RITSUMEIKAN UNIVERSITY

Runner

Maintenance Manual

Authors : EL-HABR Camille & GARCIA Xavier
Supervisor : Pr. THAWONMAS Ruck

Intelligent Computer Entertainment Lab, Ritsumeikan University

Date : August 24, 2018

Contents

1 Introduction

This document contains necessary information for the maintenance or a future evolution of the platform runner game **Runner**, developed in the *Intelligent Computer Entertainment Lab*. It's intended for developers of the software.

For information on game use, refer to the User Manual.

Runner is a 2D platform game of type Endless Runner. The goal is to go as far as possible in the level without dying, by avoiding obstacles. The special feature of the game is that it adapts to the user feedback. In the body mode, it adapts the level according to the player's death and the amount of calories the player wants to burns. So the body mode has to be used with a Kinect camera. In the emotion mode, the game adapts to the player emotion. So it has to be used with a camera.

This report is divided into 4 main parts: the game itself, the main menu, the tutorial mode and the main level. Each part uses the naming conventions below :

- Unity Object & `C#` Script in bold and red color;
- Unity Component in bold and black color;
- Variable name, folder and layer in italic.

2 The Game : Runner

This game has been coded in `C#` language, on the **Unity3D** game engine in the `2018.1.1f1` version.

Runner/Assets is the main folder that contain all required elements to make the game work (Assets used by a Unity project). It contains 10 folders :

- *Affdex* : plugins and required libraries to allow emotion mode;
- *Animations* : elements animations made with **Unity3D** for the character movement (run, stop, jump, fall, bend down, left arm, right arm), coins rotation & water waves;
- *Materials* : additional physical properties;
- *Prefabs* : store a GameObject object complete with components and properties (template to create new object instances in the scene);
- *Resources* : folder to store Objects we want to find and access by including assets;
- *Scenes* : contains the environments and menus of the game. This project contains 3 scenes : one for the main menu, one for the tutorial and the last one for the main level;
- *Scripts* : `C#` scripts as Components for GameObjects;
- *Sounds* : sounds effects and musics used in the game (*mp3* format). All sounds come from free musics and sound effects websites `https://www.zapsplat.com/` & `http://dig.ccmixter.org/games`;
- *Sprites* : 2D Graphic objects (*png* format). All sprites come from a free package in the Unity3D Assets Store. The package is the "Free Platform Game Assets, Texture & Material/2D & Isometric Tiles" by Bayat Games. Backgrounds used comes from the Ritsumeikan University database on Ukiyo-e;
- *StreamingAssets* : classifier data files required by libraries in emotion mode.

To add or modify a game element, please refer to each folder function above.

To change settings of the configuration screen, please refer to the Unity3D documentation.

3 Main Menu

The main menu is the main interface to select the player mode (cf. Figure 1). It's the first screen that appears when launching the game. This corresponds to the **MainMenu** scene in the *Unity* project. Scene contents are handled by the **MainMenuController.cs** script, which is attached to the **Canvas**.

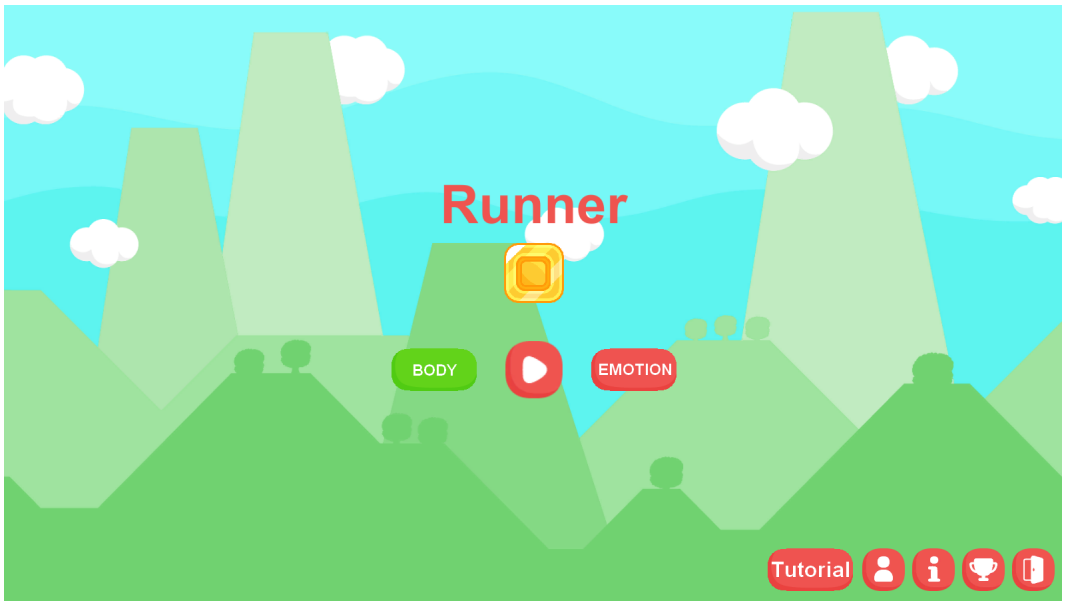


Figure 1: Main Menu Screen

3.1 Camera & Background

Unlike the main level, the **Camera** & the **Background** are static in the main menu. To modify the background image, change the *sprite* in the component **Sprite Renderer** of the **Background** GameObject.

3.2 Sound Effects

In the main menu, there are two sounds : the background music and a button sound. Musics and sounds are stored in the *Assets/Sounds* folder.

The background music is played on a loop when the game is launched. To change this music, change the *AudioClip* in the **Audio Source** component of the **music** GameObject in the **SoundsEffects** folder.

The button sound is played only when a button is clicked. To change this on-click sound, change the *AudioClip* in the **Audio Source** component of the **select** GameObject in the **SoundsEffects** folder.

3.3 Player Preferences

Unity PlayerPrefs are used for storing important values which have to be keeped in all mode game. PlayerPrefs allow value transfers between all scenes. *PlayerName* is a string to store the player name during the game. Others PlayerPrefs are all int values :

- *HighScore* stores the high score value (recover from last games);
- *CaloriesToLost* stores the amount of calories the player want to burn (0 by default);
- *Timer* stores the timer requested by the player to achieve his calories goal;
- *PreviousTime* allows to keep in memory the previous time did by the player before dying to continue the game with the last time;
- *Speed* is the character speed selected by the player;
- *BodyMode* stores the selected game mode. Body mode is selected by default and the variable is at 1, else it's the emotion mode and the variable is at 0;
- *RandomMode* stores the type of experiment. By default, not random mode is selected and the variable is at 0, it means intelligent mode;
- *TestMode* stores the type of test for the Emotion Mode (4 types : 1-coins, 2-taps, 3-speed, 4-hints).

CaloriesToLost, Timer, PreviousTime and Speed are only related to the Body Mode.

3.4 Canvas

When launching the game, only **MainMenu** GameObjects are enabled. Others GameObjects will be activated only when the specific button will be clicked on. At this time, **MainMenu** will be disabled.

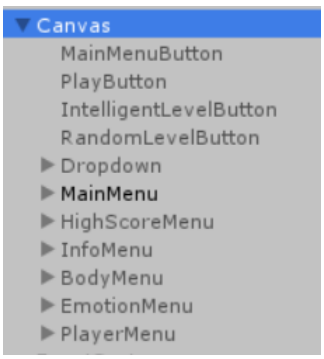


Figure 2: Main Menu Elements

3.4.1 Clickable Buttons

EventSystem allows button on-click use. All buttons work in the same way. When a button is clicked, the select sound is played, as mentioned above. To change this sound, refer to this previous part.

There are 9 buttons available in the main menu scene :

Button	Size (WxH)	Function
	128x64	Select the body mode for the game
	64x64	Select the emotion mode for the game
	128x64	Launch the game (Level Scene)
	128x64	Launch tutorial mode (Tutorial Scene)
	64x64	Display player settings screen
	64x64	Display game information
	64x64	Display the high score screen
	64x64	Exit the game
	64x64	Go back to the main menu

To change the button image, change the *Source Image* in the **Image (Script)** component of the button GameObject selected. Images are stored in the *Assets/Sprites* folder.

To change the button size, change the *Width* and/or the *Height* in the **Rect Transform** component of the button GameObject selected.

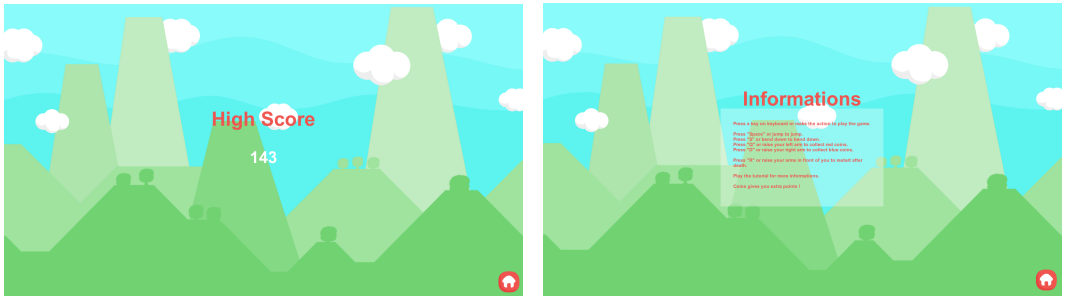
To change the button action, just change the name of the function called in the *On Click()* part of the **Button (Script)** component of the button GameObject selected. Buttons and **MainMenuController.cs** script are already attached to the **Canvas**. As mentioned above, clicking on a button disabled the main menu to run the button function. When going to another screen than the main menu, clicking on the home button allows to go back to the main menu : it means disabled the ongoing screen objects and enabled the main menu again.

3.4.2 Text On Screen

The title text appears on screen when launching the game. This text can be changed at any time by writing in the *Text* field in the **Text (Script)** component of the text GameObject

selected. It's the same thing for texts on the high score screen and on the information screen (cf. Figure 3). But the **HighScoreValueText** is set according to PlayerPrefs *HighScore*, so it's not possible to change this value by this way. Likewise, text under sliders are also set according to the slider value above, so it's also not possible to change the text value.

HighScoreMenu and **InfoMenu** are just screens to display texts. When they are activated, **MainMenu** is disabled and the **MainMenuButton** to go back to the main menu is enabled on the bottom-right side of the screen.



(a) High Score Screen (b) Information Screen

Figure 3: Main Menu Screens

3.4.3 Slider Bar

Slider bars set PlayerPrefs *CaloriesToLost*, *Timer* and *Speed*. The float value of each slider is converted into an int value. Those bars are activated only in the body mode (cf. Figure 4).

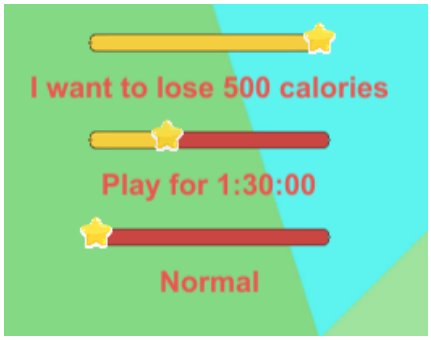


Figure 4: Body Mode Sliders

To change the slider size, modify the *Width* and/or the *Height* in the **Rect Transform** component of the **Slider**. To modify the maximum value, change the value in the *Max Value* field in the **Slider (Script)** component of the **Slider**.

Like for buttons, it's possible to change the slider action, by choosing another function in the *On Value Changed (Single)* field in the **Slider (Script)** component of the **Slider**.

To change the bar background color, change the *Source Image* or the *Color* in the component **Image (Script)** of the **Slider>Background** GameObject. To change the bar top color, change the *Source Image* or the *Color* in the component **Image (Script)** of the **Slider>FillArea>Fill** GameObject. To change the slider handle color, change the *Source Image* in the component **Image (Script)** of the **Slider>HandleSlideArea>Handle** GameObject.

3.4.4 Dropdown

The dropdown element contains players available in the database (*Assets/Resources/colors.csv* folder). "Default" is the first name that appears, checked by default (cf. Figure 5). Names allow to make links between players and their association between colors and emotions.

Figure 5: Dropdown Element

3.4.5 Colors Picker

For emotion mode, colors has to be associated with each emotion for each player, as there is no universal association between color and emotion. So, to allow the player to choose the color he wants to associate to each emotion, a color picker had been added (cf. Figure 6).

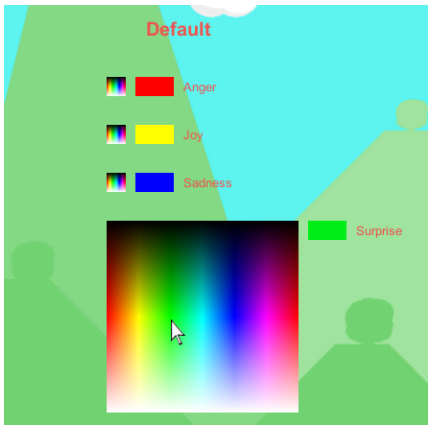


Figure 6: Colors Picker Element

DrawGUIOrderColorPickerController.cs displays on screen a color picker for each emotion: anger, surprise, joy and sadness. **ColorPickerController.cs** allows the selection of a color on a picture. Those scripts are a part of the package "Color Picker Scripting/GUI" free asset from the Unity Store by "Sergey Taraban". By default, colors and emotion are associated as follow :

Emotion	Color
Anger	
Sadness	
Joy	
Surprise	

In other case, colors information will be saved in *Assets/Resources/colors.csv* next to the player name. The reading and writing of this file is done by the **ColorFileManager.cs** script. This script manipulates objects of type **PlayerColor**. These objects contain the name of the player, and these colors associated with each emotion in RGB format.

3.5 Main Menu Interfaces

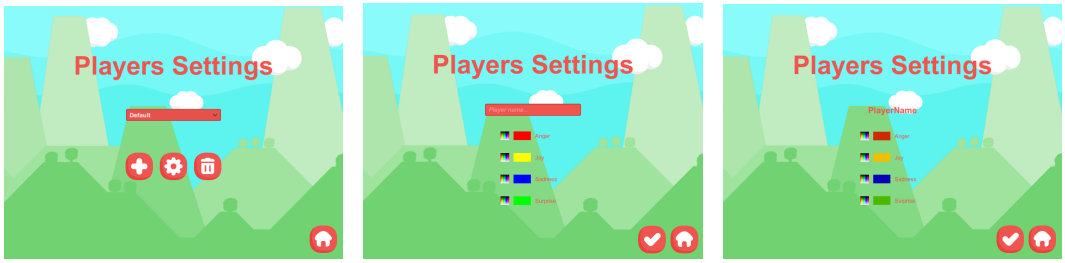
The main menu has 4 interfaces : the tutorial screen (explain in the next part), the player screen, the body mode interface and the emotion mode interface.

3.5.1 Interfaces Buttons

Button	Function
	Add a player
	Modify a player
	Delete a player
	Validate a player modification
	Return to the main menu
	Launch intelligent mode in body mode
	Launch random mode in body mode
	Launch default mode (intelligent mode in body mode and events mode in emotion mode)
	Launch an empty level with no feedback/events in emotion mode
	Launch a level with music/color feedback in emotion mode
	Launch a level with events and music/color feedback in emotion mode

3.5.2 Player Screen

The player screen is enabled when clicking on the Player button in the main menu. This screen allows the player to create a new user profile with his name and the association between color and emotion (cf. Figure 7). The player can create a new profile or modify an existing one by selecting a name in the dropdown list and modify colors recorded in the *Assets/Resources/colors.csv* file. This screen is handled by the **PlayerSettingsController.cs** script.

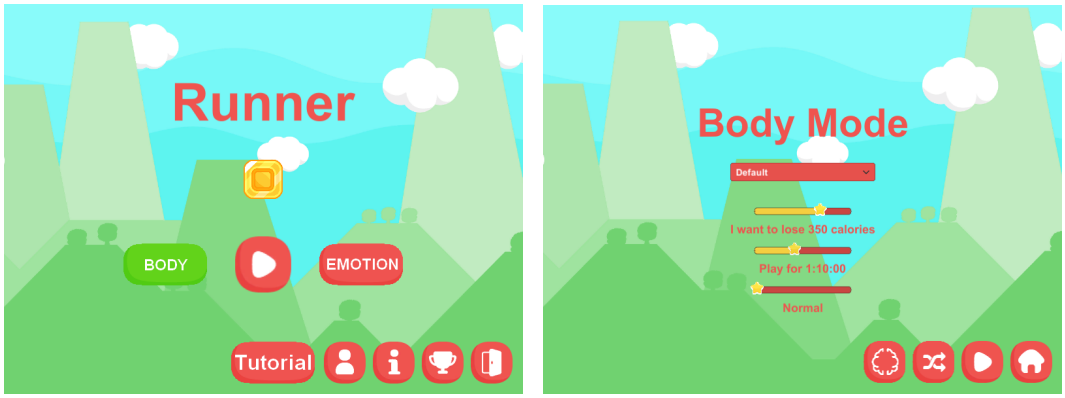


(a) Select Player (b) Add Player (c) Modify Player

Figure 7: Player Settings Screens

3.5.3 Body Mode

The body mode screen appears when clicking on the body button then on the play button in the main menu (cf. Figure 8). It provides access to a settings screen to adjust some parameters before playing the game (player name, calories to lost, timer, speed). This screen is handled by the **BodyModeController.cs** script. Specific buttons appear on that screen to launch intelligent level (brain button) or random level (arrow button). Clicking on the play button launch the intelligent mode by default.

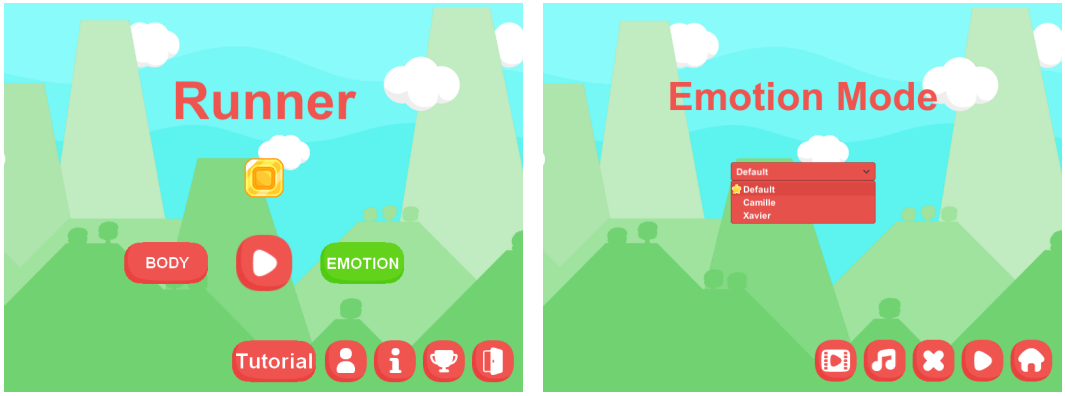


(a) Select Mode (b) Select Body Mode Settings

Figure 8: Body Mode Screens

3.5.4 Emotion Mode

The emotion mode screen appears when clicking on the emotion body button then on the play button in the main menu (cf. Figure 9). It provides access to a settings screen to adjust some parameters before playing the game (player name). Three buttons are available to launch one of the three test mode : with nothing, with music/color feedback or with music/color feedback and reward/penalty events. Clicking on the play button launches the events mode by default.



(a) Select Mode (b) Select Emotion Mode Settings

Figure 9: Emotion Mode Screens

Be careful, due to recent modifications, some buttons are disabled in the emotion menu. To allow there use again to make more tests, active them in the menu (cf. Figure 10). Those specific buttons allow to launch one of the four test levels (coins, traps, speed, hints).

Disabled Buttons	Function
	Launch a level with coins only
	Launch a level with traps only
	Launch a level with random speed
	Launch a level with hints at the position where you have to do the action

Figure 10: Additional Tests Buttons

4 Tutorial Mode

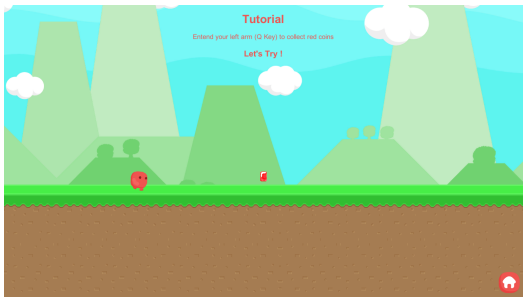
The tutorial mode is enabled when clicking on the *Tutorial* button in the main menu. This mode allows the player to practice game movements before playing the game. This corresponds to the **Tutorial** scene in the *Unity* project. Scene contents are handled by the **TutorialController.cs** script, which is attached to the **Player**.

4.1 Camera, Background & Platforms

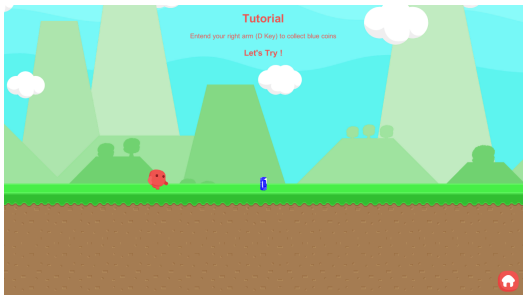
Block platforms, background and camera are also statics in this mode. To modify the background image, change the *sprite* in the **Sprite Renderer** component of the **Main Camera>Background** GameObject.

4.2 Player Movements

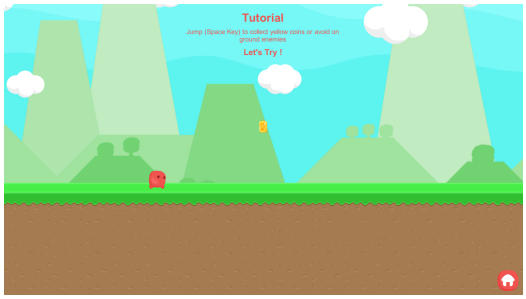
The character moves forward alone with a constant speed and the player have to make movement in order to see the next action. There are 4 movements/actions to do : tend left arm/Q key, tend right arm/D key, jump/Space key and bend down/S key (cf. Figure 11). For more details on the character dynamic, please refer to Section 5.2.1.



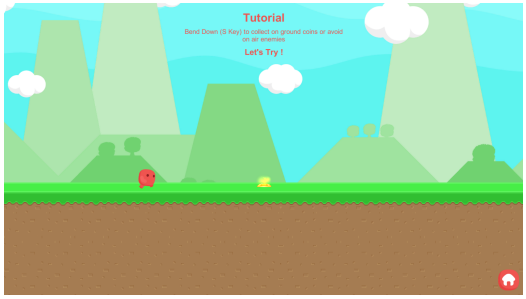
(a) Red coin for left arm



(b) Blue coin for right arm



(c) Yellow coin for jump



(d) Block for bend down

Figure 11: Tutorial Mode Actions

4.3 Generation Points

As long as the player does not make the move, the character goes back to the starting point when he arrived to the ending point (cf. Figure 12). Position of those points can be modified by changing X and Y *Position* in the **Transform** component of each point.

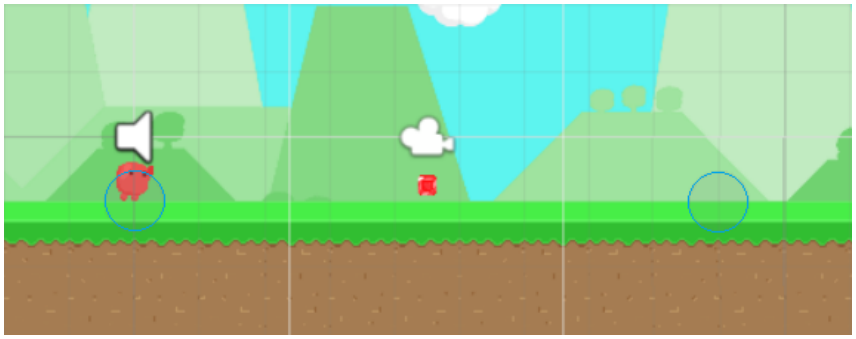


Figure 12: Starting and Ending points of the tutorial mode

4.4 Sound Effects

In the tutorial mode, there are five sounds for player actions:

- "jump" sound is playing when the character jump;
- "run" sound is playing in a loop when the character is running on ground;
- "coin" sound is playing when there is a collision between the box collider of the player and the box collider of a coin;
- "congrats" sound is playing when the player finish the tutorial mode successfully;
- "select" is playing when the player click on the main menu button.

Musics and sounds are stored in the *Assets/Sounds* folder. To change a sound, change the *AudioClip* in the **Audio Source** component of the sound GameObject in the **SoundsEffects** folder.

4.5 Canvas

4.5.1 Clickable Button

The tutorial mode has only one button to go back to the main menu. To change on-click sound, image, size or action, please refer to the section 3.4.1. Obviously, this button action is related to the **TutorialController.cs** script.

4.5.2 Text On Screen

Tutorial mode has 3 texts for the title, instructions and the order. It's possible to change the mode title by changing *Text* of the **Text (Script)** component in the **TitleText** GameObject. Other texts are set according to the tutorial step, so it's not possible to change them.

5 Main Level

The main level corresponds to the **Level** scene in the *Unity* project. Main scene contents are handled by the **GameController.cs** script, which is attached to the **Game Manager**. In this scene, main objects have a script attached.

5.1 Modes Transition

The level is launched when clicking on the *Play* button in the main menu. This mode allows the player to play the game in the body or emotion mode. The mode is loaded thanks to the *BodyMode* PlayerPrefs. PlayerPrefs are used to transfer variables through different scenes (here from the main menu to the main level). The *BodyMode* variable is an int type and correspond to the mode : 0 for the Body mode, 1 for the Mental mode. So it's possible to add more mode. Each mode allows the player to play with the keyboard, but in the body mode, it's possible to play by moving your body thanks to the Kinect camera. In the emotion mode, it's possible to see feedback according to your facial expression.

GameController.cs is the main interface between all scripts. At the beginning of a game, it charges the correct mode by activating or disabling elements. As the body mode is the default mode, the **PlatformGenerationEmotionController.cs** in the **PlatformGeneration** GameObject and **Detector (Script)** and **Camera Input (Script)** in the **MainCamera** GameObject are disabled when launching the game. If the emotion mode is selected, those elements will be enabled and the **Canvas>CaloriesManager** and the **PlatformGenerationController.cs** in the **PlatformGeneration** GameObject will be disabled.

5.2 The avatar in the game

The avatar is the character controlled by the player. This avatar has physical characteristics, animations, and has to react to actions of the player in order to make a playable game. These features are related to the **Player** object and the **PlayerController.cs** script.

Figure 13: The Avatar

5.2.1 Avatar Physics

Physical Characteristics : In order to move coherently the avatar in the game environment, it must have several physical characteristics :

- a mass
- a weight

- a velocity

The avatar needs also to have a collision box. It is thanks to this element that we are be able to detect collisions between the avatar and other elements of the environment (provided that these elements also have a collision box).

In *Unity*, associating the **Rigidbody 2D** component with our avatar allows to put it under the influence of the physical engine. In particular, he will be affected by gravity.In addition, the **Box Collider 2D** component adds a hitbox around the avatar, which will later allow to detect collisions between the avatar and other elements of the game.

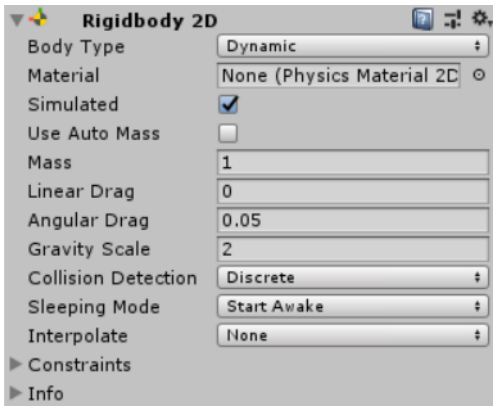


Figure 14: Rigidbody 2D

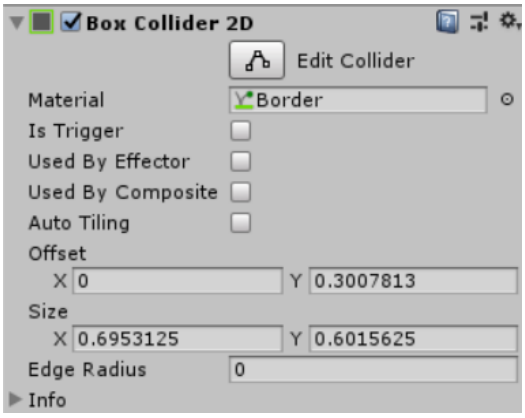


Figure 15: Box Collider 2D

Ground Detection : It is also necessary to associate an hitbox with the platforms to prevent the player going through the platforms of the game :

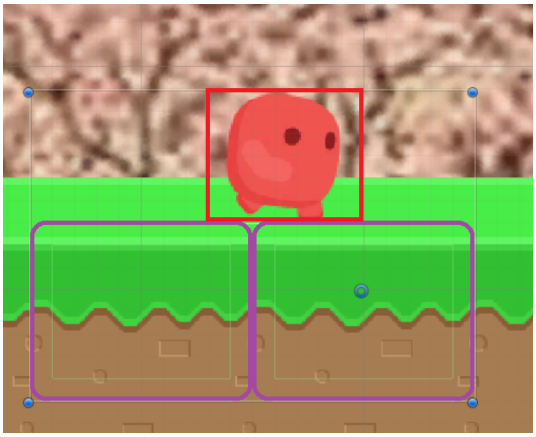


Figure 16: Hitbox (platforms and avatar)

An important mechanics of platform games is the jump control. In our game the avatar

is able to jump once, provided it is in contact with the ground. Therefore, it is necessary to know when the avatar is in contact with the ground.

One solution is to place the platforms in a specific **Layer**, called *Ground*. Thus, we will be able to determine if the hitbox of the character is in contact with a hitbox associated with *Layer Ground* by adding a boolean variable in the script associated with the character.

However, this solution can cause a bug when the character falls from a platform. If the hitbox of the avatar is in contact with one side of the hitbox of a platform, the player will be able to go up on the platform by jumping several times (cf Figure 17).

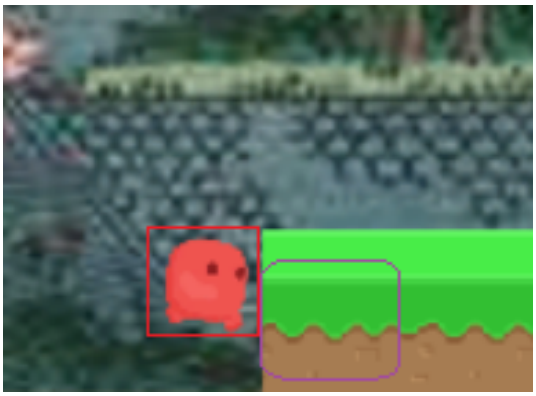
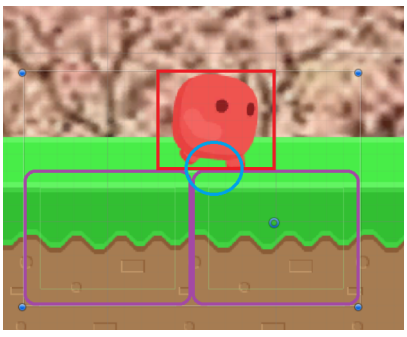
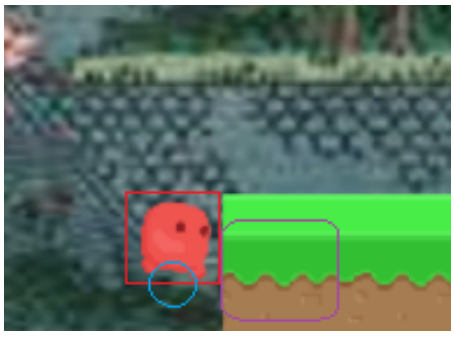


Figure 17: Ground detection issue

A better solution is to associate with the avatar a new object, called *GroundCollision*. This object is located under the hitbox of the avatar. We will then try to detect if an element associated with the layer *Ground* is in contact with a circle whose center is located at the location of the *GroundCollision* object, using the method **Physics2D.OverlapCircle**. This solution will solve the collision problem explained previously.





5.2.2 Avatar Control

Keyboard Input Detection : The player is able to control the avatar using the keys on the keyboard. The **Input.GetKey** method allows to know if the player is pressing a particular key or not.

Avatar Animation : The avatar has 6 different animations:

- run : basic animation of the avatar when no key is pressed
- left : the avatar stretches the left arm
- right : the avatar stretches the right arm
- bendDown : the avatar bends down
- jump : starts when the avatar jumps
- fall : starts when the avatar is falling (after a jump or during a fall)

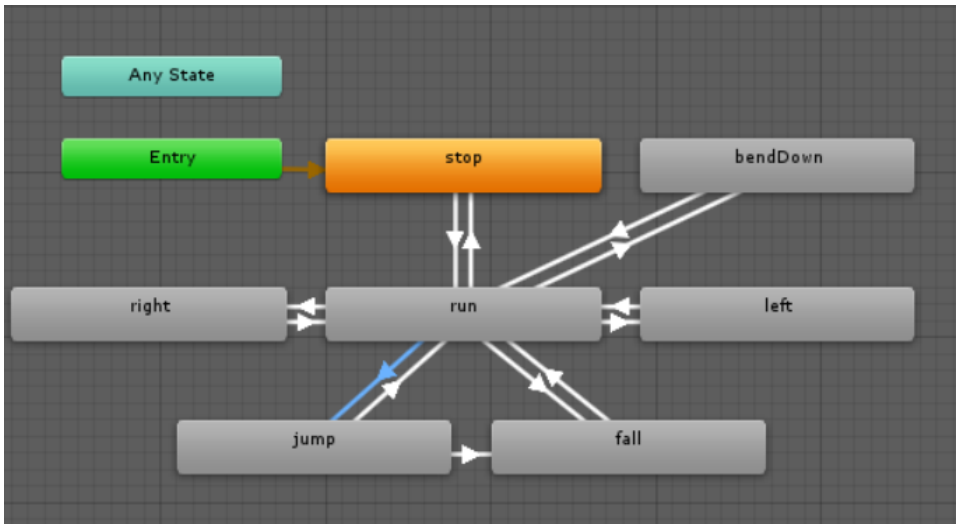


Figure 18: Animator of the avatar

All these animations are linked in a **Animator Controller** (cf Figure 18), which is a new component attached to the character. The transition from an animation to another is based on the links between the animations and conditions attached to these links. For example, in order to trigger the animation *jump*, the player must be running (in the animation *run*), must not be falling and must not be in contact with the ground (cf Figure 19).

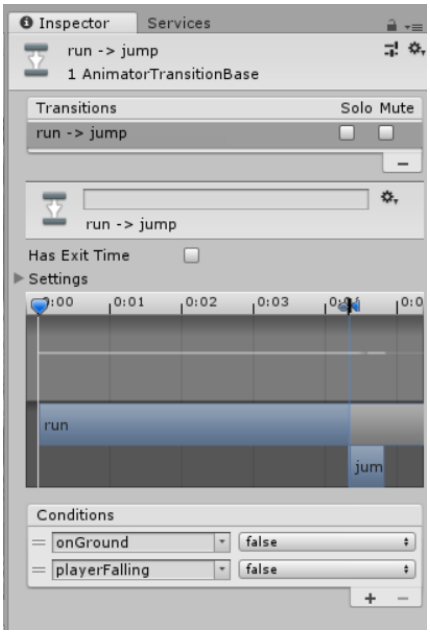


Figure 19: Transition between *run* and *jump*

Some variables related to *Animator Controller* (to organize transitions between animations) are attached to the player's keyboard input.

5.2.3 Difficulty Control

To increase the difficulty, we chose to increase the speed of the avatar according to the duration of the game. The speed will grow thanks to a logarithmic function depending on the time, but will be limited to prevent the game from becoming unplayable. Speed management is done by the **IncreaseSpeed** method (**PlayerController.cs**).

$$playerSpeed = startSpeed + \frac{log(timeSpend + 1)}{10} \quad (1)$$

5.3 Camera & Background

The camera is the element of the scene that allows the player to visualize the content of the game. The background is a set of images in the background. These objects are named **Main Camera** and **Background**, and are attached respectively to respectivement **CameraController.cs** and **BackgroundController.cs** scripts.

5.3.1 Moving The Camera

For our game, the camera has to follow the character. This gives the impression that the character runs on the spot, and that the level unfolds in front of him.

To force the camera to follow the character, it is necessary to recover the distance traveled by the avatar, and then transfer the camera of the same distance and in the same direction as the character (**UpdateCamera** method in the **CameraController.cs** script).

5.3.2 Moving The Background

Moving the background is more complex than moving the camera. The background must move less quickly than the character, and in the opposite direction in order to give a perspective effect. We chose arbitrarily that the speed of the background will be half of the speed of the avatar.

To move the background infinitely, the **UpdateBackground** method (in **BackgroundController.cs** script) will transfer the background image as soon as the player is ready to go beyond it. The background must be adapted so that this transition is imperceptible (cf Figures 20 and 21).

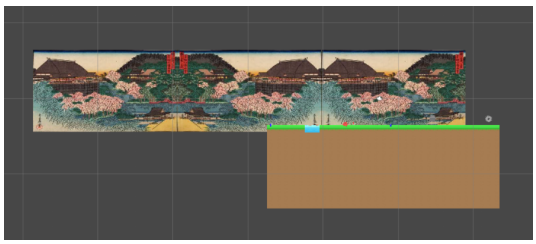


Figure 20: Background before the transfer

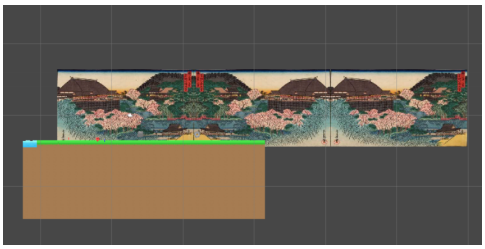


Figure 21: Background after the transfer

5.4 Platform Generation

The goal was to create the level in a procedural, dynamic and consistent way. The level must be created from rules, which themselves can be modified during the game.

The scripts used in this part are **PlatformsGenerationController.cs** and **PrefabController.cs**.They are respectively attached to the **PlatformsGeneration** object, and to the different platforms created.

5.4.1 Platforms Creation & Destruction

As mentioned above, the level is not already created when you start a game. It will be created as the player progresses in the level.

To allow the creation of new platforms, we attached to the camera a new object called **GenerationPoint**. In addition, the **PlatformsGeneration** object will always be placed at the end of the last created platform. Thus, when the **GenerationPoint** object goes past **PlatformsGeneration**, a new platform is created, and the **PlatformsGeneration** object is moved at the end of it (**GeneratePlatform** method in **PlatformsGenerationController.cs**).

In order not to saturate the memory, it is necessary to destroy the platforms that the player has already gone past and which are no longer visible on the screen. So, we attached an additional object called **DestructionPoint** to the camera . In the **PrefabController.cs** script associated with each platform, we will then check the position of the platform relative to the position of the **DestructionPoint** object : if the platform is behind this object, then it is destroyed and freed from memory.

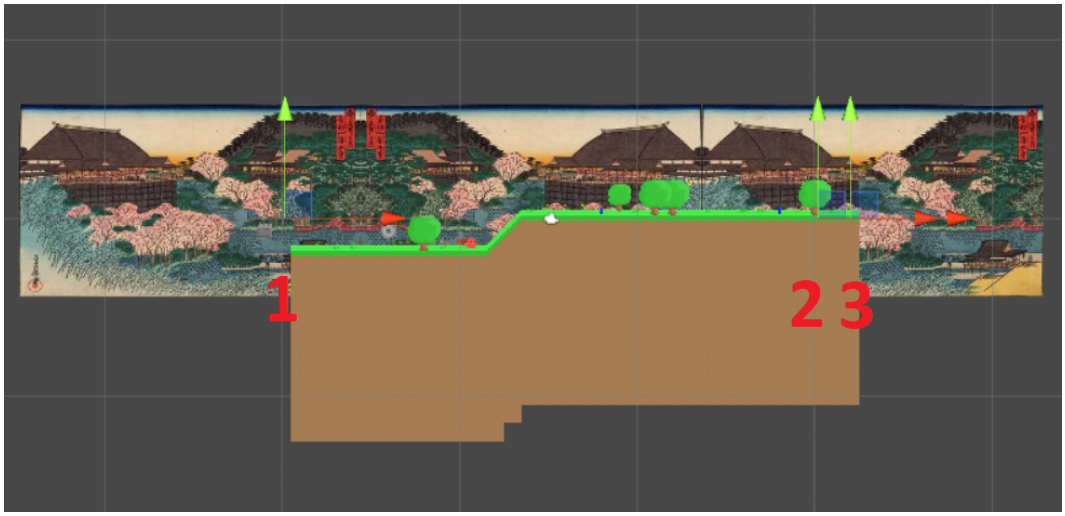


Figure 22: DestructionPoint (1), GenerationPoint (2), PlatformGeneration (3)

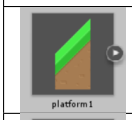
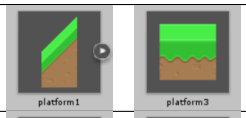
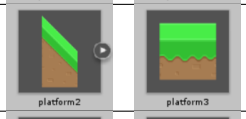
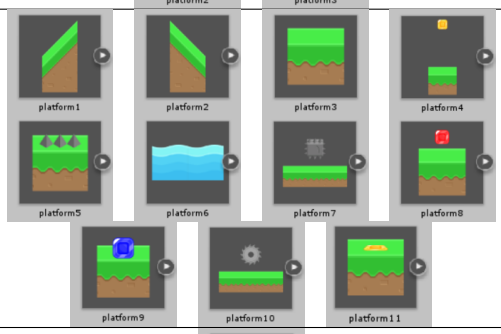
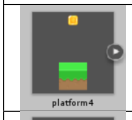
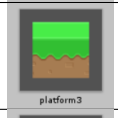
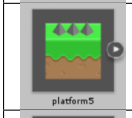
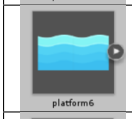
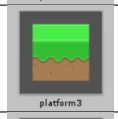
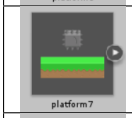
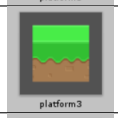
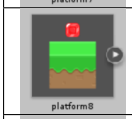
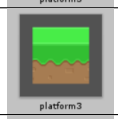
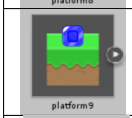
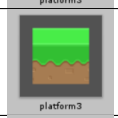
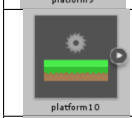
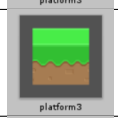
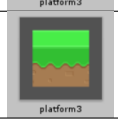
Platforms Characteristics : Different platforms are used to build the level. They are characterized by their width (*width*), their height (*height*) and by a weight (*InitialSpawnWeight*). The weight characterizes the probability of the platform to spawn.

Each platform also has a list of neighboring platforms (*possibleNeighbours*). Thus, if a platform is instantiated, it can only be followed by a platform contained in its list of neighboring platforms (cf. **PrefabController.cs**).

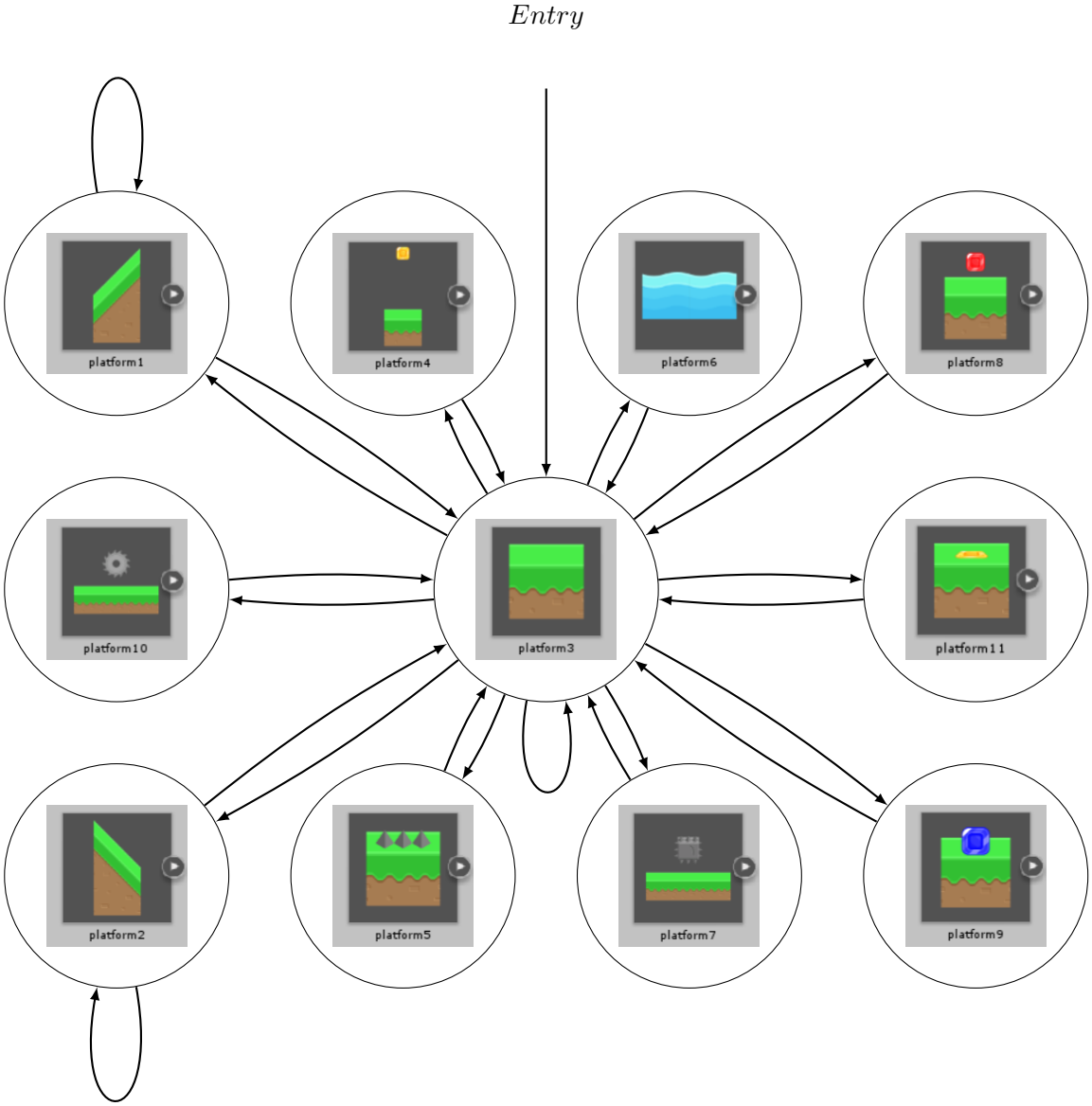
Let p_i be the i^{th} instantiated platform, N_{p_i} the list of neighboring platforms of the platform p_i, and w_p the weight of the platform p. The probability that the platform $p \in N_{p_i}$ is instantiated is equal to:

$$\mathbb{P}(p_{i+1} = p) = \frac{w_p}{\sum\limits_{p_k \in N_{p_i}} w_{p_k}} \tag{2}$$

Sample level creation rules : Here are the different platforms used to build the level :

Platform	Width	Height	Weight	Neighboring platforms
	1	1	90	
	1	-1	90	
	1	0	20	
	1	0	90	
	1	0	90	
	2	0	90	
	3	0	90	
	1	0	130	
	1	0	130	
	3	0	90	
	1	0	90	

Here is the automata associated with the creation of the level :



5.4.2 Create and add a new platform

Create a prefab : To add a new platform to our game, it is first necessary to create it on *Unity*. The simplest thing is to create it in the project scene and drag it to the *Prefabs* folder (accessible from the project file tree). This prefab object will be used during the creation of the level.

The platform must however respect some constraints : Its dimensions (width and height) must be multiples of the dimensions of a block (128px $\times$ 128px).

Set up the new platform : Once the new platform has been added to the *Prefabs* folder, you need to attach the **PrefabController.cs** script to it, then set up the platform (cf Figure 23).

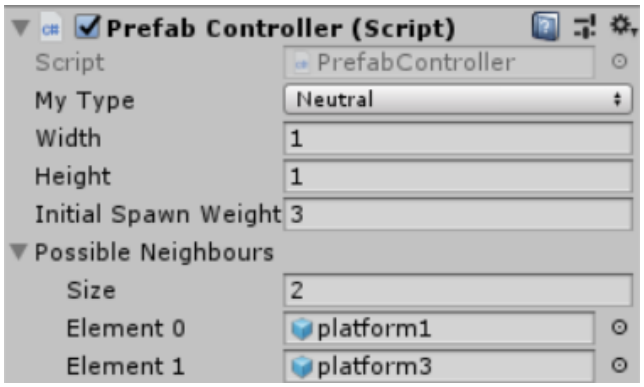


Figure 23: Platform settings

- My type : part of the body trained by the platform
- Width : width of the platform
- Height : height of the platform (negative if the platform goes down)
- Possible Neighbours : list of platforms that will be able to follow this platform

Then it is important to specify what is the type of this platform. This must be specified using the tag (here, *trapPlatform*, *water*, *coinPlatform*, or *Untagged*).

This new platform can also be added to the list of neighboring platforms of already created platforms.

After adding a new platform, it is necessary to check that the creation graph of the level does not contain a deadlock! It is also necessary to add this new platform to the *Platforms List* list of the **PlatformGeneration** object.

5.4.3 Dynamic adaptation of the level

Level height management The height of the level is be bounded by two variables associated with the **PlatformGenerationController.cs** script of the **PlatformGeneration** object: *Min Height* and *Max Height*. These two variables are respectively the upper limit and the lower limit of the height of the level. The null height (corresponding to 0) corresponds to the height of the first platform of the level (the object **StartPlatform**).

In the **PlatformGenerationController.cs** script, the *heightLevel* variable remembers the current height of the level. When a platform is instantiated, we will add the height of this platform to *heightLevel*. A platform can only be instantiated if *heightLevel* remains in the range $[minHeight; maxHeight]$.

Change the weight of the platforms : To vary the probability of appearance of a type of platform (*trapPlatform*, *water* or *coinPlatform*), simply change its weight during the game. In this project, the **TagProbabilities** method (**PlatformGenerationController.cs**) takes a tag and the value of the new weight, and changes all the weight of the platforms associated with this tag. All the weights can be reset with the **ResetWeight** method.

Force the creation of a platform : It is also possible to force the creation of a platform using the **DepthFirstSearch** method of **PlatformGenerationController.cs**. This method takes two arguments:

- start : the last platform created
- target : the platform to reach

The method use Depth First Search to find a path between the *start* and *target* platforms. Then, this method adds the platforms of the path found in a list called *path*. If it is not possible to link these two platforms, no platform is added to *path*.

As long as this list is not empty, the game instantiates the platforms of this list, while pulling the instantiated platforms.

```
function DepthFirstSearch(start, target)
    Set start as discovered
    current <- platform
    s <- stack
    s.push(start)
    while stack.length > 0
        current <- s.pop()
        if current not discovered
            Set current as discovered
            path.add(current)
            if current == target
                return
            n <- 0
            for all p in the list of neighboring platforms of current
                If p not in path
                    s.push(p)
                    n <- n + 1
            if n == 0
                path.remove(path.length - 1)
```

Gap between two traps : As we have seen previously, the speed of the avatar increases as time goes on. It is therefore necessary to calculate the minimum gap needed between two traps (in our example platforms 5, 6, 7 and 10), to make the game playable and avoid frustrating the player.

The **TrapGap** method in **PlatformGenerationController.cs** calculates the distance traveled by the avatar when the player makes a jump. This distance depends on the speed of the avatar, the strength of the jump and the gravitational force of the level. Once this distance d is calculated, the appearance of d neutral platforms is forced between two traps (here platform3).

5.5 Environment Obstacles

All environment obstacles are in the *Assets/Prefabs* folder. They all have a **Box Collider 2D** and also a *tag*.

5.5.1 Collision Detection

The box collider allows to detect a collision between the player character and the target item. When a collision between two box colliders of 2D physics objects is detected, this will send an *OnTriggerEnter2D* message. Tags are used to know which object is on collision with the player (cf. Table 1). This way, it's possible to play different sounds (cf. Section 5.6) or to make different actions regarding the tag, like killing the player character or give extra points by collecting coins.

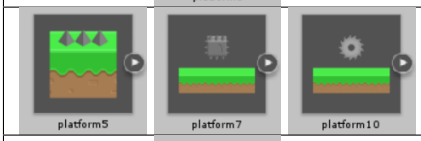
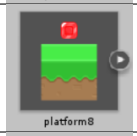
Platform	Tag	Action on collision
	water	Kill
	kill	Kill
	coin	Collect coin
	coinL	Collect coin
	coinR	Collect coin
	coinB	Collect coin

Table 1: Objects tags and action on collision

To add another object, add a new platform in the *Assets/Prefabs* folder. Then add a tag to the object attached to the previous platform and add your code in the *OnTriggerEnter2D* method in the **GameController.cs** script.

An exception is the **Catcher**. It's an empty object which also contain a **Box Collider 2D**. It is located under the ground (cf. Figure 24), at the bottom of the map and allows to

detect when the character fall under the ground (or fall in water). As mentioned above, it follows the camera to always be under the ground.

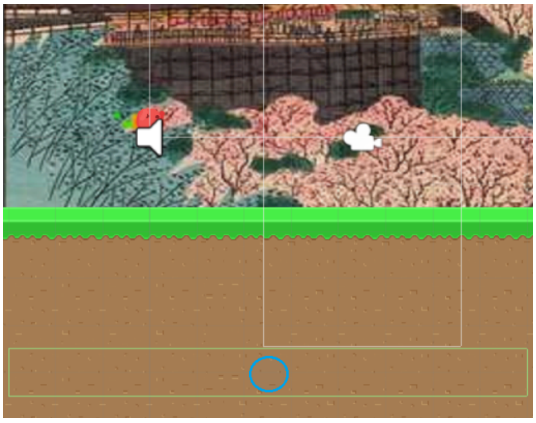


Figure 24: Catcher Object

5.5.2 Collision action & feedback

Collision action is related to the object tag. It's played only if there is a collision detection and the correct key/body input at the same time (cf. Table 2). In the body mode, the movement allows the player to trained his body. Each action in real world is converted into a keyboard input. If the correct action (movement or keyboard input) is done, this will trigger a feedback on screen.

For *coin, coinL, coinR, coinB* tags, it will collect the coin by setting at false the *SetActive* field of the gameObject. For *kill* tags, it will kill the player by disabling the character (so the camera and the background stop moving too) and loading the **deathMenu**. The *water* tag, let the player be killed by the **Catcher** gameObject : it feels like the character is drowning in the water platform.

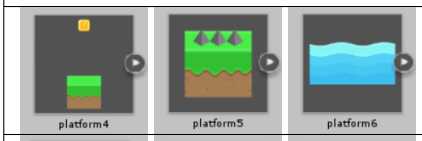
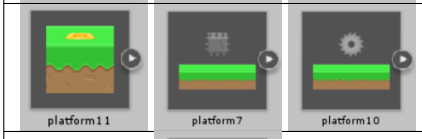
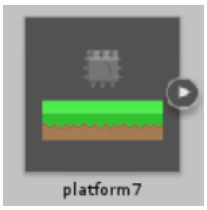
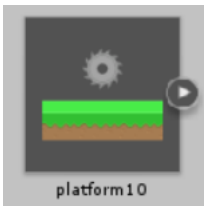
Platform	Body part trained	Movement	Activated key
	Legs	Jump	
	Legs	Bend Down	
	Left Arm	Stretch arm	
	Right Arm	Stretch arm	

Table 2

5.5.3 Enemy blocks

In our example, two different platforms force the player to bend down :





These platforms are linked with the **EvilBlockController.cs** and **EvilSawController.cs** scripts.

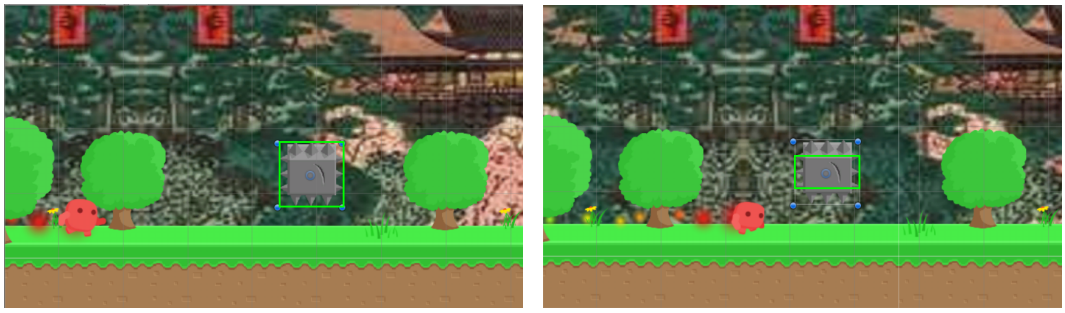
Platform7 has a block that moves horizontally. Its trajectory follows a sinusoid as a function of time :

$$transform.position = newVector2(xPosition + Mathf.Sin(Time.time), transform.position.y); \quad (3)$$

Platform10 represents a circular saw which will turn on itself at each call of **Update** method :

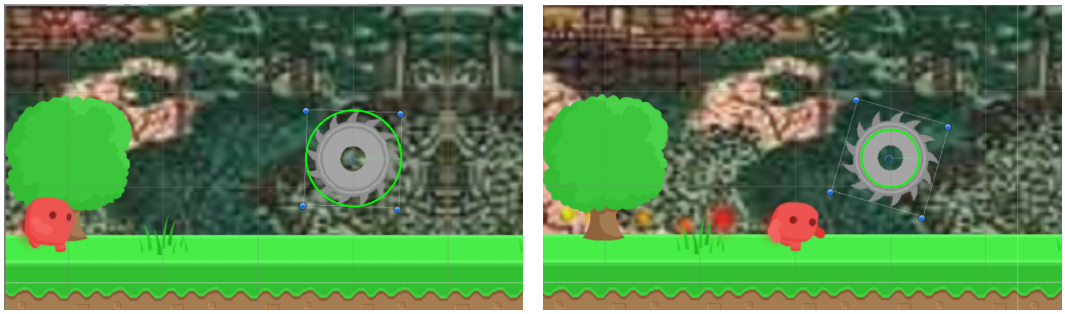
$$transform.Rotate(Vector3.forward * +2f); \quad (4)$$

When the player bends down, the hitbox of these two elements shrinks (**CheckBendDown** method). When the player gets up, their hitboxes are back to their original size. This prevents the player from colliding with the traps when he makes the right move.



(a) Before bend down (b) After bend down

Figure 25: EnemyBlock



(a) Before bend down (b) After bend down

Figure 26: EnemySaw

5.6 Music & Sound Effects

Musics and sounds are stored in the *Assets/Sounds* folder. In the **Level** scene, sounds (short **Audio Sources**) are in the **SoundEffects** empty Object and musics (long **Audio Sources**) are in the **Musics** empty Object. Sounds are used in all modes, but musics are only used in the emotion mode.

5.6.1 Sound Effect in game

Sounds appear in game only when the specific action is done (cf. Table 3). To change the sound, change the audio file in the *Audio Clip* field of the **Audio Source** component of the gameObject. Sounds can be added in the same folder, if you want to set a new action.

Sound	Trigger action
highscore	Current score exceed high score
killed	Collision between the player and an object with a *kill* tag
jump	Space key pressed
run	A loop run only when the player is not jumping
drown	Collision between the player and an object with a *water* tag
select	Click on a button on screen
coin	Collision between the player and a coin

Table 3: Sounds and their condition to be triggered

5.6.2 Musics in emotion mode

4 musics are used in the emotion mode for each emotion detected (anger, joy, sadness and suprise). Each music name corresponds to the emotion related. To change the music, just change the audio file in the *Audio Clip* field of the **Audio Source** component of the emotion gameObject. Music can be added in the same folder, if you want to set a new emotion. If you want to disabled the use of music, replace all sounds by *default.mp3*, it correspond to silence.

The script that manage music is **MusicController.cs**. But watch out ! Fade in/ Fade out methods generate an infinite loop with the pause menu as both use time.timescale functions. To use them, change the pause menu method or see unity documentation on invokerepeating/cancelinvoke methods to write new functions for fade in/fade out effects.

5.7 On screen display

The **Canvas** gameObject contains all graphic elements. Except calories information which are only on the emotion mode, all others information are used and displayed in all modes.

5.7.1 Coins Information

Coins are managed by the **CoinManager** empty object and the **CoinsController.cs** script. As mentioned above, when the player is in collision with a coin and do the proper movement, the collect sound is played but the number of coin is also increased throughout the game (*coinsNumber* variable). This number is displayed in a text gameObject on the top-left side of the screen, near the coin image gameObject. The *coinValue* allows to give multiplier value to each coin. This value is fixed to 1. Coins give extra points to the final score.

5.7.2 Score Information

Score is managed by the **ScoreManager** empty object and the **ScoreController.cs** script. The score is the distance between the character current position and the character starting position. The score is displayed on the top-left side of the screen in a text gameObject. It is updated at each frame.

The high score is stored in the *HighScore* PlayerPrefs and is only used in the High Score screen of the main menu and in the Death Screen of the level.

5.7.3 Calories Information

Calories are managed by the **CaloriesManager** empty object, the **CaloriesController.cs** script and the **TextFileController.cs** script.

Text File Management : the *Assets/Resources/informations.txt* file contains information received by the **UKI Module**. Currently, it only contains calories information. But the **TextFileController.cs** script parses the *txt* file to recover all data in a dictionary. It is therefore possible to add new data to be processed later.

Calories Slider & Text : Information about calories consumption from the text file is displayed on the top-right side of the screen in all body mode throughout the game. But the slider bar only appears in the body mode, when the player had entered a calorie goal in the main menu before starting the game. If he had, the slider progress regarding according to the calories lost read in the text file until the calories goal is reached. When the player reach his goal, an achievement sound is played. To modify the slider bar, refer to Section 3.4.3.

5.7.4 Time Information

The timer is managed by the **TimerController.cs** script. It appears only in the body mode when a calorie goal and a time goal is selected thanks to sliders. It displays the time between the beginning of the level and the current time in the top-middle of the screen. At the three quarters of the timer, if the calorie goal is not reached, the display becomes orange. When the timer is overtake and the calorie goal not reached yet, the timer become red. If the calorie goal is reached on time, the timer become green. When the player dies, the timer is not reset, the player keeps his time and his calories lost until reaching his goal.

5.7.5 Clickable Buttons

There is only one button in the game screen which allows the player to make a pause. This button is in the bottom-right corner of the screen. When the button is clicked, this calls the *Pause* method of the **PauseController.cs** script (cf. Section 5.10.1).

EventSystem allows button on-click use. When the button is clicked, the select sound is played. To change this sound, refer to section 3.2. To change the button image, size or action, refer to section 3.4.1.

5.8 Body Mode

As part of physical health promotion, an option in the game was added to integrate the movement of the player. So this mode has to be used with a Kinect camera.

5.8.1 Random Version

During the random mode, the generation of the level is basic. The platforms are generated randomly according to their initial weight. No changes are made in this mode.

5.8.2 Intelligent Version

Make the level more interesting : In order to make the level more interesting, two conditions were added in the **GeneratePlatformRandomly** method : On the one hand, it is impossible for a platform to appear twice (except for platforms where *My type* is equal to *neutral*). On the other hand, it is not possible to work two times the same part of the body (except for platforms where *My type* is equal to *neutral*).

Saving of the player's data : In this mode, the goal is to adapt the level difficulty. For this, the number of platform with a trap that the player managed to cross (here *platform5*, *platform6*, *platform7* and *platform10*), and on which trap he died (here *spikes*, *Catcher*, *evilBlock* and *evilSaw*) are recorded at the end of each game (**SetData** and **SendData** method in **PlayerController**). More generally, all recorded data are specified in **PlayerData**.

It is necessary that these data use get and set are accessors, so that we can use the C# Reflection! The data are saved in *Assets/Resources/data.csv*. The reading and writing of this file is done by the **DataFileManager** script. If **PlayerData** is changed, the **FindData** and **NextId** methods may need to be changed as well (however it is important that the player name is still the first attribute). All other functions should work through Reflexion : it allows to dynamically get the name, the type and the value of all the attributes from an existing object.

Data processing and level adaptation : At the beginning of each game, the **AdaptProbabilities** method will be called. This method receives information about the previous game of the player by calling the **FindData** method from **DataFileManager**. **FindData** method returns a dictionary that contains for each platform with a trap (here *platform5*, *platform6*, *platform7* and *platform10*), the ratio

$$\frac{\text{number of deaths on platform } p}{\text{number of platform } p \text{ crossed}} \tag{5}$$

If this ratio is equal to 1, the player is dead on all platforms p. If this ratio is equal to 0, the player never died on the platform p.

If this ratio is greater than 0.5, the difficulty will be adapted :

$$diffWeight = Weight_p * ratio_p * adjustment; \tag{6}$$

This difference in weight will be removed for the weight of the platform p, and added to the weight of a bonus platform (with the tag *coinPlatform*) making work the same part of the body. This lowers the number of platform p difficult for the player, while maintaining the balance between different parts of the body to work.

If the player dies on the first trap of his first game, the ratio for this trap will be equal to 1. The difference in weight is equal to the initial weight of the platform with the trap. The *adjustment* variable prevents a platform from completely disappearing from the game, by setting the adjustment variable different from 1 (for example 0.9).

5.9 Emotion Mode

As part of mental health promotion, an option in the game was added to integrate the emotion of the player. So this mode has to be used with a camera.

The goal is to modify the level content according to the emotion of the player. However, the objective is not to force the player to change his facial expression (which can be painful and frustrating from the point of view of the player), but rather to add feedback and fun if the player wished.

5.9.1 Emotion Detection

The emotion of the player is detected from his facial expression. To do this, we use the API **Affectiva** (https://knowledge.affectiva.com/v4.0.0/docs/getting-started-with-the-emotion-sdk-for-unity). This module identifies key points on a face (corners of the eyes, corners of the lips ...), then uses Machine learning algorithms to classify the facial expression (*Facial Action Coding System* (FACS) and *Action Units* (AUs).

The API retrieves the video stream from the computer's webcam, then return a float for each desired emotion. The greater the number, the more the emotion is significant.

5.9.2 Link the emotion to the creation of the level

The **PlayerEmotionController.cs** script save the weights of the desired emotions in a dictionary (these weights are returned by the affectiva API). In this script, the **EmotionDetected** method returns the name of the most significant emotion. If no emotion is detected, this method returns a particular string.

Then, the **PlatformGenerationEmotionController.cs** script takes care of the generation of level when the player plays in emotion mode. This script inherits from **PlatformGenerationController.cs**. The **SpawnPlatform** method is overrided to only reveal elements of the scenery when an emotion is detected. This script also contains additional methods for managing visual and auditory feedback related to emotions.

5.9.3 Basic Version

This version corresponds to the basic game (like the random version of body mode). However, the trail does not appear behind the avatar.

5.9.4 First intelligent Version

Save the color of the player : When an emotion is detected, the elements of the decoration must change color. Their new color must be related to emotion. However, there is no universal association between color and emotion. Therefore, it is important to allow the player to choose the color he wants to tie to a particular emotion.

The data are saved in *Assets/Resources/colors.csv*. The reading and writing of this file is done by the **ColorFileManager** script. This script manipulates objects of type **PlayerColor**. These objects contain the name of the player, and these colors associated with each emotion in RGB format.

Changes during the game : When a facial expression related to an emotion is detected, additional feedbacks appear in the game.

First, a colorful trail follows the player. This is an indication for the player, because it is only active when an emotion is detected. Then, new elements of decoration appear (cf Figure 27). These elements and the avatar change color depending on the most significant emotion (cf Figure 28). In our game, we decided to look at four emotions: anger, surprise, joy and sadness.

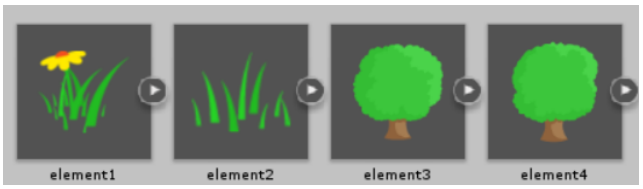


Figure 27: Additional decorations

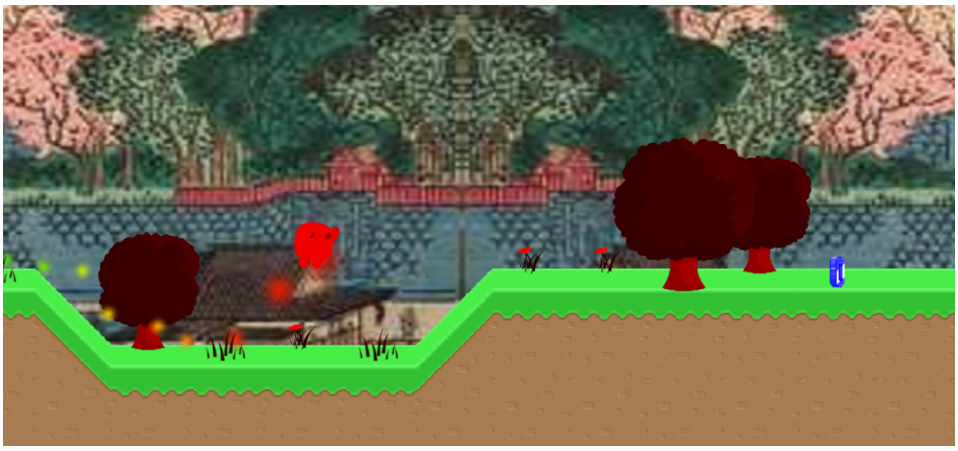


Figure 28: Color change (anger)

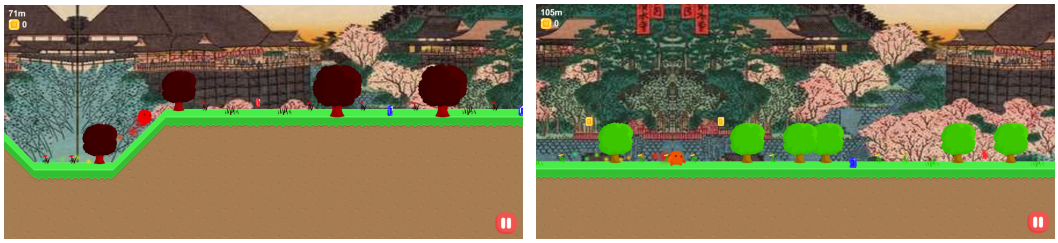
Finally, different music are played depending on the emotion detected.

Additional elements of the scenery should be added to the *Prefabs* folder and dragged into the *Elements* variable of **PlatformGeneration**. In our project, these elements will only appear on platform 3.

5.9.5 Second intelligent Version

In this version, the generation of the level change when a particular emotion is detected. The game will adapt to the emotions of the player as follows (cf Figure 29) :

- If the player is angry or sad, the probability that traps appear increases.
- If the player is happy or surprised, the probability of coins appearing increases.



(a) Anger (b) Joy

Figure 29: Emotions

Two other feedback are available, but not used :

- Indications can appear to indicate to him where to make the movements (for example, when jumping at the right moment in front of a trap).
- The speed of the avatar can be reduced.

5.9.6 Additional Version

This section is about an older version of the intelligent mode of emotion mode. This version was intended to save the emotions of the player during 4 test levels.

Save the emotional reactions of the player : The **PlatformGenerationPreExpController.cs** script takes care of the generation of test levels regarding the emotional reactions of the player. This script inherits from **PlatformGenerationController.cs**. The **Start** and **Update** methods are overrided to change some parameters at the beginning of the game or during the game.

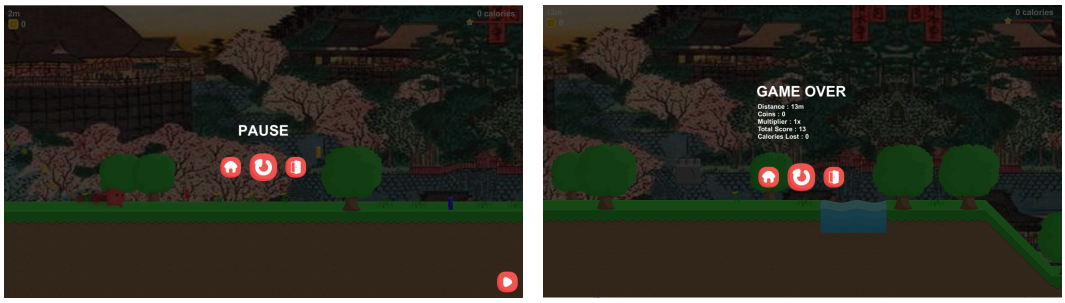
Four levels of testing can be used to observe the emotional reactions of the player :

- **CoinLevel** : The level contains only coin platforms.
- **TrapLevel** : The level contains only trap platforms.
- **SpeedLevel** : The speed of the character changes randomly after a fixed duration.
- **IndicationLevel** : There are indications in front of each platform to indicate to the player when to generate the keyboard input.

During these tests, the values returned by *Affectiva* are recorded at a regular time interval (**SaveEmotion** method). The data will be saved in *Assets/Resources/PreExp* folder, and the name of the file is composed of the name of the player and the number of the test.

5.10 In Game Menus

In all mode games, there are two kind of menu, one for the pause screen managed by the **PauseController.cs** script (cf. (a) Figure 30), and one for the death screen when the player die managed by the **DeathController.cs** script (cf. (b) Figure 30).



(a) Pause Screen (b) Death Screen

Figure 30: Level Menus

5.10.1 Pause Screen

When the pause button is pushed in game mode, the **PauseMenu** is activated. It pauses the run sound loop and pauses the game using the *Time.timeScale* Unity API. When *timeScale* is set to 0, the game is paused and when textittimeScale is set to 1, the time is passing as fast as realtime.

The pause screen has a title text, a background image and 4 buttons:

Button	Size (WxH)	Function
	64x64	Go back to main menu
	85x85	Restart the game
	64x64	Exit the application
	64x64	Resume the game

5.10.2 Death Screen

When the player box collider enter in collision with the box collider of an object having a kill tag, the **DeathMenu** is activated. Unlike the pause screen, the death screen only set the **Player** gameObject to false. As for the pause menu, the death screen has a title text, a background image and 3 buttons with same function has describe above.

The death screen also display :

- The current score (distance);
- The number of coins collected;
- The multiplier value;
- The total score (`Total Score = Distance + (Coins * Multiplier)`);
- In body mode only, calories lost.

If the total score exceed the current high score, a text will be displayed to notice the player. Likewise, a text will be displayed if the player reach his calories goal (only for body mode).

6 Contacts

For any use problem or any question, please contact one of the person bellow :

- Camille El-Habr : camille.el.habr@gmail.com
- Xavier Garcia : xav.tps@gmail.com

7 Annexes

Fitbit Experiment First Results

- **Exp1** : No mouvement
- **Exp2** : Move the arm on which fitbit is not attached
- **Exp3** : Move the arm on which fitbit is attached.
- **Exp4** : Move the arm on which fitbit is not attached

	Exp1		**Exp2**		**Exp3**		**Exp4**	
Time (min)	HB	Calories	HB	Calories	HB	Calories	HB	Calories
0	78	0	74	0	79	0	79	0
1	83	2	100	1	100	3	98	1
2	79	3	98	2	133	8	95	3
3	81	4	99	4	132	15		
4	80	6	100	5	163	23		
5	81	7	103	7	163	32		
6	83	8	100	14	143	43		
7	85	10	101	21	142	53		
8	79	11	102	22				
9	81	12	104	23				
10	75	14	103	25				

Table 4: Fitbit Experiment

During the first 5 minutes, the experiments 1 and 2 have similar results. The larger increase in calories consumption at the 6th and 7th minutes in Exp1 could not be explained.

In Exp3, the number of calories consumed indicated by Fitbit is higher than in Exp2. Thus, we can assume that accelerometers are present inside the watch, and help to calculate the number of calories lost.

However, Fitbit also indicates in Exp3 a significant increase in heart rate. This increase is not visible during the first two minutes of experiment 4. Therefore, we can think that this increase in heart rate is not due to fatigue, but to a design defect of the watch.