



**INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA**

**Área Departamental de Engenharia Electrónica e Telecomunicações e de  
Computadores**



**Design methodologies to implement computer games  
(Super Milkman)**

**NUNO MARQUES CARDOSO**  
(Licenciado)

Trabalho Final de Mestrado para obtenção do grau de Mestre  
em Engenharia Informática e de Computadores

Orientador: Doutor Pedro Miguel Florindo Miguens Matutino

Júri:

Presidente: Doutor Nuno Miguel Soares Datia

Vogais: Doutor Hugo Tito Cordeiro

Doutor Pedro Miguel Florindo Miguens Matutino

**Setembro, 2019**





**INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA**

**Área Departamental de Engenharia Electrónica e Telecomunicações e de  
Computadores**



**Design methodologies to implement computer games  
(Super Milkman)**

**NUNO MARQUES CARDOSO**  
(Licenciado)

Trabalho Final de Mestrado para obtenção do grau de Mestre  
em Engenharia Informática e de Computadores

Orientador: Doutor Pedro Miguel Florindo Miguens Matutino

Júri:

Presidente: Doutor Nuno Miguel Soares Datia

Vogais: Doutor Hugo Tito Cordeiro

Doutor Pedro Miguel Florindo Miguens Matutino

**Setembro, 2019**



# Acknowledgements

I would like to thank:

My parents and brother for being present and supportive.

Carolina for understanding me and being there when I needed.

My friends for allowing me to decompress during this period.

Pedro Matutino and Diogo Cardoso for guiding me in the right direction and clearing all my doubts.

A special thanks to Diogo Sérgio Esteves Cardoso, even though not mentioned in the cover, he was also a supervisor for this work, alongside with Pedro Matutino, and I learned a lot from him.



# Abstract

Video games nowadays are the first form of entertainment, exceeding films and music. The process of creating a video game involves many areas of expertise. Starting from the definition of the architecture, there are also the physics and graphical engines, the art assets like 3D models or 2D sprites. Furthermore, some of these art assets are also animated.

The visual effects, the audio, the user interface, the mechanics, the camera and sometimes the artificial intelligence, are connected creating the gameplay system. Each one of these different areas requires different methodologies to be implemented. Herein, it is depicted a prototype of a video game, namely *Super Milkman*, where are described the different methodologies for each area, and discussed the option chosen. The prototype developed is playable, can be used as a guide for beginner developers of video games, and also can be extended through the addition of new game levels.

# Keywords

Video game, 2D video game, Unity, game engine, computer game.





# Resumo

Jogos de vídeo actualmente são a principal forma de entretenimento, superando filmes e música. O processo de criação de jogos de vídeo envolve muitas áreas de especialização. Começando pela definição da arquitectura, existem os motores de física e gráfico, os recursos artísticos, como modelos 3D e imagens 2D. Além disso, alguns destes recursos artísticos são animados.

Os efeitos visuais, o áudio, a interface de utilizador, as mecânicas, a câmara, e às vezes, inteligência artificial, juntos criam o sistema de jogabilidade. Cada uma destas diferentes áreas requer diferentes metodologias para ser implementada. É apresentado um protótipo de um jogo de vídeo, chamado *Super Milkman*, onde são descritas as diferentes metodologias para cada área e são discutidas as opções escolhidas. O protótipo desenvolvido é jogável, podendo servir de guia para programadores iniciantes na área dos jogos de vídeo e podendo também ser estendido através da adição de novos níveis de jogo.

## Palavras-chave

Jogos de video, Jogos de video 2D, Unity, motor de jogo, jogo de computador.



# Contents

<b>Abstract</b>	<b>vii</b>
<b>Resumo</b>	<b>ix</b>
<b>List of Figures</b>	<b>xiii</b>
<b>List of Tables</b>	<b>xv</b>
<b>Listings</b>	<b>xvii</b>
<b>List of Acronyms</b>	<b>xix</b>
<b>1. Introduction</b>	<b>1</b>
1.1. Objectives.....	4
1.2. <i>Super Milkman</i> , the video game .....	4
1.3. Outline .....	4
<b>2. State of the art</b>	<b>5</b>
2.1. Approaches.....	10
2.2. Development stages.....	11
<b>3. Video game development</b>	<b>13</b>
3.1. Concept stage .....	13
3.2. Pre-production stage.....	14
3.3. Production .....	15
3.3.1. Arts and graphics .....	17
3.3.2. Level layout .....	21
3.3.3. User interface (UI) .....	24
3.3.4. Audio.....	26
3.3.5. Programming.....	27
3.3.6. <i>Super Milkman</i> engine organization.....	41
3.4. Post-production .....	43
<b>4. Conclusions</b>	<b>45</b>
4.1. Followed methodologies .....	45
4.2. Future work .....	49
<b>Bibliography</b>	<b>51</b>
<b>A. One page design document</b>	<b>i</b>
<b>B. Design document</b>	<b>ii</b>



# List of Figures

1.1: Growth of forms of entertainment, extracted from [2].....	1
1.2: Traditional project architecture. ....	2
1.3: Application architectures.....	3
1.4: Video game architecture.....	3
2.1: Video game dissection.....	5
2.2: Pac-Man game, extracted from [7].....	6
2.3: Super Mario Bros. 1985 [8], parts of level 01-01 extracted from [9]. ....	7
2.4: Flipbook, kineograph by John Barnes Linnett, extracted from [10]. ....	7
2.5: Frames from the short film “ <i>Steamboat Willie</i> ” by Walt Disney and Ub Iwerks in 1928.....	8
2.6: Skeletal animation (3D, extracted from [12], and 2D).....	8
2.7: Animation using bones.....	8
2.8: Explosion particle effect example [14]. ....	9
2.9: Example of a third party shaders package in the game Minecraft [15].....	9
2.10: Super Mario World [8] with a side-scroller camera.....	9
2.11: The Legend of Zelda [16] with a top-down camera.....	10
3.1: Game characters concept art.....	13
3.2: Playable character animator graph. ....	19
3.3: Enemy animator graph. ....	20
3.4: Playable character sprite.....	20
3.5: Keyframe using bones. ....	21
3.6: Rectangular asset as terrain, extracted from [42].....	21
3.7: Sprite shaped as terrain, extracted from [43]. ....	22
3.8: Tiles as terrain, extracted from [44]. ....	22
3.9: Terrain tiles.....	22
3.10: Rule tile. ....	23
3.11: Different tilemaps that compose the level, in perspective.....	23
3.12: Screen transition graph.....	25
3.13: Heads-Up Display (HUD).....	26
3.14: <i>Super Milkman</i> diagram. ....	31
3.15: Actor controller class diagram, first iteration.....	32
3.16: Actor colliders and boundaries.....	33

3.17: Actor controller class diagram, second iteration. ....	33
3.18: Weapon abstract class and its children. ....	34
3.19: Final iteration of the ActorController class. ....	35
3.20: Ability class diagram. ....	36
3.21: HurtOnContact class diagram. ....	36
3.22: Enemy ground checker. ....	37
3.23: Enemy field of view. ....	38
3.24: End level token script in the Unity inspector. ....	38
3.25: Teleport door 001 A in the Unity inspector. ....	39
3.26: Slide door script in the Unity inspector. ....	40
4.1: Project branches feeding and merging to complete the project. ....	47

# List of Tables

3.1: Project plan.....	14
3.2: Project tasks.....	16
3.3: Graphics iterations.....	18
3.4: Animation list for each actor. ....	19
3.5: Prototype level iterations.....	24
3.6: Screens and their purpose.....	25
3.7: Screens progress. ....	26
3.8: <i>Super Milkman</i> basic audio list. ....	27
3.9: Component priority table in development.....	31
3.10: Actor and its inherent children iterations. ....	32
4.1: Project plan example. ....	45
4.2: Detailed tasks and their respective development branch.....	46





# Listings

3.1: Level and World scriptable objects. ....	30
3.2: Code that checks if a boundary overlaps a colliders. ....	33
3.3: Code folder structure. ....	41
3.4: Unity project folder structure. ....	42
3.5: Level scene organization. ....	43
4.1: Project organization examples.....	49



# List of Acronyms

<b>API</b>	Application Programming Interface
<b>DLC</b>	Downloadable Content
<b>GDD</b>	Game Design Document
<b>GPU</b>	Graphics Processing Unit
<b>HUD</b>	Heads-Up Display
<b>N/A</b>	Not Applicable
<b>NPC</b>	Non-Playable Character
<b>SFX</b>	Sound Effects
<b>UI</b>	User Interface
<b>VFX</b>	Visual Effects



# 1. Introduction

The video game industry has been the number one form of entertainment for almost two decades, surpassing both film and music industries, and has been growing every year. There's a huge audience to be reached when developing video games [1], the growth in this industry is higher than other entertainment industries combined and it continues growing, as depicted by the profit growth in Figure 1.1.

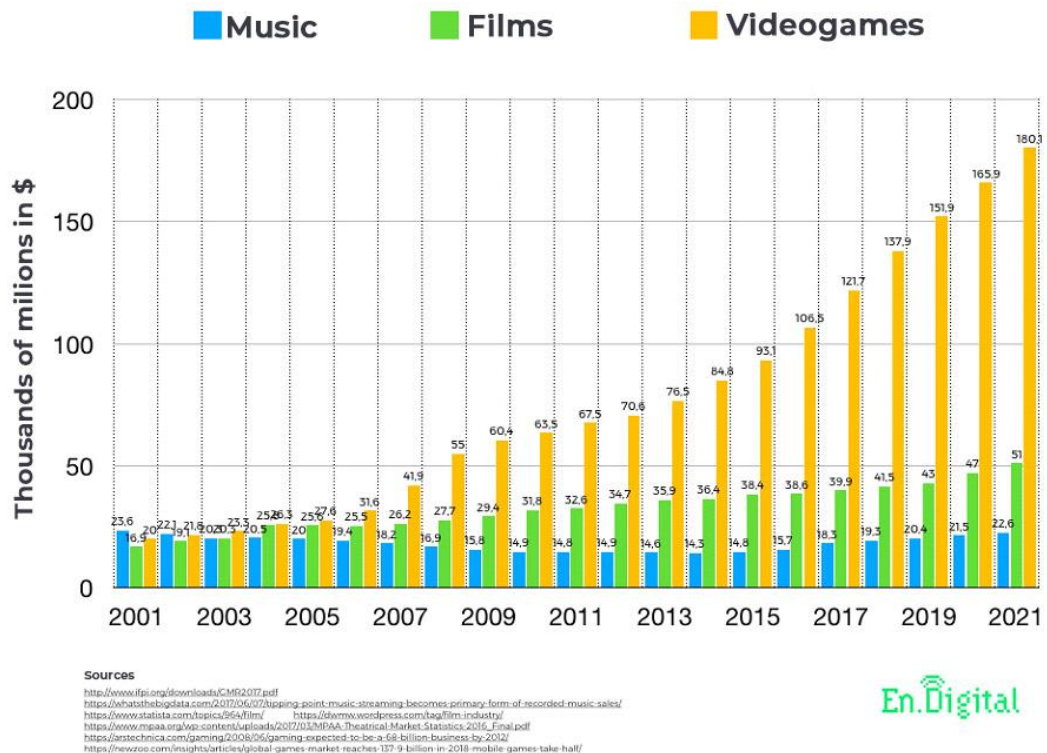


Figure 1.1: Profit growth of forms of entertainment, extracted from [2].

The development of traditional applications and video games follow different approaches, the workflow is not the same and crosses different areas of expertise. Traditional applications may have, or not, a graphic designer to embellish the user interface. These applications also have software designers that create the solution for the system that will be implemented, for that they use different techniques and design patterns to create a robust software architecture, that can be easily worked and maintained. Common software architectures use a base structure, as depicted in Figure 1.2, and their implementations only differ in small parts, as depicted in Figure 1.3.

Although different application types have similar architectures, video games differ in that aspect, the system architecture is approached in a different manner, as is presented in Figure 1.4. Video games are composed of components that can have several blocks of logic, in traditional applications the layers usually have a single responsibility. Components, in video games, can have multiple blocks until they reach the presentation layer. An example is a game that has roads and traffic lights, the traffic light components need a system to coordinate them, just like in real life. This small system can be

considered a small application. If autonomous cars are added to the game, then the car component requires blocks to drive in the road and respect the traffic lights, to not collide with other cars and to not run over pedestrians. The car component will have at least four logical blocks that can be considered as four independent small applications.

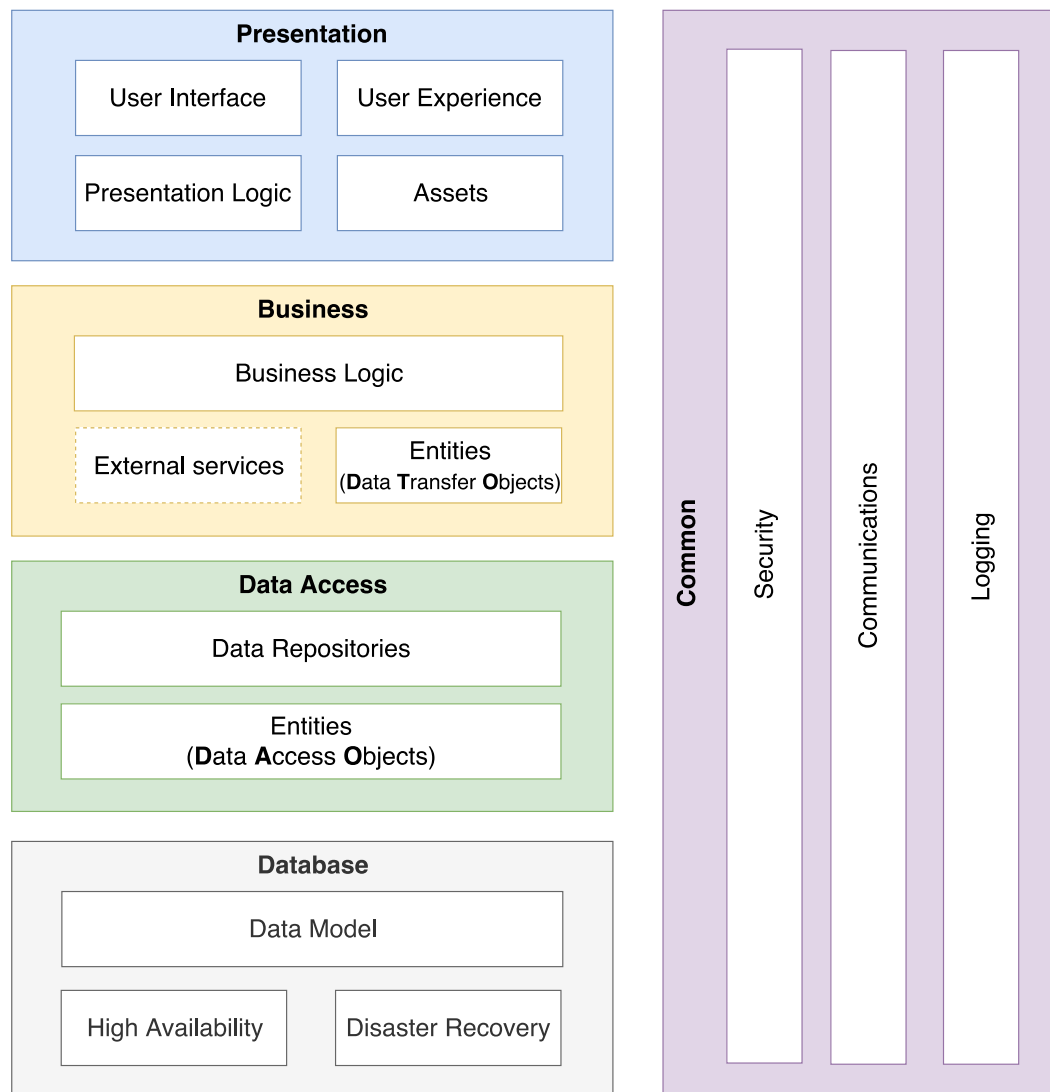


Figure 1.2: Traditional project architecture.

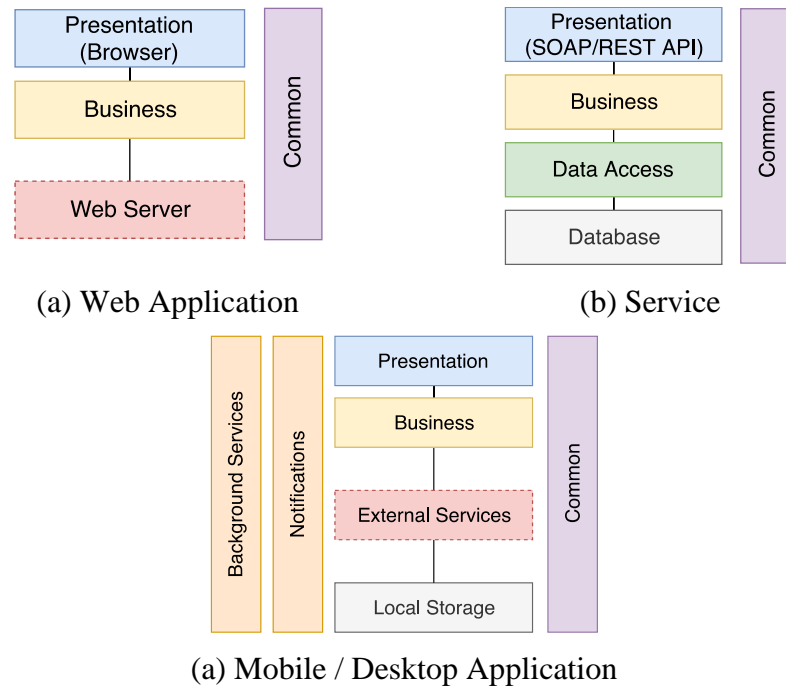


Figure 1.3: Application architectures.

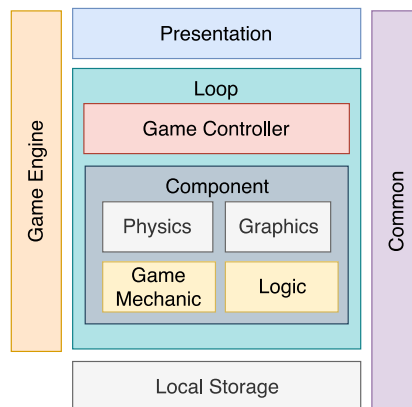


Figure 1.4: Video game architecture.

Video games, being a multimedia experience, cross many areas of expertise like animation, graphics, sound/audio, level design, programming, and others. The designers that conceptualize and materialize ideas are an essential part of the development process. For example, a game designer envisions the whole gameplay experience and storyline, on the other hand, a graphical designer conceptualizes the visual aspect of characters, backgrounds and User Interface (UI), etc.

Video games are a form of entertainment and are designed to engage the player on the storyline and gameplay. This work details the different methodologies that were used in different areas of expertise to create a 2D platformer video game. The result of that was the development of a video game, namely as *Super Milkman*.

## 1.1. Objectives

The goal of this project is to create a methodology for each disciplinary branch that composes a video game. Since every branch covers a different area of expertise, these methodologies differ from branch to branch. To accomplish this goal, the production and development of a video game were studied and the followed methodologies were detailed in this work.

In order to develop the video game for this project some objectives were established: *i*) conceptualize an original and well-established game idea; *ii*) write the game storyline; *iii*) create the Game Design Document (GDD); *iv*) develop gameplay, with responsive character controls; *v*) create visual and audio effects appealing to the player; and *vi*) organize the project and code structure to facilitate adding new components to the game.

## 1.2. *Super Milkman*, the video game

*Super Milkman* is a 2D platformer video game that was developed in this project.

The player needs to surpass challenges and collect items that will grant him access to the next level. The levels are divided into zones, having each zone a distinct theme, enemies and mechanics. The storyline progresses with the player clearing every level. To clear a level, the player needs to collect token items and a key that will unlock the door that leads to the next level. The items are spread around the level, some are hidden and others are protected by enemies.

## 1.3. Outline

The remaining document is organized in three additional chapters. Chapter 2, describes the state of the art, giving an overview of what makes a video game. Chapter 3 is divided into four sections that represent the development stages: concept, pre-production, production and post-production, and details how to start the development of a video game, the implementation of each component and the end of the development. Finally, Chapter 4 summarizes the main conclusions, the followed methodology and points to possible future work.



## 2. State of the art

Usually, the development of a video game requires the following lineup: *i) game designer*, that creates the storyline, level design and idealized the whole idea of the final game; *ii) graphics designer*, that creates all the 3D models or 2D sprites [3], graphics and animations; *iii) writer*, to create all the dialogues and manuals; *iv) sound designer and composer*, that create the music and sound effects; and *v) programmer*, that scripts the whole game. In projects with a smaller budget and less manpower one person may have to take several of these lineups, in contrast, larger budget projects have larger dedicated teams for each. In [4] a section is used to detail each lineup individually.

Being a multi-disciplinary project, the structure of a humbler video game is more complex than a simpler application. As usually, in software development the system is composed of several pieces. Herein, it is considered the following main structure of a video game: *i) storyline*; *ii) rules*; *iii) characters*; *iv) game engine*; *v) game loop*; *vi) game mechanics*; and *vii) level design*. The diagram in Figure 2.1 depicts the main components that constitute a video game.

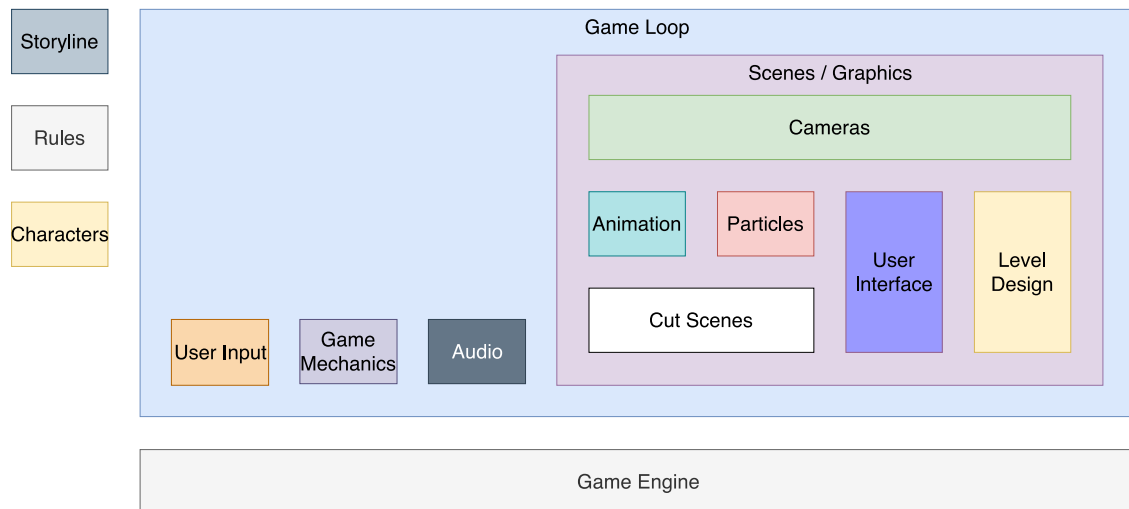


Figure 2.1: Video game structure.

The storyline is the most important topic in the development process since everything will be created around it, such as level design and game mechanics. This narrative is a sequence of events that the player will experience, as every narrative, the storyline has a beginning, a middle and an ending.

The player experiences the storyline by playing the game, the game executes inside an infinite loop namely the *game loop*. This loop always performs the same steps, three of those main steps are herein considered: *i) process player and/or network input*; *ii) update game state*; and *iii) generate output*, like graphical updates, play audio, show dialogue messages, and many others. The loop executes several times per second while the game is running. Nowadays, game engines usually have two main loops that execute independently, one for the graphics and other for the physics [5].

The game is not defined by the loop, but by rules. Rules can be categorized into two types, rules that the player follows and rules that the game follows. For the first type, player's rules, are achieved by validating the player actions before executing them, if the action is not defined within the rules, then the action is discarded. An example of this type of rules can be observed when playing the game Pac-Man [6], depicted in Figure 2.2, the player can only move in one of four directions at a time, or less if a wall is near. The second type, game rules, establish conditions that are checked inside the game loop. These conditions are the identity of the game and define what can and can't be done. Using Pac-Man, again as an example, the level is only completed when all the dots have disappeared, having this condition validated means that another level can be loaded and then the game can start again.



Figure 2.2: Pac-Man game, extracted from [7].

A concretization of rules are the game mechanics, force restrictions upon the player and the game, they consolidate the game and are an essential component of the game design. Using the Pac-Man example, the main mechanic in Pac-Man is turning the monsters into weaklings by eating the big dots in the level, doing that the player can defeat the monsters gaining points.

The levels can be created after the rules and mechanics are defined. For a greater gameplay experience, some thought must be taken when designing the levels, an example of level design can be found in the tutorial level 01-01 of Super Mario Bros (1985) [8]. In this level, the player learns the basic mechanics of the game by himself, without a user manual or in-game indications. Figure 2.3 (a) depicts the first part of the level, with components like a question mark blocks and an enemy. The question mark blocks reward the player, however, the enemies cause the player to take damage. In the same scene, there are pipes indicating that the player can go down, but some don't work. The pipes functionality, besides level decoration, is warping the player to another area in the level, Figure 2.3 (b) depicts a working pipe that warps the player to an underground area, depicted in Figure 2.3 (c). Having pipes that work and others that are decoration, indicates to the player that the level isn't as linear as it appears. These simple examples allow the player to learn the important mechanics and elements of the level without external help.

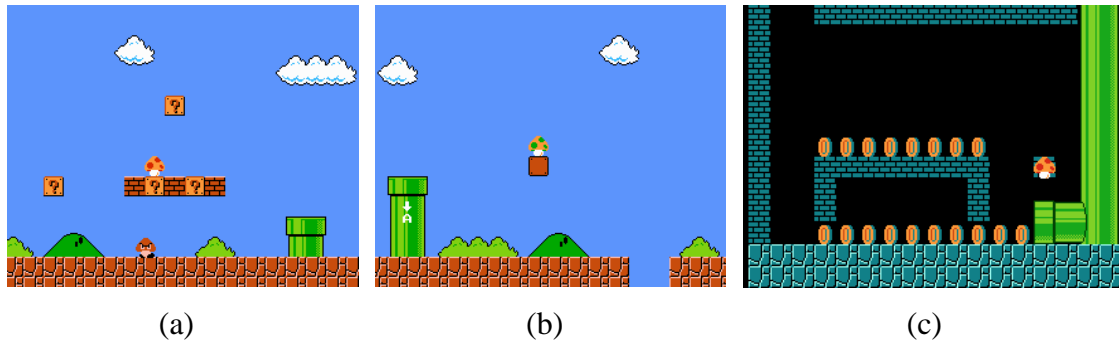


Figure 2.3: Super Mario Bros. 1985 [8], parts of level 01-01 extracted from [9].

Characters are the ones that perform actions and interact with the game. There are two types of characters, the non-playable and the playable. The Non-Playable Characters (NPC) have a defined set of actions, like moving and interact with the player. Usually, in platformer video games, the enemy characters are considered NPCs, since their interaction with the player is attacking or just damaging it, and they have a specific movement pattern.

The graphical component is what the player sees and interacts with, is where the game materializes itself. Other components make use of the graphical component such as user interface, cameras, animations, particle effects, cutscenes, terrain and others.

The player interacts with the game by the user interface, allowing navigation, it also displays information to the player, such as errors, objectives, dialogue/text, game state, events and the pause screen.

Animations make the object appear alive and fluid. Animations can be applied to all elements that can be seen by the player, like user interface buttons, character models or sprites, backgrounds and world objects. Frame by frame animation and skeletal animation are the two commonly used techniques to animate in video games.

The frame by frame technique has been used for a long time, can be seen in real life with flipbooks, as depicted in Figure 2.4, where each page is a different image (frame) and when the pages are flipped they create an animation in our brain. This technique takes a lot of human power and time to create because every frame must be drawn individually. The first animated films used this technique, as depicted in Figure 2.5.

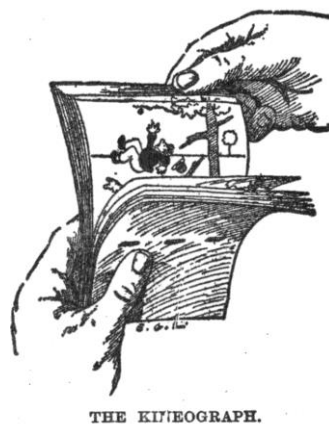


Figure 2.4: Flipbook, kineograph by John Barnes Linnett, extracted from [10].

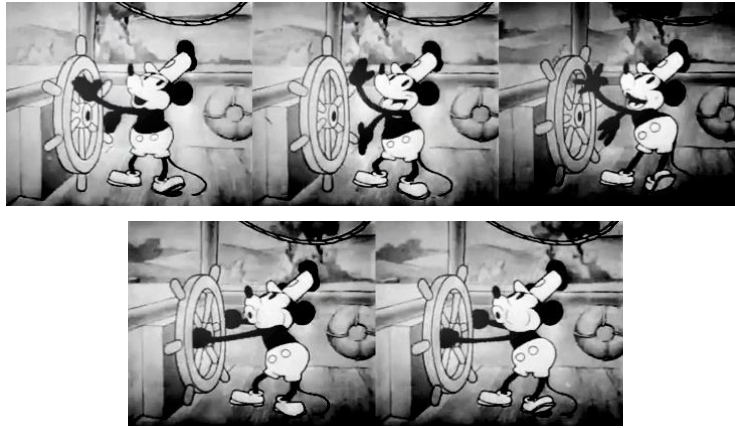


Figure 2.5: Frames from the short film “*Steamboat Willie*” by Walt Disney and Ub Iwerks in 1928.

The skeletal technique uses a series of bones as structure and can be performed using 3D models or 2D images, as depicted in Figure 2.6. The animation process works by moving the bones to a determined position in space and mark it as a new keyframe. Playing all the keyframes in sequence creates the animation, an example is depicted in Figure 2.7. As simple as the process can appear, the difficult part is creating a smooth and real-world accurate animation, like animating a humanoid character. An animator using this technique must be knowledgeable about what he’s animating, else the animation won’t appear fluid and will feel unnatural, an in-depth explanation is detailed in [11].

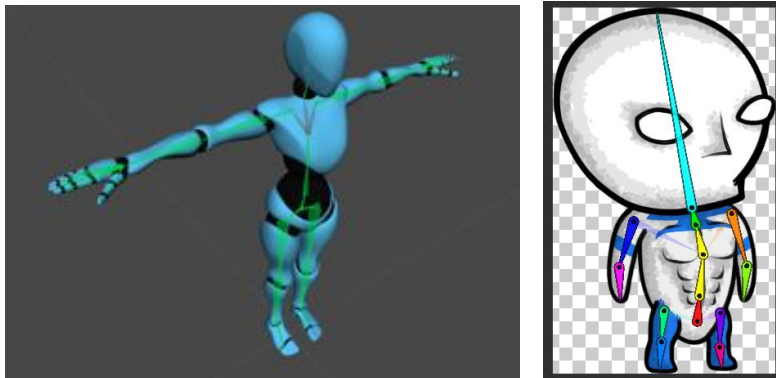


Figure 2.6: Skeletal animation (3D, extracted from [12], and 2D).

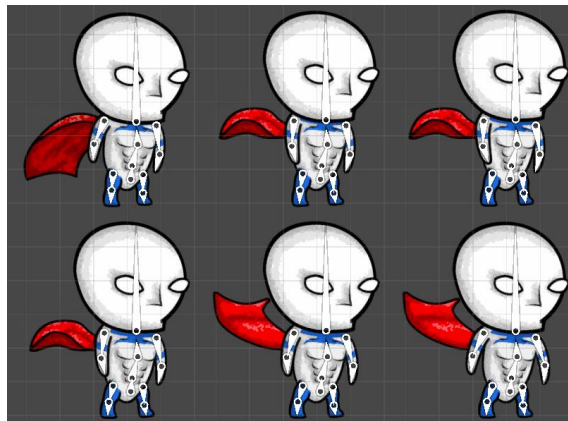


Figure 2.7: Animation using bones.

The Visual Effects (VFX) transmit action to a scene. These effects are often used in power-ups, character effects, explosions, fire, fluids, destruction, etc. Some types of visual effects are particles and shaders. Particles don't have a well-defined shape and can represent moving liquids, clouds, flames, and smoke. To create a particle effect, many 2D images are generated and animated, as can be observed in Figure 2.8. Shaders are scripts that contain calculations and algorithms, that calculates the colour of each rendered pixel, an example is depicted in Figure 2.9, these scripts run directly in the graphics processing unit (GPU) and are integrated into the rendering pipeline. A more detailed explanation can be found in the blog article [13].

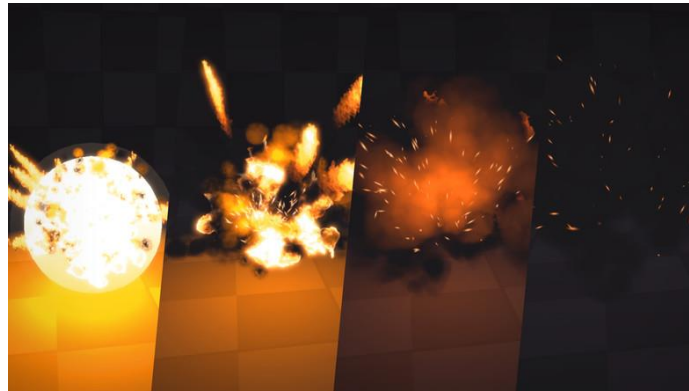


Figure 2.8: Explosion particle effect example [14].



Figure 2.9: Example of a third party shaders package in the game Minecraft [15].

The camera is the window into the game world. Camera position and perspective depends on the game and its genre, for example, the side-scroller game depicted in Figure 2.10 places the player by the side, other genres can use a top-down view as depicted in Figure 2.11. The camera position can be static, following the playable character in the middle of the screen or dynamic, changing with the player movement and level layout.



Figure 2.10: Super Mario World [8] with a side-scroller camera.





Figure 2.11: The Legend of Zelda [16] with a top-down camera.

Cutscenes are usually used to break the gameplay, and the player doesn't have control over his character. Having cutscenes transmits to the player important details of the storyline by focusing the camera on the details of the scene. Usually, at the beginning of a game, there's an introductory cutscene that sets the initial storyline, as well as give information of what is going on in the game, in form of animation or video. Another use of cutscenes is for dialogues where the characters interact with each other, this can be compared to a play or a movie scene. Cutscenes are very useful when the player triggers some event, for example, when the player goes to a bridge and in the storyline that the bridge falls, having a cutscene here makes sense, the player movement is disabled, the camera can have a cinematic movement being focused on the bridge and some story dialogue can be displayed to inform the player about the event taking place.

A video game wouldn't be a multimedia experience without audio. The audio sets the atmosphere of the scene, by having background music and characters sound effects the action presented to the player is highlighted.

The act of playing a video game requires some sort of input device, the most common are: touchscreen, gamepad, console controller, mouse and keyboard. These input devices are platform dependent.

All the previous components need a foundation, that foundation is the game engine. The engine provides core functionalities for creating video games, like rendering 3D and/or 2D graphics, physics simulation, collision, sound, animation, scripting, and others. In [5] is described all the different components in a game engine and their complexity.

## 2.1. Approaches

There are no standard approaches that work for every game development, usually, every team has their own ad hoc development process.

Project management in game development is very similar to traditional software engineering project but with several multi-disciplinary branches that join their work at some point in time. The most used development approaches for games are agile and waterfall.

The waterfall approach [17] is very linear; each step must be concluded before the next one starts. The development is divided into phases: requirements, design, code and test, verification, finished product, and maintenance.

The agile approach [18] is based on four phases that are repeated and iterated until the video game is fully complete. These four phases are the following: discover, design, develop and test. The main idea is to do small features in small periods of time, each iteration can be seen as a short project itself.

A commonly [19] used document is the Game Design Document (GDD), describing the whole game and what it aims to be in the future. All the ideas that are planned to be in the game are detailed in the document. The document also serves as a guide while the game is being developed and must be updated frequently to reflect the actual state of the game and what it aims to be. After reading this document the reader should be able to understand the whole game, how to play it and its main objective.

An optional document is the Technical Design Document (TDD), that is a more technical document than the GDD and it is targeted to programmers. In this document are presented a detailed list of all the features and game mechanics, as well as the reasons for some choices software-wise, like application logic and artificial intelligence.

## 2.2. Development stages

The development of a game is usually divided into five stages: *i)* concept; *ii)* pre-production; *iii)* production; *iv)* vertical slices; and *v)* post-production.

The first stage is where the concept of the game is developed. A market analysis is made to assess if the concept is viable or not. Design goals are created and the game price is fixed so the next stages can be managed accordingly, and teams created.

After the concept is defined, the pre-production starts and the initial documents are created, the main objective is to create the Game Design Document, GDD. Usually, to create this document, several iterations over the concept and initial design document are required. Before production starts, this document must detail very clearly all the ideas and options. Typically, some ideas are prototyped to analyse if they are worthwhile to be implemented in the production stage.

In the production stage, all the lineups are involved. In this stage, the user interface is defined, alongside the creation of the sprites/3D models, the sound effects, and music. Finally, the programmers start to code all these components. Meanwhile, the game designers continue to change the overall idea of the game with new features, or details, or they just redo an idea from scratch.

Vertical slices although not considered as typical stages, are still an important milestone in the development process. Vertical slices represent the progress made across all the components of a project, in this case, the video game. The main goal of these slices is to demonstrate the progress made so far. The main ones are alpha, beta and gold master.

Alpha, a very rough playable game with still a lot of bugs and not polished. Not all the features are yet implemented, only the major features are implemented. Beta, still rough but almost release ready, still some bugs to polish. All the features are implemented. Gold Master, ready to be distributed, all major bugs that prevent the normal flow of the gameplay are fixed and the game is ready to be played.

In the post-production stage, the last one, user bug reports are reviewed and fixed, in some cases, some unreleased content is completed and added to a new version, as downloadable content (DLC) and published.





## 3. Video game development

The development of the video game *Super Milkman* was divided into four different stages, as previously detailed in Chapter 2.2: *i)* concept; *ii)* pre-production; *iii)* production; and *iv)* post-production.

The development starts with the concept stage where everything is sketched, thought and planned. After the game has been roughly thought, the pre-production follows. This is the stage where decisions that influence the future development will be taken, the game starts to take shape. Following is the main stage, production, that implements all the ideas from the previous stages and concretizes the video game. After the game is completed, the next stage is the post-production where some features are added and bugs are fixed. The following sections will detail the development process of *Super Milkman*.

### 3.1. Concept stage

This is the no consequence stage; everything can be changed at any time. This allows for a brainstorm of ideas in order to choose the best ones for the production stage. At the end of this stage some aspects of the game must be defined: *i)* budget; *ii)* target audience; *iii)* game genre; *iv)* game storyline; *v)* gameplay ideas.

A helpful way to aggregate all these aspects is writing the GDD, as referred at the state of the art, this document will accompany the project until the end. At this stage, the GDD must have the general idea of the game, in just one page.

The concept stage in the development of *Super Milkman* started with the writing of the one-page GDD (depicted in Appendix A) describing the storyline, the gameplay roughly explained and all the other aspects mentioned above. A project plan was also made, and the goals have been prioritized. Table 3.1 depicts the plan and the estimated duration of each task. Also, some conceptual art was drawn, specifically the playable character, an enemy and some world objects, as depicted in Figure 3.1.



(a) Playable character; (b) NPC, enemy.

Figure 3.1: Game characters concept art.

Task	Priority	Duration
Game storyline and gameplay idea	High	Two weeks
Write the GDD	High	Two weeks
Conceptual art	Low	Two weeks
Learn and study the game engine	High	Six weeks
Placeholder artwork	High	One week
Level terrain	Medium	One week
Characters	High	Six weeks
Main mechanics	High	Four weeks
World objects	Med	Three weeks
User interface	Medium	One week
Final artwork	Low	Two weeks
Sound and music	Low	Two weeks

Table 3.1: Project plan.

## 3.2. Pre-production stage

The previous stage started the GDD, the pre-production stage requires a more elaborated and complete version before the production stage begins. The GDD is a living document and must be updated frequently or else will become obsolete and outdated as the development progresses. All questions related to the game must be able to be answered just by reading the GDD.

The GDD of *Super Milkman* (depicted in Appendix B) is divided into the following sections: *i)* storyline, where all the story is explained and the characters are elaborated; *ii)* playable character, its movement and mechanics; *iii)* gameplay, details the objectives of the game, how the levels are designed and how to navigate inside the game; *iv)* game world zones and their enemies; *v)* feel of the game, how the camera works and where the music and sound effects must be placed; *vi)* core mechanics explained; and lastly, *vii)* enemies and their attacks. All this information added to the GDD will be extremely important in the production stage, as every detail about the game is located in one document and can be easily accessed.

Also, in this stage, the game engine needs to be defined, this choice will condition the rest of the development workflow. There are two alternatives, an existing engine or creating an engine from scratch. If the choice is an existing game engine, a lot of work is already done. In this case, the developers can be more focus on the game story and gameplay, since there's no waiting between idea and execution. The development can be gameplay focused. On the other hand, creating a game engine from scratch is more challenging and opens the possibility to create custom features. Until the game engine is created the scripting component of the game is on hold, meaning that the development time must be extended. If the development time is short, gameplay centred, with a tight budget, and the objective is to create a game as fast as possible, then using an existent one is the better choice. There's no right choice when choosing between these two

alternatives, both have their pros and cons. If the goal is to create something new and the development time is vast, then it could make sense to create a new engine from scratch. Another factor to consider is the knowledge that is required to create a new engine versus an existing one. To create a new engine the developers need to have extensive knowledge in mathematics, physics, graphics, implementing software architectures and support all the different video game target platforms.

Herein, the purpose is to implement a video game and not to create an engine, so an existent engine will be used. The two most used engines nowadays [20] are Unreal Engine [21] and Unity [22].

Unity allows deploying the game to over 25 platforms, including Windows, macOS, Linux, Nintendo 3DS, Nintendo Switch, Xbox One, PlayStation 4, iOS, Android and many more [23]. This engine can be used to develop 3D or 2D games. The minimum specifications to run the editor and some games created in it is very budget-friendly, lower-end computers can run it [24]. Advantages: *i)* it's free to use, perfect for beginners [25]. Free when the generated annual revenues or raised funds less than \$100k; *ii)* uses a high-level programming language, quicker to program than with lower-level languages; *iii)* allows rapid prototyping. Disadvantages: *i)* proprietary engine, developers can't tweak the engine because is vendor locked.

Unreal Engine 4, like Unity, allows developers to deploy to many platforms, but not as many. Many large budget video games that have high demanding graphics [26] use this engine. Advantages: *i)* a very powerful engine, in the graphics department Unreal Engines is great with lighting, visual effects and textures, great for creating photorealistic games; *ii)* the engine is open-source [27]; *iii)* has a visual scripting system (Blueprints) enabling quick prototyping for non-programmers. Disadvantages: *i)* price, 5% royalties [28]; *ii)* uses a low-level programming language.

The choice for this project was the Unity engine since it offers a modern fully-featured managed programming language, C# [29], making it easier to program and straight to the objective that is the creation of a video game. Furthermore, other factors for choosing Unity was its wider active community and better-written engine documentation.

Like the previous stage, pre-production is also an exploratory stage where prototypes are built and decisions can be made to transition new ideas to the production stage. In *Super Milkman* this exploration process was learning the engine API [30] and its editor, and also to get used to game development workflow and project structure.

### 3.3. Production

This is the main stage of the development; all the exploratory work was already done in the previous stages and with all the goals set, the video game can start to be developed.

The approach used in the development is iterative and incremental, every iteration improves upon the previous one. Each iteration must be doable, have clear goals and the result should be evaluated. A good example can be explained with an iteration over the playable character, assuming this component is not complete yet, the sprites, animations, and sound are still missing, but the behaviour is already programmed, future iterations can add these missing parts to the character finalizing it.

To take advantage of this iterative approach, the production stage of *Super Milkman* was divided into five independent branches: *i*) arts and graphics; *ii*) level layout; *iii*) user interface; *iv*) audio; *v*) programming.

Each branch can be developed independently since crucial pieces can be placeholders until they are replaced when available, this way the development flow continues without waiting for other branches to complete their part.

Before all branches begin their development, global goals must be set. The GDD can provide that help, but it doesn't specify what each branch is assigned to do. Table 3.2 provides guidance to complete each task, achieving completion may take several iterations in the different branches, culminating at the end with the finished component.

Task	Subtasks	Branch
Screens navigation	Buttons and backgrounds	Arts and graphics
	Screens	User interface
	Navigation	Programming
	Screen music and SFX	Audio
Actors { Playable character and enemy }	Sprites and animation	Arts and graphics
	Animation scripting	Programming
	Mechanics	Programming
	Movement	Programming
	SFX	Audio
World objects	Sprites	Arts and graphics
	Mechanics	Programming
	SFX	Audio
Level	Level controller	Programming
	Death zone scripting	Programming
	Death zone placement	Level layout
	Actor placement	Level layout
	World object placement	Level layout
	Camera	Level layout
	Heads-Up Display (HUD)	User interface
	HUD scripting	Programming
	Create tiles	Arts and graphics
	Design the level	Level layout

Table 3.2: Project tasks.

### 3.3.1. Arts and graphics

This branch creates the graphic component of the videogame, which is what the player sees. For other branches to start their work, they may need some art assets. The first iteration created the placeholder assets, usually, they are basic shapes or assets from another game.

The following iterations refine the placeholders and improve upon the basic shapes. This can be accomplished by drawing digitally the components in an image editor program like GIMP (free) [31], Inkscape (free) [32] or Adobe Photoshop (paid) [33]. The drawing must follow the GDD guidelines. Another way is to acquire sprites and images already made, this can be done by outsourcing or getting them for free on websites like OpenGameArt [34] or Itch.io [35].

In *Super Milkman* this branch was divided into four iterations: *i*) placeholder; *ii*) paper drawing; *iii*) digital drawing; and *iv*) animation. The first three iterations were for sprites development and the progress can be observed in Table 3.3, the detail increases at each iteration.


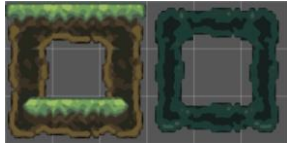










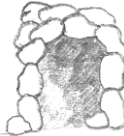

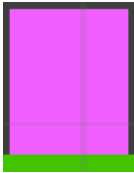














Object	Placeholder	Paper drawing	Digital drawing
Terrain (digital outsourced from [36])		N/A	
Playable character			
Enemy			
Collectable token			
Teleport door			
Final door			
Button			
Push block			
Chest	N/A		
HUD			
Background (outsourced from [37])	N/A	N/A	

Table 3.3: Graphics iterations.

The final iteration is animation. Not every component needs to be animated, in this case, the only components animated were the playable character and the enemy. In order to keep track of what animations are required, a list as depicted in Table 3.4 can be created. Having the animations written down prevents the creation of unnecessary animations.

For both actors were created conditional graphs, called animators [38], each edge of the graph has a condition making only possible to transition to another node when the condition is met. The animator graphs use all the animations listed in Table 3.4, the graphs are depicted in Figure 3.2 and Figure 3.3.

Character	Animation name
Playable character	Idle
	Walking
	Jumping
	Crouch
	Attacking
	Crouch
	Falling
Enemy	Idle
	Walking
	Targeting player
	Attacking
	Fainted

Table 3.4: Animation list for each actor.

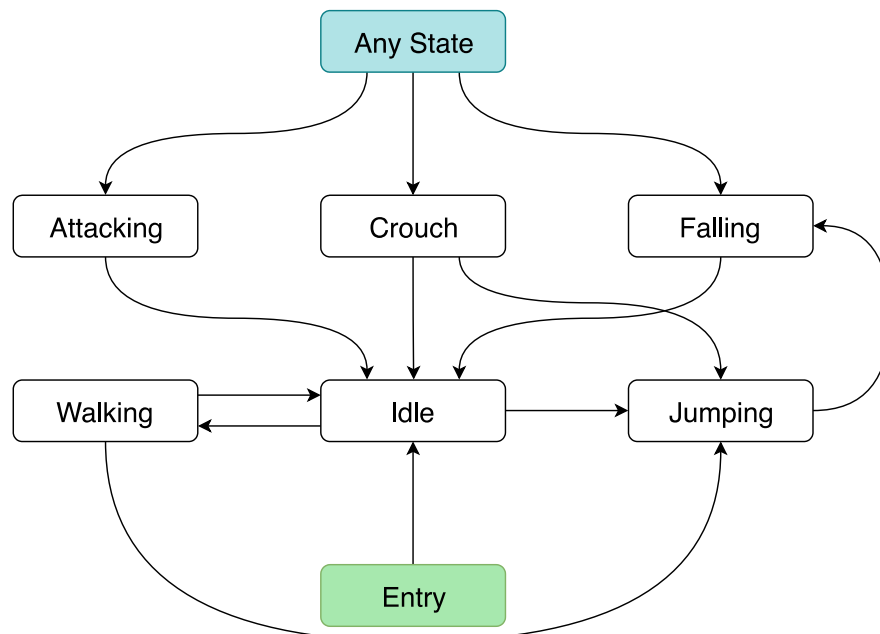


Figure 3.2: Playable character animator graph.

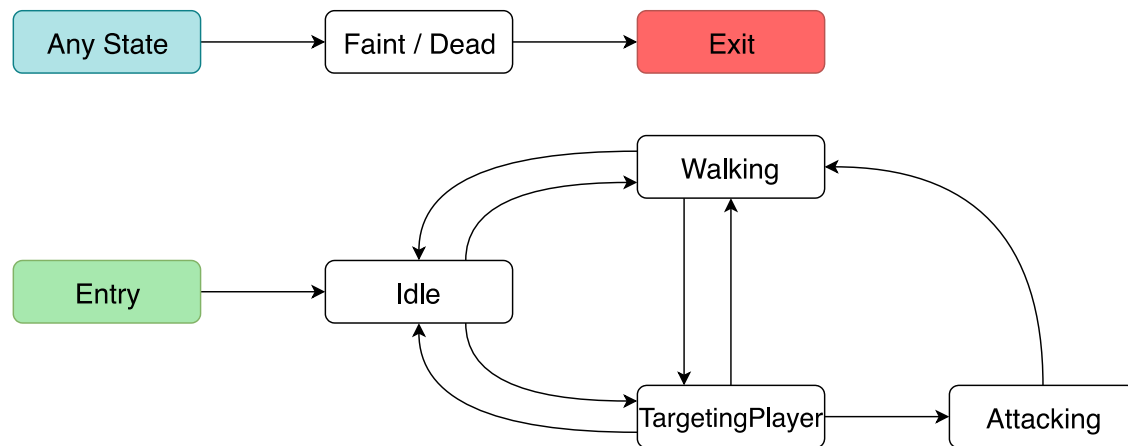


Figure 3.3: Enemy animator graph.

In *Super Milkman*, the animation process was performed using a technique called skeletal animation, described in Chapter 2. For this technique, the character sprite needs to be split by its members, as depicted in Figure 3.4 (a), this way the bones can be attached to them. The bone structure over the sprite is depicted in Figure 3.4 (b). The animation process starts by moving the bones to a position and creating a keyframe. An animation is a set of keyframes that are played in sequence, an example keyframe of the attack animation is depicted in Figure 3.5. An extensive explanation on how to attach the bones to the sprite and how to create the animations on Unity can be found in [39].

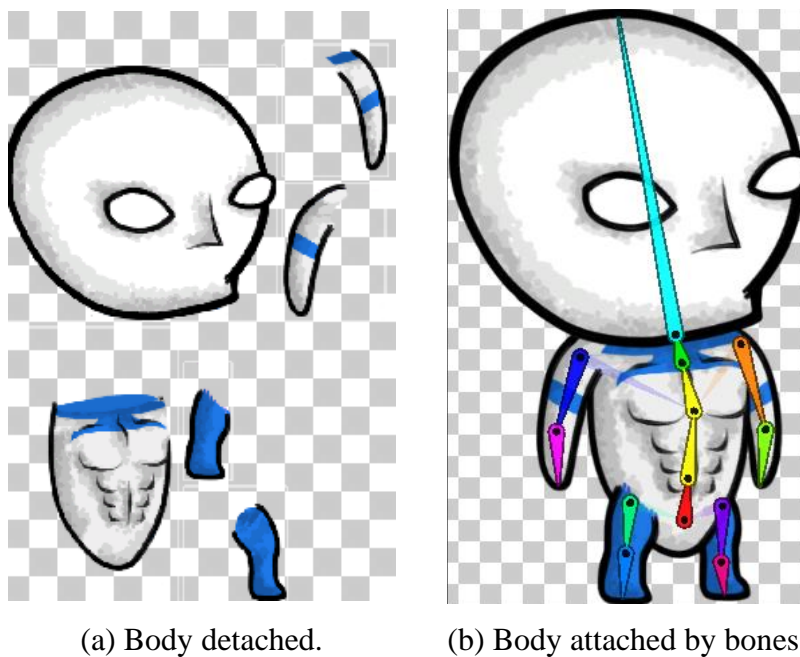


Figure 3.4: Playable character sprite.



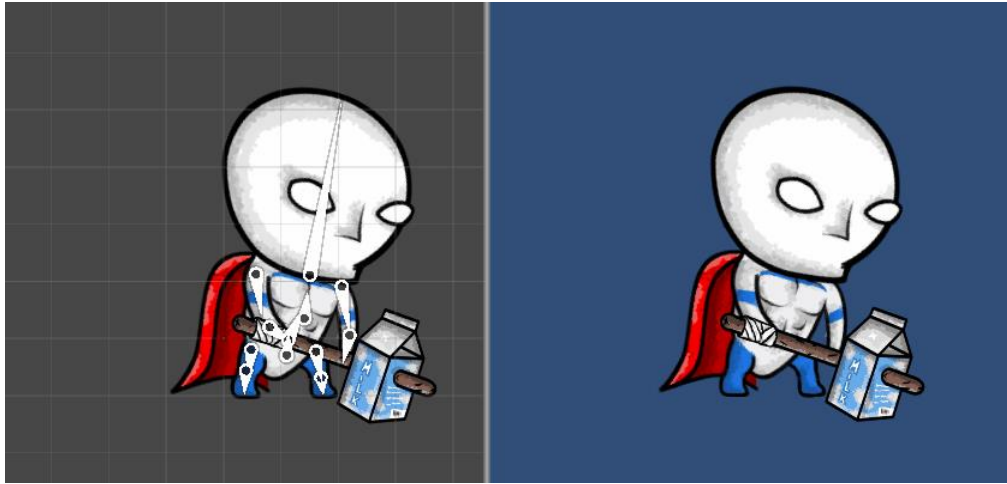


Figure 3.5: Keyframe using bones.

### 3.3.2. Level layout

This branch designs and creates each level of the game. Level design is a very important part of the overall gameplay experience and is influenced by game mechanics and characters in the level. Since level design is not the focus of this work it won't be herein fully explored.

The approach to level design in *Super Milkman* was to present the mechanics incrementally to the player, introducing them one at each time. This way the player gets used to it and learns how it works on their own.

In a 2D platformer game, the level layout is usually implemented in one of three methods. The first is placing the terrain assets in the level and resizing them accordingly to the level layout, as depicted in Figure 3.6. Another way is to use the same method as before but adding shape to the sprites, using sprite shapes [40], allowing for a curved terrain from straight sprites, an example is depicted in Figure 3.7. The last method is to use a tilemap [41], a tilemap is a rectangular grid where the level designer can place tiles, regular sprites as depicted in Figure 3.8.

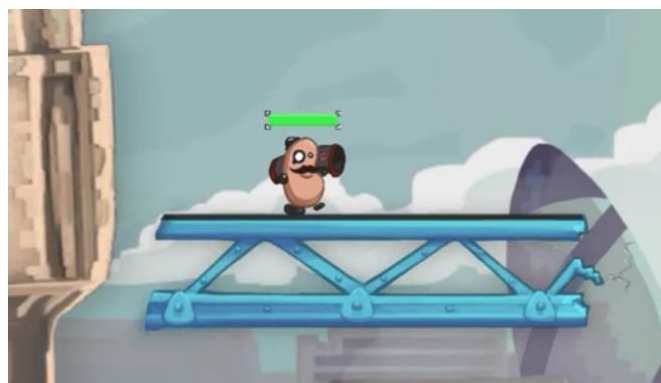


Figure 3.6: Rectangular asset as terrain, extracted from [42].

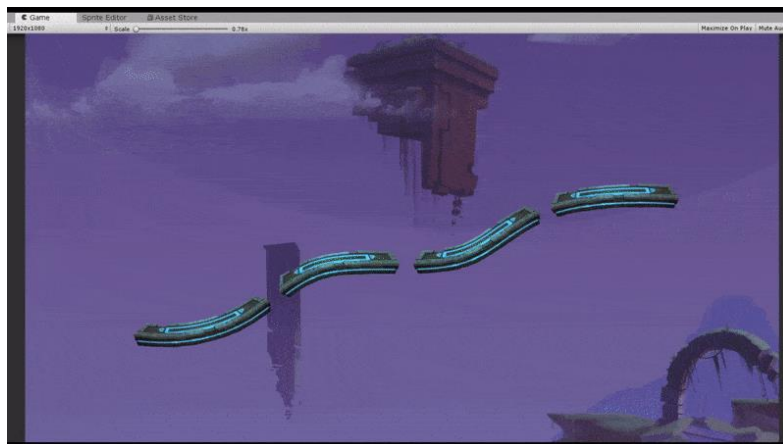


Figure 3.7: Sprite shaped as terrain, extracted from [43].

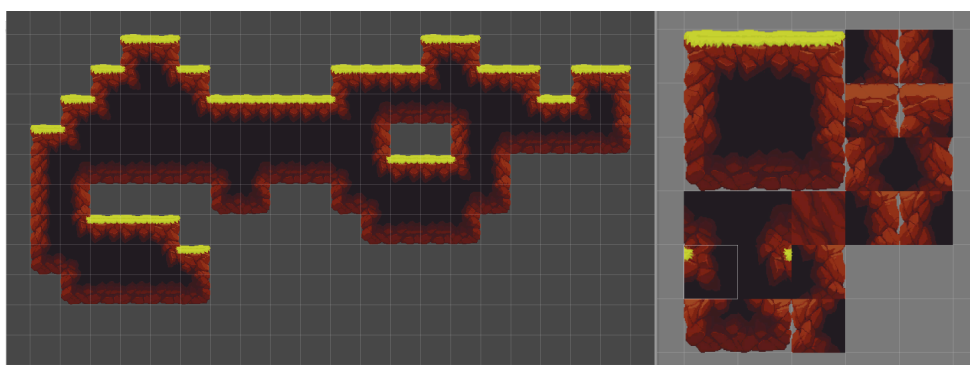


Figure 3.8: Tiles as terrain, extracted from [44].

Every method presented before is valid, although in *Super Milkman* the method used was the tilemap because it allowed faster level creation time and the uniformity across levels. However, this choice requires several different tiles that need to be placed manually, the tileset use is herein depicted in Figure 3.9. Placing the different tiles by hand takes a huge amount of time as the number of levels grows and it is not time-efficient, this problem can be overcome by creating rule tiles. A rule tile, as depicted in Figure 3.10, is a set of tiles encapsulated in one tile that can adapt to its surroundings. Having rule tiles is a huge time saver because the rules only need to be created one time, after that the tile can be used and the surrounding tiles change dynamically in the level as they are placed.

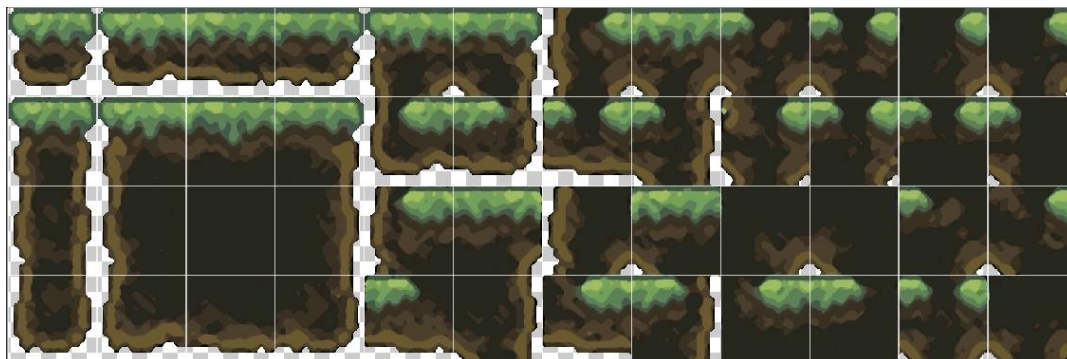


Figure 3.9: Terrain tiles.



Figure 3.10: Rule tile.

Each level is composed by three main tilemaps, as depicted in Figure 3.11: *i*) background, for decoration only, can have several layers with different depths; *ii*) foreground, is the visual boundaries of the level; *iii*) terrain boundaries, invisible sprites that contain the terrain collider.

The terrain collider was purposely isolated from the foreground tilemap in order to control the tilemap collider shape. Usually, terrain sprites have complex shapes, meaning the collider wraps around that sprite creating a complex collider (Figure 3.9), using another tilemap and rectangular sprites creates the possibility to have a more efficient collider.

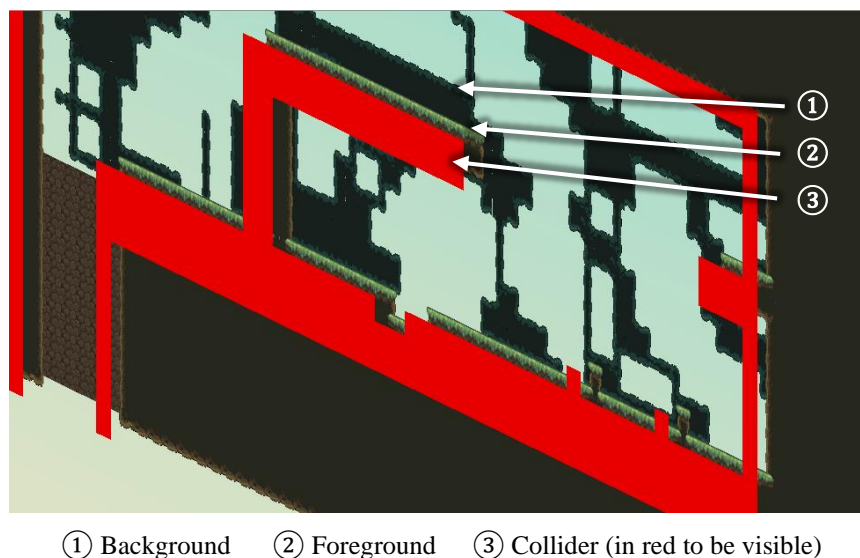


Figure 3.11: Different tilemaps that compose the level, in perspective.

The development of this branch was delayed several times since it had to wait for the other branches to conclude their work. The first iteration was the placement of the terrain using a tilemap. The tilemap was composed of placeholder sprites, this allowed to design

the level structure and place world objects and enemies. The second iteration was the refinement of the objects and enemies, also added the death triggers when the player falls into pits. The final iteration was the creation of tile rules, finishing touches like sound and level decoration, and finally the camera work. Table 3.5 presents the progress on a prototype level.

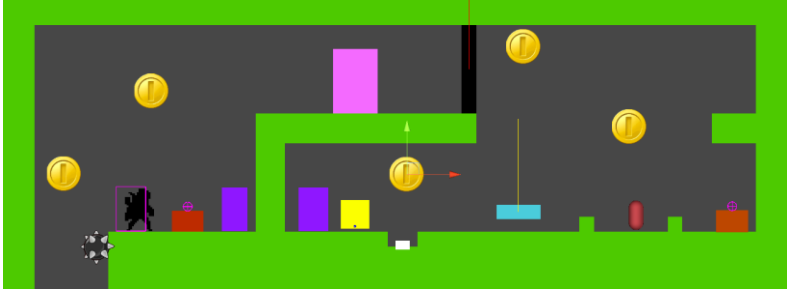
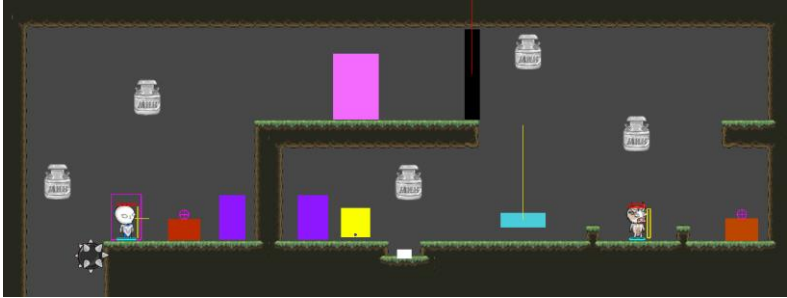

Iteration	Result
1 - Level designed with placeholder art.	
2 - Tiles placed manually and added some final artwork.	
3 - Final art, sounds, decoration and set camera.	

Table 3.5: Prototype level iterations.

After these iterations a systematic way of construction levels was defined: *i)* create the terrain using the rule tiles; *ii)* place the common objects of every level: final door, five tokens, chest with key and enemies; *iii)* death zones; and *iv)* place camera boundaries.

### 3.3.3. User interface (UI)

This branch is responsible for two components, the screen navigation and display in-game information.

In order to keep track of how many screens there is, a list of all of them and their purpose can be made, Table 3.6 presents the list of screens for *Super Milkman*. After the list is created, a graph showing the screen transitions can be designed, this way the programming branch has a reference when linking the screens, depicted in Figure 3.12.

Screen name	Purpose
Main menu	Present options (Credits, Play, Options and Exit game).
Credits	Lists all the elements involved in the development and acknowledges external help.
Options	Change overall music volume.
Level selector	Displays all the available levels that the player can choose.
Level	Gameplay.
Pause	Puts the gameplay in a suspension state and waits for the player to resume it.
Finish screen	Presents information when the player completes the level.

Table 3.6: Screens and their purpose.

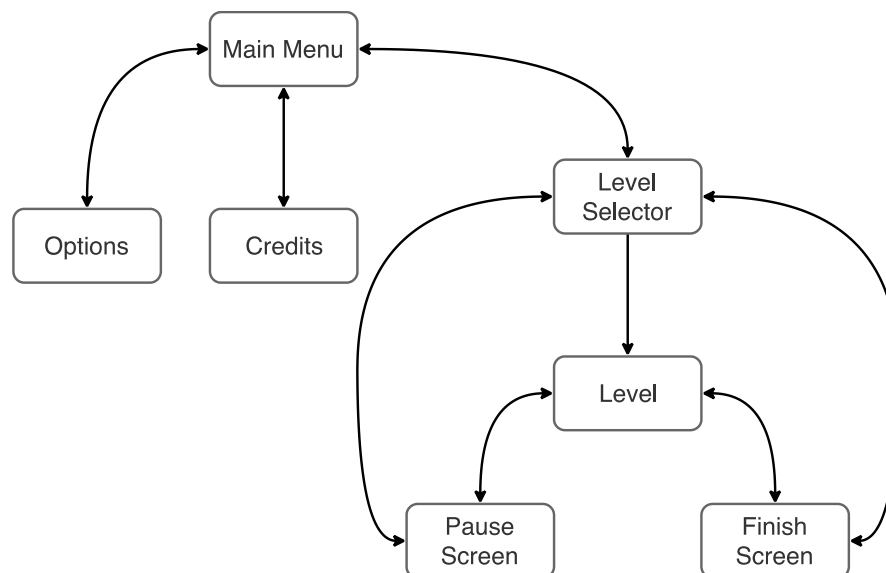


Figure 3.12: Screen transition graph.

In *Super Milkman* the navigation and displaying information had lower priority during development. The development process was composed of four iterations: *i*) display level and player information while playing; *ii*) create the screen's outline; *iii*) replace the placeholders with the real graphics; *iv*) not in this branch but still part of the process, the programmers bind the UI elements to the game scripts, reflecting the game state and creating navigation.

The first component created was the Heads-Up Display (HUD), this component resides in the level screen and displays the player and level state. The HUD is divided into three zones as depicted in Figure 3.13. The first zone is the key, when coloured means that the player has collected it; the second is the health, it is represented by five hearths and when the health decreases the hearths became grey. The last zone is where the five tokens appear when they are collected.

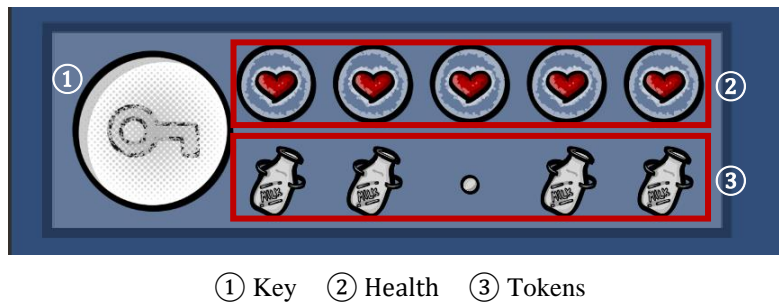


Figure 3.13: Heads-Up Display (HUD).

The next iteration created all the screens that compose the screen graph, after outlining the screens and waiting for the graphics, the final iteration of this branch is completed, the screen iterations are presented in Table 3.7.

After the UI was complete all that was left was linking the screens, that was assigned to the programming branch.

Screen	Placeholder	Final
Main Menu		
Credits		
Level Selector		

Table 3.7: Screens progress.

### 3.3.4. Audio

This branch is responsible for creating the music and SFX. For the audio clips, there are two ways to approach it: *i)* create them in software, record them with a microphone or use instruments; and *ii)* outsourcing them.

In *Super Milkman* all the audio clips were outsourced. Commonly the audio files can be acquired by contracting a sound engineer, by buying them from the engine store or by



downloading them for free in websites like Freesound [45], 99Sounds [46] or OpenGameArt [34].

To ensure interoperability across all the branches a list of the sounds has been defined, Table 3.8 resume the basic audio for *Super Milkman*.

Type	Audio name	Location	Description
Music	UI ambient	UI screens, excluding level	Calm music
Music	Level ambient	Level, excluding Pause screen	Nature music
SFX	Teleport	Doors	Fast sound
SFX	Open chest	Chests	Opening chest
SFX	Hurt	Enemies	Quick hurt sound
SFX	Hammer attack	Player / Weapon (Hammer)	Dirt explosion
SFX	Jump	Player	Jump sound
SFX	Push	Push block	Drag rock sound
SFX	Token acquisition	Tokens	Collect sound
SFX	Token placement	End level door	Placement sound
SFX	Button click	Button	Click sound
SFX	Sliding	Sliding door	Elevation sound
SFX	Success	When the level is complete	Success sound

Table 3.8: *Super Milkman* basic audio list.

### 3.3.5. Programming

The programming branch is responsible for transforming the rules and mechanics that were described in the GDD into code that enforces them in the video game.

The following three sections explain the programming process of *Super Milkman*. The first section explains the basic key concepts in Unity, the other two dissect the UI implementation and level implementation.

#### 3.3.5.1. Unity basics

This project was developed using the Unity engine and its editor. In order to understand the following sections some key concepts in Unity will be explained: *i)* scenes; *ii)* the MonoBehaviour class; *iii)* layers and tags; *iv)* colliders; *v)* rigidbodies; *vi)* prefabs; *vii)* coroutines.

A scene is what the player sees, can be UI and/or the game world. Each scene is composed of several gameobjects<sup>1</sup> that act as a container for components and scripts, all these gameobjects are positioned in the scene in a cartesian coordinate system. Each gameobject can contain components such as UI elements, sprites, 3D models, scripts, animators, etc.

<sup>1</sup> <https://docs.unity3d.com/Manual/class-GameObject.html>

The behaviour of a gameobject is defined by its scripts, these scripts derive from the base class `MonoBehaviour`<sup>2</sup>. This class has all the methods and fields used by the Unity engine already defined. The life cycle [47] of every instance of `MonoBehaviour` is managed by the game engine and its methods should not be called programmatically, the engine takes care of it. In Unity the physics loop executes the *FixedUpdate* method of every *MonoBehaviour* and the graphical loop executes the *Update* method each frame, so code that manipulates physics or graphics should be placed in their respective method.

Every gameobject has a layer attributed to it, this layer can be used as a filtering condition when evaluating for collision using a `LayerMask`<sup>3</sup>. An example usage for layer masks is for object detection on an actor, a ray can be cast in front of the actor and be parametrized with the respective layer mask, if the ray hits some object with a layer belonging to the mask then there's an object in front.

Tags<sup>4</sup> allow gameobject identification. An example usage for tags, is the clear condition of a level, being the condition to defeat all enemies, given that all enemies have the tag "Enemy", is possible to find all gameobjects with that tag and validate if all of them were defeated.

A collider delimits an object and can be categorized as physical or trigger. A physical collider registers when it touches another without overlapping their boundaries. A trigger collider registers when another collider enters its bounds. Whenever the physics engine registers a touch or a trigger, it sends a message to the respective method. This message is received by the *OnCollision* or *OnTrigger* method suffixed with *Enter*, *Exit* or *Stay*; if the gameobject has 2D physics then the 2D suffix is added to the method name like *OnCollisionEnter2D* that receives a message when a 2D object touches another collider. The *Enter* and *Exit* methods are called in the moment of the contact or trigger, and the *Stay* method is called every frame the collider is touching or in trigger bounds.

A `rigidbody`<sup>5</sup> is a component that applies physics to an object. The manipulation of this type of objects must be performed in the physics loop. Like colliders, rigidbodies can be either 2D or 3D, since *Super Milkman* is a 2D game, whenever is referenced *Rigidbody* what is meant is *Rigidbody2D*<sup>6</sup>. A *Rigidbody2D* has one of three body types: i) *static*, disables every force to be applied to the object, it is supposed to be attached to objects that never move; ii) *kinematic* is similar to static but only collides with dynamic body types and is only moved by applying velocity to it or by explicitly changing its position; last type iii) *dynamic*, this reacts to gravity and external physic forces from other dynamic or kinematic rigidbodies. Each one of these body types has its purpose. The following details some examples: i) *static* type can be used for walls or ground, something that doesn't move; ii) *kinematic* type can be used to move the rigidbody through code like a platform that moves from point A to point B; iii) *dynamic* type can be used for actor movement.

---

<sup>2</sup> <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>

<sup>3</sup> <https://docs.unity3d.com/ScriptReference/LayerMask.html>

<sup>4</sup> <https://docs.unity3d.com/Manual/Tags.html>

<sup>5</sup> <https://docs.unity3d.com/ScriptReference/Rigidbody.html>

<sup>6</sup> <https://docs.unity3d.com/Manual/class-Rigidbody2D.html>



A prefab<sup>7</sup> is a template of a gameobject that is saved as a file with all the components and their properties. Prefabs can be reused in many scenes and several times in a scene. A good use example is to reuse the main character since it has several components like the animator, sprite, movement script and health. Also, creating a prefab character allows it to be reused on a different level while maintaining all its components and properties across all levels, even after modified the changes will be reflected in every scene.

Lastly, coroutines<sup>8</sup>, are functions that can pause their execution. An example usage is the movement wall from position A to B when a switch is active. In order to move the wall a loop is needed. Normally the moving loop will execute until completion, so the wall will be moved to the last position within one frame. The moving loop needs to be paused so that the new position is updated in the graphical engine. To resolve this problem coroutines can be used since they can be paused and return control to the engine. And then continue performing the loop where it left off on the following frame. Now the moving loop pauses every position change until the last position is reached.

### 3.3.5.2. User Interface dissection

Scenes represent the screens that the player can interact with. As mentioned before, in Chapter 3.3.3, the UI branch completed the design and creation of each screen as well as the navigation graph. What was left was linking each scene, this was accomplished by using the Unity *ScreenManager* API<sup>9</sup>. Each UI button has an *OnClick* method that was programmed to call the *ScreenManager* API in order to load the next scene, the API requires the index or the scene name. The approach used for storing the scene names was to store them in a class with constant values, the disadvantage is that when a scene is renamed the constant value needs to be changed, creating a need to recompile the code again. Another approach to resolve the naming problem is to use an external file that has a key-value structure with the key being the scene identifier and the value the name of the scene. Implementing this last approach would be similar to the previous since both approaches store the name of the scene with an identifier. In the approach used the identifier is a class constant and the scene name is its value. In the key-value approach, the identifier is the key and the value is the identifier, this structure can be stored in an XML or JSON file. This last approach still makes use of a class to store the class identifiers that are mapped to the structure keys, but the actual scene names remain in an external file that is editable without changing any code. The approach used in *Super Milkman* was the first one since it is simpler and the number of screens is small.

The level select screen lists all the levels dynamically with pagination, each page is called world and has its own set of levels. To store the level and world information were used *Scriptable Objects*<sup>10</sup>, since they provide a way to store data in individual files according to the class fields, as in Listing 3.1. The level select scene has a script that takes an array of *World* instances of the *ScriptableObject* and these worlds have a set of *Level* instances themselves. The script loads into the UI every world and places buttons that represent each level, as well as create the navigation between worlds. The level scriptable object has a field called *sceneName* and when the level button is clicked it calls the *ScreenManager* API to load that level scene with the specified name.

---

<sup>7</sup> <https://docs.unity3d.com/Manual/Prefabs.html>

<sup>8</sup> <https://docs.unity3d.com/Manual/Coroutines.html>

<sup>9</sup> <https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html>

<sup>10</sup> <https://docs.unity3d.com/Manual/class-ScriptableObject.html>

```
1. [CreateAssetMenu(fileName = "W00L00",
2.   menuName = "Level")]
3. public class Level : ScriptableObject
4. {
5.     public string levelName;
6.     public string sceneName;
7. }

1. [CreateAssetMenu(fileName = "W00",
2.   menuName = "World")]
3. public class World : ScriptableObject
4. {
5.     public string worldName;
6.     public int worldNumber;
7.     public Level[] levels;
8. }
```

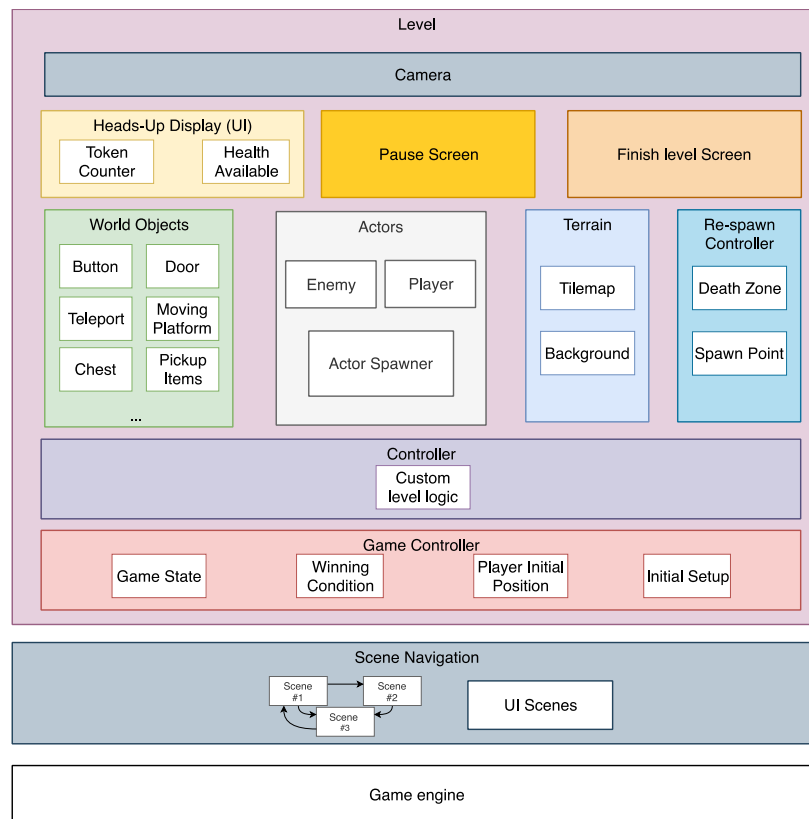
Listing 3.1: *Level* and *World* scriptable objects.

### 3.3.5.3. Level dissection

In this section, the level will be dissected since it is the main part of this video game. Each level in *Super Milkman* has the same structure and components; the only difference between levels is the level layout, the enemy placement and what and where the world objects are placed.

The development started with the planning what composes a level, the result is the diagram depicted in Figure 3.14. The main building blocks that compose the level are: *i*) game controller, containing the rules that the level must follow and sets it up so it's ready to play; *ii*) level controller, an optional component and can have custom level logic, making the level unique; *iii*) actors, also called characters have movement and perform actions; *iv*) world objects, populate the game world and interact with the actors; *v*) HUD, displays important information always on the screen; and *vi*) camera, is windows where the player sees the game.

After that planning, each component in the level had designated a priority in which it would be developed. This planning is resumed in Table 3.9, that details the component, its priority and the reason for having that priority.

Figure 3.14: *Super Milkman* diagram.

Component	Priority	Explanation
Camera	Medium	The default one can be used until further progress is done.
World objects	High	The objects interact with the player and the mechanics are design and developed around these interactions.
Game controller	Low	Coordinates when the level is complete, only needed further in development.
Level controller	Low	Level coordinator, only needed further in development.
Terrain	High	Contributes to testing mechanics and actors movement.
Respawn controller	Medium	Not essential but helps when testing the health component in the actor, also translates the player to a previous position.
Enemy	High	Interacts with the player and shares components with the Player.
Player	High	Is the actor that the player controls, this character interacts with all the level components.

Table 3.9: Component priority table in development.

The development was iterative like the previous branches, firstly was developed the base actor that had the development iterations as presented in Table 3.10, the actor was later extended to be a player and an enemy since all these characters shared the same base components.

Iteration #	Actor	Player	Enemy
1	Movement interface	Input movement	Linear movement
2	Actor bounds	Abilities	Linear movement without falling
3	Weapons and Health	Hurt on contact	Awareness of the player and when close use weapon
4	Animation	Weapons	Pursue player

Table 3.10: Actor and its inherent children iterations.

The first iteration created the skeleton of the actor controller. An actor needs several fields in order to be able to interact with its surroundings. The main fields are: *i*) a box collider, called hitbox, that surrounds the sprite and collides with terrain, world objects and enemies; *ii*) a layer mask, used to contain the layers where the actor can move, also for checking if the actor is touching the ground, ceiling or has something in front of him; *iii*) sprite direction, to know if the sprite is facing the correct direction, this is used when the actor changes movement direction so it can flip the sprite to face the new direction; and *iv*) default movement speed is used by the *Move* method. With these fields and some methods, the first iteration of the abstract actor controller is complete, the simplified class diagram is depicted in Figure 3.15.

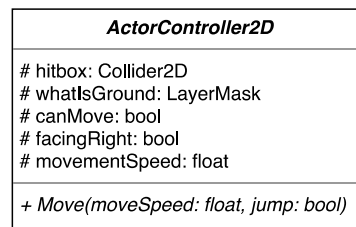


Figure 3.15: Actor controller class diagram, first iteration.

The second iteration added boundaries to the actor in order to validate if it is touching the ceiling, the ground or if it has something in front. Since *ActorController* is an abstract class, its children inherit all these fields and can access them. There are three boundaries for ceiling, ground and front. These boundaries help calculating when the actor is touching its surroundings. The approach used to update the checkers was to have three rectangular boundaries using *Bounds*<sup>11</sup> struct and placing them overlapping the actor surroundings trigger collider, as depicted in Figure 3.16.

<sup>11</sup> <https://docs.unity3d.com/ScriptReference/Bounds.html>

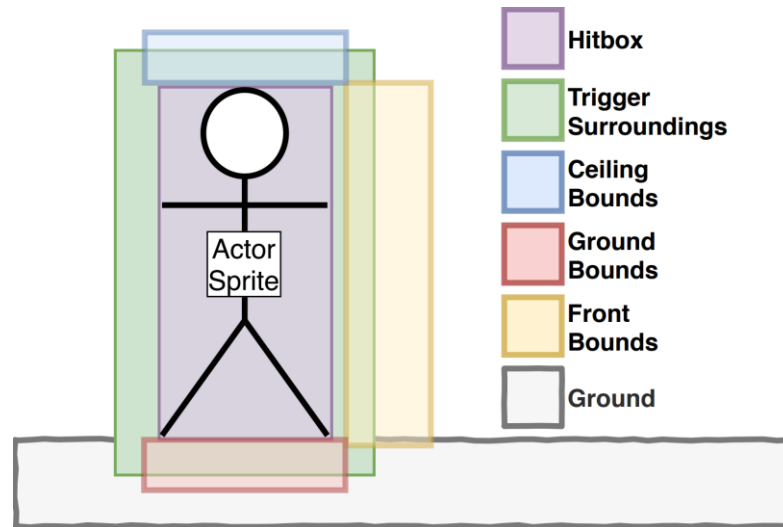


Figure 3.16: Actor colliders and boundaries.

This trigger collider acts as a proximity sensor sending messages to the *OnTrigger{Enter, Exit and Stay}2D* methods, the inherent classes of *ActorController* can implement the methods *OnTriggerEnter2D* and *OnTriggerExit2D* to call the *CheckCollision* method, this method updates the three checkers with similar code as in Listing 3.2. A child class shouldn't use the remaining method, *OnTriggerStay2D*, since it would execute the *OverlapBox* function several times per seconds without being necessary and adding CPU processing time. The result of the checkers needs to be saved in a field, using that field the children of the *ActorController* can read its state, Figure 3.17 depicts the updated class diagram.

```
1. float angle = 0;
2. bool checker = Physics2D.OverlapBox(actorPosition + boxBounds.center,
    boxBounds.extents, angle, groundLayerMask);
```

Listing 3.2: Code that checks if a boundary overlaps a collider.

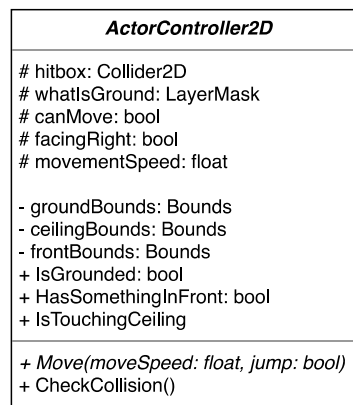


Figure 3.17: Actor controller class diagram, second iteration.

After the base actor controller is defined, new components could be added, the third iteration adds health and weapons.

The actor has a health component, that controls its state, alive or dead. This component has four events that other objects can subscribe: *i) OnChangeHealth*, when the health changes, gets higher or lower; *ii) OnDeath*, when the actor has no more health points; *iii)*

*OnDamage*, when the actor takes damage and the health points decrease; and *iv) OnHeal*, when the health points increase. These events can be used by the actor controller to play particles effects or to update the UI. Since the decrement of health points happens in the *Update* method, usually when collisions happen, that means that the actor can be left with no health point if the collisions occur very fast. There's a need to have a cooldown timer between hits or the actor can die almost instantly, before any decrease in health points the cooldown timer is checked and after the actor takes damage the cooldown timer is reset.

In order to have different types of weapons, projectile and handheld, was created an abstract class that every weapon must extend. This class has a few fields like the cooldown time between usages and a layer mask that defines what gameobjects in the specified mask the weapon can damage and has a single method for using the weapon, the simplified class diagram with its children is depicted in Figure 3.18. The children classes were developed in the enemy and player iterations.

This iteration added a few fields to the *ActorController* since it only added the Health and *Weapon* components.

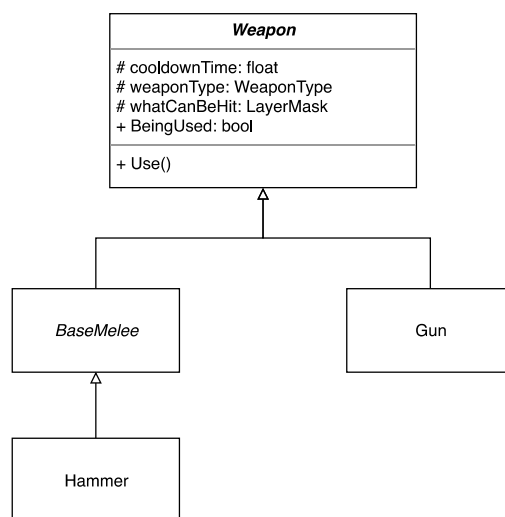


Figure 3.18: Weapon abstract class and its children.

The next iteration didn't require much more additions since the *ActorController* is the base class that others must extend. But several abstract methods were added: *i) UpdateMethod*, called by the base class in the method *Update*; *ii) AnimateActor*, called by the base class before the *Update* method exits; *iii) FixedUpdateMethod*, called by the base class in the method *FixedUpdate*. This approach of creating new methods that the children use instead of the ones provided by the engine allows to encapsulate numerous validations and removes the responsibility from the child classes.

The *Update* method in the *ActorController* firstly verifies if the actor is alive then calls the *UpdateMethod* of the child class and lastly the *AnimateActor*. This way the child class only implements the logic of their specific actor. The same is done for the *FixedUpdate* method, it verifies if the actor is alive and then calls the *FixedUpdateMethod* method from the child. There's still the possibility of overriding the *FixedUpdate* and *Update* methods but then the actor behaviour would be discarded. Since the new methods are abstract, they must be always implemented.

Having an *AnimationActor* method simplifies the animation process since it encapsulates all the animation code in one method. This method is called by the base class, removing

this responsibility from the child class, every actor type sets proper values, so their animator can transition to another animation.

After all these iterations, the *ActorController* class is complete and ready to be used as a base for the *PlayerController* and the *EnemyController*, its class diagram is depicted in Figure 3.19

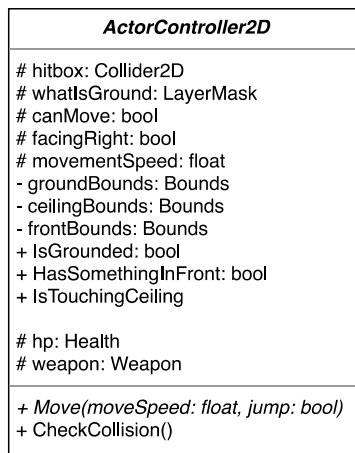


Figure 3.19: Final iteration of the ActorController class.

Starting with the *PlayerController*, the initial iteration was about the input. The class responsible for handling the inputs has a reference for the *PlayerController*, every input is read in the *Update* method and physics-related code is executed in the *FixedUpdate*. In the *Updated* method, the movement values are stored. The movement is executed in the *FixedUpdate* method by calling the *Move* method in the *PlayerController* with the values read in the *Update* method. The player input is very modest in this iteration, the verification is done by testing if the key in the keyboard is being pressed.

The second iteration was the abilities. The abstract *Ability* class has three main methods, the *BeginUsing*, *WhileUsing* and *EndUsing*, and an *AbilityState*: *InUse*, *ReadyToUse*, *CantUse*. The ability states determinate if the ability can be used or not, or if it is already being used. The *InUse* state is when the player is using the ability; the *ReadyToUse* state is when the player is not using the ability but can use it; the *CantUse* state prevents the user from using it. The three methods, in the *Ability* class, assure that the state of the ability changes correctly, the class diagram is depicted in Figure 3.20. Two abilities were made to ensure that this approach would work, the abilities were *push block* and *dash*.

Since the input method still uses keypresses, a dictionary with keycode and ability was created in the *PlayerController*, this allows to loop over every ability checking its state and if the key is pressed. If there aren't abilities in use or is the same ability but in a different state, then the begin, while or end method will be called accordingly.

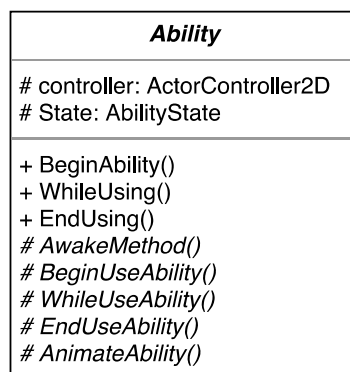


Figure 3.20: Ability class diagram.

The third iteration implemented collision damage between actors. The *HurtOnContact* script decreases the health of an actor when it collides with a gameobject that has their layer in its layer mask. In order to have visual impact when the player is hurt on contact a force is applied in the rigidbody of the player projecting it backwards. The class diagram of *HurtOnContact* is depicted in Figure 3.21.

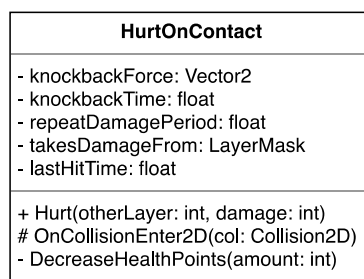


Figure 3.21: HurtOnContact class diagram.

The last iteration on the *PlayerController* were the weapons, these weapons are not exclusive for the player since all actors can have a weapon. Some weapon can even function without an actor, for example, a gun in a wall. Two weapons types were created, a melee and gun.

A melee class was created to be the base to more weapons of this type. Melee weapons don't shoot, they only check if there are targets in front of the actor when used. The only melee weapon implemented was a hammer. The *Hammer* class implements the *Use* method from the *Weapon* class and uses the *Physics2D.OverlapBox* function to validate if any gameobject found is within the specified layer mask. The output of the *OverlapBox* function is needed, differently from the checkers used on the *ActorController* where the only purpose was to verify if there was something there, the output is a *Collider2D* array with the targets. After obtaining the targets, there's an iteration over them where the health points are decreased and the knockback force is applied.

To finalize the weapons, the gun type. The *Gun* class has a field for the bullet prefab, this field will be used to create an instance of that prefab. In order to instantiate a child of a *MonoBehaviour* there's a function<sup>12</sup> provided by the engine that clones that prefab gameobject and instantiates it. This instantiation of the bullet is performed in the implementation of the *UseWeapon* method and then a force is applied so the bullet moves in a direction. Unlike the melee weapons, what damages the target is not the weapon but

<sup>12</sup> <https://docs.unity3d.com/ScriptReference/Object.Instantiate.html>



the bullet, so the bullet is responsible for decreasing the health points of the target actor. The approach used for the bullet was to have it disappear after a certain amount of time or to disappear when in contact with a collider. When there's a collision between a bullet and an actor, the *OnCollisionEnter2D* method of the *Bullet* class is called and the health points of the actor are decreased.

The *EnemyController* is in charge of the enemy behaviour, this behaviour is controlled by an enemy state. These states can be: *i)* free movement, where the enemy walks forward; *ii)* targeting the player, the enemy moves towards its target, the player; and *iii)* attacking the player, the enemy stops and performs an action to decrease the player health points.

The first iteration was the creation these states and their corresponding movement controllers that implement a common interface that has a single method, *ExecuteMovement*, this way the movement is separated from the actor controller and can be reused by others if needed. Movement, in this first iteration, was always linear, even if there's a wall in front or a pit, the enemy always went forward. The second iteration solved this problem by checking if there was something in front of the enemy, *HasSomethingInFront* property inherent from *ActorController*, and to use the *OverlapCircle* method to detect pits, as depicted in Figure 3.22, if the conditions to change direction are met then movement direction is inverted.

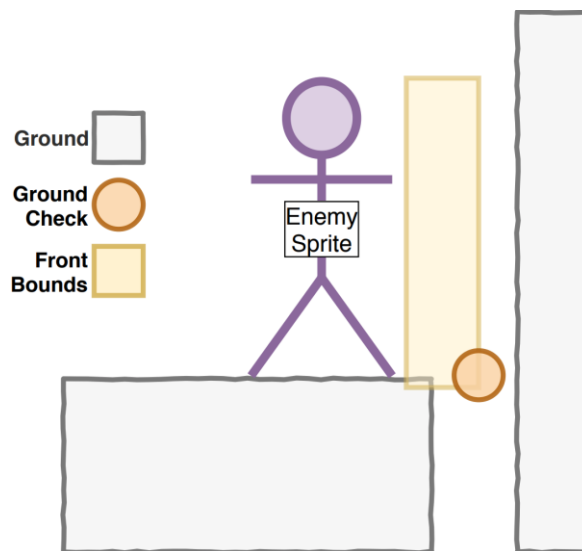


Figure 3.22: Enemy ground checker.

The third iteration was related to player awareness, to accomplish this the controller scans every frame for the player, if found then the enemy state is changed. Scanning for the player is performed by verifying if the player position is within the enemy field of view. The field of view is a portion of a circle, as depicted in Figure 3.23, and the values of distance and the cone aperture can be changed in the Unity editor. When the player is found the state changes from free movement to target player and when the player is very close then the state changes to attack. This state change is performed in the *Update* method every frame, with the following sequence: *i)* check if enemy can target player, then scan and check if player is within range, if positive change state; *ii)* check target is out of range but within cooldown time; and *iii)* use weapon, when available, if the player is in range.

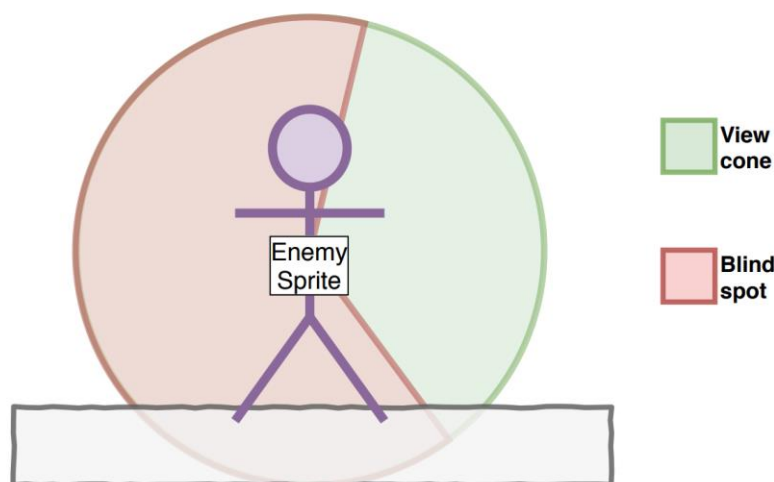


Figure 3.23: Enemy field of view.

Is possible to disable this active behaviour towards the player in the enemy controller, the field *canTargetPlayer* can disable this behaviour so the player isn't chased.

In Chapter 3.3.1 was listed the main world objects with their sprites, these objects were: *i)* collectable token; *ii)* teleport door; *iii)* final door; *iv)* button; *v)* push block; and *vi)* chest.

Collectable tokens are the part of winning condition to finish the level, so it is necessary for them to be collected, this happens when the player enters the surroundings of the token. The approach was to create a trigger collider that verifies if the collision was with a gameobject tagged with as "Player" and then the token is marked as collected, *isCollected* field. There's also a *UnityEvent*<sup>13</sup> field that is executed after the *isCollected* is set. The *UnityEvent* has a set of callbacks that can be added directly in the Unity editor and executed in the script by calling the *Invoke* method, this was used to play a sound and to disable the sprite and collider without needing to code, as depicted in Figure 3.24. This approach results in less code and it's easy to add more callbacks if needed, feedbacking to the UI or the game controller.

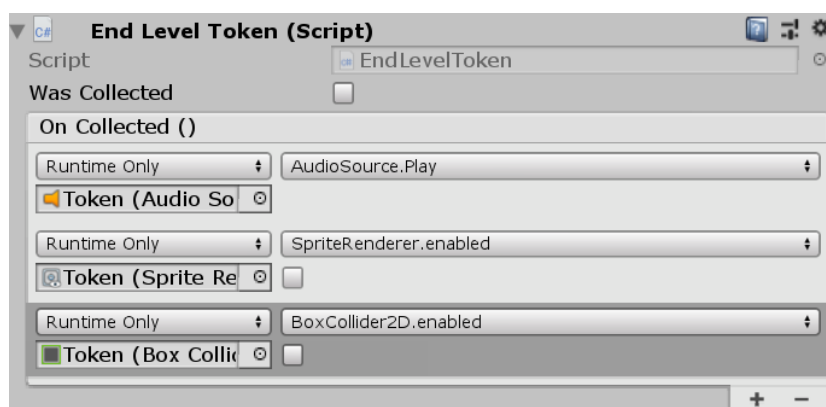


Figure 3.24: End level token script in the Unity inspector.

Teleport doors warp the player from one location to another by altering its coordinates. As the collectable tokens, teleport doors also use *UnityEvent*. When the player is within

<sup>13</sup> <https://docs.unity3d.com/ScriptReference/Events.UnityEvent.html>

the door trigger collider and presses a specific key the *UnityEvent* executes all its callbacks. There are two callbacks, one for a sound clip and another for the linked door. In the teleport door script, there's a method called *Teleport* that changes the player position to the door's coordinates. For example, door A added door B *Teleport* method resulting in the player being teleported to the door B, the inverse must be done to door B, so it links back to door A. In Figure 3.25 is depicted door A in the Unity editor and one of the callbacks is the door B *Teleport* method.

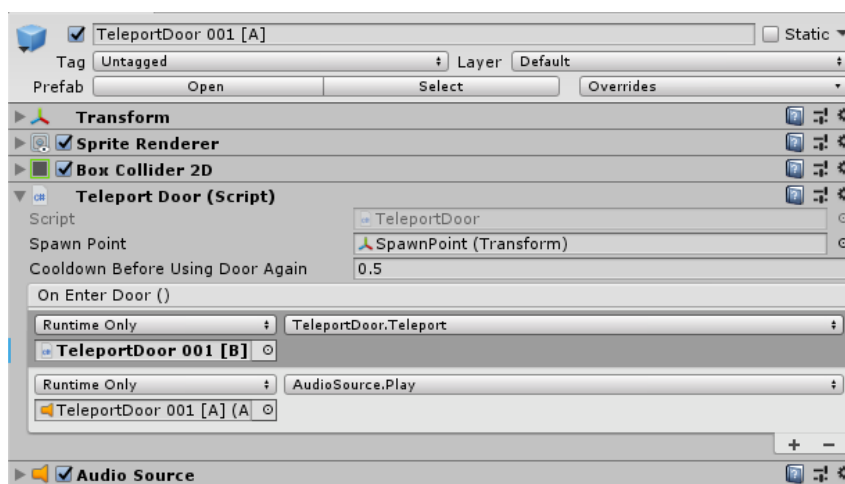


Figure 3.25: Teleport door 001 A in the Unity inspector.

Buttons are paired with sliding doors, similar to the previous gameobjects, when the button is pressed the registered callbacks are executed, there's a sound clip that plays and the *OpenDoor* method of the sliding door is called. After being pressed one time the button says locked, so it doesn't execute the callbacks again.

Sliding doors have two positions, the open position and the closed position, when the *OpenDoor* method is called the door is moved from one coordinate to another at a specific speed using a coroutine. There are two *UnityEvents*, one called *onOpening* and another *onFullyOpen*, the first one plays a sound clip and the other stops it and disables the collider and enables a new smaller collider when the door is fully opened. The script in the inspector has some buttons that were added there programmatically that help setting the door in the initial position and in the final position, as depicted in Figure 3.26, this custom *Editor*<sup>14</sup> script adds the buttons and is totally optional, doesn't add any functional requirements, but helps the game designer when he's arranging the level.

<sup>14</sup> <https://docs.unity3d.com/ScriptReference/Editor.html>

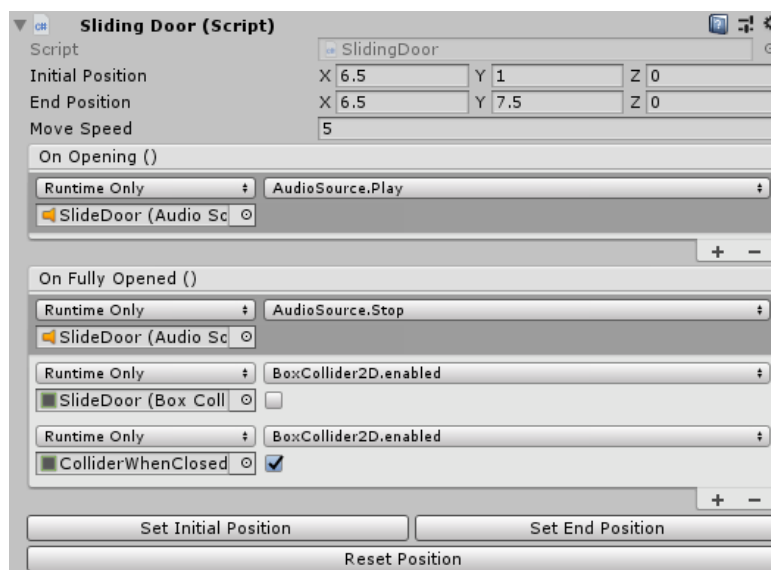


Figure 3.26: Slide door script in the Unity inspector.

The push block object is a static rigidbody until the player interacts with it, this is done with a player ability. The ability connects the player rigidbody with the block rigidbody using a *FixedJoint2D*<sup>15</sup>, so when the player moves the block moves as well. The block can only be moved when it's grounded and when the ability is active, else the rigidbody is static. The manipulation of these rigidbody types is done in the *FixedUpdate* method since it makes changes to physics-related objects.

The final world object is the chest. Within the chest is a pickupable object, an object that the player can collect, but in order to show this pickupable object it needs to be open. The opening process is done by having a trigger collider that only reacts to the player, after that the *onChestOpen* UnityEvent plays a sound, similar to the previous objects, and changes the sprite and collider to a smaller one. After this process, the pickupable object is shown at the middle of the open chest. This approach of having *UnityEvent* facilitates calling whatever callbacks are needed, for example adding particle effects when the chest is open or pause the scene and show the item for more dramatic effect. The *Pickup* abstract class is used for increasing the health points and to collect the key to be able to finish the level.

Summarizing the world objects, almost all the world objects contain sprites and a physical or trigger collider. After the player interacts with this collider some actions are applied to the player and/or callbacks are executed.

At last, the game controller and level controller. These two controllers are responsible for the rules and conditions of the level and the game. The game controller is responsible for the generic verifications and rules that every level needs to do, like verifying if the player has all the tokens and restoring the player back to life when his health points go below zero; this controller also has the responsibility to end the level when the player is at the end door with all the tokens and the key, when this happens the level state changes and the finish level screen appears telling the player that the level is complete. The level controller is for custom code and interactions within a certain level, for example, if in a

<sup>15</sup> <https://docs.unity3d.com/ScriptReference/FixedJoint2D.html>

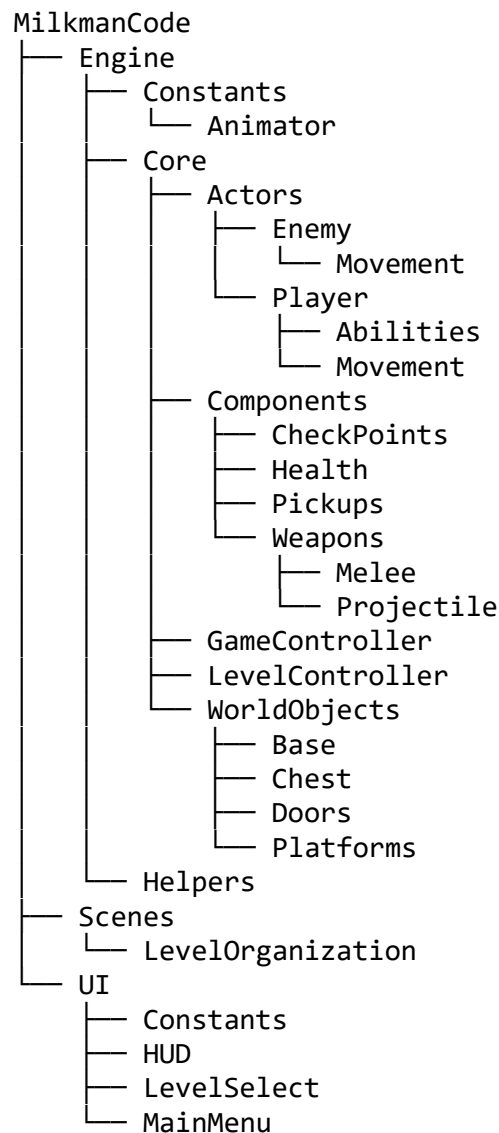
level there's a cutscene or a zone where the player needs to wait and some action is triggered then it can be scripted in this controller.

### 3.3.6. *Super Milkman* engine organization

In this chapter, Production, the main details in this work were explained individually and divided by its branch of speciality. This section explains the project organization, the combination of all the branches that created *Super Milkman*.

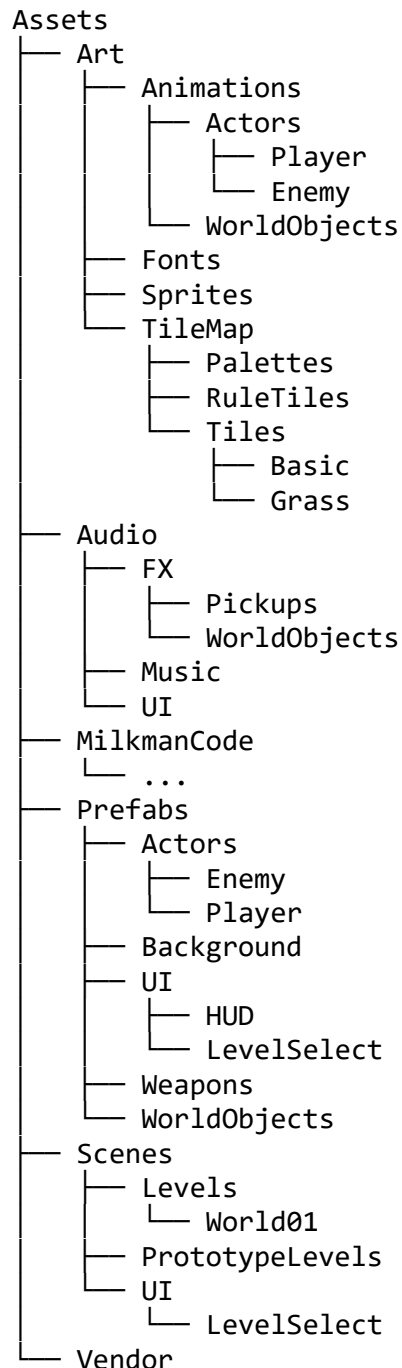
There were three main different organizations: *i*) code, the scripts that control the whole game; *ii*) project assets, the Unity project with all the assets that compose the game; and *iii*) level, the scene where the gameplay takes place.

Code organization, as listed in Listing 3.3, was divided into three main folders: *i*) engine, scripts related to gameplay; *ii*) scenes, container objects used to populate the user interface; *iii*) UI, scripts related to the user interface like the screen actions, and HUD.



Listing 3.3: Code folder structure.

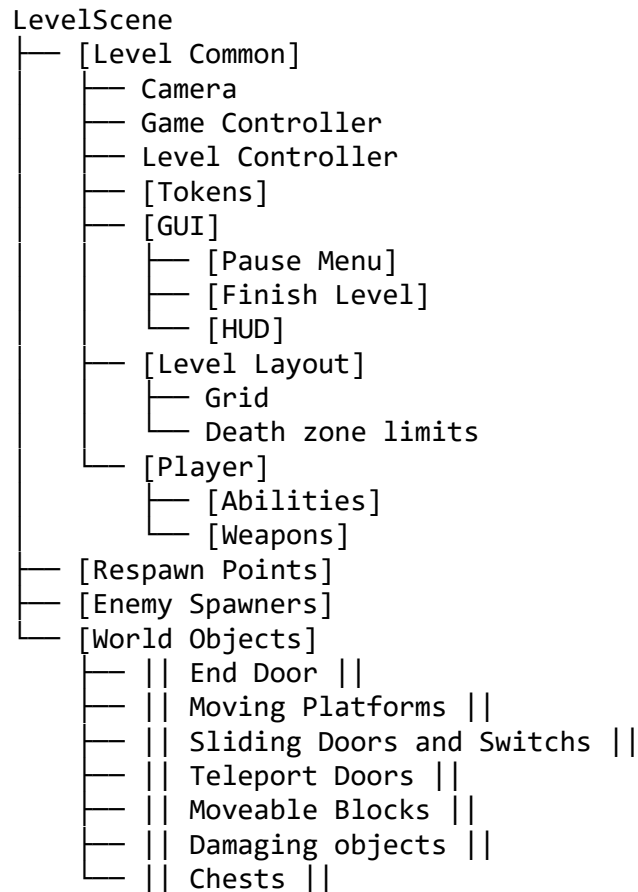
Project organization in the Unity editor is very important since the project folder holds everything about the game. The approach to organize the folders, inside the editor was to divide all the same type of files per folders, as listed in Listing 3.4, this way is easier to work in each different branch. Another approach would be organizing the project by unique themed folders, like in the case of the player it would have a “Player” folder with all the other specific folder: the art folder with sprites, animations and animator, sound folder with audio related to the player, prefab folder with the player and its weapons.



Listing 3.4: Unity project folder structure.

The level scene organization must be coherent across all the different levels, to create a uniform structure where the level designer can create and arrange the level with ease. The scene has three main object parents, as listed in Listing 3.5: the “Level Common”,

the foundations of the level, the main components that compose a level; “Respawn Points”, the triggers that are meant to be placed in pits where the player can fall so it can be placed on ground again; “Enemy spawners”, game objects that are responsible for managing the enemies in the level and spawning more if some is defeated; “World Objects”, this parent has several children in order to better organize the scene.



Listing 3.5: Level scene organization.

## 3.4. Post-production

This stage of development only occurs for completed videogames already released.

Since *Super Milkman* is not a finished videogame, this stage didn't happen. As previously described in Chapter 2.2, this stage serves the purpose of bug fixes, maintenance and possibly to add new features as DLC.





## 4. Conclusions

The work depicted in this document describes the process used to create a 2D platformer video game, *Super Milkman*. Some of these objectives set at the beginning of the project weren't totally met since they were on the artistic side, like visual and audio effects.

Since a video game is a multimedia experience, a difficult process was creativity. Is very difficult to be creative, to create and design a video game by yourself. Ideally, there are different teams that are responsible for each branch.

Despite these difficulties, the main parts of the *Super Milkman* video game were concluded and a prototype video game was created. The methodologies followed resulted in the creation of *Super Milkman* and will be described in the following section, 4.1. This video game is far from complete, but the general structure was completed and future work is related in section 4.2.

### 4.1. Followed methodologies

There isn't a methodology that works for the development of all video games since there are many genres and each development team and project is different, but the following worked for the development of *Super Milkman*.

Starting with the concept stage, was created a simpler GDD with only one page to express the main idea of the video game, some conceptual artwork was also drawn to help visualize some ideas. Lastly in the concept stage, was the prioritization of tasks for the next stage, as it's depicted in Table 4.1.

Task	Priority	Duration
Task 1	{ High, Medium, Low }	$x$ weeks
Task 2	{ High, Medium, Low }	$y$ weeks

Table 4.1: Project plan example.

In the pre-production, a more elaborated version of the GDD was written to further develop the game details. Is very important to keep the GDD up to date, since it contains all the details about the video game being developed and describes what aims to become in the future. It also provides an easier way to share information among others and should serve to guide along the development progress. The GDD doesn't have a standard structure since there are many different genres of video games, where some aspects make sense in one game and not so much in others, [48] gives a good starting point on how to write a GDD. Before starting the production stage, the game engine was chosen since it will condition the workflow of the next stage. Unity was the choice since it has a wide active community and the scripting language is a modern managed programming language, C#, allowing for quick prototyping and faster development.

Before the production stage started developing its work, tasks to be completed were established. These tasks were further divided by branch, as depicted in Table 4.2, since the task usually was multidisciplinary, crossing different areas of expertise. The branches considered were: *i*) arts and graphics; *ii*) level layout; *iii*) user interface; *iv*) audio; and *v*) programming. The progress in all the branches was iterative and developed in parallel, meaning that each iteration would add small features, and to join the progress the branches would merge. Figure 4.1 depicts all the different branches feeding each other when they need to progress, and finally, all the different branches merge together finalizing an iteration or complete a task.

Task	Subtasks	Development branch
Player actor	Sprites and animation	Arts and graphics
	Animation scripting	Programming
	Mechanics	Programming
	Movement	Programming
	SFX	Audio

Table 4.2: Detailed tasks and their respective development branch.

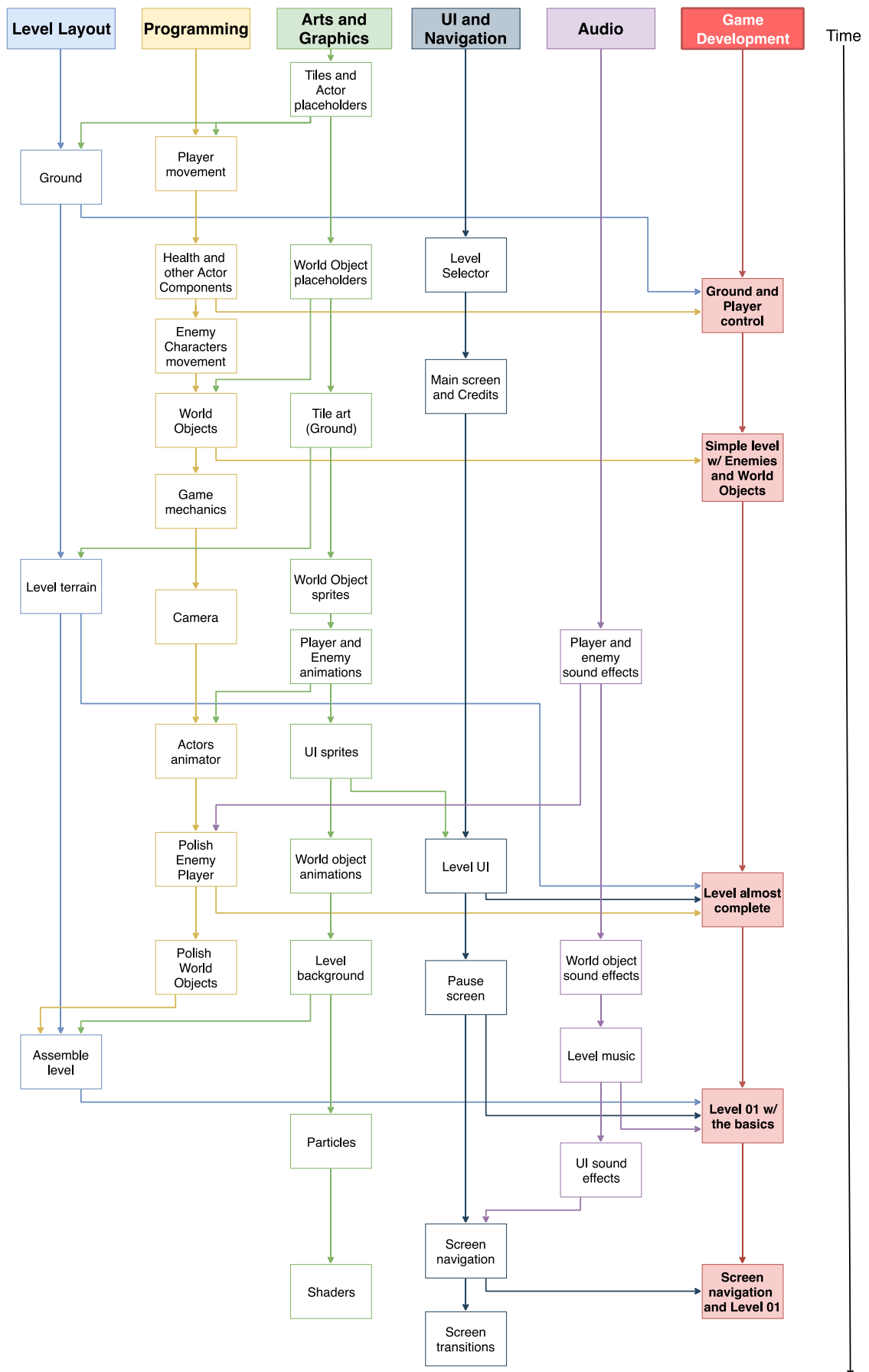


Figure 4.1: Project branches feeding and merging to complete the project.

Some assets in the arts and graphics branch were digitally drawn and others were obtained for free on websites. The technique used for animating was skeletal animation, allowing to create animation of a single image. This technique was chosen because it meant that the character didn't need to be drawn again for every frame of the animation.

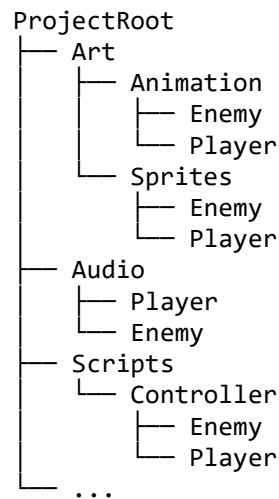
The level layout branch created the levels. To smooth the creation process was used a tilemap grid to create the level terrain. A rule tile was used to encapsulate a whole tileset with many different tiles, this tile changes the surrounding tiles to adapt to its rules. After the rules were created, the creation of each level went faster since the different tiles didn't need to be placed individually. Other tilemaps were used to create a visual depth perspective, the different tilemaps were: foreground, background and an invisible collider. This invisible collider needed to be created in order to create uniformity across the level, since the collider in the terrain tilemap would wrap to its irregular shape, causing strange behaviour with other components in the game. Lastly was created a systematic way of creating each level, starting with the terrain, adding the common objects, the death zones and finally the camera boundaries.

The user interface (UI) branch was divided into two, outside the level and inside the level. Outside the level UI was all the navigation between screens, a diagram was created to keep track of all the screens transitions. Inside the level two UI elements were needed, the HUD and the end level screen, these were created so they could be re-used in the different levels without having to change anything since they communicate with the game controller.

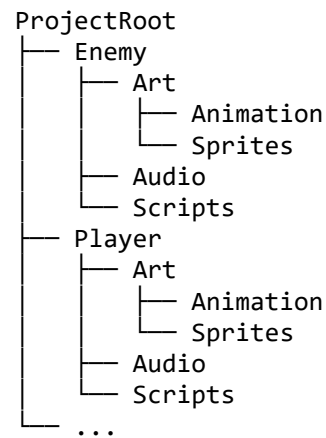
The audio branch obtained its sounds in free websites since it is an artistic area that requires specialized skills.

Lastly, the programming branch created a global diagram of the game, depicting every major building block of the game, helping to plan the next steps. The next steps were to develop the world objects, the actors and its components, like animation, weapons, movement scripts, etc. An approach used for some world objects was to allow the game designer, in the engine editor, to add callbacks that would be executed. This approach meant that the script attached to the object didn't have the responsibility to know what it would be executed, allowing for simpler code and allowing to view the list of all callbacks registered in the engine editor.

To ensure the project structure maintains coherence, the folder structure to store the project assets chosen, there are many different ways to organize the project structure, two examples are listed in Listing 4.1. There isn't a right way to organize the project, but it should be consistent across the project to ensure it is easy to find any asset needed. The choice for this project was the organization presented in Listing 4.1 (a) since it creates fewer folders and allows for quick navigation to find a file since it is sorted by type.



(a) Organization by type.



(b) Organization by component.

Listing 4.1: Project organization examples.

The scripting should also be well organized and should follow the good practices and design patterns of software development. There are also video game-specific programming patterns [49] making cleaner code and easier to maintain.

## 4.2. Future work

The work developed in this project didn't leave the prototype state. It would be interesting to build upon the foundation that was developed and improve it.

One addition that should be done is to explore branches that weren't explored since they were out of scope of software engineering. These branches were left out but they add visual interest to the game, and these are branches that create an immersive experience and capture the player attention.



# Bibliography

- [1] T. Wijman, “Newzoo,” 30 April 2018. [Online]. Available: <https://newzoo.com/insights/articles/global-games-market-reaches-137-9-billion-in-2018-mobile-games-take-half/>. [Accessed June 2019].
- [2] “The Video Games' Industry is Bigger Than Hollywood,” 10 October 2018. [Online]. Available: <https://livesports.com/e-sports-news/the-video-games-industry-is-bigger-than-hollywood> . [Accessed September 2019].
- [3] “Unity - Manual: Sprites,” [Online]. Available: <https://docs.unity3d.com/Manual/Sprites.html>. [Accessed September 2019].
- [4] S. Rogers, “Level Up! The Guide to Great Video Game Design,” Wiley, 2014, pp. 17-27.
- [5] J. Gregory, Game Engine Architecture, A K Peters/CRC Press, 2009.
- [6] B. N. Entertainment, “Pac man,” [Online]. Available: <https://www.bandainamcoent.com/games/pac-man>. [Accessed June 2019].
- [7] “File:Pac-man.png - Wikipedia,” [Online]. Available: <https://en.wikipedia.org/wiki/File:Pac-man.png>. [Accessed August 2019].
- [8] Nintendo, “The official home for Mario - Super Mario games,” [Online]. Available: <https://mario.nintendo.com/history/>. [Accessed August 2019].
- [9] “Mario Wiki,” [Online]. Available: [https://www.mariowiki.com/images/e/e4/World\\_1-1\\_SMB.png](https://www.mariowiki.com/images/e/e4/World_1-1_SMB.png). [Accessed August 2019].
- [10] “File:Linnet\_kineograph\_1886.jpg,” [Online]. Available: [https://commons.wikimedia.org/wiki/File:Linnet\\_kineograph\\_1886.jpg](https://commons.wikimedia.org/wiki/File:Linnet_kineograph_1886.jpg). [Accessed August 2019].
- [11] Raluca, “Skeletal Based Animation,” 30 May 2016. [Online]. Available: <https://marionettestudio.com/skeletal-animation/>. [Accessed June 2019].
- [12] robertl, “Mecanim Humanoids,” [Online]. Available: <https://blogs.unity3d.com/2014/05/26/mecanim-humanoids/> . [Accessed July 2019].
- [13] F. Ordóñez, “80 level,” 11 May 2017. [Online]. Available: <https://80.lv/articles/vfx-for-games-explained/>. [Accessed June 2019].
- [14] “Unity Particle Pack 5.x - Asset Store,” [Online]. Available: <https://assetstore.unity.com/packages/essentials/asset-packs/unity-particle-pack-5-x-73777>.

- [15] “Minecraft Shaders | Shaderpacks & GLSL Shaders,” [Online]. Available: <https://shadersmod.net/>. [Accessed June 2019].
- [16] “The official home of Legend of Zelda - About,” [Online]. Available: <https://www.zelda.com/about/>. [Accessed August 2019].
- [17] “What Games Are,” [Online]. Available: <https://www.whatgamesare.com/waterfall-development.html>. [Accessed June 2019].
- [18] W. G. Are, “Agile Development - What Games Are,” [Online]. Available: <https://www.whatgamesare.com/agile-development.html>. [Accessed June 2019].
- [19] S. Rogers, “Level Up! The Guide to Great Video Game Design,” 2nd Edition ed., Wiley, 2014, pp. 67-91.
- [20] Game Designing, “The Top 10 Video Game Engines,” [Online]. Available: <https://www.gamedesigning.org/career/video-game-engines/>. [Accessed January 2019].
- [21] “Unreal Engine,” [Online]. Available: <https://www.unrealengine.com/>. [Accessed September 2019].
- [22] “Unity,” [Online]. Available: <https://unity.com/>. [Accessed September 2019].
- [23] Unity3D, “Unity - Multiplatform - Publish your game to over 25 platforms,” [Online]. Available: <https://unity3d.com/pt/unity/features/multiplatform>. [Accessed January 2019].
- [24] Unity3D, “Unity - System Requirments,” [Online]. Available: <https://unity3d.com/pt/unity/system-requirements>. [Accessed January 2019].
- [25] Unity3D, “Unity Store,” [Online]. Available: <https://store.unity.com/>. [Accessed January 2019].
- [26] D. Kayser, “The Gaming Industry Gets Set for an Unreal 2018,” 5 January 2018. [Online]. Available: <https://www.unrealengine.com/en-US/blog/the-gaming-industry-gets-set-for-an-unreal-2018>. [Accessed 20 June 2019].
- [27] Unreal Engine 4, “Unreal Engine 4 on GitHub,” [Online]. Available: <https://www.unrealengine.com/en-US/ue4-on-github>. [Accessed January 2019].
- [28] Unreal Engine 4, “What is Unreal Engine 4,” [Online]. Available: <https://www.unrealengine.com/en-US/what-is-unreal-engine-4>. [Accessed January 2019].
- [29] “C# Guide | Microsoft Docs,” Microsoft, [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/csharp/>. [Accessed October 2019].
- [30] “Unity - Scripting API,” [Online]. Available: <https://docs.unity3d.com/ScriptReference/>. [Accessed January 2019].
- [31] “GIMP - GNU Image Manipulation Program,” [Online]. Available: <https://www.gimp.org/>. [Accessed August 2019].



- [32] “Draw Freely - Inkscape,” [Online]. Available: <https://inkscape.org/>. [Accessed August 2019].
- [33] “Adobe Photoshop,” [Online]. Available: <https://www.adobe.com/products/photoshop.html>. [Accessed August 2019].
- [34] “OpenGameArt.org,” [Online]. Available: <https://opengameart.org/>. [Accessed August 2019].
- [35] “Top free game assets - itch.io,” [Online]. Available: <https://itch.io/game-assets/free>. [Accessed August 2019].
- [36] K. Shadewing. [Online]. Available: <https://opengameart.org/content/grasstop-tiles>. [Accessed July 2019].
- [37] CraftPix.net. [Online]. Available: <https://opengameart.org/content/horizontal-2d-backgrounds>. [Accessed July 2019].
- [38] “Unity - Manual: Animator Controller,” Unity Technologies, [Online]. Available: <https://docs.unity3d.com/Manual/class-AnimatorController.html>. [Accessed August 2019].
- [39] “Getting Started with Unity’s 2D Animation Package - Unity Blog,” 9 November 2018. [Online]. Available: <https://blogs.unity3d.com/2018/11/09/getting-started-with-unitys-2d-animation-package/>. [Accessed August 2019].
- [40] A. Hilton-Jones, “Intro to 2D World Building with Sprite Shape,” 20 September 2018. [Online]. Available: <https://blogs.unity3d.com/2018/09/20/intro-to-2d-world-building-with-sprite-shape/>. [Accessed August 2019].
- [41] “Unity - Manual: Tilemap,” Unity Technologies, [Online]. Available: <https://docs.unity3d.com/Manual/class-Tilemap.html>. [Accessed August 2019].
- [42] “2D Platformer- Asset Store,” [Online]. Available: <https://assetstore.unity.com/packages/essentials/tutorial-projects/2d-platformer-11228>. [Accessed July 2019].
- [43] “Intro to 2D world buildng with Sprite Shape,” [Online]. Available: <https://blogs.unity3d.com/2018/09/20/intro-to-2d-world-building-with-sprite-shape/>. [Accessed July 2019].
- [44] “2D tilemap asset workflow from image to level,” [Online]. Available: <https://blogs.unity3d.com/2018/01/25/2d-tilemap-asset-workflow-from-image-to-level/>. [Accessed July 2019].
- [45] “Freesound - Freesound,” [Online]. Available: <https://freesound.org/>. [Accessed August 2019].
- [46] “99Sounds | Free Sound Effects & Sample Libraries,” [Online]. Available: <http://99sounds.org/>. [Accessed August 2019].
- [47] U. Technologies, “Unity - Manual: Order of Execution for Event Functions,” [Online]. Available: <https://docs.unity3d.com/Manual/ExecutionOrder.html>. [Accessed August 2019].

- [48] S. Rogers, “Level Up! The Guide to Great Video Game Design,” Wiley, 2014, pp. 81-107.
- [49] R. Nystrom, Game Programming Patterns, 2014.

# A. One page design document

**Game name:** Super Milkman

**Game genre:** 2D platformer / fantasy

**Intended game systems:** MacOS and Windows

## **Game summary:**

Some day in a land far away, a humble milkman wakes up and finds out that all his cows are missing. Not only his, but all the cows in the world vanished. A portal appears near his house that leads to the cow world. This world is ruled by the cow princess and she has her own army of cows and allies.

This milkman has the mission to defeat the cow world occupants and bring the cows back to earth, for that he must travel around the cow world and defeat every tribe of cows. Each tribe has its own location, weapons, abilities and protector; the milkman learns new abilities as he progresses in this new world.

After beating all the tribes, the only place to go is the castle, where the cow princess lives. The castle is protected by all the tribes in hope to protect their princess and the milkman needs to defeat them all to complete his mission.

## **Selling points :**

- Multiple zones to explore, from hell to heaven and everything in the middle;
- Different elemental enemy tribes with unique attacks;
- Puzzles to solve.

## **Similar products:**

- Super Metroid
- Shovel Knight
- Cave Story+
- Hollow Knight

## B. Design document



### **Game genre**

2D platformer / fantasy

### **Intended game systems**

Windows

### **Version**

1.0

### **Written by**

Nuno Cardoso

06 November 2018

## Version history

Version	Date (yyyy.mm.dd)	Changes
1.0	2018.11.06	Initial writing

## Game Outlines

### Game storyline summary

Some day in a land far away, a humble milkman wakes up and finds out that all his cows are missing. Not only his, but all the cows in the world vanished. A portal appears near his house that leads to the cow world. This world is ruled by the cow princess and she has her own army of cows and allies.

This milkman has the mission to defeat the cow world occupants and bring the cows back to earth, for that he must travel around the cow world and defeat every tribe of cows. Each tribe has its own location, weapons, abilities and protector; the milkman learns new abilities as he progresses in this new world.

After beating all the tribes, the only place to go is the castle, where the cow princess lives. The castle is protected by all the tribes in hope to protect their princess and the milkman needs to defeat them all to complete his mission.

### Game flow

Super Milkman is a 2D platformer game that takes place in another dimension, a world ruled by cows. This world is divided by tribes, each tribe has its own location and levels. The only way to progress is to beat every level in each tribe zone.

To be able to beat a level the player needs to collect five special items and a key to unlock the gate to the next level. These items are spread around the level, some hidden and some protected by enemies.

When progressing through the levels and zones the player will acquire new weapons and abilities, this will make possible overcoming new obstacles that were previously impossible to get through.

With new abilities comes greater challenges so at the end of each zone there's a boss to ensure the player is ready for the next zone. This boss will be more challenging than normal enemies and has different and more powerful attacks. Defeating a boss will grant access to a new zone. This repeats until the last zone, where the final boss will be the princess cow.

## Character

### History

The playable character lives on earth and he's a humble milk delivery man. He is an adult man called Ed and wears blue/white clothes. He has his own cows that provide him with the milk. Someday all his cows vanish. So he has to find out what happened to them, he finds a portal to the cow world and the adventure begins.

In the cow world, the milkman has to find his way through challenges until he faces the Cow Princess and brings the cows back to earth.



### Player movement

The movement is the basics of any platformer game, moving left, right and jumping.

Crouch and slide will make the player explore tighter places and avoid obstacles in the way.

### Weapons

The only way the player has to defeat an enemy is by using a weapon or jumping in the head, other than that any direct contact between the main character and an enemy will result in the player getting hurt.

All weapons are obtained along the journey. The first weapon is a hammer made from a milk carton, the other obtainable weapons are bombs, gun and a grappling hook, more in the ***Game Mechanics*** section of this document.

# Gameplay

Super Milkman is a 2D platformer game where the player surpasses obstacles and challenges to collect items that will grant him access to the next level.

The game is broken into zones, each zone has its own set of levels, theme, enemies and distinct features. The storyline progresses with the player clearing every level in a zone, this will unlock the next zone until the final zone.

To beat each level the player will have weapons at his disposal to confront and defeat enemies in his way. These weapons will be obtained along the way in key levels or zones to overcome tougher enemies. The player will also be able to craft/obtain weapons with some sort of currency.

When the player opens the game he will be prompt to the main screen with a menu to choose from “Play”, “Options” and “About”.

“About” is just a letter with some text with credits and other stuff.

“Options” is related to gameplay controls and audio.

“Play” will show the level select screen.

The level select screen is composed of the zones and the map of the selected zone with its all levels. Under each level in the map there are three spots that represent the tokens collected. Selecting the level launches it and the game experience starts. The player walks and jumps on platforms and interacts with enemies, the enemies deal damage by throwing/shooting at the player or by physical contact. Within the level there are collectable items that are required to open the gate that grants access to the next level.



## Game World

The cow world is ruled is divided into five different zones protected by the princess's friends.

The world is represented by a sphere and divided into five zones. Each zone expands and has its own map where the different levels are displayed and the player can go from one to another if the level is unlocked.

The **meadows** (Zone 2), is a simple and humble zone where the cows relax and walk around, this zone is protected by *Pegasus* (winged white horse). This zone is very calm and is for the player to get used to the game world, the levels are simple and show the basics mechanics about the game (can be called also a tutorial zone).

The **glaciers** (Zone 5), an icy zone where the cows shoot ice bullets, the floor is slippery and there's ice-cold water, this zone is protected by *Whity* (artic fox). This zone is the first real challenge for the player, this is where the skills acquired in the tutorial zone will be handy.

The **clouds** (Zone 3), a fluffy zone, the ground is like cotton candy and the cows have wings and fly above the player. These cows are special, they shoot thunders downwards that stun and hurt the player. The protector of this zone is *Volteep* (sheep).

The **hell** (Zone 4), the floor is lava, the cows have armour, weapons and are angry! This zone is protected by *Kame* (armoured turtle). Here the enemies have different behaviour, the cows are aggressive, they hunt the player and attack him, here the enemies are warriors and tough to beat.

The final zone is the **castle** (Zone 1), this zone is where the princess resides. As zone this is the kingdom castle area, all the tribes protect it. As this is the last zone, the difficulty must be higher than before, so this zone is a mix of all the enemies that appeared before, all the enemies cooperate in order to try to stop the player from reaching the final boss, the princess.

World zones summary table:

Zone #	Theme	Name	Boss (boss name)
2	Meadow / Fields	Meadows	Winged Horse ( <i>Pegasus</i> )
5	Ice	Glaciers	Artic fox ( <i>Whity</i> )
3	Cloud / Bounce	Clouds	Sheep ( <i>Volteep</i> )
4	Hell / Fire	Hell	Armoured Turtle ( <i>Kame</i> )
1	Castle / Ruins	Castle	Princess Cow

## Game Experience

### The feel

All the art is very simple and humble.

The game is visually simple and does not have many elements at once on the screen that will distract the player.

The game is meant for casual players, with quick levels to beat and a feeling of accomplishment when the level is over.

### Music and sound design

The mood in each zone is defined by its theme, so the music in the levels must complement the zone theme.

Sound effects are a must to bring life to the characters and actions happening on screen. Jumping and landing are mandatory. Other like enemies fainting, attacks, ambient sound (like birds, wind, water, etc.) and enemy and boss speech and attacks are a nice touch and will bring the game together.

### Camera

The camera follows the player movement and has a 'dead zone' where the player can move within a certain distance from the centre and the camera doesn't move.

When the player is near something of interest the camera changes and detaches itself from the centre and reveals the entire scenario, in this mode the player can move and the camera position doesn't change.

### Movies / Cutscenes

For now none.

Maybe just static drawings of the cutscene without any animation, if it is really necessary to understand the game story.

## Enemies

There are different types of enemies depending on the zone.

In the **meadows** zone, the player will encounter a simple enemy, that just walks back and forward and don't fall into pits. The only way this enemy inflicts damage is by touching the player or the player touching the enemy.

Meadows final boss is '*Pegasus*'.

Enemy name	Description
<b>Cow</b>	Just walks, damage by contact. Does not attack, hurts in contact.
<b>Bunny</b>	Walks and jumps the height of one unit. Does not attack, hurts in contact.

In the **glaciers** zone, the enemies will have new abilities, they will shoot ice forward.

Glaciers final boss is '*Whity*'.

Enemy name	Description
<b>Cool Cow</b>	Same as Cow but shoots Ice bullets.
<b>Snow Cow</b>	Throw giant snowballs in from of him.

In the **clouds** zone, the enemies will be able to fly and shoot downwards.

Clouds final boss is '*Volteep*'.

Enemy name	Description
<b>Electric Cow</b>	Flies above the player in a specific route and shoots down periodically.

In the **hell** zone, the enemies will be tougher and stronger, these enemies will shoot in a circle while jumping.

Hell final boss is '*Kame*'.

Enemy name	Description
<b>Armoured Cow</b>	Has armour, more health and attacks while jumping (circle of bullets).
<b>Heavy Cow</b>	Throws fire boulders in front of him.

The last zone is the **castle**, in this zone every enemy from other zones appear. This zone is intended to be protected by all the cow tribes that have a mission in common that is protecting the princess. No new enemy is introduced in this zone.

Castle final boss is '*Princess Cow*'.