view on multi-player. In single-player mode, the player will select between the Adventure and the Party mode. Each level map will be a 17 by 13 grid, made with blocks of 16 by 16 pixels.*". After decoding the description above, the game features needed for this game are the following:

- 2D graphics and camera handling to create a 2.5D projection for single-player mode
- AI for bots (human playing behaviour) and monsters (similar behaviour to Bomberman's enemies)
- animation on 2D sprites for player, bot and monster movement, or bomb's detonation (similar to Bomberman game)
- User Interface (UI) for game mode selection, character selection, bots' addition and players' lives
- and audio for making the game more vivid and engaging.

During the implementation, the thread unravels and each developer may face more specifications or new challenges that were not anticipated. In this project, we attempt to justify the decisions made from the developer according to the requirements, on every step of the implementation.

3.2. Implementation Steps

Game programmers are usually asked to design and develop a requested game in the most efficient way regarding the game performance and the resources utilised. Sometimes superiors assign specific tasks, while other times they allow developers to take initiatives. In the latter case, a programmer needs to distinguish and organise the steps that need to be followed until the game is finished. In the following subsections, these steps have been sorted by their ideal implementation time. Most of them were created in the described timeline, but some that were formed as new ideas were added later on. However, we have chosen to present them this way to follow the developer' actions trail.

3.2.1. Game levels and maps design. At first, we needed a 2D environment to build our game on and a sprite sheet from a designer to use for this purpose. According to the description given, we had to create a 17 by 13 grid. Unity as a game engine provides a tilemap component[4], that comes with a helpful design tool called Tile Palette[5], giving the user the ability to add as many sprites as they need on it and to draw on the grid by simply dragging their mouse. Figure 2 demonstrates the palette used in this project. The sprites included are either for ground tiles, breakable or static objects.

Figure 2: Tile Palette used for level maps design

In our case, we preferred to design every level map on three separate tilemaps (Figure 3). The first one, called 'background', contains the background of the level that is basically the ground of the stage. The second one, called 'collision', includes the indestructible objects that would either stop the player from going outside of the arena or block their way. To achieve this purpose, the 'collision' object also needed a 2D collider - combined with a composite collider 2D to form a new compact geometry of colliders - and a static 2D rigid body. The last tilemap is the one containing the destructible elements whose purpose is to delay the player from roaming freely from the one side of the map to the other. This one is called 'bricks' and a simple 2D collider was added to it. Creating three different layers is beneficial especially when looking for clear

[4] https://docs.unity3d.com/ScriptReference/Tilemaps.Tilemap.html

[5] https://docs.unity3d.com/Manual/Tilemap-Palette.html

paths or checking if a player's next step is a destructible object that they should break using a bomb.

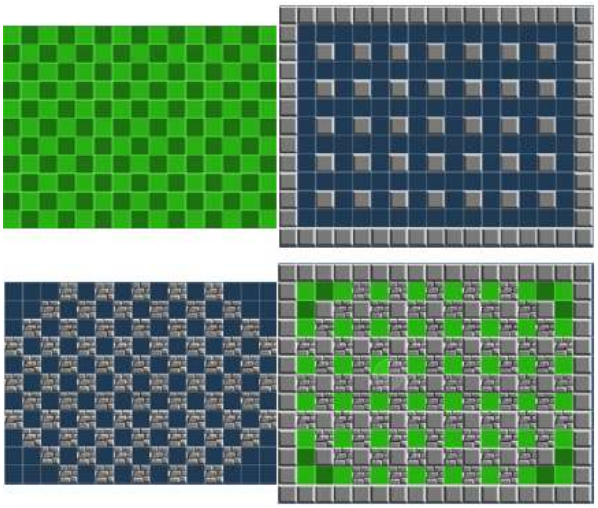

Figure 3: Tilemaps 'background', 'collision', 'bricks' and the complete level 1

By adding a tilemap, a grid component[6] is automatically generated as a parent game object on it. A grid component accepts variables such as a 3D vector for Cell Size, another 3D vector for Cell Gap, an enum for Cell Layout with options 'Rectangle', 'Hexagon', 'Isometric' and 'Isometric Z as Y' and another enum for Cell Swizzle with various options of swizzling between the axises X, Y and Z. In our case, we simply adjusted the Cell Size vector on the size of our tiles, to eliminate space around our sprite tiles.

3.2.2. Game objects and their behaviour. Having the 2D sprites prepared from the designer, we are able to create our game objects. First, we need to consider what we want from the object we are planning to add in our game. In particular, we have to answer the following questions in our mind:
- Are we going to need more than one of these objects? Should we create an empty game object that contains them first?
- Do we need a sprite on this object? Or should we add it as a child object?
- Do we need this object to interact with other objects around it? Does it need a collider? What type of collider does it need?
- Do we want this object to be affected by colliding with other objects or is it going to have forces applied to it? Do we need a rigidbody added?
- Will this object be animated? Should we add an animator to it? Should we add an animator to its sprite?
- Are there any other objects that need to be attached to it?
- Should this object be attached to another object?
- Will this object have a behaviour? Will it be controlled by the user? Do we need to add a script on it?

When we have answered these questions, we are finally ready to add the game object in the Hierarchy window of Unity.

3.2.2.1 The player object. According to our game specifications, we know that the player objects should be four, i.e. the maximum number of players requested. Thus, we need an empty game object 'Players' to include all the player entities. Our players need a sprite to be visible in our game, a collider and a rigidbody since they will interact with surrounding objects. They will be animated and controlled by users so they need some scripts. We have also mentioned that in single-player the camera will be attached on the player, and the player – together with other objects - will be rotated towards the camera. However, we need to keep in mind that our game is still a 2D, similar to the 2D colliders used, which means that our colliders are not supposed to be rotated creating the third dimension. As a result, when single-player adventure mode is activated the sprite renderer component should be on a child object, which will be rotated, while the collider will remain in the original player object, and the camera will be attached temporarily on the tilted sprite child object.

Regarding the Player script that will be added on every player object, the main behaviour is the player's movement and bombs dropping from input controls. Particularly, the user can move the player either by the left joystick of the controller or through the D-pad. Regarding the dropping-bombs functionality, the bomb object has to be created first, but it is decided that it will be triggered by the 'X' button on PS4 controllers, or 'A' on Nintendo Switch and Xbox controller since these buttons are the most commonly used on these controllers and therefore the most convenient for the user. Another functionality that will be needed when the power-ups get created, is the bomb picking up when the 'throw bombs' power-up is enabled, because the user will pick up the planted bomb, by pressing '□' on PS4 controller or 'X' on Nintendo Switch and Xbox controller.

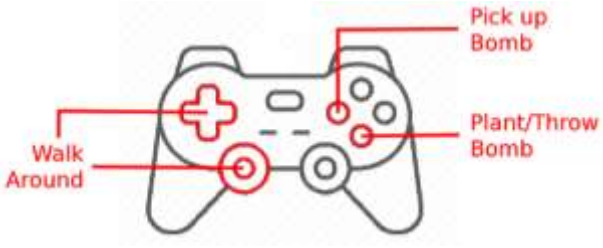

Figure 4: Player Controls on every controller

The Player script should also include functions to enable/disable the power-ups collected. An indicative list of Player script's functions, omitting the Start() and Update() functions, is:

```
private void UnlockTheFirstBomb();
public Transform SelectAnUnusedBomb(Vector3 direction);
private void ActivateBomb(Transform bomb, Vector3 direction);
public void Die();
private IEnumerator Revive();
public void ReturnToInitialPosition();
public void Reset();
```

[6] https://docs.unity3d.com/ScriptReference/Grid.html

```
private void InputHandling();
private void MenusInputHandling();
private Vector2 GetDirectionFromPlayerSprite();
private void SetHoldingBombSprite();
public void AssignInput(Rewired.Player p);
public void AddLives(int extraLives);
public void ActivateKicking(float kickAbilityDuration);
private IEnumerator DeactivateKicking(float waitingTime);
public void SpeedUp(float speedUpDuration);
private void SpeedDown();
private IEnumerator SpeedDownAfter(float waitingTime);
public void ActivateGhostBombs(float ghostModeDuration);
private void DeactivateGhostMode();
private IEnumerator DeactivateGhostModeAfter(float
waitingTime);
public void ActivateXRay(float xRayVisionDuration);
private IEnumerator DeactivateXRayAfter(float waitingTime);
public void UnlockExtraBomb();
public void IncreaseBombsFireRange();
private void CancelAllPowerUps();
```

In addition to scripts, many different animations are needed for the player:
- a walking animation, that consists of eight animations – four for the male sprite moving up, down, left and right and four similar ones for the female sprite,
- a death animation, that represents a burnt figure that is just blinking,
- an empty 'Idle' animation which has as a primary purpose to maintain player's pose as is when no movement takes place, and
- a holding-bomb animation that, similarly with the walking animation, consists of four animations for the male sprite and four for the female sprite.

The animation controller of the player object evolved into the structure shown in Figure 5: Player's Animator. In general, we wanted the transitions to be sharp and immediate, so we did not need to use a blend tree.

Every transition on movement animations has the following parameters in its conditions: 'HorizontalAxis' or 'VerticalAxis' depending on the direction, the 'IsHurt' variable that needs to be false, otherwise, it transits to the death animation, the 'IsFemale' variable that distinguishes animations for the female sprite, from the ones with the male sprite, and the 'IsHoldingBomb' variable that when is true, leads to the holding-bomb animation. The parameter 'IsRevived' is triggered at the end of a coroutine a few seconds after a player has lost a life, re-colouring the player's sprite.

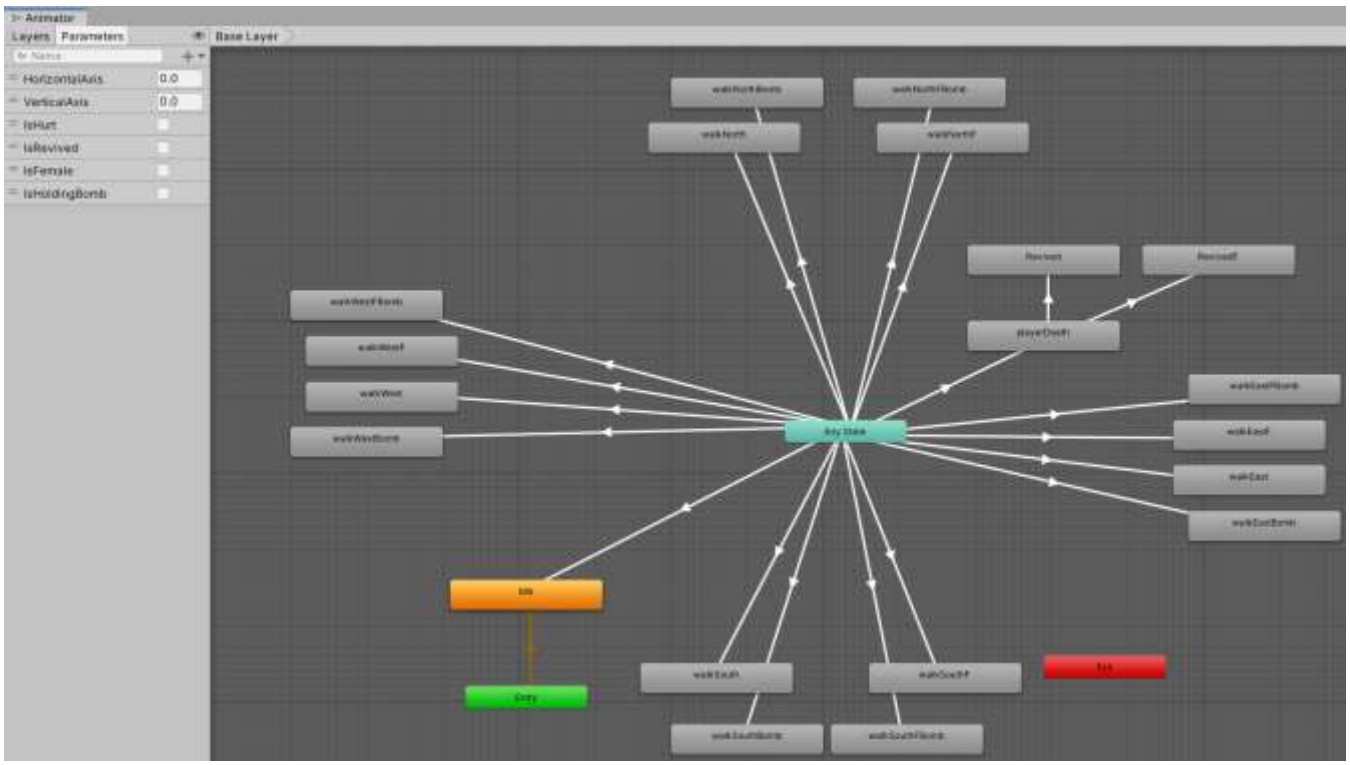

Figure 5: Player's Animator

Figure 6: Male and Female Animation Sprites

Furthermore, some power-ups create the necessity of some extra children objects of the player. For instance, a bomb detector object which has a circle collider that extends beyond the player is useful for the 'throw bomb' power-up, offering an insight of the surroundings and suspending the bomb when a player with this power-up is about to pick it up. Another interesting and helpful addition is a sprite mask as a child of the player, which gets enabled when the 'X-ray Vision' power-up is acquired. This mask along with the power-up object's design - that will be evaluated in the following paragraphs - allows the user to see 'behind destructible objects'. The sprite mask has a custom range for rendering layers therefore it filters objects that are above bricks and walls.

3.2.2.2 The bomb object. Every player owns four bombs that are assigned to them. Having bombs per player is beneficial to easily apply modifications on them every time that the player picks up a power-up like 'extra bomb', 'ghost bomb' or 'detonation range increment'. If we used shared bombs for all the players, we would need to apply these alterations according to players 'condition' on each bomb every time they were about to plant one and revert them after the explosion.

Bombs are supposed to detonate after a small time-period after being planted, so first, we needed a way to indicate on the users that there is an evolving countdown. An animated fuse that was placed as a child of the bomb and was gradually shortened until it was hidden in the bomb itself, fulfilled the aforementioned purpose. Every time a bomb is placed, it gets activated as an object. A countdown variable gets set to be equal to the fuse's animation length and as the frames pass it gets reduced until it reaches zero. When it does, it triggers the explosion.

Regarding the explosion, each bomb has its own explosion parts, consisting of the rounded edges and the extensions (as shown in Figure 7) - used to extend the detonation range. Each one of the parts has its own animation too to create the 'fade out' effect (Figure 8).

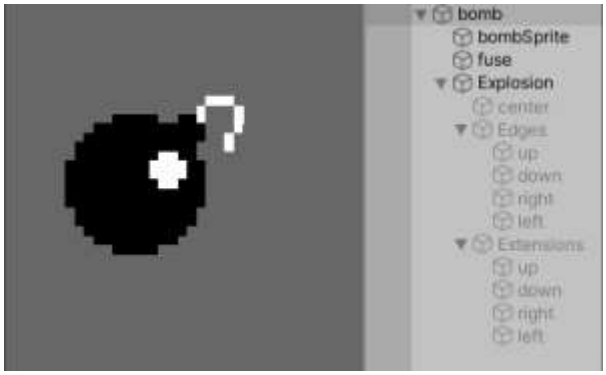

Figure 7: Bomb's game object

Figure 8: Bomb explosion animation

The appearance of the extensions is related to the bomb's level variable. They only appear if the bomb's level is greater than one. The extensions sprites' Draw Mode in Unity is set to Tiled because we need them to appear repeatedly in proportion to bomb level. For instance, the sprite's size will be calculated as:

```
new Vector2(_gridSize, _gridSize * (explodedTiles - 1))
```

with explodedTiles variable to be lower than or equal to the bomb level. The case of non-equality is faced when a bomb is blocked by an indestructible or destructible object. In the former situation, the bomb detonation stops right before the indestructible object, while in the latter, it stops right on the destructible object and destroys it. There is an exception to the last case. When the 'ghost bomb' power-up is enabled, then the detonation can go through the destructible objects, to its full range and destroy anything behind them, thus a boolean variable 'IsGhostBomb' was added.

A primary behaviour of any bomb is to trigger any other bombs in range, so we included a function that practically sets the explosion's countdown to zero immediately, without waiting for the fuse's animation. To make it look realistic though, we had to disable the fuse's animation and hide the fuse at the same time. Another functionality needed for the power-ups is the search for a good landing spot when a bomb is thrown. In this case, before the bomb starts moving over the objects, we need to check all the tiles in the wanted direction for ground tiles. After some experiments, the minimum realistic distance for the throwing has proven to be three tiles.

An indicative list of Bomb's functions is:

```
private void Awake();
private void Update();
private void Explode();
private IEnumerator ExplosionFadeOut();
private void HideExplosion();
private void HideExplosionChildren(Transform explosionPart);
public void SetCountdownTo(float newCountdown);
public void ThrowBomb(Vector3 direction, float gridSize);
private void StartTweening(Vector3 from, Vector3 to);
public void ResetLocally();
public void Reset();
public void FollowPlayer(int playersIndex);
public void UnFollowPlayer();
private void OnTriggerExit2D(Collider2D other);
```

3.2.2.3 The power-up objects. Similarly to the traditional Bomberman, there are many power-ups for the player to pick up in the current project as well. All the power-up objects share a common PowerUp script which is responsible to detect the colliding player or bot and call the proper function on the detected object's script, according to the type of the power-up object. As mentioned in the previous section, the power-ups implemented in the current game are:

	Extra bombs: At the beginning of each level, every player/bot has only one usable bomb. The rest are locked. By picking up this power-up, one more bomb gets unlocked and usable.
	Extra life: Every player/bot has three lives initially. This power-up works either as a refill, if any life has been lost by the time the power-up is picked up, or as an extra life, making them four in total.
	Fire range: The bomb's initial level is one, which forms a small cross with no extension parts when detonating. This power-up increases bombs level by one, adding/tiling extension parts per side.
	Ghost bomb: Bombs do not penetrate destructible objects by default. This power-up allows the player to burn game objects hidden behind obstacles or even destroy many obstacles at once.
	Throw bomb: Blocking your own way out with a bomb is the most common mistake in Bomberman. With the specific power-up, bombs get suspended when you are too close to them and you can pick them up and throw them in any direction you wish.
	Speed up: Every time this power-up is picked up, player's/bot's speed gets increased by 0.06. This power-up is permanent until the player or bot gets hurt or the level is completed.
	X-Ray Vision: This power-up reveals the hidden power-ups' position on a certain range around the player/bot. However, it is a tricky power-up because, in multi-player mode, other players can also see the hidden power-ups around you.

Most of the power-ups' implementation was quite straight-forward, but there were also some exceptions. The most demanding ones were the 'Throw bomb' and the 'X-Ray Vision'. The former needed an extra player animation for holding the bomb, a bomb detector collider as a child for player object and a few functions to traverse through the row/column which the player was facing towards when throwing the bomb, to find empty landing tile. As for the throw motion, it was created as a linear interpolation on the horizontal/vertical axis, along with some curves, to create a bounce effect.

On the other hand, the 'X-Ray Vision' was implemented with a circle-shaped sprite mask, attached to the player object. An easy way to fulfil our purpose was to make the destructible objects, - under which power-up objects are hidden - 'Visible outside of the mask'. However, this was proven ineffective, as the player could not see the obstacles around them inside the range of the sprite mask and therefore kept walking towards dead-ends. That said, we followed a different approach: we added on every power-up object a copy of their sprite as a child object, sorted it on a higher rendering layer and made it 'Visible inside the mask'. As for the sprite mask, it filters certain rendering layers' range, which is higher than the layer of other map objects but contains the power-ups sprites layer.

3.2.2.4 The bot object. In this game, we have used two types of AI, namely the bots and the monsters. The Bot is supposed to imitate a person's playing behaviour, so it always looks for the closest opponent (either real players or other bots), as analysed in Figure 11. After finding its target, it uses an A* algorithm [5] to calculate the best path to follow, including paths through destructible objects. Then it moves towards every tile in the found path one by one until it finds an obstacle. If the obstacle is a destructible object or another player/bot, it drops a bomb. If the obstacle is a bomb or after it has planted a bomb, it starts path-finding for a position a few tiles backwards and on a turn. The found path towards the player is temporarily replaced by the new path for the hiding spot. When it reaches that spot, it freezes for a second to avoid the bomb's detonation. Then, it recalculates the path towards the player and starts over.

Figure 9: Bot planting a bomb and hiding

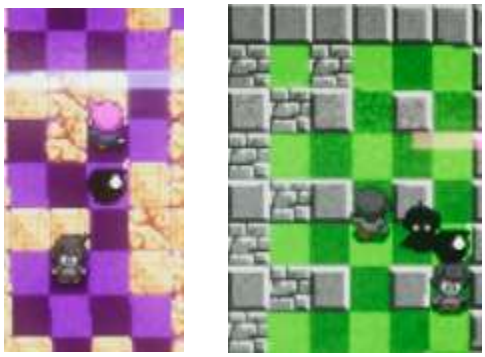

Figure 10: Bots planting bombs to attack their enemies

Figure 11: The Bot AI logic

3.2.2.5 The monster object. The Monster is the second type of AI used in this game. It appears only in the Adventure mode of the game. At first, Monster AI worked similarly to the bots, except that it could not plant bombs. It always had the player as a target, but unless the player cleared a path, the Monster could not reach its target. However, an 'aggressive' AI like that was a quite tough opponent for our player, and in original Bomberman, the monsters were not that smart. Moreover, the player loses a life by simply colliding with the monster. As a result, we ended up separating our monsters in two kinds: the 'dummy' and the 'smart' one.

The Monster AI object has three states: Idle, Roaming and Hunting. The way the 'dummy' one operates is by checking for all four directions around it if they are clear or blocked by wall, obstacle or bomb, storing the clear directions in a list.

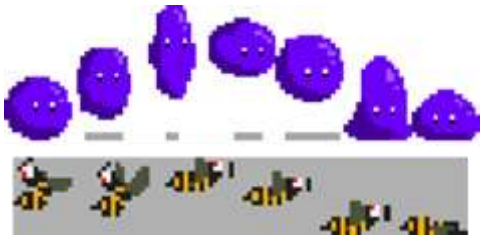

Figure 12: Top - Dummy AI, Bottom - Smart AI

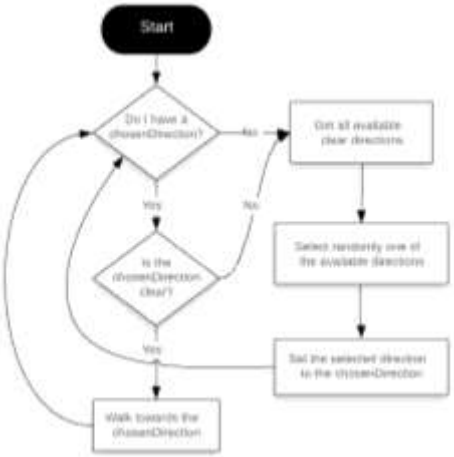

Figure 13: The 'Dummy' AI logic

If there are no clear directions, it turns to Idle state. If there are clear paths, it selects randomly one of the stored directions, turns to Roaming state and starts moving towards it. As long as the chosen direction is clear, it keeps moving towards it, otherwise, it repeats the process from the beginning. Hunting state is not used by the 'dummy' AI at all.

The 'smart' one, on the other side, operates as the 'dummy' one on roaming, until it spots – by being on the same row or column with the player – and has a clear path towards the player (Figure 14). In this case, it turns to the Hunting state and starts following the player. If it reaches a bomb, it moves backwards and turns, similarly to the bots.

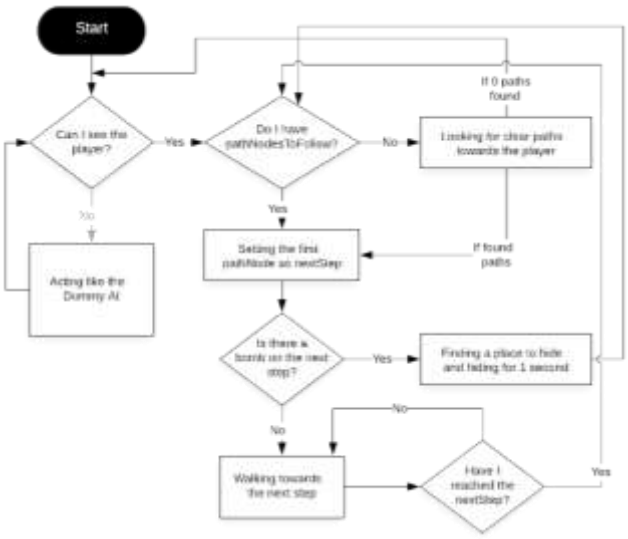

Figure 14: The 'Smart' AI logic

3.2.3. Object generators. In every game, most of the dynamic objects get spawned in runtime, but for performance issues, they usually get pooled from the beginning and are activated/deactivated when in need. In this game, three generators were created: one for levels, one for the power-ups and one for monsters.

Each generator has some variables exposed making it easily configurable. For instance, the power-up generator allows the developer to adjust the type and the number of the power-up objects appearing on the levels, having an exposed array where the developer easily fills in the prefabs of the wanted power-up objects and the times of their appearance. It also stores some generic settings, like the cool down time of the power-ups that we might want to be temporary. However, if these durations/cool-down times are set to zero, the power-ups are transformed into permanent ones, which means that they will be deactivated either when the level is completed or the player is hurt.

The power-up generator is located on the 'PowerUps' game object that also functions as the parent for all the power-up objects. Regarding its functionality, every time a level starts the generator scans current level's tilemaps storing the positions of destructible objects in a list.

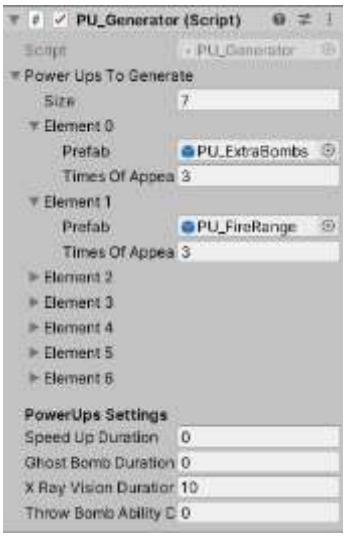

Figure 15: Power-up generator configurations

If there are no children under the 'PowerUps' game object for every power-up in the configured array and for every time of appearances set for it, the generator has to spawn an object. To do that, it gets a random index for the list with the positions of the destructible objects, it instantiates a power-up of the given prefab on the randomly chosen position and removes this position from the list. Otherwise, if the power-up objects have already been generated – for example when the current level is not the first one – the generator just re-positions the power-ups randomly on the new positions available.

The level spawner is responsible for generating a certain amount of levels as set in the object's settings, using the level prefabs given by the developer. If prefabs are less than the number of levels set, the spawner instantiates multiple instances of the given prefabs.

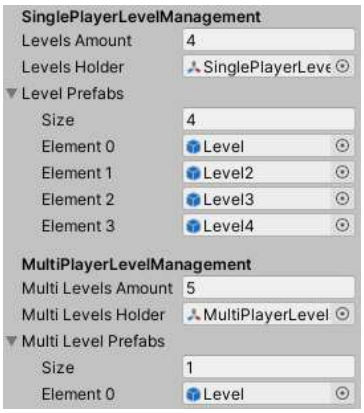

Figure 16: Level generator configurations

An idea that came up later on this project was having different camera's background colour per level, which is also handled by the level generator. Every level map object has its configurations for

the camera's background colour, its tilemaps and the seconds passed before the map starts shrinking (Figure 17). The shrinking arena is a feature for multi-player or party mode, where the current map starts appearing an inner circle of indestructible objects every x seconds (x are the seconds set in the configuration of each level), forcing the players to face each other.

Figure 17: Level settings and shrunk map

Regarding the monster spawner, unspawned monsters are stored under a hidden object on the root. Every time a level starts, the appropriate monsters, as set on the spawner configuration, are transferred under the 'Monsters' object and positioned randomly on ground tiles of the current map. An addition for better user experience is filtering the aforementioned random position to avoid spawning monsters next to the player and consequently sacrifice the player's lives. Whenever a monster dies, it resets and is moved back under the unspawned monsters object. In the spawner settings, the dropdown for the monsters type - apart from the specific monster types and the random one - contains also the options 'dummy only' and 'smart only', covering the possibility of the existence of multiple monsters of each intelligence.

Figure 18: Monster generator configurations

3.2.4. User interface (UI). Game developers have a certain way of communicating with gamers and is called 'user interface'. Any information that users need to know – for instance, the players' lives – has to be displayed somewhere on the player's screen. Moreover, various screens are used to allow users to follow the game flow, while others give them the ability to change the game according to their preferences.

Figure 19: Start screen & Game Over screen

Therefore, in the current project, amongst the usual screens like Start and Game Over (Figure 19), we have also Game Mode and Character Selection screens. In single-player mode, the screens used are the Start screen and the Mode Selection screen. According to the chosen mode, screens are differentiated. In Adventure Mode, the player will see the Next Level, the Game Over and the Win screens. The first two do not need further explanations. The Win screen appears when the current level has reached the Monsters per Level array size, set on the Monsters Spawner, making sure that every level has enemies.

Figure 20: Next Level screen & Win screen

In Party Mode, where the player has to survive against the bots, the player is given the ability to modify their character's gender, skin and clothes, besides, the number of the bots to which they will play against. It is worth mentioning that in the Character Selection screen, one bot is pre-activated because the player cannot play alone. Another screen - that appears both in Party and multi-player mode - is the Winner screen, which displays the winner's image as it was configured in the Character Selection screen, along with the winner's number, for example, 'Player B Wins'.

Figure 21: Character Selection screen – default and modified

Figure 22: Winner screen

In multi-player mode, the UI screens are similar to the Party Mode screens, with a small alteration. The Character Selection screen, instead of having a bot pre-activated, has all the connected players visible, with arrows enabled to transform their characters. If the connected players are fewer than four, then there are 'Add Bot' buttons visible to fill the empty positions (Figure 23). Furthermore, in order to inform the other players that a player is ready, we hide the selection arrows. Last but not least, each player's lives are visible at the top of the screen, together with the player's number and a thumbnail of their image (Figure 24), in every game mode.

Figure 23: Character Selection screen with 2 Players & 1 Bot

Figure 24: Main Game UI

3.2.5. Camera manager. Game specifications requested different types of camera for our game modes, therefore a camera manager was necessary. In Party mode and multi-player game, a top-view camera showing the whole map was asked. Thus, in these modes, the camera manager places the camera under the root game object, turns it to orthographic and centres it on the map.

Figure 25: 2D camera for Party or multi-player mode

In the adventure mode though, the camera manager attaches the camera on the player's sprite object and turns it to perspective. In the camera handler settings, two arrays of game objects are included. One array has to be filled with all the object that in adventure mode will be rotated by -30 degrees angle including the player's sprite so that they face towards the camera and create the 2.5D effect. The other array has to contain any children items of the other array's objects, that are needed to stay unrotated, such as the Explosion object under each Bomb. Some objects, such as the monsters' sprite objects, were also added in the rotation list through the script for the developer's convenience.

Figure 26: 2.5D camera for Adventure mode

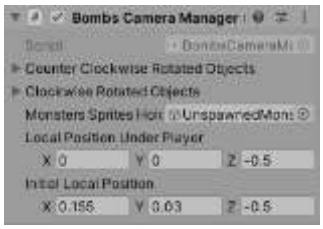

Figure 27: Camera manager configurations

3.2.6. Sound Effects and Music. Having completed the previous steps, the game still misses some features for vividness, such as background music and sound effects. For background music, an intro sound is played once when the game starts and right after it, a seamless audio clip is set to be played repeatedly. Considering the sound effects used, a list of objects was created. The objects that synchronised with sound effects are:

- bombs' explosions, with an explosion sound
- power-ups picking up, with a jiggling sound
- the game over screen, with a mocking laughter
- the next level screen, with a short satisfactory audio clip.

3.2.7. Extra features requested on a second phase. It is not uncommon during a project, for new ideas to come up adding a special touch to the game. Even the simplest game, can become a lot more interesting with small additions, that sometimes offer just an aesthetic variation. For instance, screen-shake in a game is a straightforward coding addition, that does not affect the gameplay or the story, but creates an effect that catches the user's eye. Thus, in this project, screen-shake was implemented for bombs explosions.

Furthermore, getting inspired by other games and creating a unique mixed result, is also beneficial for every project. An idea that was implemented and differentiated this project from the original Bomberman game, was a shrinking functionality for the tilemaps used as a level map. It was decided that in every mode except for the adventure one, every few seconds, the arena had to shrink by the addition of an inner wall of indestructible items. This way, the players would be forced to face each other, instead of procrastinating to 'explore' the arena.

3.3. Challenges

Programmers are expected to solve problems and deal with challenges on a daily routine. In the current project, two issues required more attention.

The first issue was that Unity's physics were causing an annoying bounce on the walls of the map, even when physics materials with no bounciness were used on both the walls and the players. Therefore, we followed a different approach. Instead of modifying the player's position gradually, we decided to use the function *Rigidbody.MovePosition*[7], that accepts the new position as a parameter. By replacing the initial movement code (Figure 28) with the one shown in Figure 29, we managed to make the players stop immediately when facing a wall, thus avoiding the bounciness.

Another minor issue that needed to be resolved was the unsmooth turns. When the player was not located perfectly in an intersection tile, they could not turn. They got stuck on any obstacle on the directions they were trying to turn to. From the very beginning, the player objects' collider was set to Circle collider 2D, to make their movement more flexible, but that was not enough. At last, we proceeded with setting the player's rigidbody interpolation option to Extrapolate[8], resulting in smoother turns around corners.

```
float horizontalAxis = Input.GetAxis("LeftStickHorizontal");
float verticalAxis = Input.GetAxis("LeftStickVertical");

if (Mathf.Abs(horizontalAxis) > Mathf.Abs(verticalAxis))
{
    direction = horizontalAxis > 0 ? 1 : -1;
    transform.position += Vector3.right * direction * Time.deltaTime * Speed;
}
else if (Mathf.Abs(horizontalAxis) < Mathf.Abs(verticalAxis))
{
    direction = verticalAxis > 0 ? 1 : -1;
    transform.position += Vector3.up * direction * Time.deltaTime * Speed;
}
```

Figure 28: Initial movement code

```
float horizontalAxis = Input.GetAxis("LeftStickHorizontal");
float verticalAxis = Input.GetAxis("LeftStickVertical");

if (Mathf.Abs(horizontalAxis) > Mathf.Abs(verticalAxis))
{
    direction = horizontalAxis > 0 ? 1 : -1;
    _rigidbody.MovePosition(new Vector2(
        transform.position.x + direction * Time.deltaTime * Speed,
        transform.position.y));
}
else if (Mathf.Abs(horizontalAxis) < Mathf.Abs(verticalAxis))
{
    direction = verticalAxis > 0 ? 1 : -1;
    _rigidbody.MovePosition(new Vector2(transform.position.x,
        transform.position.y + direction * Time.deltaTime * Speed));
}
```

Figure 29: Final movement code

The most challenging task in this project was the various AI types we needed, mostly because of their path-finding. Searching around the grid repeatedly is not cheap, especially when the grid's contents keep changing, thus it should be limited to the minimum possible. In the beginning, both the bots and the monsters were smart and they were supposed to hunt down the players/bots. As mentioned in Sections 3.2.2.4 and 3.2.2.5, it was decided to divide the monsters into the 'dummy' and the 'smart', because it felt quite tough for the gamer. In this section, we will mention the ways we limited the use of path-finding.

The 'dummy' AI does not do any path-finding. It simply checks for its surrounding four tiles (above, below, left and right), every time it comes across an obstacle. The 'smart' one always checks if it is positioned in the same row or column with the player. If it is not, it omits the path-finding algorithm and behaves like the 'dummy' one. Otherwise, it performs one search to find a clear path towards the player. If the search returns any paths, then it starts hunting, doing path-finding every time the player moves. However, if no paths are returned, it roams around like the 'dummy one', as presented in Figure 14.

On the contrary, the bot has to always target the closest one player/bot, as described in the previous sections. It is programmed to recalculate the available paths every time the targeted player has

[7] https://docs.unity3d.com/ScriptReference/Rigidbody.MovePosition.html

[8] https://docs.unity3d.com/ScriptReference/RigidbodyInterpolation.Extrapolate.html

changed position. When it comes across an obstacle, it drops a bomb and looks for a close place to hide that is approachable, without having to clear the path. When it reaches the chosen place, it needs to recalculate the best path towards the closest player and starts hunting again. In order to make this procedure as cheap as possible, when we load the current level, we create two arrays of path nodes. The one stores all the tiles on the map and marks them as approachable (ground tile) or breakable (destructible tile). The other array holds only the clear approachable tiles. Every time a bomb explodes, we update the proper values on these two arrays. The path-finding algorithms used are two as well. The one, which uses the first array, is used from the bots, to find the best path even with breakable tiles in it. The second one uses the second array that contains only the clear tiles when searching for clear paths towards a position.

4. Results and Evaluation

When a project is finished, we need to confirm that what was requested in the given specifications has been completed and is functional. In the current game, after the implementation described previously, there is now a Bomberman-type game, called BombDudes, that has three modes, namely an adventure, a party and the multi-player one. Furthermore, it has three heuristics AI used in the aforementioned modes. The camera gets positioned accordingly, based on the game mode selected, while the levels have the wanted dimensions and are created as grids. As a result, all the requirements have been covered.

From the technical point of view, in this project, we followed specific guidelines to optimise its performance. More specifically the techniques that were used are:

- object pooling for items that we could reuse, like power-ups, monsters, players and bombs
- memory handling, by allocating as many variables we could on the stack and avoid to overload the heap
- caching, by declaring any component-type objects and initialise them on the Start or Awake methods, instead of doing it repeatedly on the Update function, or by storing the level's layout, instead of searching for each tile wanted.
- coroutines' use, for any function that needed to run in more than a single frame, or when we needed some function to be called after some time.

The screenshots below (Figure 30-Figure 37) were taken on a laptop with 64-bit Windows 10, a 4-core CPU at 1.80GHz, 8GB RAM and 2 GPUs with 8GB total GPU memory. An example of the usual condition of the current laptop, when having the Unity active, but not running the game, can be found in Figure 30.

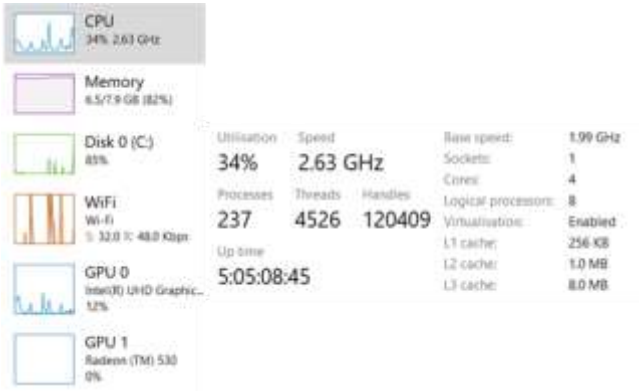

Figure 30: Laptop's condition before running the game

The statistics in Figures Figure 31-Figure 37 were taken when the game was running, but they cannot be indicative exclusively for the 'BombDudes' game, because this game was created as a part of a larger industrial project, that is also loaded on the background, while this game is running.

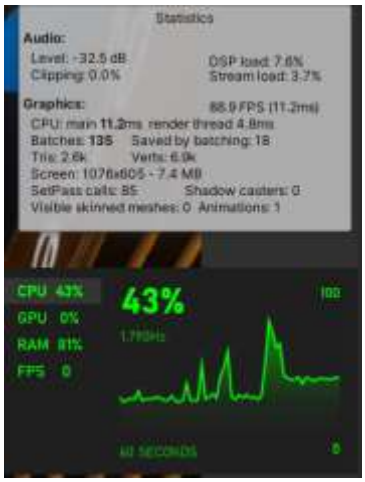

Figure 31: Statistics on Adventure mode

At a glance, we notice that there are trivial variations between the laptop's condition. It is reasonable to consume more CPU, considering the processes running. By filtering the resources used only for the Unity Editor, the numbers seem to be balanced. In general, comparing the framerates on the screenshots, the alterations are very small and insignificant. In particular, the absence of sudden peaks or drops means that there are no unusual renderings. The following snippets (Figures Figure 35, Figure 36, Figure 37) were taken while playing in multi-player mode, and performance seems to be balanced, regardless of the players' and bots' number.

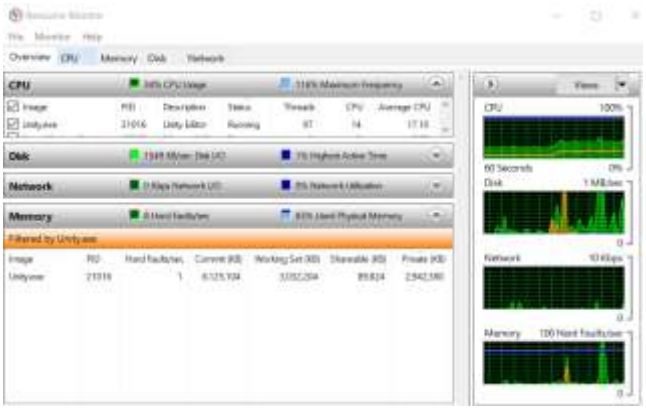

Figure 32: Windows Resource Manager filtered on Unity Editor

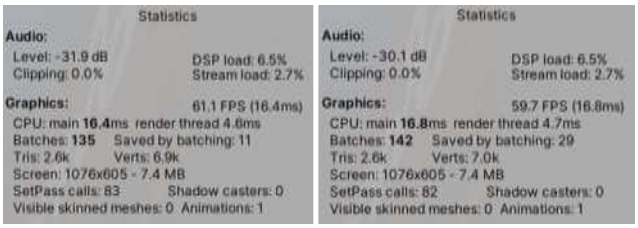

Figure 33: Statistics from Party mode with 1 bot (left) and 2 bots (right)

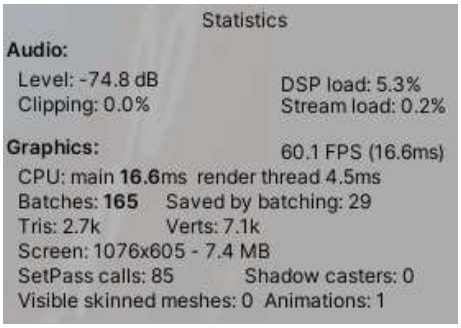

Figure 34: Statistics on Party mode with 3 bots

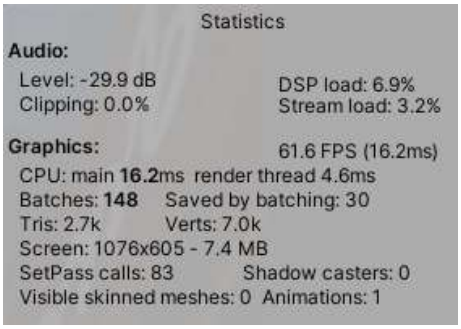

Figure 35: Statistics on multi-player with 2 players

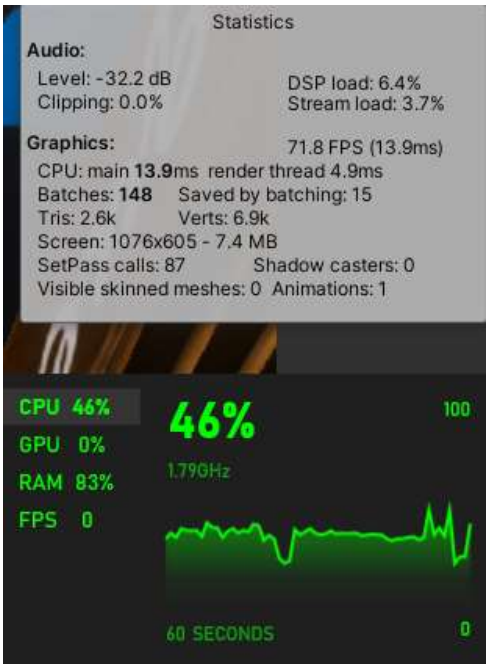

Figure 36: Statistics on multi-player with 2 players & 1 bot

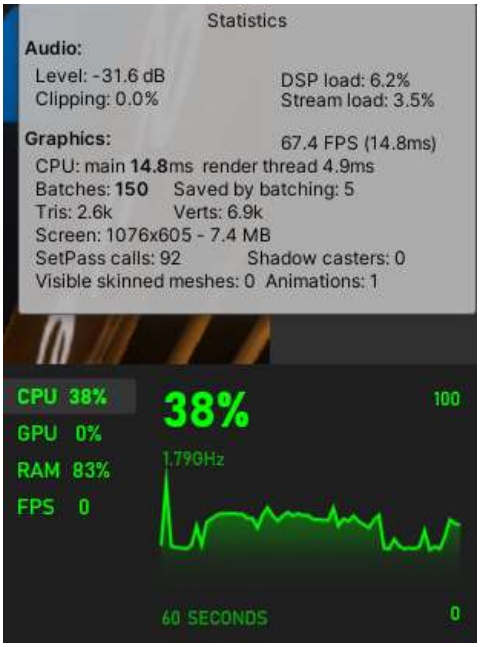

Figure 37: Statistics on multi-player with 2 players & 2 bots

Figure 38 is derived from the Unity Profiler, while the game was running in adventure mode. The average response time and framerate are justified by the fact that other processes are running in the background.

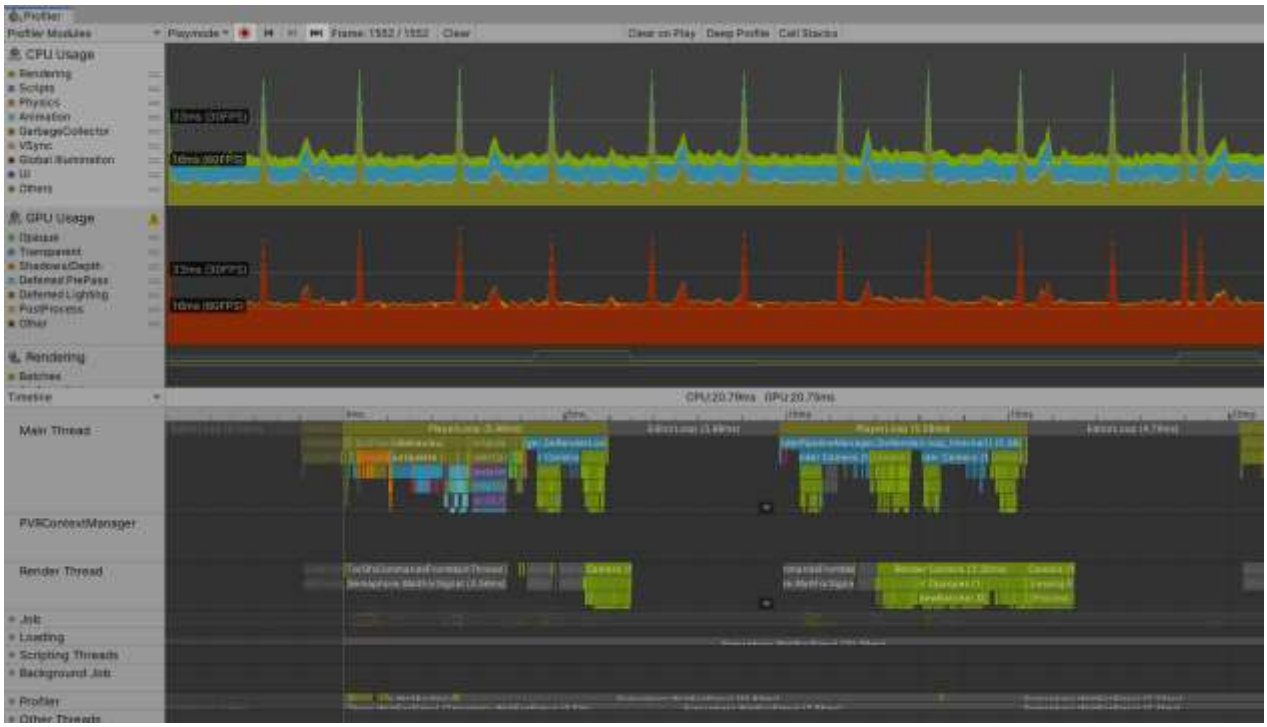

Figure 38: Profiler window when game is running

5. Conclusions

Given the limited research on game development methodology, we felt the need to demonstrate an indicative way of building a Bomberman-type game, presenting the developer's perspective and implementation process. We described the trail of actions from the beginning until the final complete game and we made an effort to justify any decision made. By reading Section 3, it becomes obvious that developers have to think in advance about possible future requests or ways to extend functionality. It is also important to write configurable code, that does not need script re-writing often, but can be modified from the Unity Editor's environment.

In general, this project allowed us to take initiatives, become more independent and discover better implementation techniques. As it is said 'Practice makes perfect' and in the gaming industry, everyday developers face new challenges. By being able to foresee features that may be requested and adjusting our scripts, we can save a great amount of refactoring time. To sum up, the final result is a fully developed game that will be included in a game that will be published in the near future.

Regarding possible future work, considering that this game was based on Super Bomberman built for Super Nintendo (known as SNES), the creation of various enemy types for the adventure mode with distinctive abilities would be interesting. For instance, in the original game, there are:
- 'bomb enemies' that roam around and explode every seven seconds,
- 'bomb-eater enemies' that look like an eating-bomb Pacman
- 'floating enemies' that can pass over breakable objects
- 'camouflage enemies' that partially disappear every few seconds
- 'coin enemies' that look like moving coins and reward the player with double points when they are killed.

Possible additions could also be 'boss levels', a progress map to offer a visual illustration of the player's level and more power-ups. The only limit is the developer's ideas and time.

REFERENCES

[1] Rogers Scott. 2014. Level Up! The guide to great video game design. John Wiley & Sons.
[2] de Lope Rafael Prieto, et al. 2015. Design Methodology for Educational Games based on Interactive Screenplays. *CoSECivi 1394, pp.: 90-101*.
[3] Milam David. 2013. Game Design Framework and Guidelines Based on a Theory of Visual Attention.
[4] Groot Kormelink Joseph, Drugan Madalina & Wiering Marco. 2018. Comparison of Exploration Methods for Connectionist Reinforcement Learning in the game Bomberman. DOI:https://doi.org/10.5220/0006556403550362.
[5] Mikael Fridenfalk. 2014. The design and implementation of a generic A∗ algorithm for search in multidimensional space. *2014 IEEE Games Media Entertainment, Toronto, ON, pp. 1-2*. DOI:https://doi.org/10.1109/GEM.2014.7048081.
[6] da Cruz Lopes, Manuel António. 2016. Bomberman as an artificial intelligence platform.
[7] Resnick Cinjon, et al. 2018. Pommerman: A Multi-Agent Playground. *arXiv preprint arXiv:1809.07124*.
[8] Segal, Omri Ben Dov Meirav, and Gal Katzhendler Varda Zilberman. Pommerman Agent.