EPS
Escola Politècnica
UdG Superior

Escola Politècnica Superior

Universitat de Girona

Development of a 2D lateral action videogame for Android platforms. Desenvolupament d'un videojoc d'acció lateral per a plataformes Android.

Projecte/Treball Fi de Carrera

GEINF. Pla 2016

Document: Memòria

Autor: Robert Bosch

Director: Gustavo Patow

Departament: Informàtica, Matemàtica Aplicada i Estadística

Àrea: LSI

Convocatoria: JUNY/2016

Contents

Chapter 1

Introduction

In this chapter we are going to explain the main motivations, objectives and purpose of this project. Also at the end, we will present a brief summary of the distinct sections of this memory.

1.1 Introduction

A videogame is a game played in an electronic device, where one or diverse players interact with it.

Nowadays, the videogame industry is an ever growing sector able to compete with both cinema and music industries, actually surpassing their profits. The future of videogames is constantly changing and open to new markets, for example games on smartphones and tablets. 1.1

An endless runner game is one where the player keeps running forward, avoiding or destroying certain obstacles and/or enemies. The game developed in this project is specifically a 2D endless runner, developed for Android.

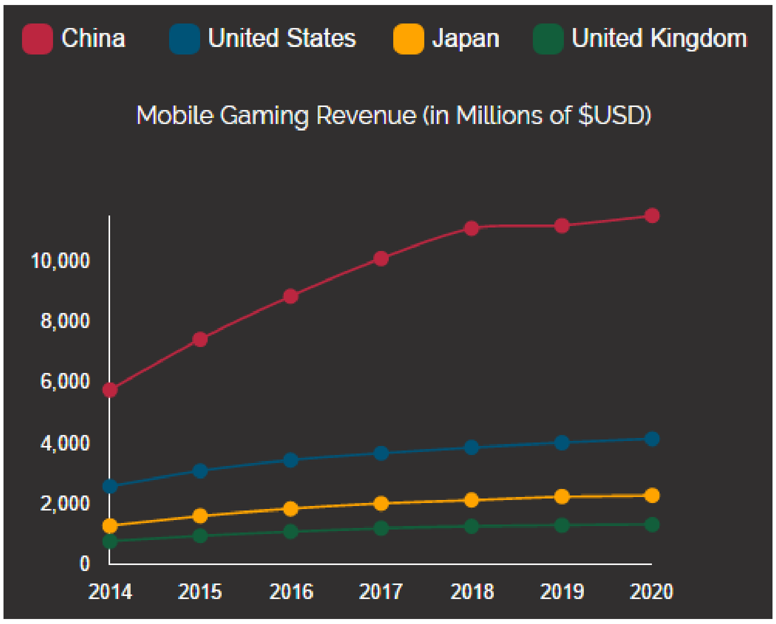


Figure 1.1: Mobile Gaming Revenue (in Millions of $USD)

1.2 Personal motivations

I personally find videogames an interesting aspect of the computer engineering degree, and a good way to put together knowledges about programming, software engineering, multimedia, computer graphics, and also art, which also interests me on a personal level.

Furthermore, I am also interested in learning about the Unity game engine and getting used to its functionalities, and this project is a great way to learn thoroughly about a game development platform.

1.3 Project motivations

The game of this project is a very simple idea, which can be further developed into a more complex game, simply by adding features. So, this game could be further developed and become a good seller in the Android store, called Play Store, or at least give the creator some credit and publicity.

1.4 Project purposes

The main purpose of this project is to develop a game of the 2D endless runner game for Android.

The main character would be able to interact with the game through swipes and touches on the screen, without any control GUI. Enemies and obstacles will keep appearing from the right side of the screen, moving to the left. Then the player will have to use one of the available gestures to either avoid or destroy the menace. Certain enemies require certain gestures, and the player is required to react quickly at the sight of a hostile object approaching.

This videogame will be developed using a free game development platform: Unity.

1.5 Objectives

The main objective of this project is to develop a 2D endless runner game for Android. The tasks can be split in the following:

- Study of the Android platform
- Study of the Unity platform
- Study of Unity of programming languages (C#)
- Planning of the game. Script and defining the elements of interaction.
- Implementation of possible actions by the player and interaction with the environment.
- Development of a prototype of the game
- Design and implementation of algorithms of artificial intelligence of enemies.
- Implementation of designs and animations of sprites for Android.
- Creating the final model of the game
- Refinement of the documentation.

1.6 Structure of this memory

This project report is divided into 15 chapters, which include all project information and conclusions.

- Chapter 1. Introduction, motivation, purposes and objectives of the project. It explains the reason of its development, its objectives and the organization of this document.
- Chapter 2. Feasibility study. It specifies the parameters that make possible the development of the project.
- Chapter 3. Methodology. This chapter defines the used methodology, explaining the reasons for its use.
- Chapter 4. Planning. It defines the strategy followed to develop and finalize the objectives and the timing.
- Chapter 5. Framework and previous concepts. This chapter explains the various issues surrounding the development of the project to better understand the project.
- Chapter 6. System Requirements. Contains various system requirements for full operation of the application.
- Chapter 7. Studies and decisions. It describes the hardware and software libraries used during the development project.
- Chapter 8. Analysis and design. This chapter describes the design used in the system, as well as specifications, deployment diagrams, classes and methods of the project.
- Chapter 9. Implementation and testing. In this chapter it is explained how the different parts of the system are implemented and tested.
- Chapter 10. Deploying and results. Describes the process to deploy the game and presents the final results.
- Chapter 11. Conclusions. The conclusions are set out at the end of the project.
- Chapter 12. Future work. It describes improvements and features for new versions of the game.
- Chapter 13. Bibliography. This chapter contains biographical references used by the project.
- Chapter 14. User manual. The last chapter includes a manual that explains the operation and installation in order to use the game correctly.

Chapter 2

Feasibility study

2.1 Resources needed to develop this project

The Unity engine, while being itself free, has a series of system requirements, different for developing and for playing games.

In bold we highlight the available resources for the development.

2.1.1 Developer requirements

OS: Windows 7 SP1+, **Windows 8**, 10; Mac OS X 10.8+. Windows XP & Vista are not supported; and server versions of Windows & OS X are not tested.

GPU: Graphics card with DX9 (shader model 2.0) capabilities. Anything made since 2004 should work.

Additional platform development requirements:

1. iOS: Mac computer running minimum OS X 10.9.4 version and Xcode 6.x.
2. **Android: Android SDK and Java Development Kit (JDK).**
3. Windows 8.1 Store Apps / Windows Phone 8.1: 64 bit Windows 8.1 Pro and Visual Studio 2013 Update 2+.
4. WebGL: Mac OS X 10.8+ or Windows 7 SP1+ (64-bit editor only)

So the OS and GPU requirements were already ok, I only had to download and install Unity, the Android SDK and JDK (Java Development Kit).

2.1.2 Player requirements

1. Desktop:
   (a) OS: Windows XP SP2+, Mac OS X 10.8+, Ubuntu 12.04+, SteamOS+
   (b) Graphics card: DX9 (shader model 2.0) capabilities; generally everything made since 2004 should work.
   (c) CPU: SSE2 instruction set support.
   (d) Web Player (deprecated): Requires a browser that supports plugins, like IE, Safari and some versions of Firefox

2. iOS: requires iOS 6.0 or later.
3. **Android: OS 2.3.1 or later; ARMv7 (Cortex) CPU with NEON support or Atom CPU; OpenGL ES 2.0 or later.**
4. WebGL: Any recent desktop version of Firefox, Chrome, Edge or Safari
5. Windows Phone: 8.1 or later
6. Windows Store Apps: 8.1 or later

All the devices used in testing fulfill this requirements.

2.2 Initial budget

This project requires no initial investment. For the future, if we decide to publish it on Steam Greenlight, it will only require about 100€ and then to pay 30% of profits to Steam once we start selling. The program used for the development of the games is Unity 5, which requires no special given license for use. If the sells were to exceed 100000€ annually, then it would require a professional license.

2.3 Human resources

The optimal team for the realization of this project required a project manager, programmers, designers, analysts and artists of several disciplines. The tasks would be defined and divided depending on the role of the worker and the needs of the moment.

- The task of a programmer is to transform all the game ideas into functional code.
- The task of a designer would be to define the structure of the code and algorithms used.
- The task of the analyst would choose what is the best tool developments, investigate market trends and future technologies.
- The tasks of the artists are to create scenarios, sprites, soundtracks, sound effects, backgrounds, interfaces and all the artistic aspects of the videogame.
- The task of the project manager would be to set priorities and manage the project team.

In this project all roles are performed by myself, with the help of the tutor of the project.

2.4 Technological viability

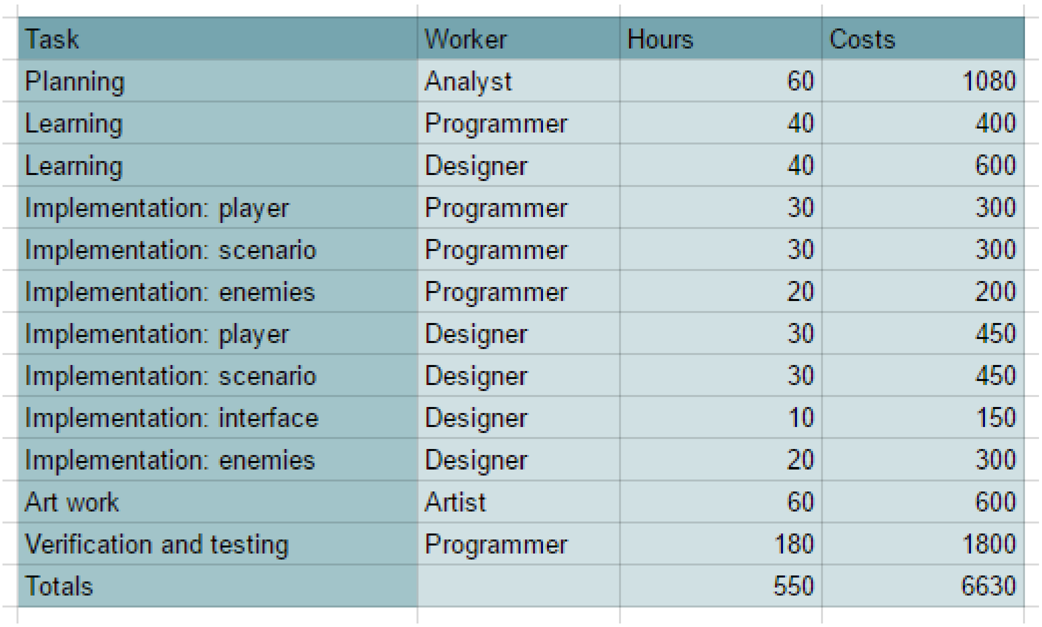
2.4.1 Economic viability

Estimated economic viability of the project can be divided into two parts, and human costs costs of equipment and software.

2.4.2 Human costs

The human costs would be based on the following costs per hour:

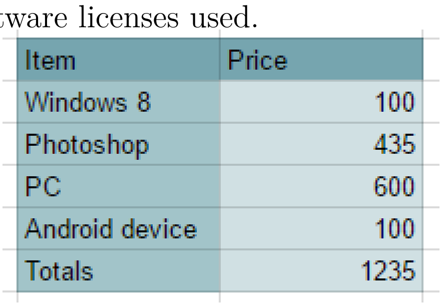
- Programmer costs: 10€/hora
- Designer costs: 15€/hora
- Analyst costs: 18€/hora
- Artist costs: 10€/hora
- Manager costs: 18€/hora



Task	Worker	Hours	Costs
Planning	Analyst	60	1080
Learning	Programmer	40	400
Learning	Designer	40	600
Implementation: player	Programmer	30	300
Implementation: scenario	Programmer	30	300
Implementation: enemies	Programmer	20	200
Implementation: player	Designer	30	450
Implementation: scenario	Designer	30	450
Implementation: interface	Designer	10	150
Implementation: enemies	Designer	20	300
Art work	Artist	60	600
Verification and testing	Programmer	180	1800
Totals		550	6630

2.4.3 Equipment costs

The cost of machinery and software should take into account the costs of used computers and software licenses used.



Item	Price
Windows 8	100
Photoshop	435
PC	600
Android device	100
Totals	1235

2.4.4 Total costs

In conclusion the estimated cost for the development of the project would be about 7865€.

Chapter 3

Methodology

We have not used a standard methodology for the general development of this project, instead we choose to use a personalized methodology, due to its advantages in the concrete problem of developing a videogame.
The different steps of this methodology are as follows:

1. Choosing the tools and programming language.
2. Learning the language and tools chosen.
3. Dividing the project into different parts.
4. Planning the work according to the different parts in which the project has been divided.
5. Developing each part of the project.
6. Checking correctness of each part of the project. If not correct, return to the above-mentioned step.
7. Merging the different parts of the project. If errors are found, return to the fifth point.
8. Generating different scenarios for the proper functioning of the project. If errors are found, return to the fifth point.
9. Nuancing the documentation written during the project development.

In the following diagram 3 the steps used are graphically represented.

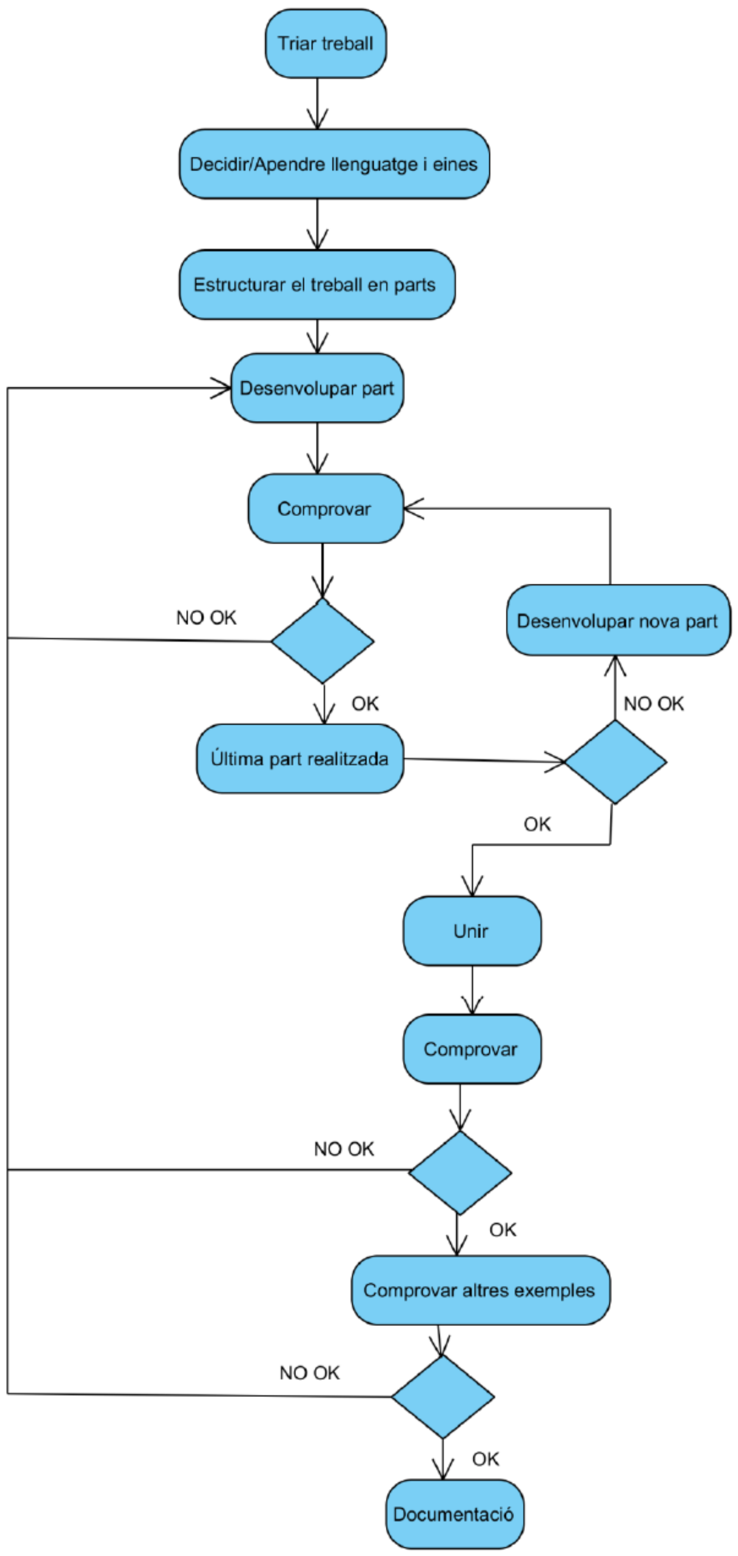


Figure 3.1: Project Methodology

Chapter 4

Planning

In this chapter the various tasks of the project and the timing assigned to them will be described.

4.1 Working plan

The working plan of the videogame is divided in tasks, which broadly are planning, learning, implementation, verification and documentation.

4.2 Planned tasks

4.2.1 Planning

The first task is to decide the tasks and the scheduling, and also how the game is going to be, and the general software structure.

4.2.2 Learning

In order to begin working with Unity and C#, it is necessary to learn how they work and how they interact.

4.2.3 Implementation

The implementation consists of software engineering, design, programming and artwork. It can be divided in the specific game parts.

4.2.3.1 Player

The interaction with the player is done through the main character. This task consists in managing the player input/interaction and merging it with the actions of the main character, and also making the character react to the environment.

4.2.3.2 Scenario

The scenario is where the game takes place. It is mostly a design task, but also has algorithms for its generation and movement.

4.2.3.3 Interface

The interface of this game is minimal, it is meant to display the score and allow the player to restart the game. There is no ingame interface, as the player has only one life.

4.2.3.4 Enemies

The enemies and obstacles require a certain behavior scripting, and also they must interact with the environment and player.

4.2.3.5 Art design

The art design consists of:

1. Sprites and animation
2. Backgrounds
3. Scenarios
4. Music and sound
5. Interface design

For the music and sound part, free material (creative commons) has been used. As for the other aspects, they have been developed by me.

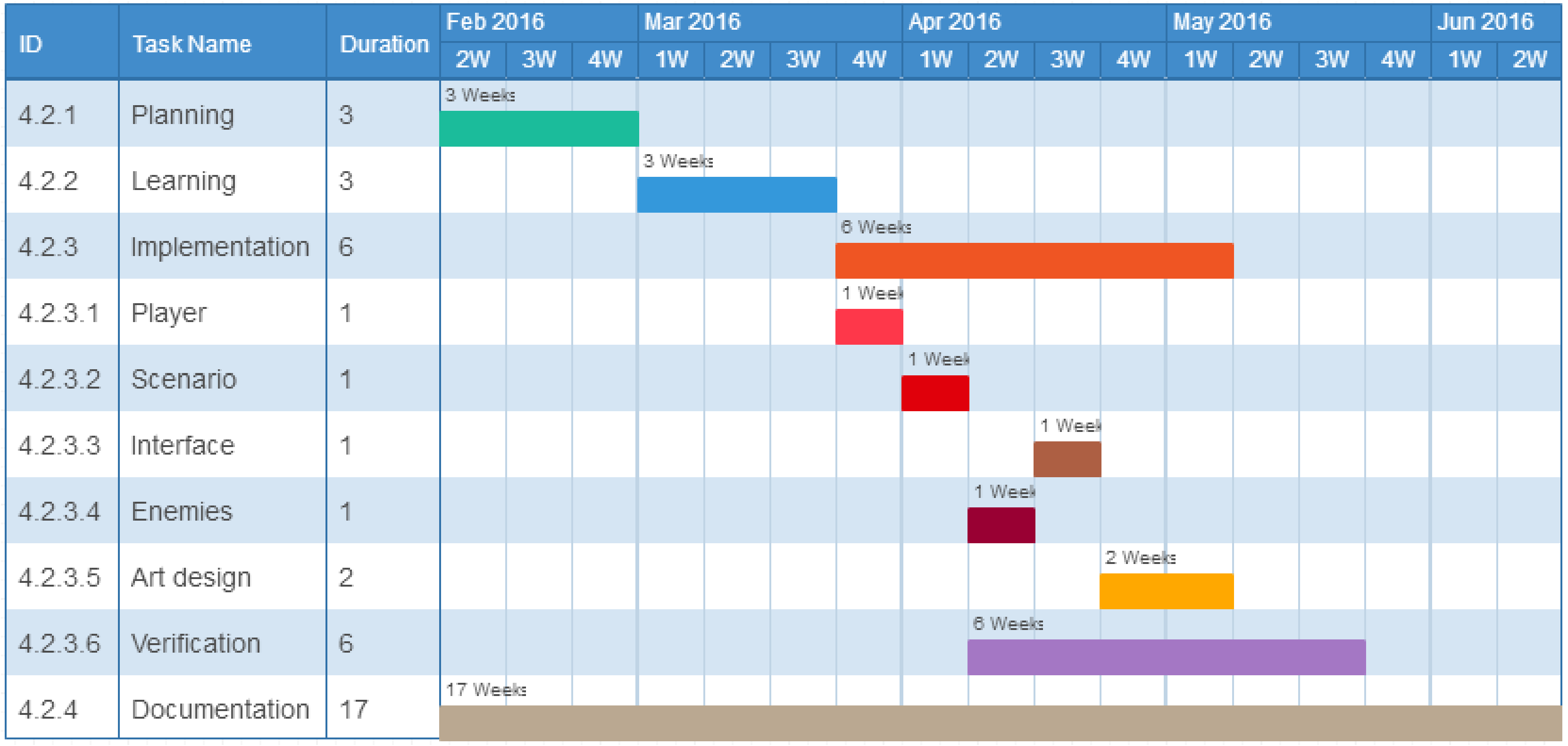
4.2.4 Verification

During the developing process the algorithms have been verified and dynamically modified to suit the needs of the game.

4.2.5 Documentation

The documentation written during the development process is then refined and perfected.

4.3 Estimated scheduling



ID	Task Name	Duration	Bar
4.2.1	Planning	3	3 Weeks
4.2.2	Learning	3	3 Weeks
4.2.3	Implementation	6	6 Weeks
4.2.3.1	Player	1	1 Week
4.2.3.2	Scenario	1	1 Week
4.2.3.3	Interface	1	1 Week
4.2.3.4	Enemies	1	1 Week
4.2.3.5	Art design	2	2 Weeks
4.2.3.6	Verification	6	6 Weeks
4.2.4	Documentation	17	17 Weeks

4.4 Expected results of every task

4.4.1 Planning

The planning is expected to suit the timeline.

4.4.2 Learning

It is expected to adquire the knowledge necessary to develop the project.

4.4.3 Implementation

It is expected for the game to be functional/playable.

4.4.3.1 Player

The player must be able to interact with the game, and of course the main objective of it is to be fun to play.

4.4.3.2 Scenario

The scenario must be quickly understandable but also aesthetically pleasing.

4.4.3.3 Interface

The interface of this game has the only purpose to show the score, so it's not meant to be very elaborate.

4.4.3.4 Enemies

The enemies are meant to difficult the game and make the player react quickly, thus creating the game action.

4.4.3.5 Art design

The art is expected to be monothematic and visually appealing, but also clean and easy to understand by the player.

4.4.4 Verification

The resulting game must be bug-free and fully tested.

4.4.5 Documentation

The resulting documentation must reflect all the decisions taken, and it must be easily readable.

Chapter 5

Framework

5.1 Videogame engines

In order to develop a game, the most frequent option is to use a game engine. A game engine is a software that enables the creation, design and display of a video game. They are based on different parts, such as rendering, physics, collision detection, sound, scripting, animation, artificial intelligence, networking, streaming, memory management, multithreading, location and graphic scenes.
This kind of software is designed for game developers so that they don't have to worry about direct manipulation over the graphics libraries on a concrete operating system.
Also, game engines aren't used only in videogaming ambits, but are also used in architectural visualizations, training simulations, modeling and physical simulation tools for realistic animations and film scenes.

5.2 Examples of videogame engines

5.2.1 Unreal Engine

It is an engine created by Epic Games. It allows developing for the following platforms: Android, HTML5, iOS, Linux, Mac, Oculus, PC, PS4, SteamOS, Xbox One. It is free. It is based in C++.

The big difference of this engine over others is that it has a system of nodal design, suitable for users with little programming knowledge who want to use the advanced features of the system. It also has a a good amount of tutorials and feedback from other users.

5.2.2 CryEngine

It is an engine created by Crytek. It allows developing for the following platforms: Android, iOS, PC, PS3, PS4, Xbox 360, Xbox One, Wii U. It is free.

It has been used in all of their titles with the initial version being used in Far Cry, and continues to be updated to support new consoles and hardware for their games. It has also been used for many third-party games under Crytek's licensing scheme, including Sniper: Ghost Warrior 2 and SNOW.

5.2.3 GameMaker

It is an engine created by YoYo Games. It allows developing for the following platforms: Android, Browser, iOS, Mac, PC, PS3, PS4, Vita, Windows Phone. It is free, but the paid version has more features. It is based in GML. It is oriented to 2D development.

GameMaker accommodates the creation of cross-platform and multi-genre video games using drag and drop action sequences or a sandboxed scripting language known as Game Maker Language.

5.2.4 Havok Vision Engine

It is an engine created by Havok. It allows developing for the following platforms: Android, iOS, PS3, Vita, Wii, Wii U, Windows 8, Xbox 360. It is based in C++.

Havok provides a physics engine component and related functions to video games. Since the SDK's launch in 2000, it has been used in over 600 video games. Havok supplies tools (the "Havok Content Tools") for export of assets for use with all Havok products from Autodesk 3ds Max, Autodesk Maya, and Autodesk Softimage.

5.2.5 Project Anarchy

It is an engine created by Havok. It allows developing for the following platforms: Android, iOS, PC, Tizen. It is an Android oriented version of the Havok Vision Engine.

5.2.6 ShiVa

It is an engine created by ShiVa. It allows developing for the following platforms: Android, Blackberry, iOS, Linux, Mac, PS3, SteamOS, Vita, Wii, Windows, Windows Phone 8, Xbox 360. It is based in Lua, but has a C++ plug-in system.

Numerous applications have been created using ShiVa, including the Prince of Persia 2 remake for Mobiles and Babel Rising published by Ubisoft. With ShiVa 2.0, the next version of the Editor is currently under heavy development. ShiVa users with licenses newer than Jan 1st 2012 are invited to download the beta builds, test thoroughly and provide feedback.

5.2.7 BigWorld

It is an engine created by Wargaming. It allows developing only for the following platforms Browser and PC. It's oriented to MMOG (Masive Multiplayer Online Games) development.

5.2.8 GameSalad

It is an engine created by Gamesalad. It allows developing for the following platforms: Android, HTML5, iOS, Mac, Windows. It is free, but has a paid version with more features.

5.2.9 Leadwerks

It is an engine created by Leadwerks. It allows developing for the following platforms: Linux, Mac, PC. It is based in OpenGL.

It is aimed primarily at non-programmers for composing games in a drag-and-drop fashion.

5.2.10 App Game Kit

It is an engine created by The Game Creators. It allows developing for the following platforms: Android, iOS, Mac, PC. It is not free, but has a free trial version. It is based in C++ or AGK BASIC.

Software produced with the App Game Kit is written in a language called AGK Script. This language has powerful inbuild commands including commands for 2D graphics, physics and networking. The commands make use of the platforms' native functions to improve performance. They are also designed to enhance code readability. The AGK Script commands have extensive online documentation.

5.2.11 FPS creator Reloaded

It is an engine created by The Game Creators. It allows developing for the following platforms: Oculus Rift, PC. It is not free.

Its main objective is to develop an FPS game (First Person Shooter).

5.2.12 Reach3dx

It is an engine created by Gamebase. It allows developing for the following platforms: Android, Flash, HTML5, iOS. It is not free. It is based in OpenFL and Haxe.

5.2.13 HeroEngine

It is an engine created by Idea Fabrik. It allows developing only for PC. It is not free. It is designed for massively multiplayer and online games.

At first developed for the company's own game Hero's Journey, the engine won multiple awards at tradeshows, and has since been licensed by other companies.

The engine has online creation. For example, one developer can be creating a house and the entities inside, while another works on the landscaping and terrain around it. Each sees the other's work in real time.

5.2.14 Marmalade

It is an engine created by Marmalade. It allows developing for the following platforms: Android, BlackBerry, iOS Mac, Windows, Tizen. It is not free. It is based in C++, but also allows Lua, HTML or Objective-C.

The underlying concept of the Marmalade SDK is write once, run anywhere so that a single codebase can be compiled and executed on all supported platforms rather than needing to be written in different programming languages using a different API for each platform. This is achieved by providing a C/C++ based API which acts as an abstraction layer for the core API of each platform.

5.2.15 Turbulenz

It is an engine created by Turbulenz. It allows developing only for Browser. It is based on HTML5.

5.2.16 Unity

It is an engine created by Unity Technologies. It allows developing for the following platforms: Android, BlackBerry, iOS, Linux, Mac, PS3, PS4, Xbox 360, Xbox One and more. It is free to use, but if the game earnings exceed 100000€ annually, a payment license is required. Also, if the games developed are commercial, a certain amount of the earnings will go to Unity Technologies. It allows programming in C# or Javascript.

5.3 Chosen engine

The chosen engine for this project is Unity.
The main criteria in order to choose an engine are the following:

- It has to allow Android developing.
- It has to be free.
- Its scripting language has to be a familiar one.
- It has to have a good amount of tutorials.
- Its community has to be large in order to get support.
- Preferably it has to allow multiplatform development.

So by following these criteria I have chosen Unity, for I think it fulfills the needs of this project.

Chapter 6

System requirements

This chapter describes the requirements to be met by the system, both functional and nonfunctional. On one hand the functional requirements are the services offered by the application regardless of its implementation. On the other hand, the non-functional requirements are constraints imposed by the platform or the problem.

6.1 Functional requirements

The application requirements are:

- Begin the game
- Move the character to avoid enemies and obstacles
- Die when a character is damaged
- Show the score and be able to restart

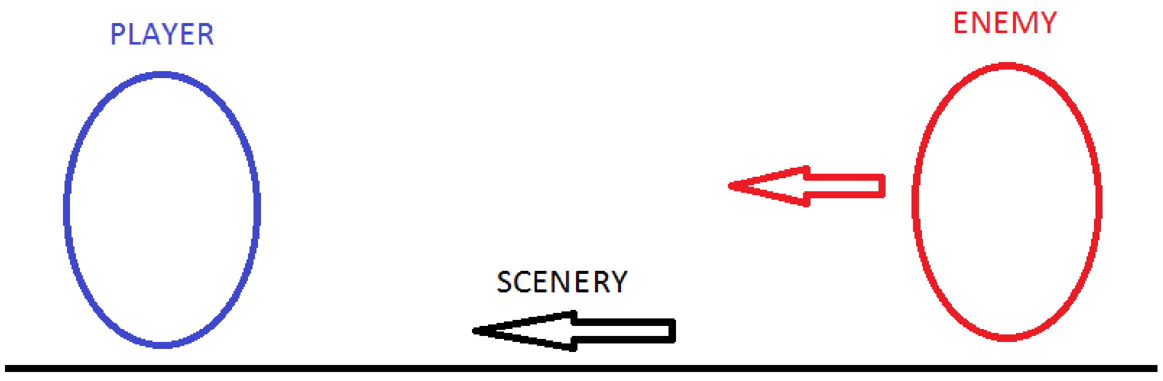


Figure 6.1: Game schematic diagram

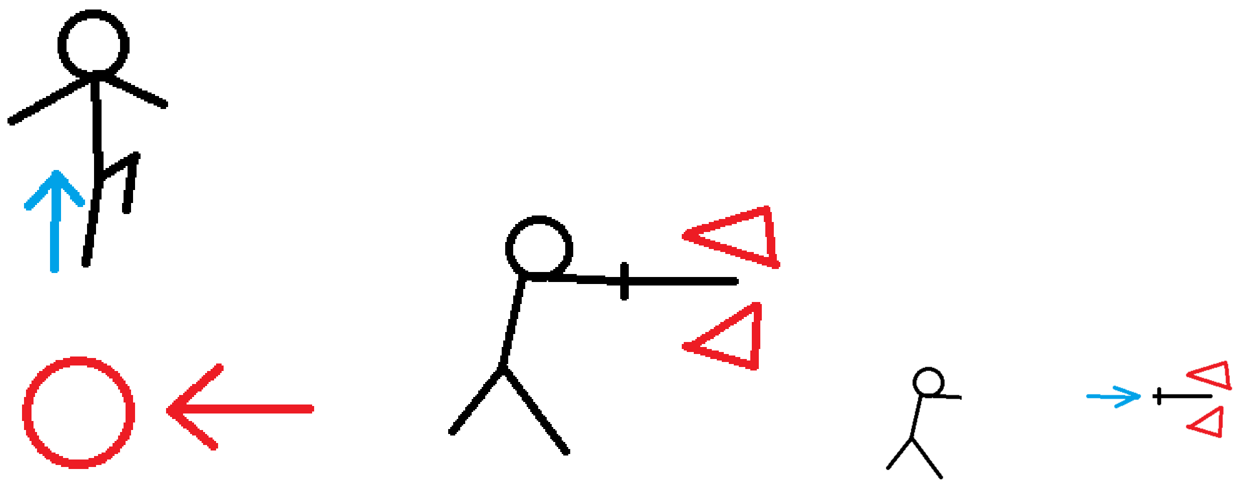


Figure 6.2: Jump, attack and throw respectively.

The player is always in the same horizontal position, but the enemies and the scenery move to the left. The enemies spawn on the right. The player must be able to either avoid or destroy the enemies.

The player can do three actions: jump (swipe up), attack (swipe down) and throw (swipe right). Attacking and throwing can both be performed in the air while jumping, but actions cannot be interrupted.

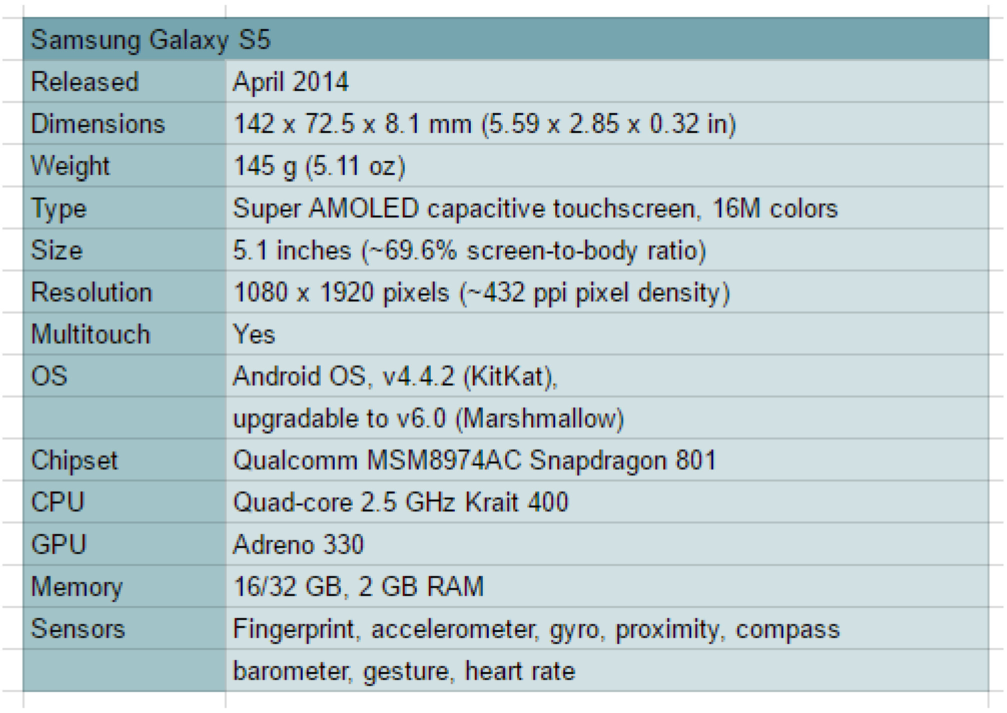
Jumping will allow the player to avoid obstacles, and it is the most basic defensive action. Attacking will destroy an enemy who is very close to the player, and throwing will hit an enemy at a longer distance.

The game rhythm changes creating relaxed temporal zones and stressful temporal zones. This allows the player to relax from stressful game parts and also not get bored in a very easy zone.

The rhythm changes can also be appreciated by the player through a change in the illumination. The darker the light gets, the faster everything will be moving, and viceversa. The speed does not change discretely but in a continuous fashion, and the rhythm changes in a prefixed random interval, between 1.5 - 2.5 minutes.

6.2 Nonfunctional requirements

Basically, non-functional requirements relate to the availability of resources, security or external interfaces. As for the security matters, there is no access control requirement for the program. Also, the program does not keep any confidential data. The implementation and testing of the game have been carried out in the following devices:



Samsung Galaxy S5	
Released	April 2014
Dimensions	142 x 72.5 x 8.1 mm (5.59 x 2.85 x 0.32 in)
Weight	145 g (5.11 oz)
Type	Super AMOLED capacitive touchscreen, 16M colors
Size	5.1 inches (~69.6% screen-to-body ratio)
Resolution	1080 x 1920 pixels (~432 ppi pixel density)
Multitouch	Yes
OS	Android OS, v4.4.2 (KitKat),
	upgradable to v6.0 (Marshmallow)
Chipset	Qualcomm MSM8974AC Snapdragon 801
CPU	Quad-core 2.5 GHz Krait 400
GPU	Adreno 330
Memory	16/32 GB, 2 GB RAM
Sensors	Fingerprint, accelerometer, gyro, proximity, compass
	barometer, gesture, heart rate

Chapter 7

Studies and decisions

Hereafter we will describe the characteristics and features of Unity used in the development of this project. For the Unity class diagram, please refer to section 7.8.

7.1 Basic concepts

7.1.1 GameObject

This is the Base class for all entities in Unity scenes.

It inherits from Object, which is the base class for all objects Unity can reference.

In Unity, **Prefabs** are non instantiated objects (class only) which can be instantiated into the scene in any moment.

7.1.1.1 Public functions

- **AddComponent**: Adds a component class named className to the game object.
- **BroadcastMessage**: Calls the method named methodName on every MonoBehaviour in this game object or any of its children.
- **CompareTag**: Checks if this game object tagged with a given tag
- **GetComponent**: Returns the component of Type type if the game object has one attached, null if it does not.
- **GetComponentInChildren**: Returns the component of Type type in the GameObject or any of its children using depth first search.
- **GetComponentInParent**: Returns the component of Type type in the GameObject or any of its parents.
- **GetComponents**: Returns all components of Type type in the GameObject.
- **GetComponentsInChildren**: Returns all components of Type type in the GameObject or any of its children.
- **GetComponentsInParent**: Returns all components of Type type in the GameObject or any of its parents.
- **SendMessage**: Calls the method named methodName on every script in this game object.

- **SendMessageUpwards**: Calls the method named methodName on every script in this game object and on every ancestor of the behaviour.
- **SetActive**: Activates/Deactivates the GameObject.

7.1.1.2 Static functions

- **CreatePrimitive**: Creates a game object with a primitive mesh renderer and an appropriate collider.
- **Find**: Finds a game object by name and returns it.
- **FindGameObjectsWithTag**: Returns a list of active GameObjects tagged tag. Returns empty array if no GameObject was found.
- **FindWithTag**: Returns one active GameObject tagged tag. Returns null if no GameObject was found.

7.1.1.3 Inherited public functions

These are the public functions inherited from Object.

- GetInstanceID: Returns the instance id of the object.
- ToString: Returns the name of the game object.

7.1.1.4 Inherited static functions

These are the static functions inherited from Object.

- Destroy: Removes a GameObject, component or asset.
- DestroyImmediate: Destroys the object obj immediately. It is strongly recommended to use Destroy instead.
- DontDestroyOnLoad: Makes the object target not to be destroyed automatically when loading a new scene.
- FindObjectOfType: Returns the first active loaded object of Type type.
- FindObjectsOfType: Returns a list of all active loaded objects of Type type.
- Instantiate: Returns a copy of the object original.

7.1.2 Component

This is the Base class for everything attached to GameObjects. The code will never directly create a Component. Instead, we write a script code, and attach the script to a GameObject.

Component also inherits from Object, so it has the same inherited methods (Destroy, Instantiate:.)

7.2 2D Graphics

7.2.1 Camera

It is the 2D view of the scene. See figure 7.1. Its perspective and position can be changed.

7.2.2 Light

It illuminates the scene. See figure 7.2. It can be a directional light, a point light or an ambient light.

7.2.3 Sprites

A sprite is a two-dimensional bitmap that is integrated into a larger scene. All the sprites made for this project can be found in the annex.

7.2.4 Textures

A texture is an image that fills a 3D object surface.

7.3 2D Physics

7.3.1 2dCollider

Base class of all colliders: BoxCollider, SphereCollider, CapsuleCollider, MeshCollider, EdgeCollider.

A collider is used to detect collisions between Rigidbody2Ds. 7.8

7.3.2 Rigidbody2d

This class controls an object's position through physics simulation. Adding a Rigidbody component to an object will put its motion under the control of Unity's physics engine. Even without adding any code, a Rigidbody object will be pulled downward by gravity and will react to collisions with incoming objects if the right Collider component is also present.
The Rigidbody also has a scripting API allows applying forces to the object and controls it in a physically realistic way.

7.4 Interface

7.4.1 Canvas

It is an element that can be used for screen rendering. See figure 7.3. Elements on a canvas are rendered after scene rendering, either from an attached camera or using overlay mode.
It is used in this project for displaying the interface.

7.5 User Input

7.5.1 Keyboard Input

In the Input settings we configure the user keyboard input. In this project this is only used for debugging.

7.5.2 Touch Input

The Unity touch detection can detect single touches but not swipes. So, in order to detect swipes, an external library or an algorithm must be used.

7.6 Sound

7.6.1 Audiosource

Audiosource is the main Unity component for sound management. In 2D, the sounds have no 3D special effect.

7.7 Libraries used

7.7.1 UnityEngine

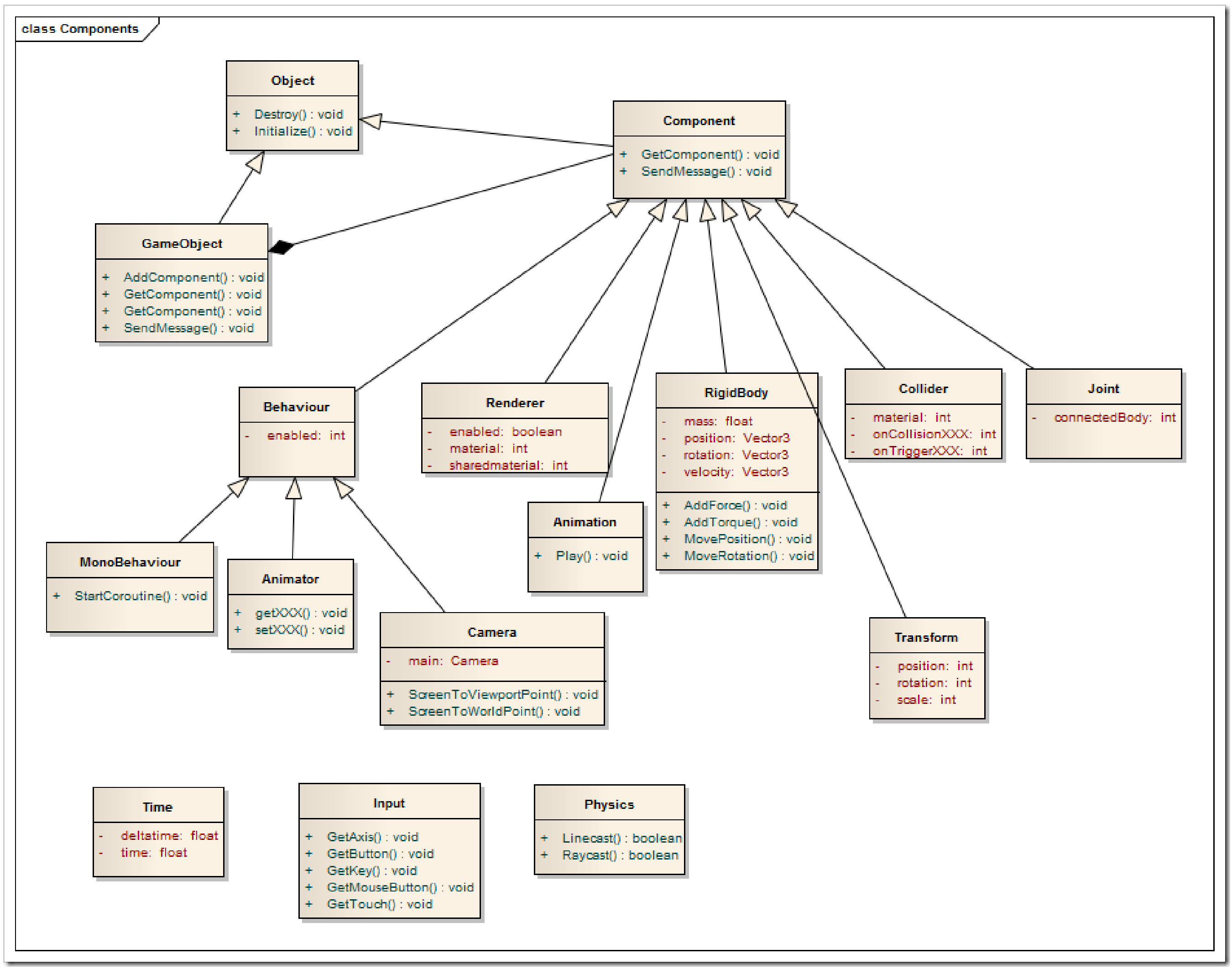
It is the main Unity library which contains the basic objects and methods described above.

7.7.2 System

System contains the C# classes and data types.

7.8 Unity class diagram

This is a simplified view of the Unity class diagram.



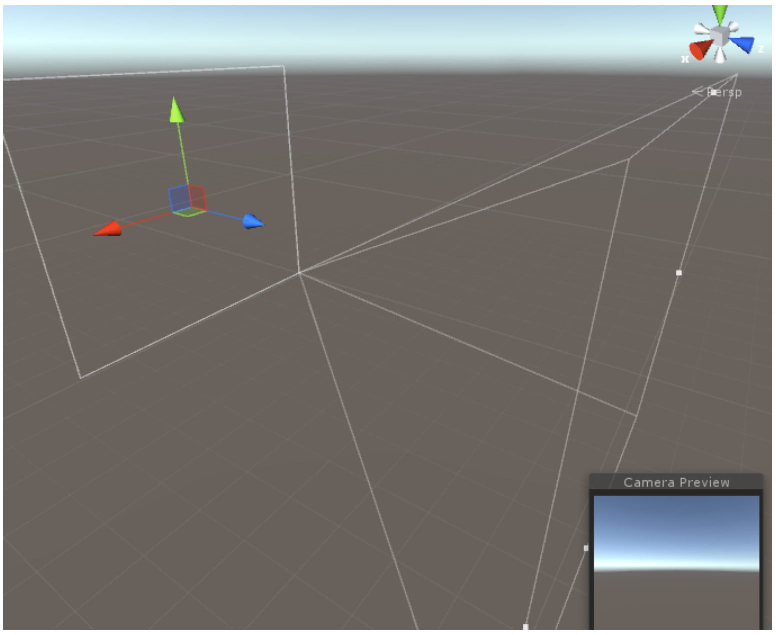


Figure 7.1: Camera

Figure 7.2: Orange point light

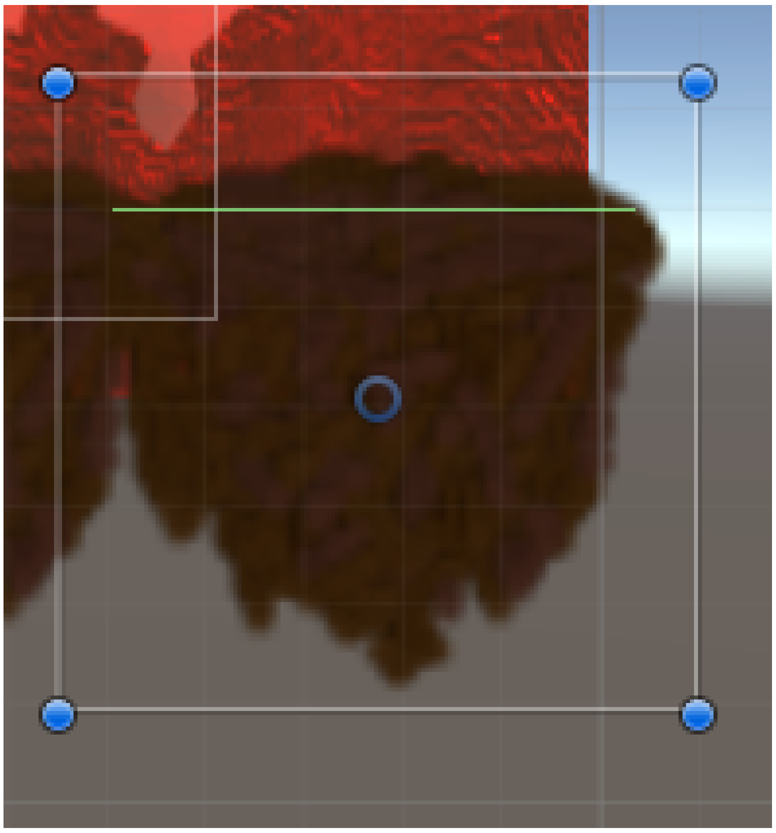


Figure 7.3: Edge Collider (Green line)

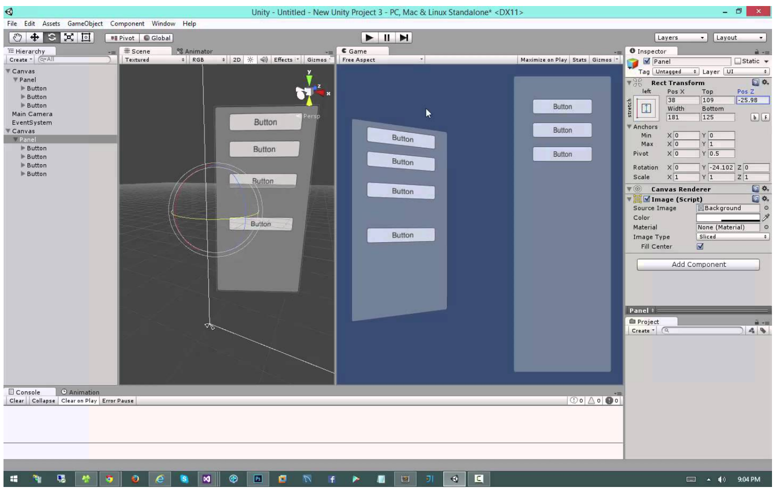

Unity - Untitled - New Unity Project 3 - PC, Mac & Linux Standalone* <DX11>
File Edit Assets GameObject Component Window Help
Hierarchy
Scene
Game
Inspector
Button
Button
Button
Button
Rect Transform
Canvas Renderer
Image (Script)
Add Component
Project
Console

Figure 7.4: Canvas

Chapter 8

Analysis and Design

In this chapter we will describe the videogame in detail and the elements of its design.

8.1 Description

As said in the above chapters, this videogame is a 2D endless runner. The player is always in the same horizontal position, but the enemies and the scenery move to the left. The enemies spawn on the right. The player must be able to either avoid or destroy the enemies.

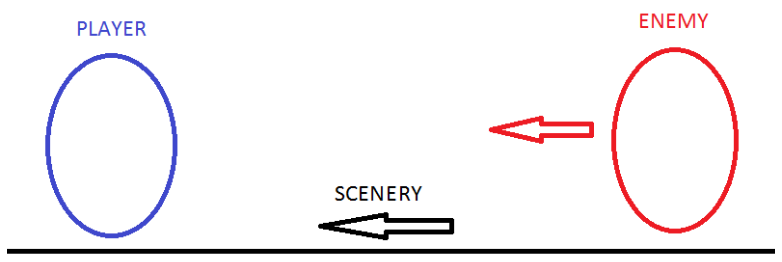


Figure 8.1: Game schematic diagram

8.2 Analysis

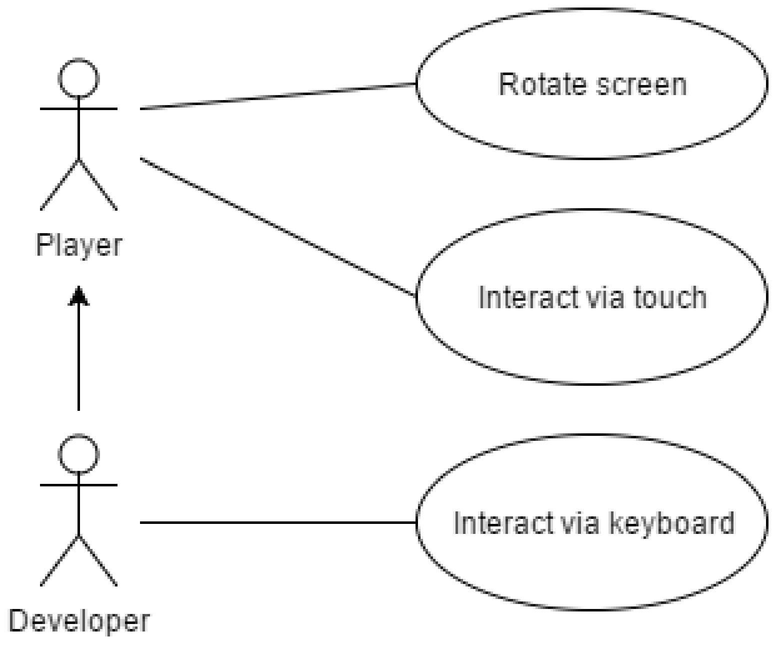
8.2.1 Actors

An actor in software engineering is a role performed by a user or another system that interacts with the subject. In this project we have only 2 actors, the player and the developer, which is an extension of the player with additional capabilities (keyboard interaction) that allow debugging the game.

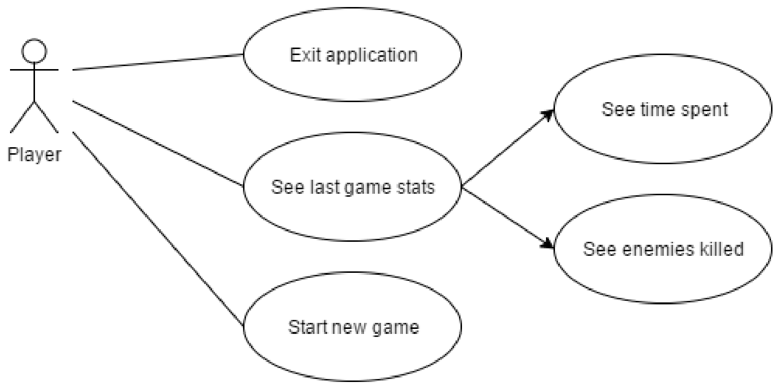
8.2.2 Use cases

A use case is a sequence of interactions that take place between the system and the actors in response to an event initiated by the actor. The use cases are divided in 3 diagrams: interaction, main menu and game.

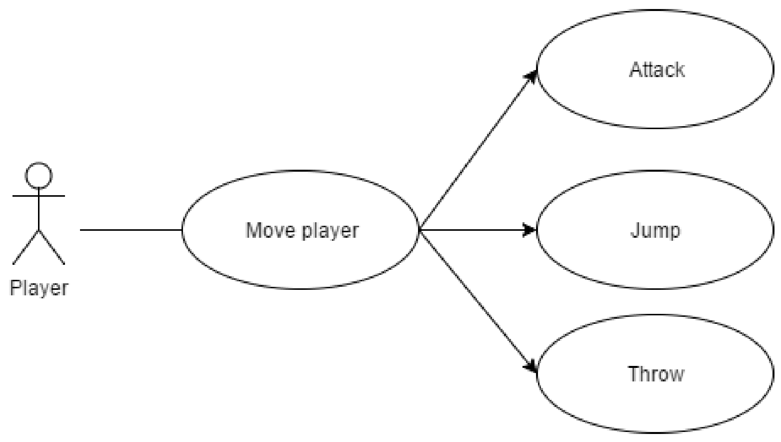
8.2.2.1 Interaction



8.2.2.2 Main menu



8.2.2.3 Game



8.2.3 Use case sheets

Use case sheet	Rotate screen
Description	Player fisically rotates device screen
Actors	Player, Developer
Precondition	Device screen rotation is activated
Main flow	1. Screen rotates
Alternative flow	
Postcondition	Screen is rotated
Notes	

Use case sheet	Interact via keyboard
Description	Keyboard key is pressed or released
Actors	Developer
Precondition	
Main flow	1. Application detects the input
Alternative flow	
Postcondition	The input is detected
Notes	For debugging purposes only

Use case sheet	Interact via touch
Description	Player touches the screen
Actors	Player, Developer
Precondition	
Main flow	1. Application detects the touch
Alternative flow	
Postcondition	The touch is detected
Notes	

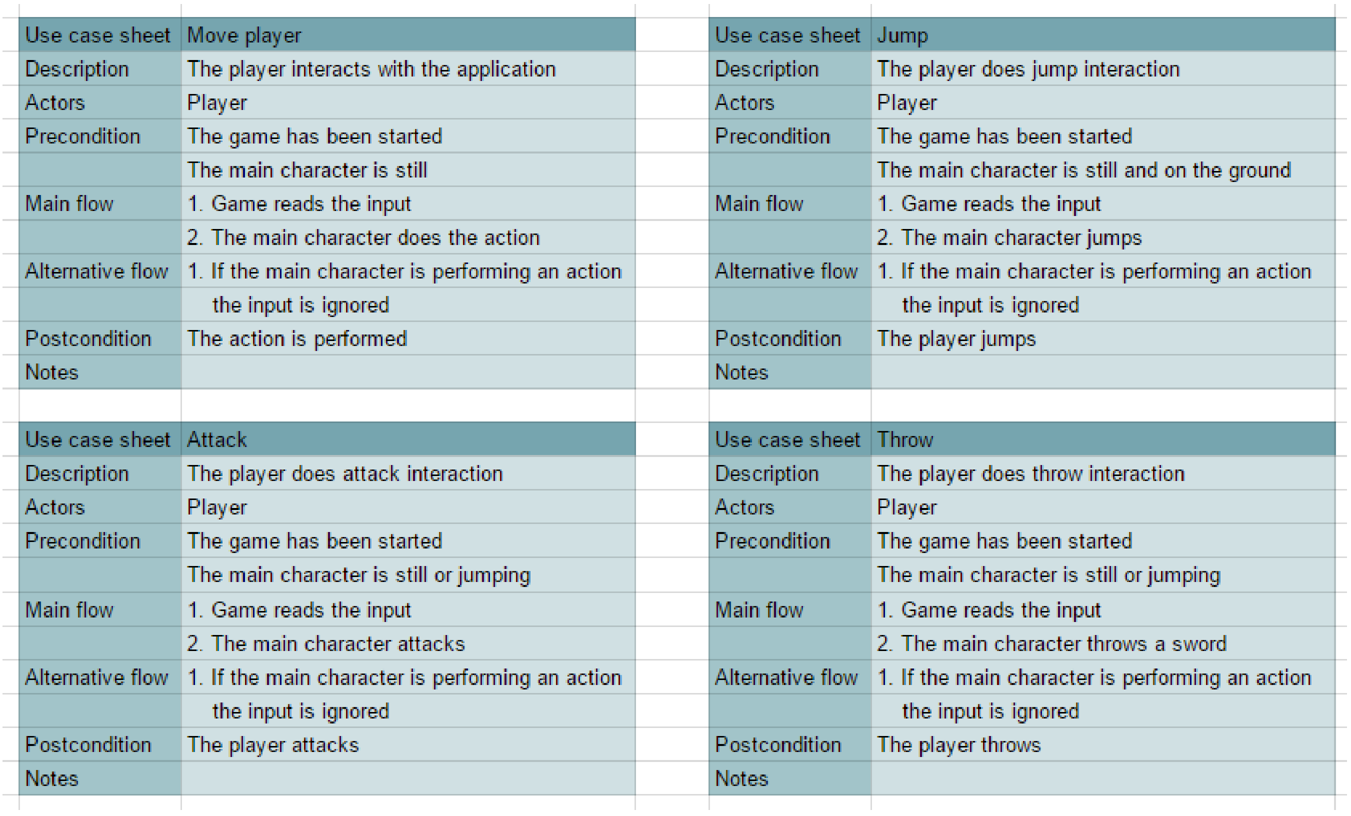
Use case sheet	Exit application
Description	The player presses the 'back' button
Actors	Player
Precondition	
Main flow	1. Application finishes
Alternative flow	
Postcondition	Application is finished
Notes	

Use case sheet	See last game stats
Description	The player visualizes the stats from last game
Actors	Player
Precondition	A last game has been played
Main flow	1. Interface displays stats
Alternative flow	1. If no last game, no stats are displayed
Postcondition	Stats are displayed
Notes	

Use case sheet	See enemies killed spent
Description	The player visualizes the enemies killed
Actors	Player
Precondition	A last game has been played
Main flow	1. Interface displays stats
Alternative flow	1. If no last game, no stats are displayed
Postcondition	Stats are displayed
Notes	

Use case sheet	See time spent
Description	The player visualizes the time from last game
Actors	Player
Precondition	A last game has been played
Main flow	1. Interface displays stats
Alternative flow	1. If no last game, no stats are displayed
Postcondition	Stats are displayed
Notes	

Use case sheet	Start new game
Description	The player presses Start button
Actors	Player
Precondition	
Main flow	1. Game starts
Alternative flow	
Postcondition	The game is started
Notes	



Use case sheet	Move player
Description	The player interacts with the application
Actors	Player
Precondition	The game has been started
	The main character is still
Main flow	1. Game reads the input
	2. The main character does the action
Alternative flow	1. If the main character is performing an action
	the input is ignored
Postcondition	The action is performed
Notes	

Use case sheet	Jump
Description	The player does jump interaction
Actors	Player
Precondition	The game has been started
	The main character is still and on the ground
Main flow	1. Game reads the input
	2. The main character jumps
Alternative flow	1. If the main character is performing an action
	the input is ignored
Postcondition	The player jumps
Notes	

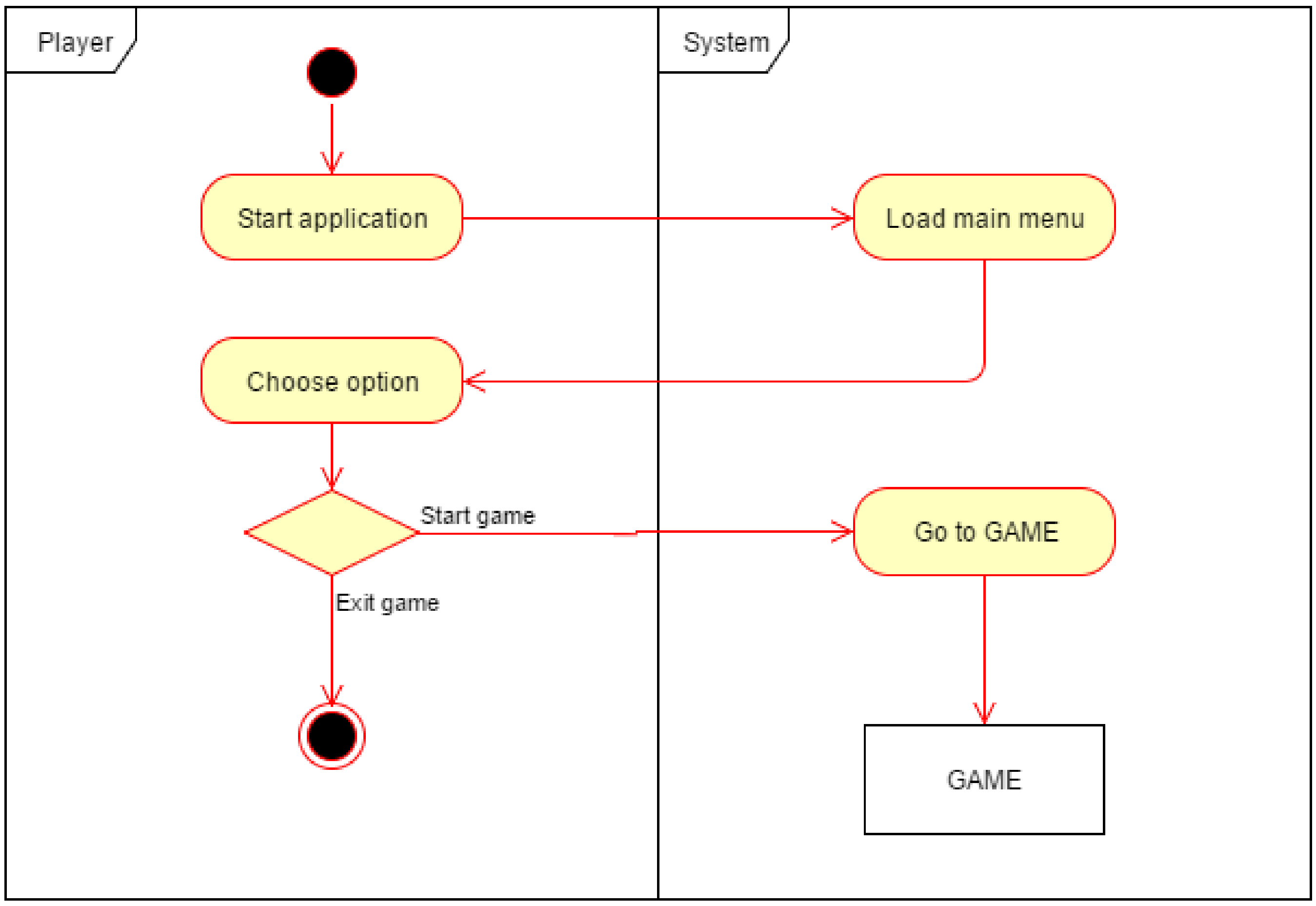
Use case sheet	Attack
Description	The player does attack interaction
Actors	Player
Precondition	The game has been started
	The main character is still or jumping
Main flow	1. Game reads the input
	2. The main character attacks
Alternative flow	1. If the main character is performing an action
	the input is ignored
Postcondition	The player attacks
Notes	

Use case sheet	Throw
Description	The player does throw interaction
Actors	Player
Precondition	The game has been started
	The main character is still or jumping
Main flow	1. Game reads the input
	2. The main character throws a sword
Alternative flow	1. If the main character is performing an action
	the input is ignored
Postcondition	The player throws
Notes	

8.2.4 Activity diagrams

The activity diagrams are divided in 2 diagrams: main menu and game.

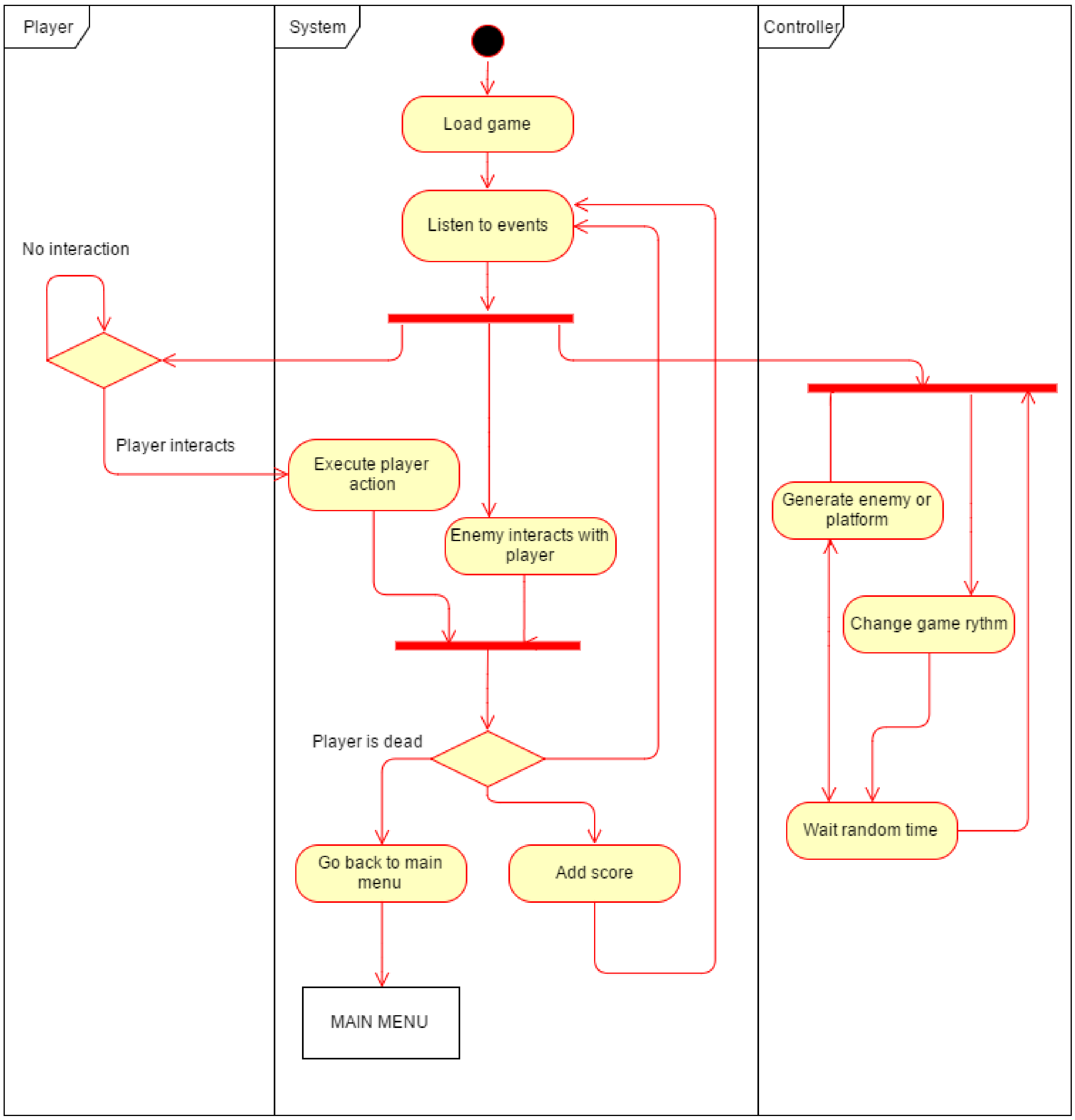
8.2.4.1 Main menu



As we can see, the player starts the application. Then the system loads the first scene: the main menu. It allows the player to start the game.

If the player starts the game, go to the next diagram: Game. If the player chooses to exit, the application finishes.

8.2.4.2 Game



The game starts loading the main game scene. It begins listening to events in parallel, and starts the game controller. These events can come from within the system or from the player interaction.

If the player interacts with the game (touching the screen), the game executes the player action. If an enemy interacts with the player (because of reaching him, or because a random event is that triggered), the game executes the enemy action.

The actions described above can result in the death of either the player or an enemy. If the player is dead, the game returns to main menu (see Main Menu diagram). If an enemy is dead, the score is increased and the enemy disappears.

The controller has two functions: object generation and game rhythm management. Both are triggered by a random time. The object generation is the creation of both enemies and platforms. They are not interdependent, and the platforms appear more often than enemies. The game rhythm changes will make the global game speed faster or slower in a progressive way. It also implies a change in the scene lighting.

8.3 Design

8.3.1 User interface

The interface is not meant to be fancy, but purely functional. Likewise, artwork will be left for future work.

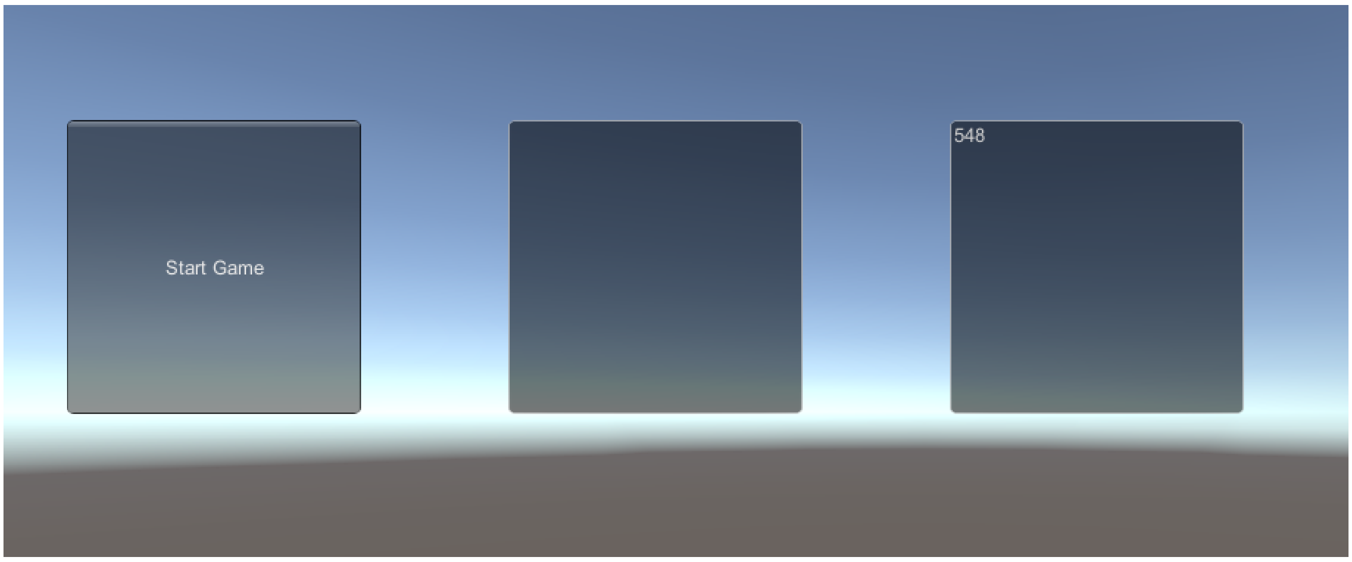


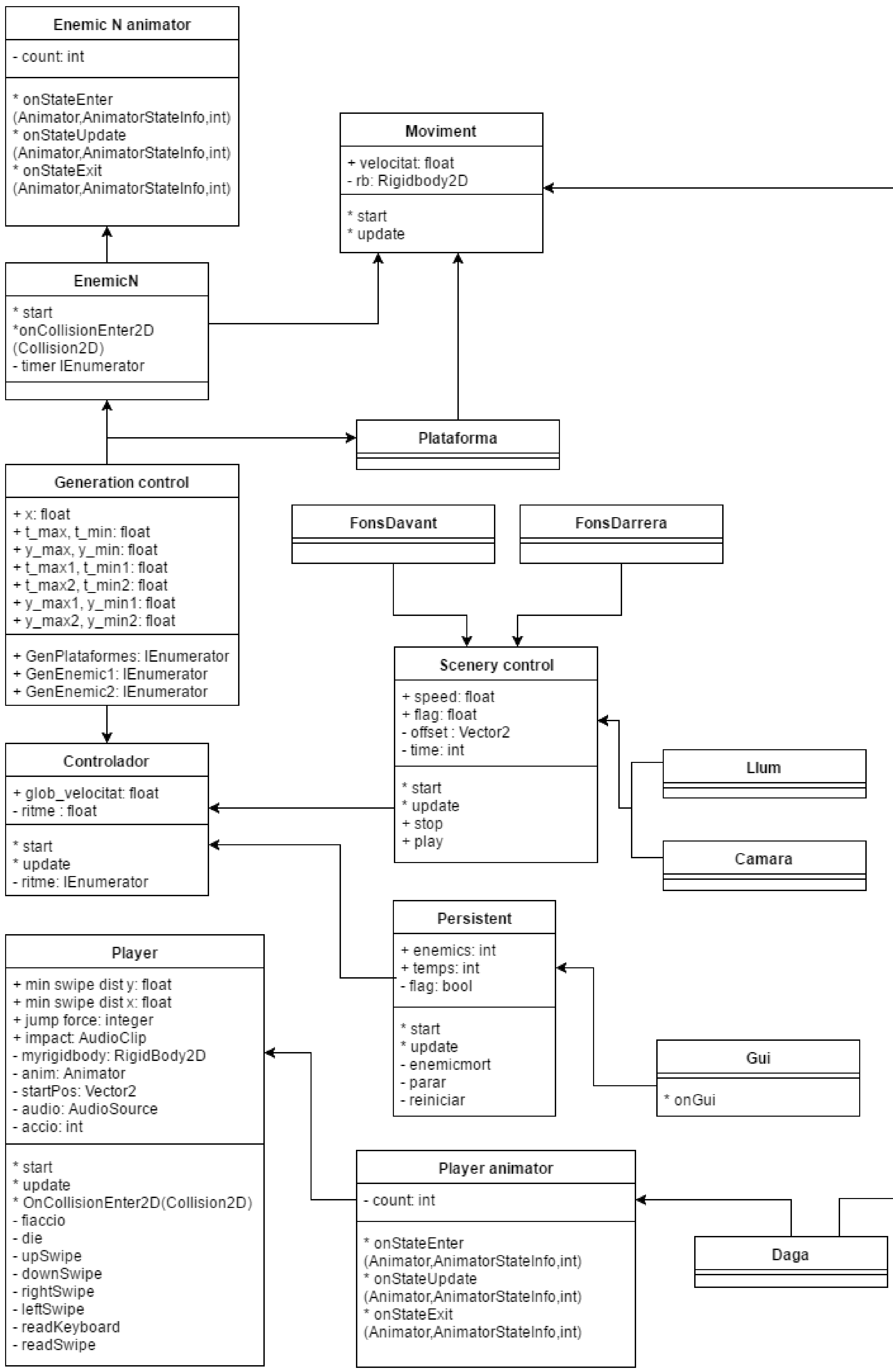
Figure 8.2: Basic interface menu

8.3.2 Artwork

In the annex are included all the spritesheets used (even when not used, expected for future work) in the game.

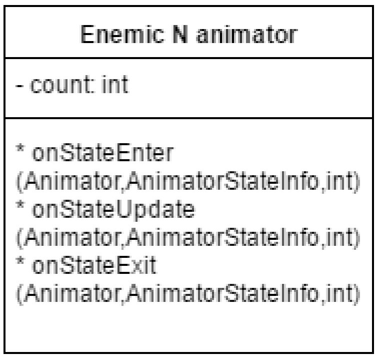
8.4 Class diagram

In the next page we can see the game class diagram, with relations, methods and members.


Enemic N animator
- count: int
* onStateEnter (Animator,AnimatorStateInfo,int)
* onStateUpdate (Animator,AnimatorStateInfo,int)
* onStateExit (Animator,AnimatorStateInfo,int)
Moviment
+ velocitat: float
- rb: Rigidbody2D
* start
* update
EnemicN
* start
*onCollisionEnter2D (Collision2D)
- timer IEnumerator
Plataforma
Generation control
+ x: float
+ t_max, t_min: float
+ y_max, y_min: float
+ t_max1, t_min1: float
+ t_max2, t_min2: float
+ y_max1, y_min1: float
+ y_max2, y_min2: float
+ GenPlataformes: IEnumerator
+ GenEnemic1: IEnumerator
+ GenEnemic2: IEnumerator
FonsDavant
FonsDarrera
Scenery control
+ speed: float
+ flag: float
- offset : Vector2
- time: int
* start
* update
+ stop
+ play
Llum
Camara
Controlador
+ glob_velocitat: float
- ritme : float
* start
* update
- ritme: IEnumerator
Persistent
+ enemics: int
+ temps: int
- flag: bool
* start
* update
- enemicmort
- parar
- reiniciar
Player
+ min swipe dist y: float
+ min swipe dist x: float
+ jump force: integer
+ impact: AudioClip
- myrigidbody: RigidBody2D
- anim: Animator
- startPos: Vector2
- audio: AudioSource
- accio: int
* start
* update
* OnCollisionEnter2D(Collision2D)
- fiaccio
- die
- upSwipe
- downSwipe
- rightSwipe
- leftSwipe
- readKeyboard
- readSwipe
Gui
* onGui
Player animator
- count: int
* onStateEnter (Animator,AnimatorStateInfo,int)
* onStateUpdate (Animator,AnimatorStateInfo,int)
* onStateExit (Animator,AnimatorStateInfo,int)
Daga

8.5 Classes

8.5.1 Enemic N animator


Enemic N animator
- count: int
* onStateEnter (Animator,AnimatorStateInfo,int)
* onStateUpdate (Animator,AnimatorStateInfo,int)
* onStateExit (Animator,AnimatorStateInfo,int)

8.5.1.1 Description

This class controls the animations and transitions of an enemy N. The transition process will be further explained in Chapter 9.

8.5.1.2 Unity components

This is an Animation Controller. Its role is to manage the state transitions between animations.

8.5.1.3 Members

- **count**: This integer holds the value of the time that has passed since the animation has entered the current state.

8.5.1.4 Methods

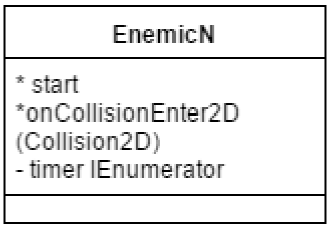
- **onStateEnter**: This is an override of the Unity method with the same name. It is triggered when an animation state is entered. Here the time counter is initialized.

- **onStateUpdate**: This is an override of the Unity method with the same name. It is triggered in every step during an animation state. Here the counter is incremented, and when it reaches a given prefixed number, an action of the enemy is performed.

```
onStateUpdate{
        counter = counter + 1
        if(counter > NUMBER){
                performAction()
        }
}
```

- **onStateExit**: This is an override of the Unity method with the same name. It is triggered when an animation state is exited. Sometimes the action described in the above method is performed here.

8.5.2 Enemic N



EnemicN

* start
*onCollisionEnter2D
(Collision2D)
- timer IEnumerator

8.5.2.1 Description

This class represents an individual enemy. The function of the enemies is to be an obstacle/menace for the player.

Figure 8.3: Example of enemy

8.5.2.2 Unity components

It is a game object (prefab) that has multiple instances. It has scripts and 2D physics elements: a 2DCollider and a 2DRigidBody, and it has an animation controller.

8.5.2.3 Methods

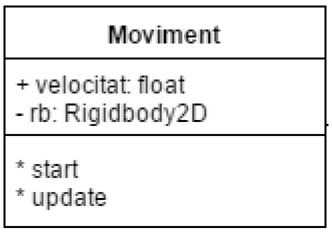
- **start**: This is an override of the Unity method with the same name. It is triggered when the object is created. It initializes the enemy, giving it its motion and starting the enumerator.

- **onCollisionEnter2D**: This is an override of the Unity method with the same name. It is triggered whenever the collider throws a collision event. It handles the collisions with the player and with the object Daga. The first one results in the enemy's death when the player is attacking, or in performing an action when it is not. The second also results in death for the player.

```
onCollisionEnter2D(object){
        //object is the object who collides
        if(object.type = daga){
                die
        }
        else if (object.type = player){
                if(player.action == ATTACK){
                        die
                }
                else{
                        performAction()
                }
        }
}
```

- **timer**: This is the method of the enumerator. It performs an action (attack the player) and reinitializes the enumerator.

8.5.3 Moviment



8.5.3.1 Description

This is an interface which allows objects to move horizontally.

8.5.3.2 Unity components

This is a script which will be used by objects who have a simple horizontal movement.

8.5.3.3 Members

- **velocitat**: This variable holds the speed of the object. Positive values mean left and negative mean right.

- **rb**: This is the RigidBody2D of the object that will be moved.

8.5.3.4 Methods

- **start**: This is an override of the Unity method with the same name. It is triggered when the object is created. It initializes the object giving it its initial motion.

- **update**: This is an override of the Unity method with the same name. It is triggered on every step. It keeps the object moving on a constant speed.

8.5.4 Plataforma

8.5.4.1 Description

This class represents a platform, on the top of which the player and the enemies move.

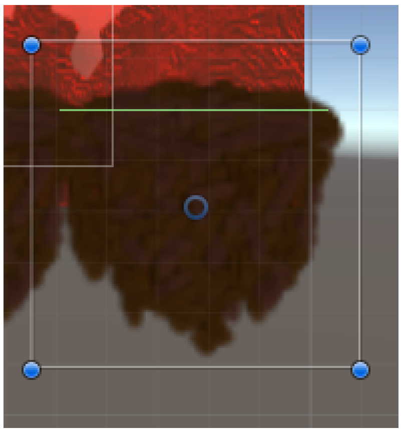
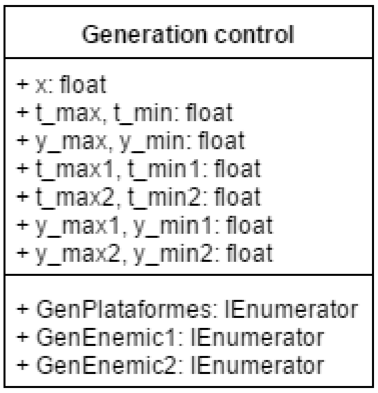


Figure 8.4: Example of platform with its collider in green

8.5.4.2 Unity components

It is a game object (prefab) that has multiple instances. It has 2D physics elements: a 2DCollider and a 2DRigidBody, and it has a fixed sprite.

8.5.5 Generation control



Generation control

+ x: float
+ t_max, t_min: float
+ y_max, y_min: float
+ t_max1, t_min1: float
+ t_max2, t_min2: float
+ y_max1, y_min1: float
+ y_max2, y_min2: float

+ GenPlataformes: IEnumerator
+ GenEnemic1: IEnumerator
+ GenEnemic2: IEnumerator

8.5.5.1 Description

This class manages the generation of enemies and platform instances in the game.

8.5.5.2 Unity components

This is an abstract Game Object with scripts.

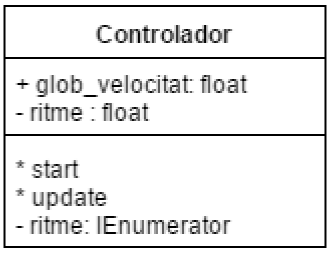
8.5.5.3 Members

- **x**: This holds the horizontal generation position value.
- **t_max**: This is the upper value for the time interval generation for platforms.
- **t_min**: This is the lower value for the time interval generation for platforms.
- **y_max**: This is the upper value for the vertical position interval generation for platforms.
- **y_min**: This is the lower value for the vertical position interval generation for platforms.
- **t_maxN**: This is the top value for the time interval generation for enemy N.
- **t_minN**: This is the low value for the time interval generation for enemy N.
- **y_maxN**: This is the top value for the vertical position interval generation for enemy N.
- **y_minN**: This is the low value for the vertical position interval generation for enemy N.

8.5.5.4 Methods

- **GenPlataformes**: This enumerator generates an instance of the platform.
- **GenEnemicN**: This enumerator generates an instance of enemy N.

8.5.6 Controlador



8.5.6.1 Description

This is the main control object. It holds the generation controller, the persistence controller and the scenery controller, and also manages rhythm variation.

8.5.6.2 Unity components

This is an abstract Game Object with scripts.

8.5.6.3 Members

- **glob_velocitat**: This variable holds the global game speed.

- **ritme**: This variable holds the rhythm speed.

8.5.6.4 Methods

- **start**: This is an override of the Unity method with the same name. It is triggered when the object is created. It initializes the rhythm enumerator and its subclasses: Scenery contol, Generation control and Persistent.

- **update**: This is an override of the Unity method with the same name. It is triggered at every step. It changes the speed according to the current rhythm.

- **ritme**: This enumerator changes randomly the rhythm, and is triggered again in a random time between a given interval (1.5-2.5 minutes).

```
Enumerator ritme{
        ritme = random
        temps = random(1.5,2.5)
        retorna esperar(temps)
}
```

8.5.7 Fons davant

FonsDavant

8.5.7.1 Description

This is the foreground of the scenery landscape.

8.5.7.2 Unity components

It is a 3D Game Object (a Quad) with a texture.

8.5.8 Fons darrera

FonsDarrera

8.5.8.1 Description

This is the background of the scenery landscape.

8.5.8.2 Unity components

It is a 3D Game Object (a Quad) with a texture.

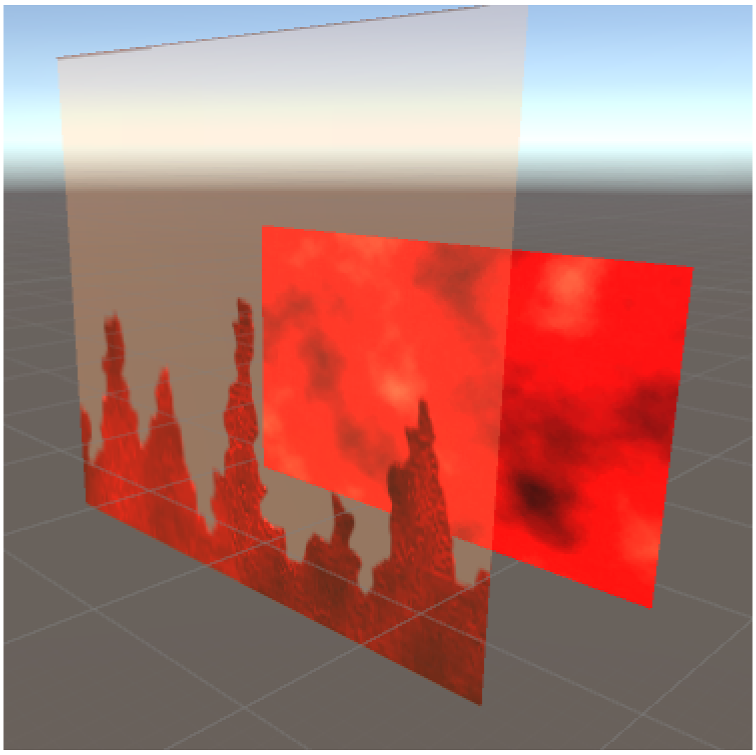
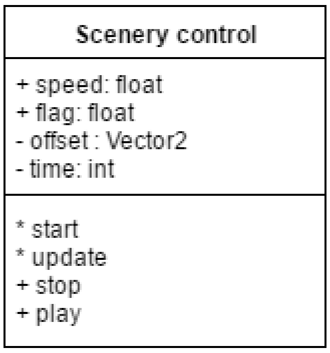


Figure 8.5: Both background objects seen in 3D view

8.5.9 Scenery control



8.5.9.1 Description

This class controls the parallax effect of the background and the lighting changes when the rhythm changes.

8.5.9.2 Unity components

This is an abstract Game Object with scripts.

8.5.9.3 Members

- **speed**: This variable holds the value of the speed of the background. It is relative to the game speed.

- **flag**: This flag is used to activate/deactivate the background movement.

8.5.9.4 Methods

- **start**: This is an override of the Unity method with the same name. It is triggered when the object is created. It initializes the movement of the background.

- **update**: This is an override of the Unity method with the same name. It is triggered at every step. It moves the background and changes the lighting according to the current rhythm.

```
update{
        GLOB_velocitat = ritme * FACTOR1
        llum.intensity = ritme * FACTOR2
}
```

- **start**: This starts the background movement.

- **stop**: This starts the background movement.

8.5.10 Llum

8.5.10.1 Description

This is the light that illuminates the scene.

Figure 8.6: Light

8.5.10.2 Unity components

This is a Directional Light, which illuminates the 2D scene from above. It has an orange color.

8.5.11 Camara

8.5.11.1 Description

This is the 2D camera that allows the player to see the scene.

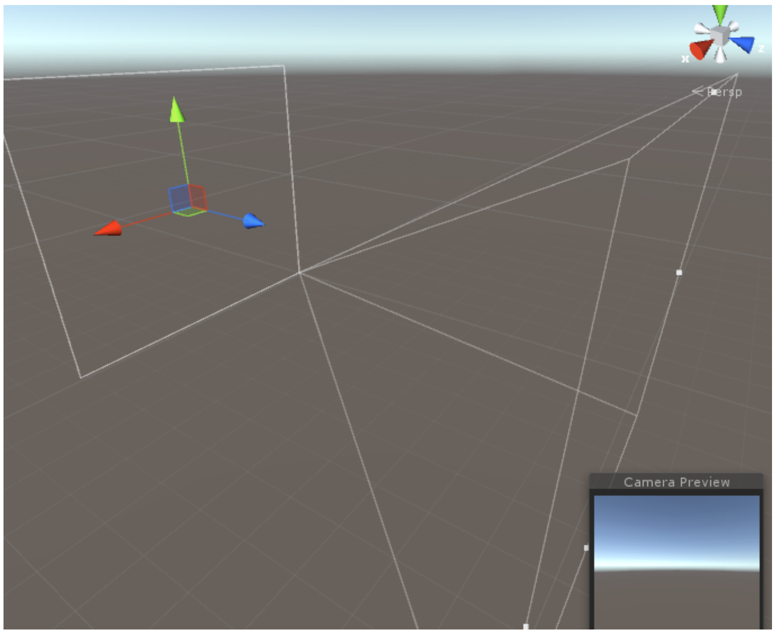
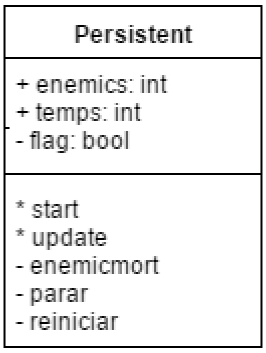


Figure 8.7: Camera

8.5.11.2 Unity components

This is a Camera, which uses a 2D projection.

8.5.12 Persistent



8.5.12.1 Description

This is a special object that survives scene changes in order to store values between scenes.

8.5.12.2 Unity components

This is a persistent Game Object, meaning it does not disappear when changin between scenes.

8.5.12.3 Members

- **enemics**: This variable holds the value of the amount of enemies killed.
- **temps**: This variable holds the value of the amount of time spent.
- **flag**: This flag blocks the time counter when in the menu.

8.5.12.4 Methods

- **start**: This is an override of the Unity method with the same name. It is triggered when the object is created. It initializes the variables to 0.
- **update**: This is an override of the Unity method with the same name. It is triggered at every step. It updates the time variable.
- **enemicmort**: This is triggered when an enemy is killed, and increments the variable enemics.
- **parar**: This stops the time.
- **reiniciar**: This restarts the variables.

8.5.13 Gui

8.5.13.1 Description

This class manages the GUI displaying.

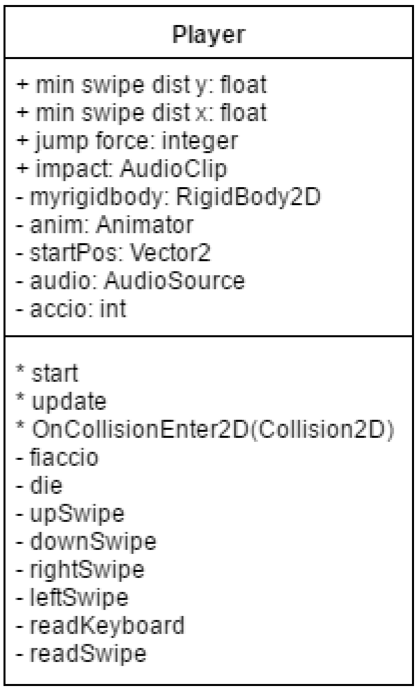
8.5.13.2 Unity components

This is a Canvas, where the menu Gui will be drawed.

8.5.13.3 Methods

- **onGui**: This is an override of the Unity method with the same name. It is triggered when the canvas is able to draw. It draws the interface.

8.5.14 Player



8.5.14.1 Description

This is the main character controller, which also reads the user input.

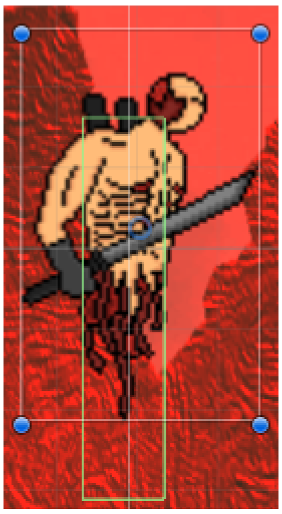


Figure 8.8: Player in the scene

8.5.14.2 Unity components

It is a game object (prefab) that has multiple instances. It has associated scripts and 2D physics elements: a 2DCollider and a 2DRigidBody, and it has an animation controller. It also has an AudioSource and a Scene Manager.

8.5.14.3 Members

- **min_swipe_dist_y**: This variable holds the minimal vertical length for a swipe to be detected.

- **min__swipe__dist__x**: This variable holds the minimal horizontal length for a swipe to be detected.
- **jump__force**: This variable holds the value of the force applied upwards when jumping.
- **impact**: This holds the sound performed when attacking.
- **startPos**: This is the initial position of the player.
- **audio**: This is the Audiosource that plays sounds.
- **accio**: This holds the value of the current action. 0 means no action, 1 means attacking and 2 means throwing.

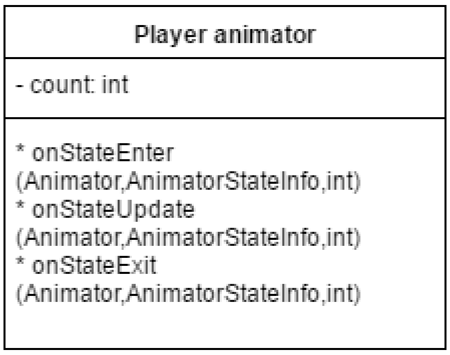
8.5.14.4 Methods

- **start**: This is an override of the Unity method with the same name. It is triggered when the object is created. It initializes the enemy giving it its motion and starting the enumerator.
- **update**: This is an override of the Unity method with the same name. It is triggered at every step. It calls readKeyboard (developer only) and readSwipe.
- **onCollisionEnter2D**: This is an override of the Unity method with the same name. It is triggered when the collider throws a collision event. It handles the collisions with the enemies. If the player is not attacking, it triggers the method the method die.

```
onCollisionEnter2D(object){
        //object is the object who collides
        if (object.type = enemy){
                if(action != ATTACK){
                        die
                }
        }
}
```

- **fiaccio**: It allows the player to perform a new action. It is called when an action is finished.
- **die**: It destroys the player and changes the Scene to the main menu.
- **upSwipe**: It performs the actions of an "up" swipe (activating the corresponding animation)
- **downSwipe**: It performs the actions of a "down" swipe (activating the corresponding animation)
- **rightSwipe**: It performs the actions of a "right" swipe (activating the corresponding animation)
- **leftSwipe**: It performs the actions of a "left" swipe (activating the corresponding animation)
- **readKeyboard**: It reads the user input via keyboard and launches one of the swipe methods.
- **readSwipe**: It reads the user input via touch and launches one of the swipe methods.

8.5.15 Player animator


Player animator
- count: int
* onStateEnter
(Animator,AnimatorStateInfo,int)
* onStateUpdate
(Animator,AnimatorStateInfo,int)
* onStateExit
(Animator,AnimatorStateInfo,int)

8.5.15.1 Description

This class controls the animations and transitions of the player.

8.5.15.2 Unity components

This is an Animation Controller, like Enemic N animator.

8.5.15.3 Members

- **count**: This integer holds the value of the time that has passed since the animation has entered the current state.

8.5.15.4 Methods

- **onStateEnter**: This is an override of the Unity method with the same name. It is triggered when an animation state is entered. Here the time counter is initialized.

- **onStateUpdate**: This is an override of the Unity method with the same name. It is triggered at every step during an animation state. Here the counter is incremented, and when it reaches a given prefixed number, if the action is throw, the player creates a Daga object.

```
onStateUpdate{
        counter = counter + 1
        if(player.action == throw && counter > NUMBER){
                createDaga
        }
}
```

- **onStateExit**: This is an override of the Unity method with the same name. It is triggered when an animation state is exited. Sometimes the action described in the above method is performed here, in the same way of onStateUpdate.

8.5.16 Daga

8.5.16.1 Description

This is the sword thrown when the player performs the action throw.

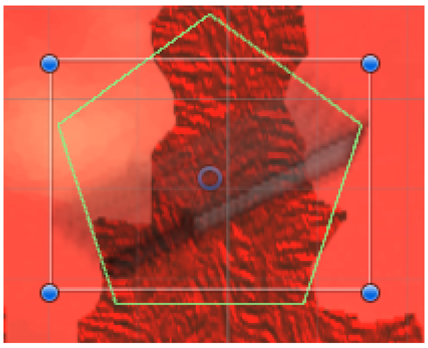


Figure 8.9: Daga flying in the scene

8.5.16.2 Unity components

It is a game object (prefab) that has multiple instances. It has 2D physics elements: a 2DCollider and a 2DRigidBody.

Chapter 9

Deploying and testing

In this chapter we will detail how the specific elements of the videogame have been programmed.

9.1 Touch/swipe recognition

The game input uses swipes as actions. In order to detect the swipe directions we use the following algorithm:

```
void readSwipe(){
if (Input.touchCount > 0){
        Touch touch = Input.touches[0];
switch (touch.phase) {
        case TouchPhase.Began:
                startPos = touch.position;
                break;
        case TouchPhase.Ended:
        float swipeDistVertical = (new Vector3
                (0, touch.position.y, 0) - new Vector3 (0,
                startPos.y, 0)).magnitude;
        if (swipeDistVertical < minSwipeDistY) {
                float swipeDistHorizontal = (new Vector3
                        (touch.position.x, 0, 0) - new Vector3
                        (startPos.x, 0, 0)).magnitude;
                if (swipeDistHorizontal > minSwipeDistX) {
                        float swipeValue1 = Mathf.Sign
                                (touch.position.x - startPos.x);
                        if (swipeValue1 > 0) {
                                rightSwipe ();
                                break;
                        } else if (swipeValue1 < 0) {
                                leftSwipe ();
                                break;
                        }
                }
        }
        swipeDistVertical = (new Vector3(0, touch.position.y, 0) -
                new Vector3(0, startPos.y, 0)).magnitude;
        if (swipeDistVertical > minSwipeDistY){
```

```
                        float swipeValue2 = Mathf.Sign(touch.position.y -
                                startPos.y);
                        if (swipeValue2 > 0) {
                                upSwipe();
                                break;
                        } else if (swipeValue2 < 0) {
                                downSwipe();
                                break;
                        }
                break;
}
                }
}
}
```

This algorithm captures the initial offset and detects the movement. If the new offset is at enough distance from the initial one, it triggers a swipe.

It uses Touchphase in order to detect whether the swipe is beginning (to capture the initial position) or ending (to capture the final position). When it has the final position, the vertical and horizontal distances are calculated (using Vector3).

9.2 Animation

For the player and enemy animation we use sprites, which we cut in Unity to create animations. In order to change from one animation to another, we use the Unity's Animation Controller, which is a deterministic state machine. The conditions of the state change are both state ending and the reading of a variable "action", which is manipulated in the code of the states or by the player/enemy.

The initial state is main_walk, and is entered on the creation of the animator. When the variable "action" has the value 1 and the state is main_walk, it triggers the transition main_walk - main_hit. When the value is 2 and state is main_walk, it triggers main_walk - main_throw. On any state, when its animation is ended, the transition will be going back to main_walk.

At figure 9.1. is the state machine for the player character

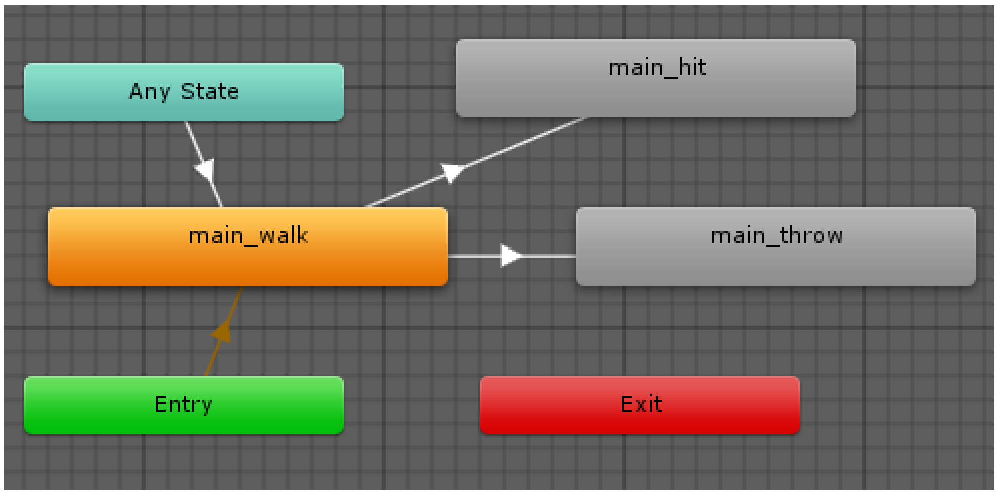


Figure 9.1: Player Animation Controller

9.3 Sound playing

The sound playing uses the Unity embedded Audiosource functionalities. These functionalities are basically the usage of AudioSource objects (which are a source of sound) and Sound objects (which are sound instances). The Audiosource is a Component, and we call its sound playing method with the sound we want to play.

```
audio = GetComponent<AudioSource>();
audio.PlayOneShot(impact, 0.7F);
```

9.4 Rhythm variation

The rhythm variation is based in an enumerator that loops infinitely, changing the rhythm when triggered (with a random time). It also alters the global speed constantly according to the current rhythm, adding to it the value of "ritme" multiplied by a scale factor, thus creating an acceleration.

```
void Update () {
        GLOB_velocitat += ritme*0.01f;
}

IEnumerator Ritme(){
        while( true ){
                ritme = Random.value * (0.01f - -0.01f) +
                        -0.01f;
                GameObject.Find ("llum").
                GetComponent<Light> ().intensity = 1.45f - ritme*200;
                yield return new WaitForSeconds(
                        Random.value * (10 - 1) + 1);
        }
}
```

9.5 Parallax background

The parallax background is the visual effect produced when 2 layers of background move in parallel planes. This produces an aesthetically pleasing effect of movement and simulates depth. In order to achieve this, we use Unity's embedded 3D functionalities, thus creating 2 different 3D objects with the textures, that will be seen in 2D.

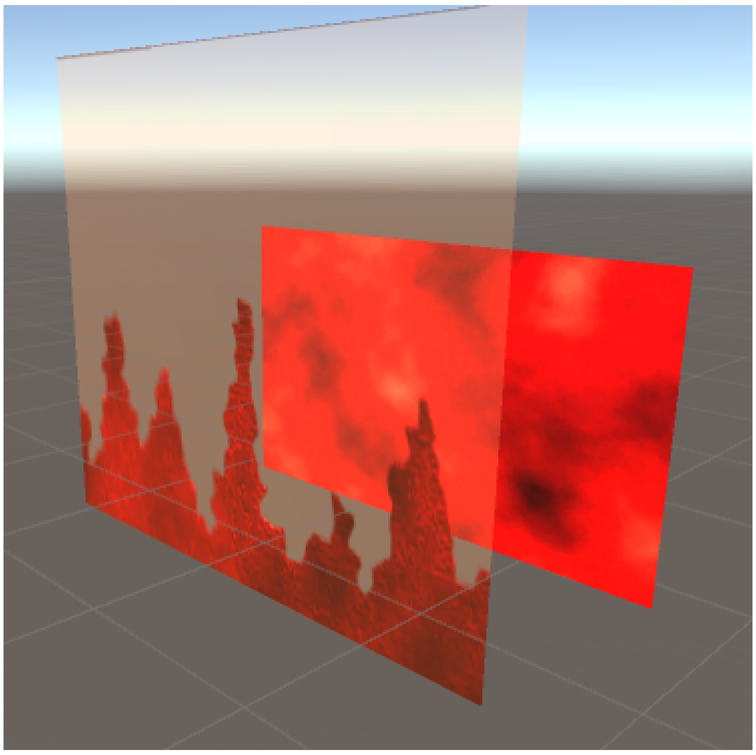


Figure 9.2: 3D Objects forming the Parallax background

Then we have to make them move according to the speed of the game, so we multiply the background speed and the global speed, finding a speed v. This will be used to calculate the new value of the texture offset in every step, using the variable time (a counter that will be incremented at every step). We multiply the time for the speed and a scale factor, and this will give us the horizontal component of the vector offset. Finally we apply this offset to the background texture.

```
void Start(){
                time = 0;
        }
        void Update () {
                if (flag == 1) {
                        time += 1;
                        float v = speed * GameObject.Find
                                ("controlador").GetComponent
                                <controlador> ().GLOB_velocitat;
                        offset = new Vector2 (time * 0.01f * v, 0);
                }
                GetComponent<Renderer>().material.mainTextureOffset
                        = offset;
        }
        public void stop(){
                flag = 0;
        }
        public void play(){
                flag = 1;
        }
```

9.6 Interface

The interface is created totally via scripting. It is purely functional and has no artistic elements.

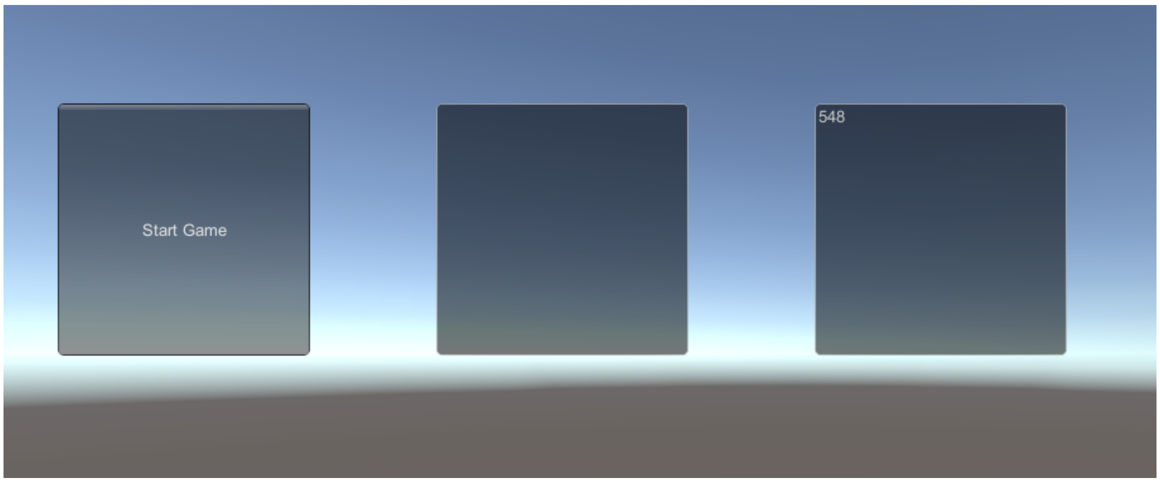


Figure 9.3: Basic menu interface

It displays the enemies killed and time spent when possible (when there is a last game), otherwise leaves the fields empty. So the first step is to try to get the value of "enemics" and "temps" which are variables from the object persistent, representing the enemies killed and the time elapsed respectively. If we can't get them, we leave the values in 0, so that they won't be shown.

The rest of the code is about drawing TextFields and giving them the variable values and configurating a button that will call the game scene when pressed.

```
void OnGUI () {
int enem = 0;
int time = 0;
try{
enem= GameObject.Find ("persistent").GetComponent<p> ().enemics;
time= GameObject.Find ("persistent").GetComponent<p> ().temps;
}
catch (Exception){}

if(enem==0)
        GUI.TextField(new Rect(400, 100, 200, 200), "",50);
else
        GUI.TextField(new Rect(400, 100, 200, 200), ""+enem,50);

if(time==0)
        GUI.TextField(new Rect(700, 100, 200, 200), "",50);
else
        GUI.TextField(new Rect(700, 100, 200, 200), ""+time,50);
if(GUI.Button(new Rect(100,100,200,200), "Start␣Game")) {
  try{ GameObject.Find ("persistent").SendMessage ("reiniciar"); }
  catch (Exception){}
  SceneManager.LoadScene ("main");
}
}
```

Chapter 10

Implementation and results

10.1 Development process

First of all, we selected the videogame type, a 2D endless runner. After that, we decided the videogame mechanics and objectives of the project. The following step was to select a videogame development engine. For this project Unity was selected. Then we decided which language of Unity we would use, which ended up being C#. The next step was to begin learning all the funcionalities of Unity, oriented to the development of the specific type of game we wanted. Afterwards we began the development of the project, following the steps described in chapter 3. During the development, testing and beta tester feedback was used.
Once finished the main game mechanism, we dedicated the last days to refining the documentation and the game itself.

10.2 Normative and legislation

This project has no problem with the current legislation and normative.
Refering to the *Ley organica de protección de datos de caracter personal (LOPD)*, this project does not store any personal data.
Refering to the *Ley de servicios de la sociedad de la información y comercio electrónico (LSSICE)*, this project is not an economic activity.

10.3 Resulting application

All the tasks of the objectives have been accomplished.
Below are several snapshots of the videogame, displaying different stages of it. The rhythm variations can also be observed in the different scene lighting of every screen.

On the very beginning of the game, we see the player floating over platforms. The visual effect is that the player is moving forward to the right, but in fact the player is still and the scenario is moving to the left.

The platforms form an irregular terrain due to their random generation in the vertical position, and cannot be individually distinguished. The background is moving at two different speeds forming the parallax effect. See Figure 10.1.

Randomly, we can see a change in the game rhythm. We also notice it due to a change in illumination.

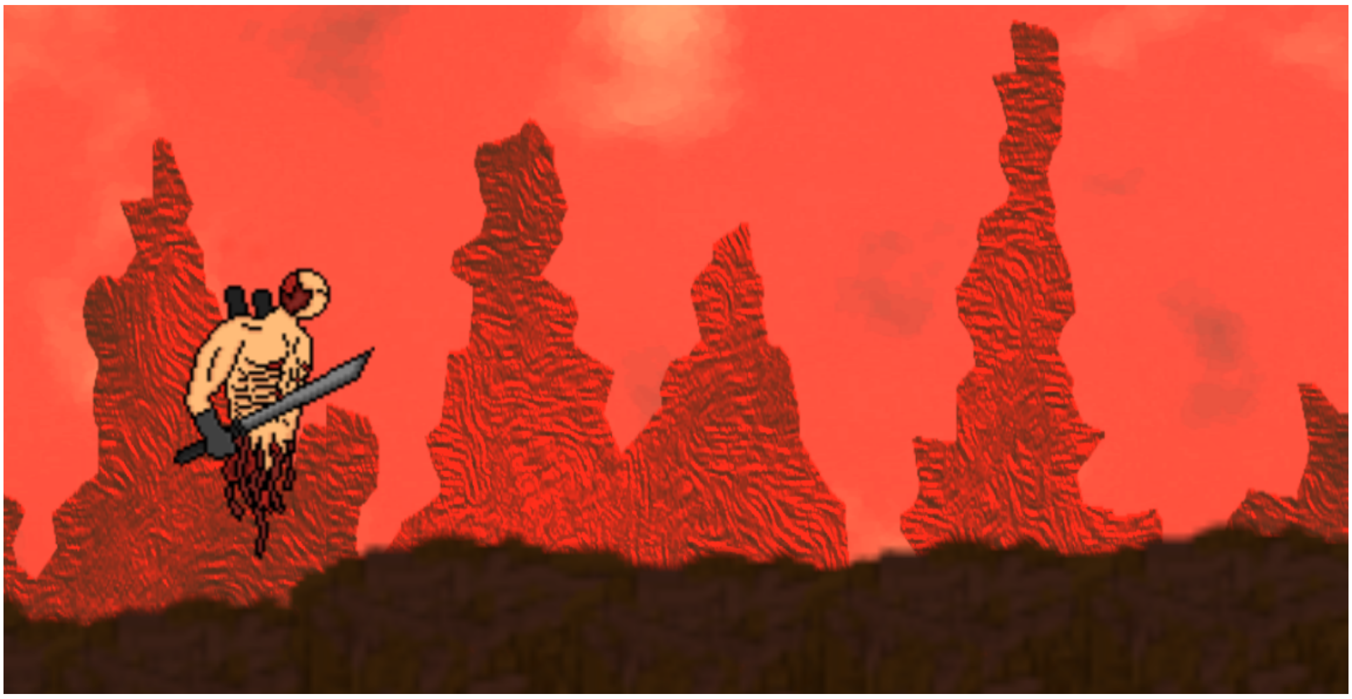


Figure 10.1: Player and scenery

The first enemy appears. Since it is far away from the player, the best idea is to use throw. The throwing animation of the player is played and a flying sword appears, moving towards the enemy. If it hits, the enemy dies.

In this screenshot we can also see a different illumination, meaning this is a faster zone than the previous one. See Figure 10.2.

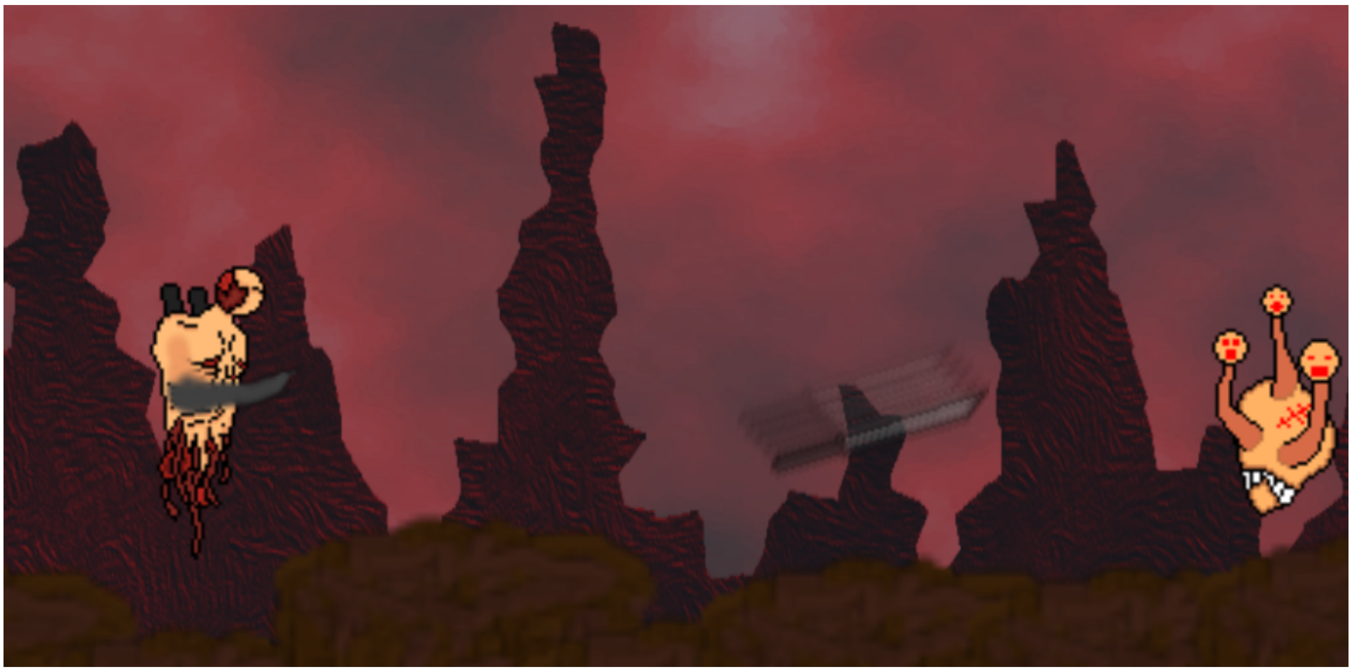


Figure 10.2: Enemies appear, player uses throw

Certain obstacles/enemies cannot be killed. Then, the only option the player has left is to jump. Jump can also be used to avoid regular enemies, or to attack or throw in the air, in order to hit a target that is floating. See Figure 10.3.

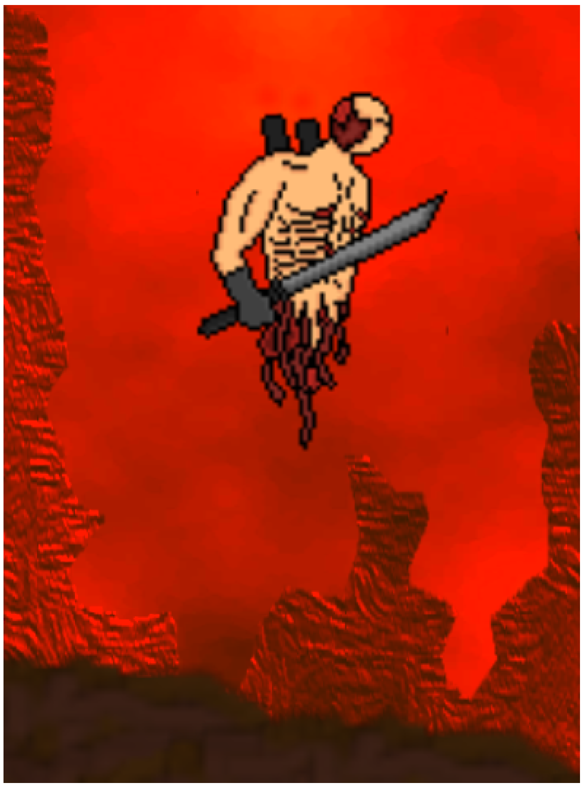


Figure 10.3: Player uses jump to avoid an obstacle

When an enemy is very close to the player, the best option is to attack: jumping would not avoid it and throwing would be too slow. So, if the player uses the attack commando, the enemy will be killed when hit.

At Figure 10.4 we can also see the fastest game pace possible. As we can see, the background has turned almost black, because the light is very weak. It means the game speed will increase strongly in the next seconds, making this area a difficult one in the game.

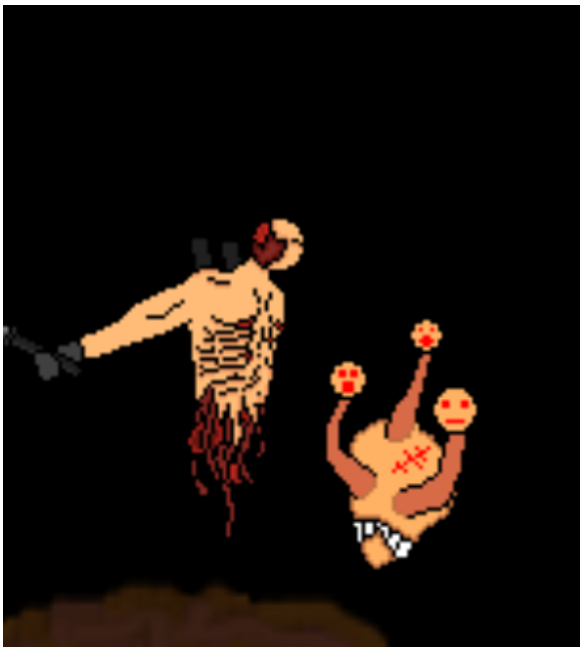


Figure 10.4: Player uses attack to kill an enemy

Chapter 11

Conclusions

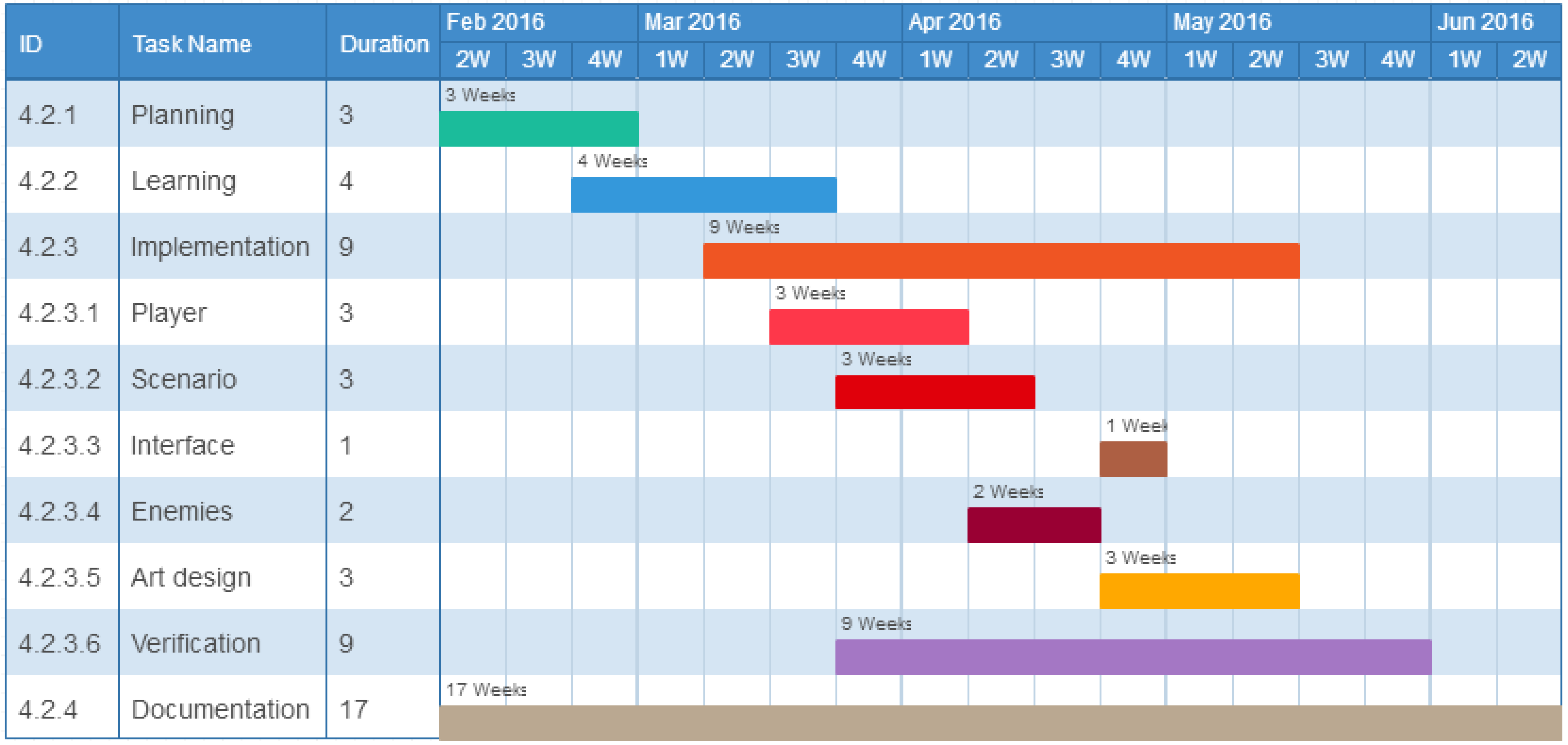
11.1 Scheduling differences

Several differences can be observed between the real and planified timeline.

- In general, everything took more time than expected.
- The learning was very practical-oriented, so the implementation began simultaneously.
- The implementation took more time than expected. We had to restart the Unity project several times due to errors in the configuration.
- The development of the art design also took some time, but in the beginning of the development prototype sprites were used.
- The verification began when the player and scenario were created and lasted longer than expected.

Since a certain margin was left for perfecting everything in the last weeks, the project has been successfully completed, but the resulting game requires a lot more work to be commercially acceptable.

11.2 Real scheduling



ID	Task Name	Duration
4.2.1	Planning	3
4.2.2	Learning	4
4.2.3	Implementation	9
4.2.3.1	Player	3
4.2.3.2	Scenario	3
4.2.3.3	Interface	1
4.2.3.4	Enemies	2
4.2.3.5	Art design	3
4.2.3.6	Verification	9
4.2.4	Documentation	17

11.3 Conclusions

The conclusions of this project are the following:

- The main objectives have been successfully accomplished.
  - Planning of the game. Script and defining the elements of interaction.
  - Implementation of possible actions by the player and interaction with the environment.
  - Development of a prototype of the game
  - Design and implementation of algorithms of artificial intelligence of enemies.
  - Implementation of designs and animations of sprites for Android.
  - Creating the final model of the game
  - Refinement of the documentation.
- I have realized developing a videogame is a very complex task, probably more adequate for a team.
- Writing a memory fully in English and in Latex was an enriching experience in both competences.
- I have learned about Android platform
- I have learned about the Unity platform
- I have learned about programming languages (C#)
- The resulting game requires way more work to be commercialized.

Chapter 12

Future work

12.1 Future work

The possible future improvements of the videogame are the following:

1. Perfect the gameplay so that the player will have a better game experience.
2. Improve the interfaces.
3. Publish the game to Play store.
4. Create an online service (maybe, a REST service) to display best scores.
5. Add more enemies with different behaviors.
6. Improve the artwork and graphics.
7. Improve the music and sound effects.
8. Add more scenarios and decorations.

Bibliography

[1] Android. https://www.android.com/. Accessed: 2016-02/03.

[2] Another open game art website. http://open.commonly.cc/. Accessed: 2016-04/06.

[3] C sharp community examples. http://www.csharp-station.com/, . Accessed: 2016-02/04.

[4] C sharp programming guide. https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx, . Accessed: 2016-02/04.

[5] C sharp tutorial. http://www.tutorialspoint.com/csharp/, . Accessed: 2016-02/04.

[6] Information about game engines. http://www.develop-online.net/. Accessed: 2016-06-01.

[7] List of open game art websites. http://v-play.net/game-resources/16-sites-featuring-free-game-graphics. Accessed: 2016-04/06.

[8] Monodevelop. http://www.monodevelop.com/. Accessed: 2016-02/03.

[9] Open game art. http://opengameart.org/. Accessed: 2016-04/06.

[10] Parallax effect. http://pixelnest.io/tutorials/2d-game-unity/parallax-scrolling/. Accessed: 2016-04-20.

[11] Sprite design. http://design.tutsplus.com/tutorials/how-to-create-an-animated-pixel-art-sprite-in-adobe-photoshop--cms-20428. Accessed: 2016-03/04.

[12] Unity. https://unity3d.com/, . Accessed: 2016-02/06.

[13] Unity+android. http://www.androidauthority.com/an-introduction-to-unity3d-666066/, . Accessed: 2016-02/03.

[14] Unity api. http://docs.unity3d.com/ScriptReference/index.html, . Accessed: 2016-02/06.

[15] Unity community. http://answers.unity3d.com/, . Accessed: 2016-02/06.

[16] Unity examples and information. https://madewith.unity.com/, . Accessed: 2016-03.

[17] Unity manual. http://docs.unity3d.com/Manual/index.html, . Accessed: 2016-02/06.

[18] Unity tutorials. https://unity3d.com/learn/tutorials, . Accessed: 2016-02/03.

Chapter 13

User manual

13.1 Instalation

The game is an android APK. It has no special requirements for its instalation, except the technical requirements stated in chapter 2. The APK just has to be opened on an Android device or emulator.

13.2 Game controls

The game controls are simple swipes:

1. Left to right swipe 13.2: throw
2. Down to up swipe 13.2: jump
3. Up to down swipe 13.2: hit
4. To interact with the interfaces, all the buttons are touchable 13.2.

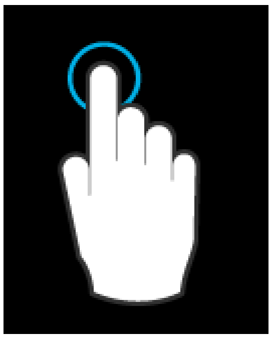


Figure 13.1: Screen touch

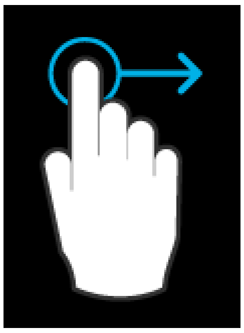


Figure 13.2: Right swipe

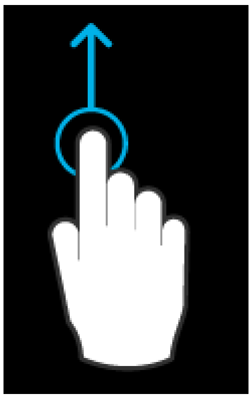


Figure 13.3: Up swipe

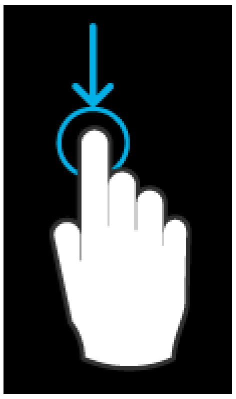


Figure 13.4: Down swipe

Chapter 14

Annex

14.1 Spritesheets and textures

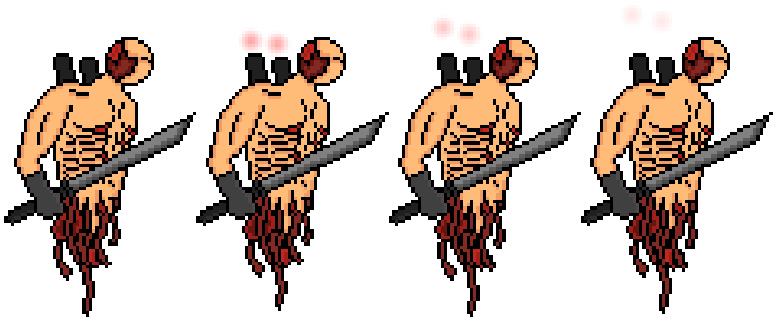


Figure 14.1: Player still

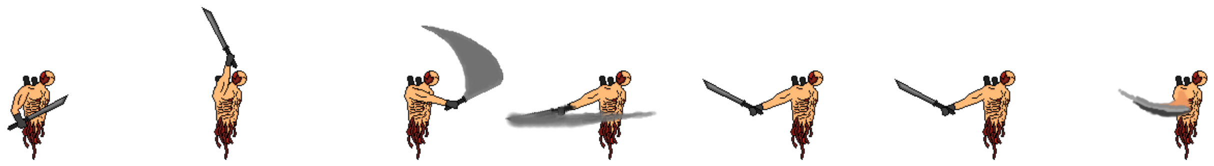


Figure 14.2: Player attack

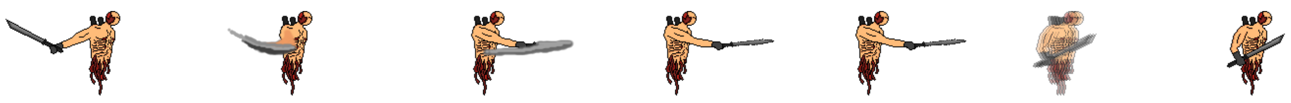


Figure 14.3: Player throw

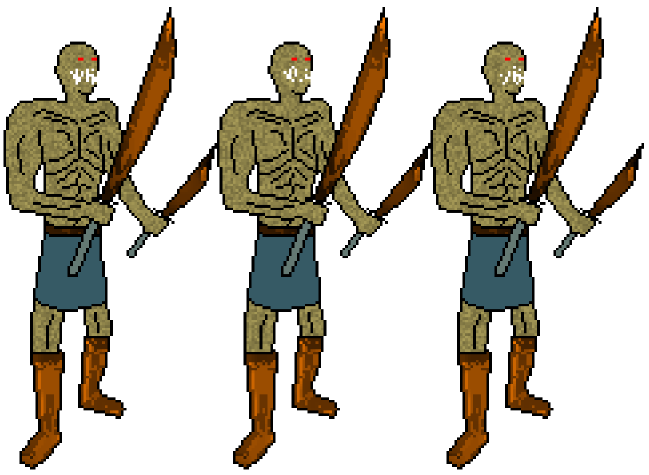


Figure 14.4: Enemy still

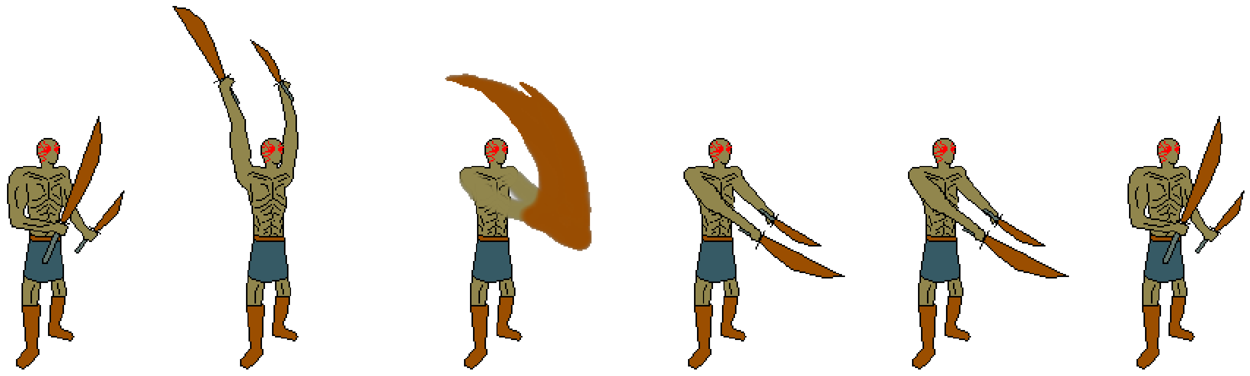


Figure 14.5: Enemy attack1

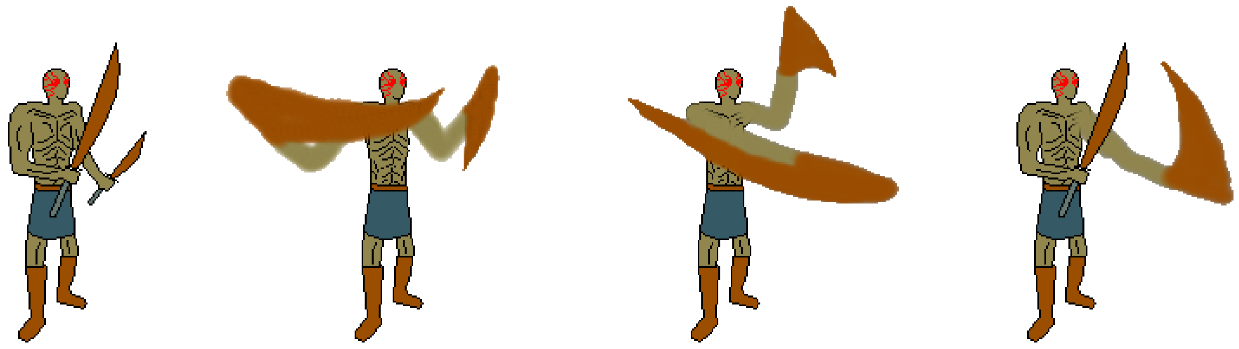


Figure 14.6: Enemy attack2

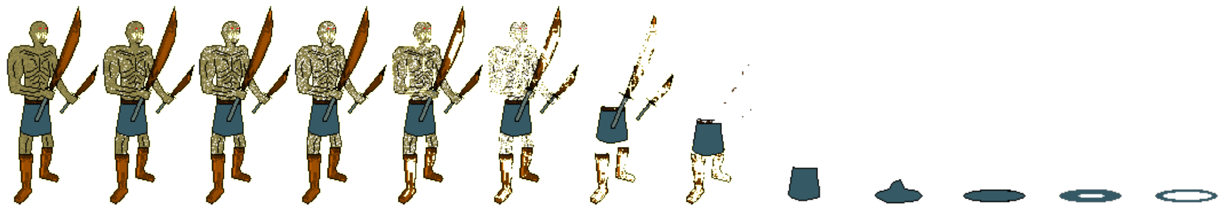


Figure 14.7: Enemy dies

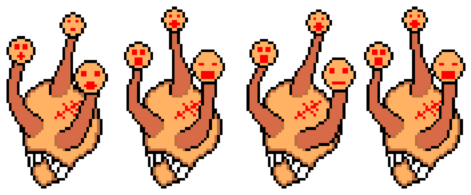


Figure 14.8: Enemy still

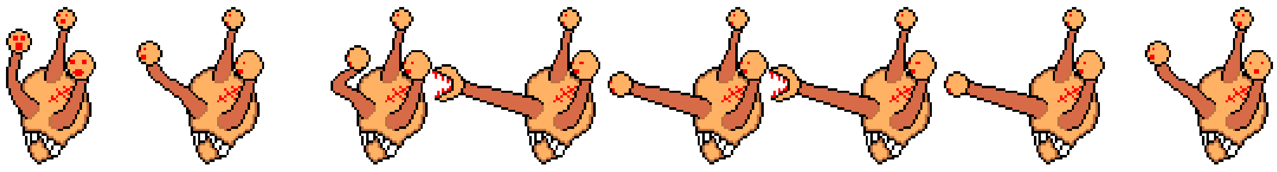


Figure 14.9: Enemy attack1

Figure 14.10: Enemy attack2

Figure 14.11: Enemy dies

Figure 14.12: Enemy attack1

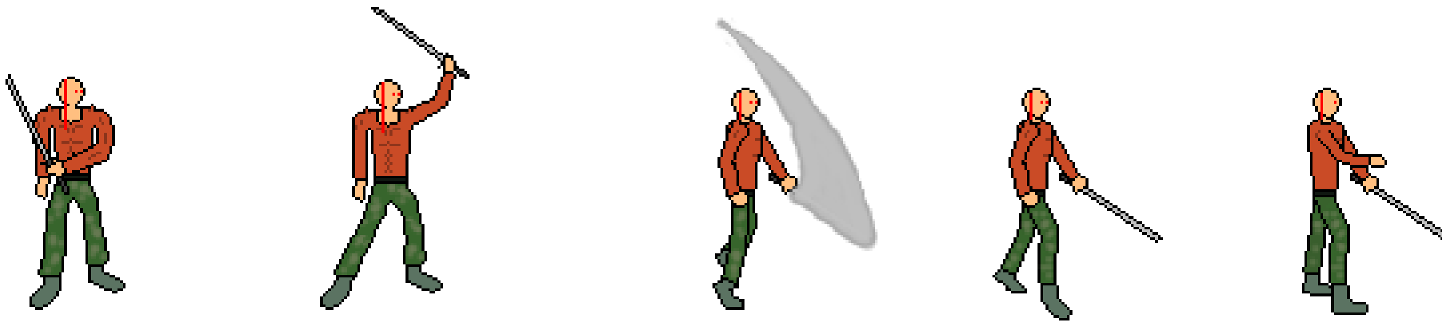


Figure 14.13: Enemy attack2

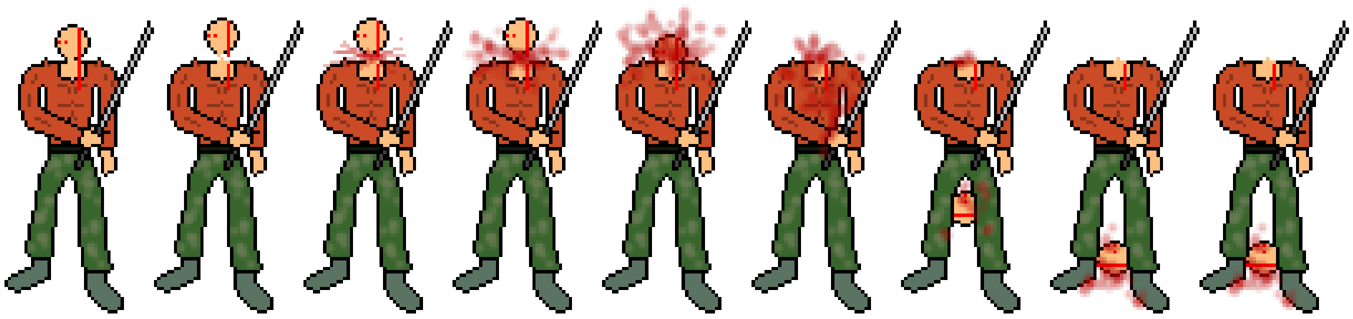


Figure 14.14: Enemy dies

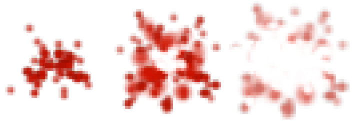


Figure 14.15: Blood splash

Figure 14.16: Platform

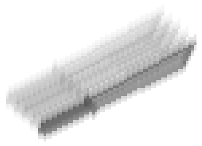


Figure 14.17: Flying sword

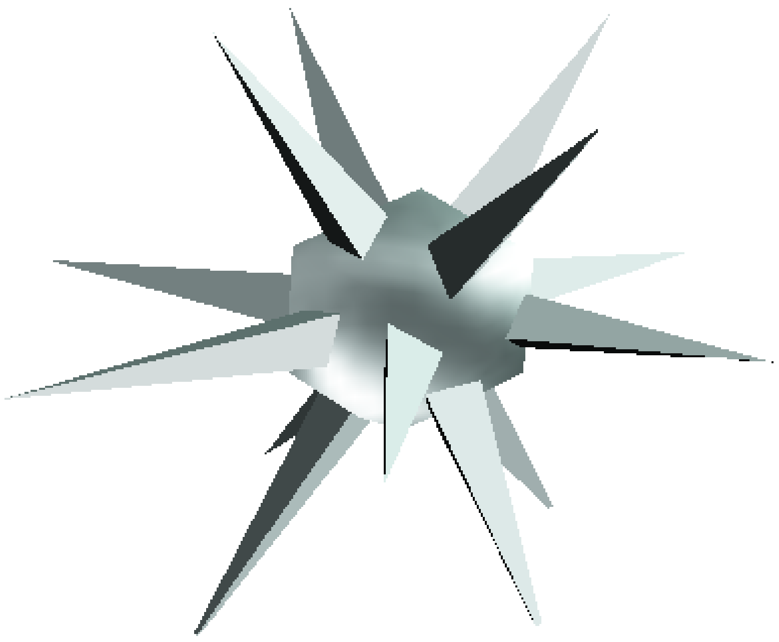


Figure 14.18: Spike

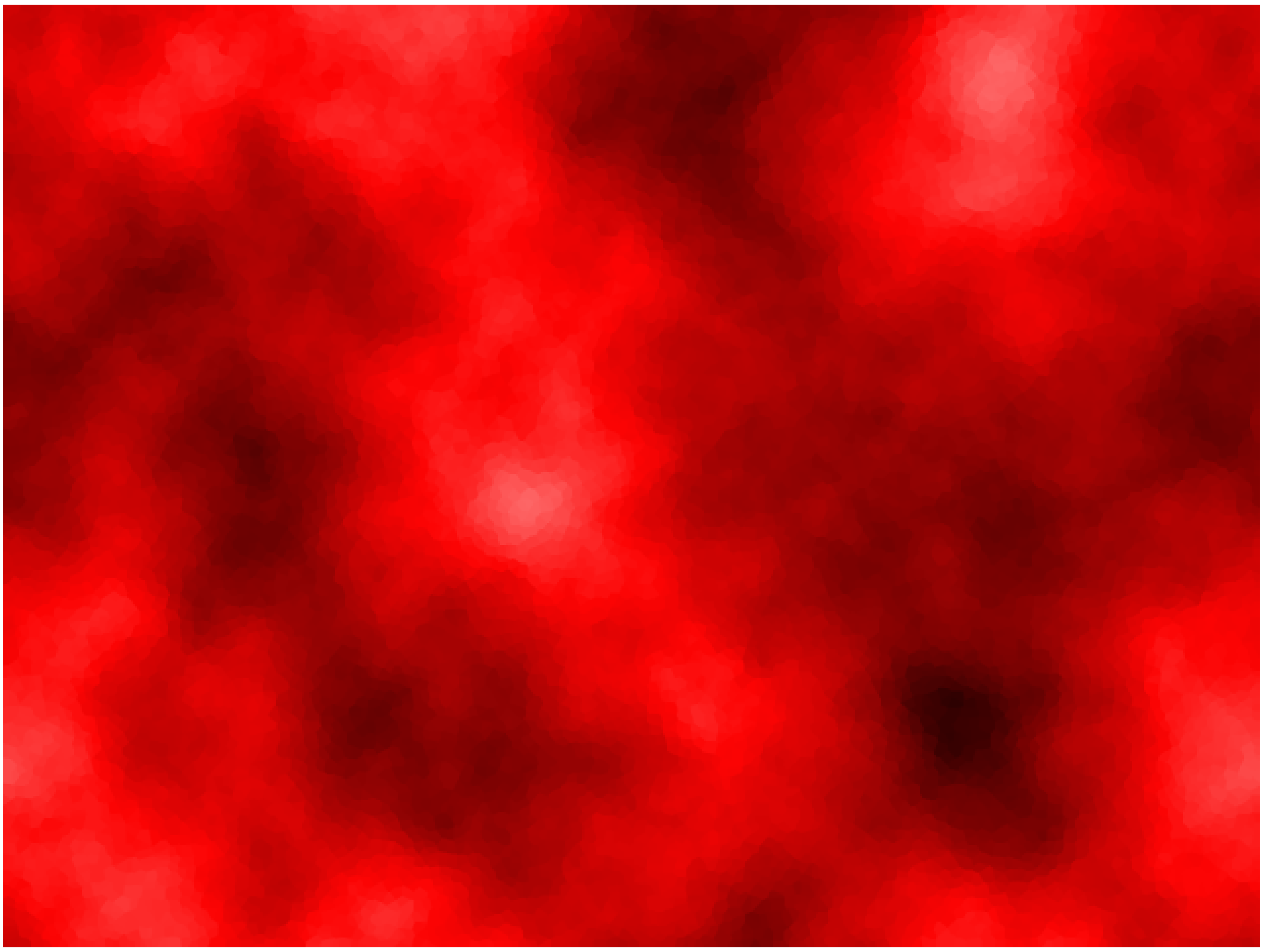


Figure 14.19: Background 1

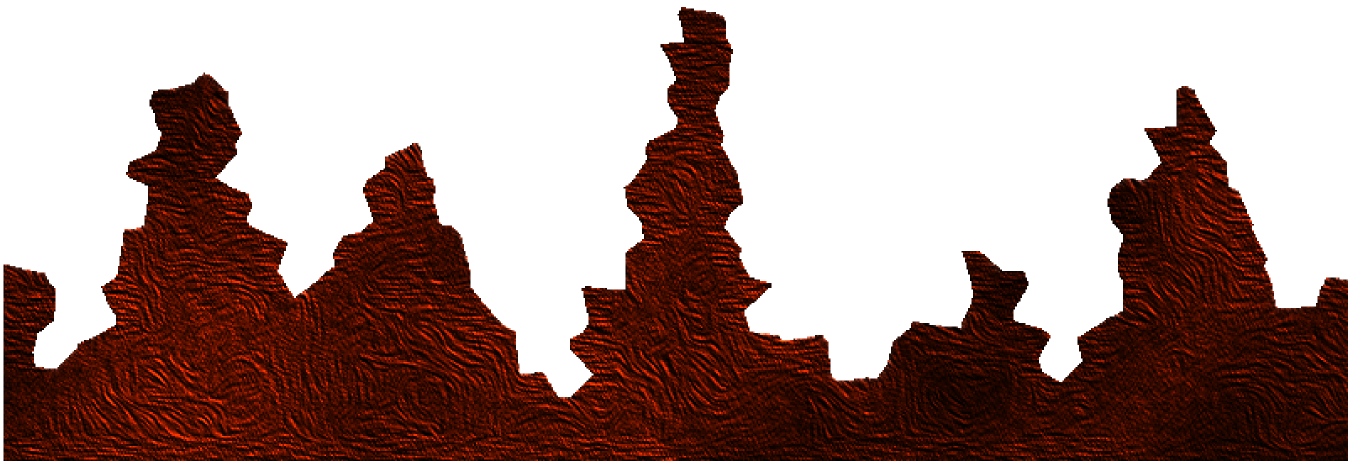


Figure 14.20: Background 2