Universitat de Girona
Escola Politècnica Superior

Treball Final de Grau/Carrera

Realitzat i defensat a

Budapest University of Technology and Economics (nom universitat) de Hongria (país)

Estudi: Grau en Enginyeria Informàtica Pla 2010

Títol: DEVELOPMENT OF A SURVIVAL GAME IN A UNITY3D ENVIRONMENT

Document: Memòria del Treball Final de Grau

Alumne: Aleix Canet Vidal

Director/Tutor: Dr. Milan Magdics
Departament: Informàtica i Matemàtica Aplicada

Convocatòria (mes/any): 06/2017

Budapest University of Technology and Economics

Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Aleix Canet Vidal

DEVELOPMENT OF A SURVIVAL GAME IN A UNITY3D ENVIRONMENT

SUPERVISOR

Dr. Milan Magdics

BUDAPEST, 2017

Table of contents

STUDENT DECLARATION

I, **Aleix Canet Vidal**, the undersigned, hereby declare that the present BSc thesis work has been prepared by myself and without any unauthorized help or assistance. Only the specified sources (references, tools, etc.) were used. All parts taken from other sources word by word, or after rephrasing but with identical meaning, were unambiguously identified with explicit reference to the sources utilized.

I authorize the Faculty of Electrical Engineering and Informatics of the Budapest University of Technology and Economics to publish the principal data of the thesis work (author's name, title, abstracts in English and in a second language, year of preparation, supervisor's name, etc.) in a searchable, public, electronic and online database and to publish the full text of the thesis work on the internal network of the university (this may include access by authenticated outside users). I declare that the submitted hardcopy of the thesis work and its electronic version are identical.

Full text of thesis works classified upon the decision of the Dean will be published after a period of three years.

Budapest, 12 June 2017

...…….
Aleix Canet Vidal

Summary

From the beginning of time, man has wanted to take advantage of his free time with activities that allow him to disconnect from his day-to-day life. From the appearance of the first video game back in 1958, called Pong - a tennis simulator, to the almost life-like video games we have nowadays, video games have evolved at a vertiginous speed.

With the evolution of technologies and with the increase in the number of players, new video game categories have appeared, such as simulators, shooters, open-world video games and survivals. This last category is the one that I want to focus on, as it is the one I have chosen.

Thus, the main purpose of this BSc thesis will be the development of a survival game using Unity [1] as the game engine.

The videogame will consist of a player whose main goal will be to survive on an island, using all the available resources he can find, such as water and food. The character will have attributes, like health, hunger, and thirst. The game will be over when the character dies.

Using the game engine Unity, alongside Blender [18] for animating items and changing some meshes from objects, and using some third party assets I have been able to create the island, populate it with trees, bushes and other items that the character will need in order to survive.

Moreover, the character will be able to create his own items using the materials he has previously found on the island, either chopping trees, mining stones to get metals or gathering from bushes. And to make it more realistic, a day-night cycle has been added.

Resumen

Desde el principio de los tiempos, el ser humano ha querido aprovechar su tiempo libre con actividades que le permitan desconectar del día a día. Y desde la aparición del primer videojuego en 1958, como era el Pong, un simulador de tenis mesa, hasta ahora que tenemos videojuegos casi tan reales como la realidad, los videojuegos han ido evolucionando a una velocidad tremenda.

Con la evolución de las tecnologías, han ido apareciendo distintas categorías de videojuegos, desde simuladores de futbol, a shooters, pasando por juegos de mundo abierto, y también survivals, la categoría que he elegido.

Dicho esto, el principal objetivo de esta tesis será el desarrollo de un juego de supervivencia utilizando el motor de juego, Unity.

El videojuego consiste en un personaje solo en una isla cuyo único objetivo es sobrevivir, valiéndose de los recursos que encuentre en la isla, así como comida o agua. El personaje tendrá varios atributos, así como vitalidad, hambre y sed. Si el personaje se queda sin vida, el juego se termina.

Utilizando el motor de juego Unity para el juego, Blender para las animaciones y algunas modificaciones en objetos y juntamente con otros assets de terceros, he sido capaz de crear la isla, poblarla de árboles, arbustos y otros objetos que el personaje necesitará para sobrevivir.

Además, el personaje podrá crear sus propios ítems utilizando los distintos materiales que hay en la isla, ya sea recolectando madera como picando piedra o recogiendo de los arbustos. También se ha incorporado en el juego un ciclo día-noche.

1. Introduction

The main task of this BSc thesis is to develop a three dimensional video game using the game engine Unity.

I started playing video games at the age of seven, with my first console, the GameBoy Color. Since then, I have gone from the GameBoy Color to the PS4, playing a wide range of video games, from simulators to shooters, alongside RPG or sandboxes.

In addition, I have always been interested in developing my own video game. So, when the opportunity was given to me, I could not resist accepting and started working on it and learning all about the development of video games.

Having played Rust and Far Cry 3, I wanted to mix these two video games into one, and tried to copy the essence of an open world as it is Far Cry (like the island where my game happens) with the only, crucial aim to survive Rust.

The chosen software for the development has been Unity, because it is very intuitive software with the integrated "scripting" part. Using all tabs that Unity offers, you can do a lot of things, from designing a 2D UI to creating all the items that will be needed in the videogame. Then, with your created items, you can add behaviours to these items by writing scripts using the programming language C# and assign them afterwards.

This document is organized with the main aspects of the game and how they have been developed. Every section will be thoroughly explained alongside the whole design process that I have followed and how they interact with the rest of the components of the video game.

1.1 Unity: Introduction

Unity3D is a game engine developed by Unity Technologies and is used to develop video games for almost all platforms. With an emphasis on portability, the engine is able to run in lots of platforms targeting the following APIs:

- Direct3D and Vulkan on Windows and XBOX 360: Used to render three-dimensional graphic applications where performance is essential. It uses hardware acceleration of the 3D rendering pipeline

- OpenGL on Mac, Linux and Windows: Abstract API for drawing 2D and 3D graphics. It has been designed to be implemented not in the software but mostly, if not entirely, in the hardware.

- OpenGL ES on Android and iOS: Subset of OpenGL API designed for embedded systems. Cross-language and multi-platform API, that is why it is the most deployed 3D graphics API.

- Proprietary APIs on video game consoles.

Unity allows developers to use the texture compression and resolution settings for each platform that the game engine supports. Another thing to highlight is that x the support Unity provides for bump mapping, reflection mapping, dynamic shadows, render-to-texture and full-screen-post-processing effects.

With shaders with multiples variants and declarative fallback specifications, it allows Unity to detect the best variant for the video hardware in use, and if none of them are compatible, Unity tries to find one that could sacrifice some features for performance.

1.2 Unity: concepts

The Unity workflow is built around the GameObjects. A GameObject is any object placed in the scene. And to give behaviours to these GameObjects Unity has what is called a Component. Unity has different types of Components, from components to control the physics of the GameObject (Rigidbodies and Colliders) to scripts that the developer has written and then added to the GameObject.

With this system, as the GameObject and the Component are not tied to each other, Components are allowed to be used in more than one GameObject. This results in a better decoupling and an increment in scalability.

To show this workflow in a practical example, we can have a look at how the Item "system" works.

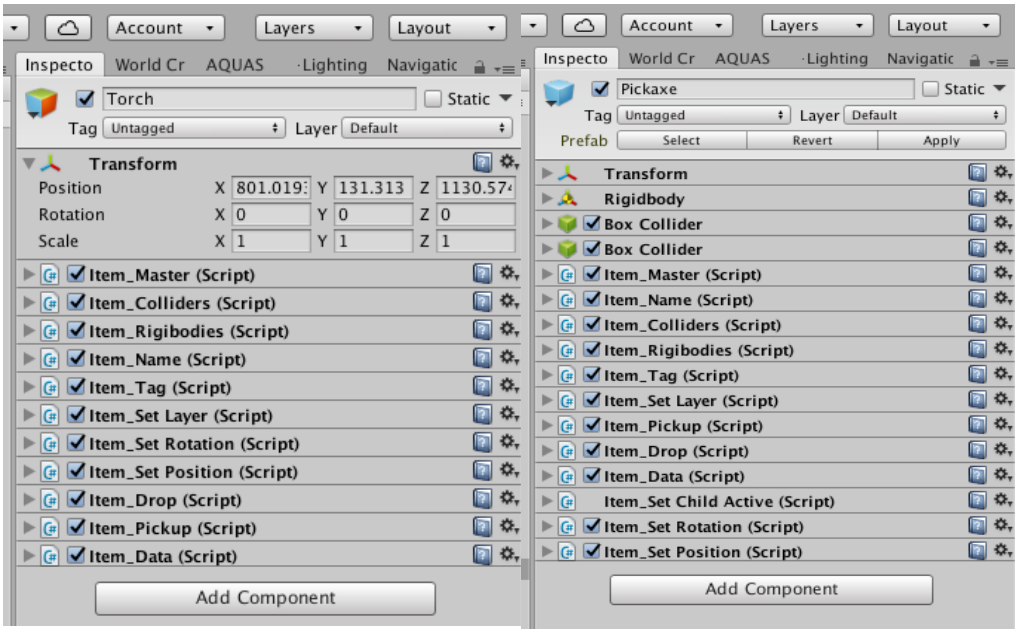

Figure 1.1 Torch component *Figure 1.2 Pickaxe component*

We can see that we have a lot of components that are included in both objects. In this example, all these script components change the properties of the item, such as the name, the position, the layer, the tag, etc…

All this gameobjects and components must be placed somewhere in the game, and this is where the scenes take part. A scene contains the objects of your game and you can use it to create from a main menu to a level with obstacles and all that you need. This workflow is useful to design and build your game in pieces and make sure that everything works as expected.

2. Explanation and analysis of the task

The video game consists of a character trapped on an island where its main aim is to survive. The character can explore the island and find resources that will allow him to survive. However, the player will not be alone on the island, as there are some enemies around the island protecting key items that the player can make use of.

As I was inspired by Rust, I wanted to recreate some features, such as being able to pick up items into the inventory, and use them. Another thing I wanted to add was weapons because I considered it would be useful to have weapons when I decided to include enemies.

Another thing I wanted my game to have was a realistic environment, starting from the island to the elements placed in it. With the player being able to interact with terrain items, his opportunities to survive would increase as well as the playability of the game. From actions like animations when trees fall to the ground because the player has chopped it to effects when a bullet hits a surface has been added to the game, and using image effects (either using the ones that come with Unity or the ones that I have created) I could increase the realism of the game.

The game ends when the player dies. It happens when the player loses all his health, by starving, thirsting, falling from height, or being killed by the enemies.

2.1 Communication between scripts

This part is the most important one, as it is where all happen. Having a good and structured communication make things much easier.

When I started the project, I had the basic idea on how scripts interacted to each other. At the beginning, when I wanted to interact with scripts, such as calling the script that will apply damage to a tree from the player, I only had to call that script to perform that action. As the project were growing, more complex actions were required. Picking up an item means that I have to make fifteen calls to different components and all of them have to be executed at the same time. So, as I considered that this would be problematic later on, I had to look for an easier and more efficient solution to handle this. I decided to implement an event system, so all the actions needed could be called with only one call.

The chosen pattern here is the Event Aggregator pattern, which fits perfectly in my case because I can subscribe multiple objects to a single event, and this event can be from different objects.

Below we can see an example of how it is:

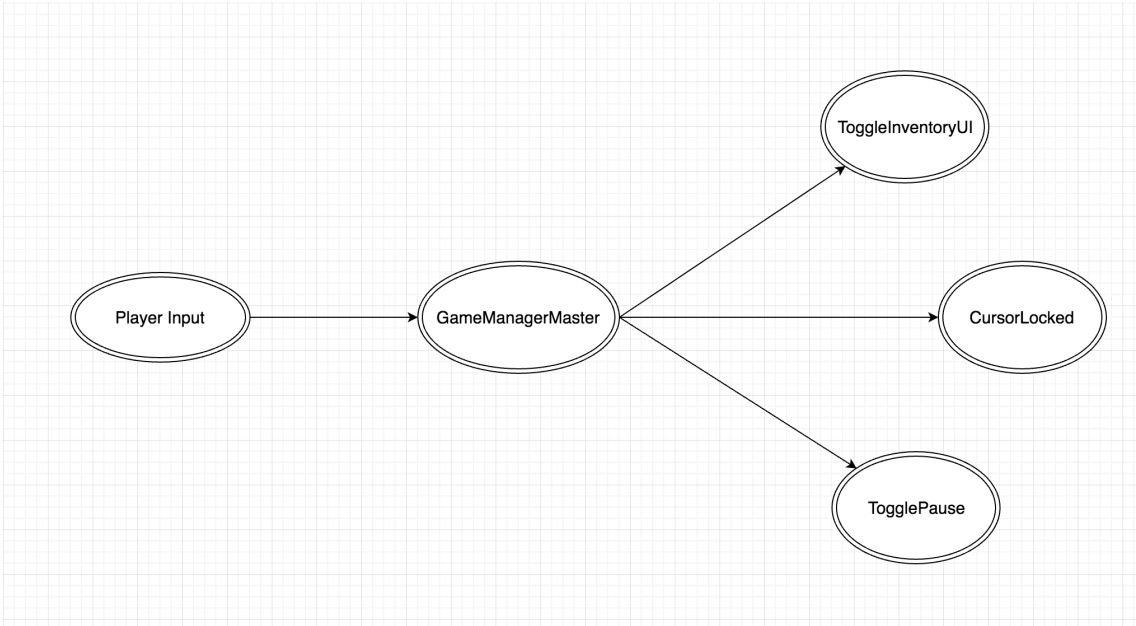

Figure 2.1 Components needed to toggle the inventory UI

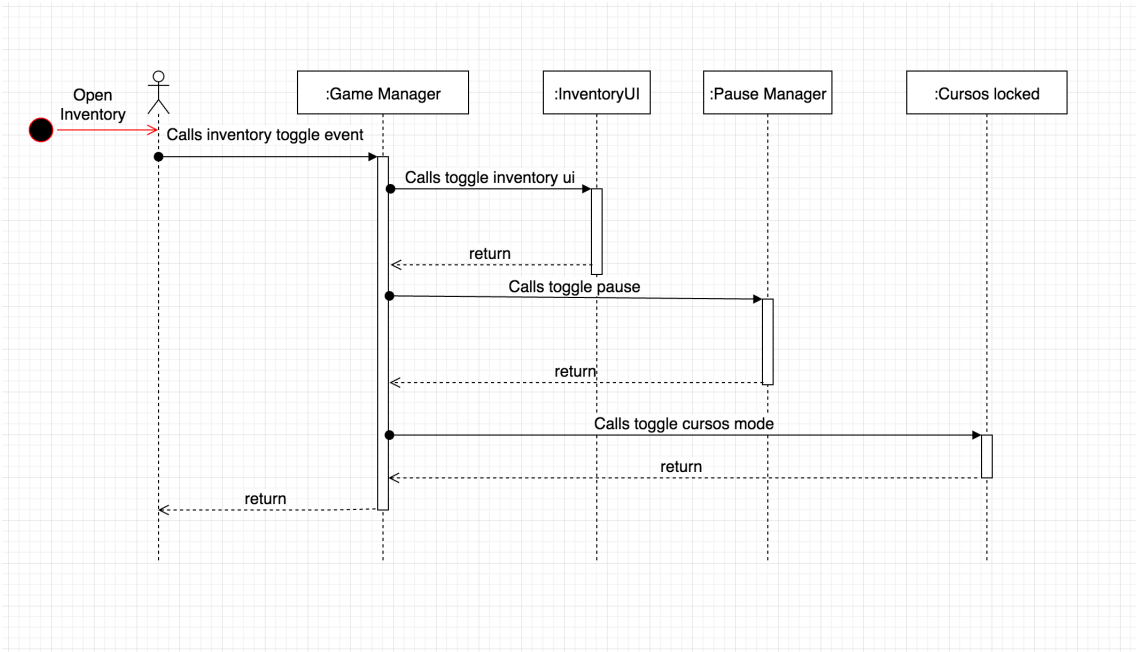

Figure 2.2 Sequence diagram from the process of toggling the inventory UI

To clarify how the event system works, there is a simple example used to toggle the inventory.

First of all, we have a kind of master script, that contains the definition of all the events that can take place.

```
public delegate void GameManagerEventHandler();

public event GameManagerEventHandler MenuToggleEvent;
public event GameManagerEventHandler InventoryToggleEvent;
public event GameManagerEventHandler RestartLevelEvent;
public event GameManagerEventHandler GoToMenuSceneEvent;
public event GameManagerEventHandler RestartIslandEvent;
public event GameManagerEventHandler GameOverEvent;
public event GameManagerEventHandler ResumeGameEvent;
```

These events handle all possible situation that can happen between the UI and the game. As we have said before, we want to toggle the inventory UI, so we would have to pause the game and make it appear to the screen. If we have a look at the TogglePause script:

```
void OnEnable()
{
    SetInitialReference();
    gameManagerMaster.MenuToggleEvent += TogglePause;
    gameManagerMaster.ResumeGameEvent += TogglePause;
    gameManagerMaster.GameOverEvent += TogglePause;
}

void OnDisable()
{
    gameManagerMaster.MenuToggleEvent -= TogglePause;
    gameManagerMaster.ResumeGameEvent -= TogglePause;
    gameManagerMaster.GameOverEvent -= TogglePause;
}
```

we can see that OnEnable() has an instruction that calls the event InventoryToggleEvent and that subscribes the void TogglePause (not only a void can be subscribed, but functions as well, what it is important here is that both headers must match) to the event. Now that we have the game paused, we want to display the UI, so if we go to ToggleInventory script we will see the following:

```
void ToggleInventoryUI()
{
    if (inventoryUI != null)
    {
        inventoryUI.SetActive(!inventoryUI.activeSelf);
        gameManagerMaster.isInventoryUIOn = !
          gameManagerMaster.isInventoryUIOn;
        gameManagerMaster.CallEventInventoryUIToggle();
        if (inventoryUI.activeSelf)
        {

weaponToolbar.transform.SetParent(inventoryWeaponZone.transform);
            weaponToolbar.transform.localPosition = Vector2.zero;
            weaponToolbar.transform.localScale = new
Vector3(0.835073f,0.835073f,0.835073f);
            UIWeaponZone.SetActive(false);
        }
        else
        {
            UIWeaponZone.SetActive(true);
            weaponToolbar.transform.SetParent(UIWeaponZone.transform);
            weaponToolbar.transform.localScale = Vector2.one;
            weaponToolbar.transform.localPosition = Vector2.zero;

        }
    }
}
```

in the highlighted instruction, we see that we make a call for the event InventoryToggleEvent, so as we have subscribed the action of pausing the game previously to the event, the subscribed action will be executed and, in consequence, the game will be paused.

As shown above, if we subscribe more functions to an event, all of them will be executed and we will not have to call to each one every time we want to perform an action.

2.2 Environment

When I thought about where would my game to happen, many ideas came to my mind, but the main one was to make it happen on an island. The island would give me two advantages: the first one is the fact that an island is a piece of land in the middle of the sea, so I would not have to limit the map. The second one is where I had to think a lot: what would the island contain in order to make it look like a real island, or a good approximation at least.

With the help of the third-party plugin World Creator [15] I have been able to create the shape of the island and populate the island with the Unity Terrain tools afterwards

The final result of the island can be seen here:

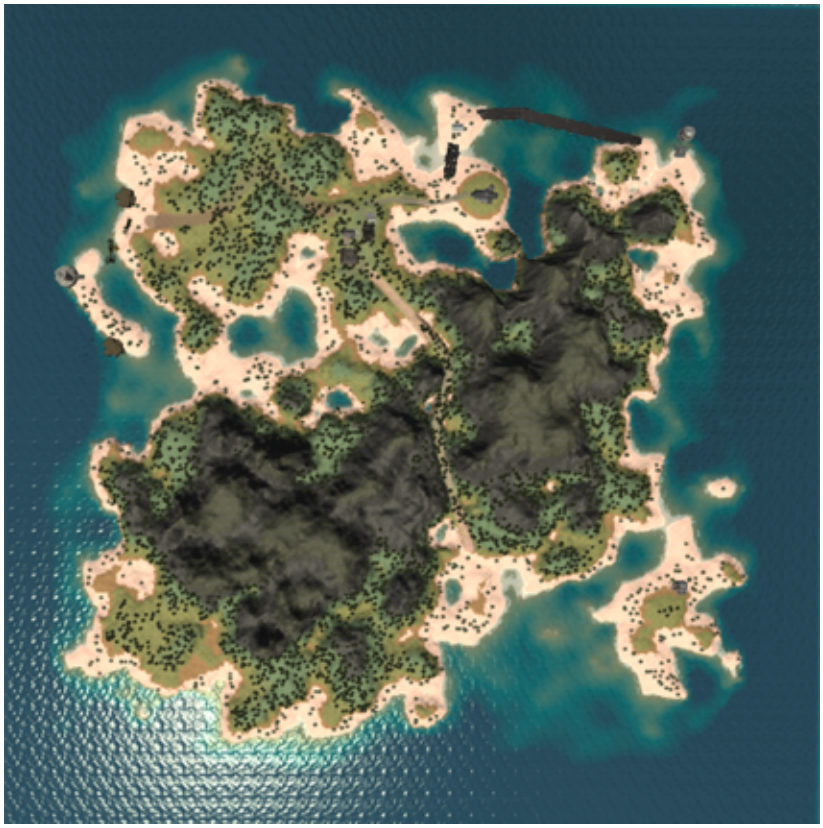

Figure 2.3 Shape of the island

2.3 Player interaction

One of the most important parts of a survival game is the interaction with the environment. And that is something that I have put a lot of effort to make it as easy as possible.

As I knew beforehand what I wanted my character to interact with, I decided to make it as simple as possible. The chosen technique to detect the interactive surrounding items is the SphereCast, as I want to check if there is an item close enough to the player to be picked up.

To get a simple but powerful and scalable solution, I have done two functions that are both called inside the void Update (). This is the first one:

```
void CastRayForDetectingItems()
{
   if (Physics.SphereCast(rayTransformPivot.position, detectRadius, rayTransformPivot.forward,out
hit,detectRange,layerToDetect))
   {
      interactiveItem = hit.transform;
      itemInRange = true;
   }
   else
   {
      itemInRange = false;
      interactiveItem = null;
   }
}
```

In this function, I check if the sphere detects a hit, and that this hit has a layer that I want to be detected. With this feature, I can filter which layers I want to detect and which ones I do not want. Now, if we check the second function:

```
void CheckForItemInteractionAttempt()
{
   if (Input.GetKeyDown(buttonPickup) && Time.timeScale > 0 && interactiveItem != null &&
      !interactiveItem.root.CompareTag(GameManager_References._playerTag))
   {
      if (interactiveItem.GetComponent<Item_Master>() !=  null)
      {
         interactiveItem.GetComponent<Item_Master>().CallEventPickupAction(rayTransformPivot);
      }
      else if (interactiveItem.GetComponent<Bush>() != null)
      {

interactiveItem.transform.SendMessage("GetElementsFromBush",SendMessageOptions.DontRequireRec
eiver);
      }
   }
}
```

In this function, I check if I have pressed the button to interact with the item, plus if the game is not paused (otherwise we could not interact with the object), plus if the character has detected something and if the character does not already have the item equipped.

Once explained how it works, we will see below with what and how the character can interact with environment.

- The first one is by interacting with the trees. There are about two hundred trees on the island from which the player can get fruit and they can be chopped to collect wood from them.

Figure 2.4 The player in front of the tree interacting with it

- The second one is by gathering fruit from bushes and mushrooms. Small pieces of fruit or mushrooms can be collected and will be stored in the inventory.

Figure 2.5 The player in front of a bush interacting with it

- The third one is the most important one. It allows the user to pick up items and use them afterwards. In the below image, we can see how a small text appears displaying the item's name and the key the user have to press to pick up the item.

Figure 2.6 The player in front of an item interacting with it

2.4 Inventory system

This part has been by far the most complex one in the whole game. You can imagine the player inventory as a switch, where you have inputs, outputs apart from processing things. Following with the metaphor, the inputs would be the picked-up items, the outputs would be the dropped items and the processed ones would be the items that come from the crafting system.

2.4.1 Picked-up items

Picking up items allows the character to interact and use them. When an item is picked, the inventory checks if the character already has the item, and depending on if the item is stackable or not, it increases the amount of the item or it creates a new item.

What the inventory checks is if the item is a tool or a weapon, as it will change the item's parent to the player, in order to use it. If not, it just performs the picks up call and destroys the item gameobject.

2.4.1.1 Dropped items

Dropping items is a difficult process. The inventory has to check if it has more than one item. If this is the case, it will instantiate a new gameobject and release. If it is the last item that the inventory has, it will check if it is an item or a weapon. If it is a weapon

it will just set its parent to null, but if it is an item, it will just instantiate a new object and will delete the slot in the inventory.

2.4.1.2 Crafted items

With resources, the character can craft its own weapons and tools.

All the character has to have is the resources needed. The resources used during the crafting process will be automatically deducted from the inventory and the item will be added.

Figure 2.7 Items that can be crafted

Figure 2.8 Item enabled to be crafted

Figure 2.9 Item crafted and added to the inventory

2.5 Effects

To make a game look realistic, it has to have a lot of effects and details. In my game, there is two types of effects: the first one is the wind and the second one is the particle systems.

2.5.1 Wind

As the island includes trees, I thought it would be a good idea and add reality to the game. Using the built-in component wind-zone in Unity, I have been able to place the wind on the terrain and recreate a pretty realistic wind effect.

To achieve the desired effect, I played around with the different parameters of the component until I got a realistic result. The final configuration is as follows:

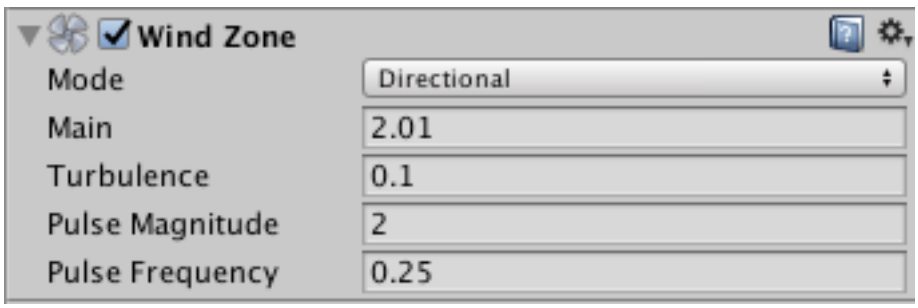

Figure 2.10 Wind zone component with the values

The chosen mode is directional as I saw it was better than the spherical mode, because the effect produced was similar to the one produced by a helicopter.

Wind can be seen mainly in trees, in bushes (there are just a few bushes in the island) and grass. In trees, wind makes the leaves to move in different directions, while in bushes and grass it moves all the mesh to the wind direction.

Figure 2.11 Tree affected by the wind

2.5.2 Particle effects

I have used particle systems to simulate fire, the effect it produces when a bullet hits the ground and when a bullet hits a human, the effect that a bullet produces when it is fired and a kind of poisonous area. As they are completely different, first we will have a closer look at the fire particle system.

2.5.2.1 Fire effect

To simulate fire, four particle effects are required. The first one is the flame itself, the second one is the white effect that can be seen at the base of the flame, the third one are those little sparks that can be seen at the top of the flame and the last one is smoke.

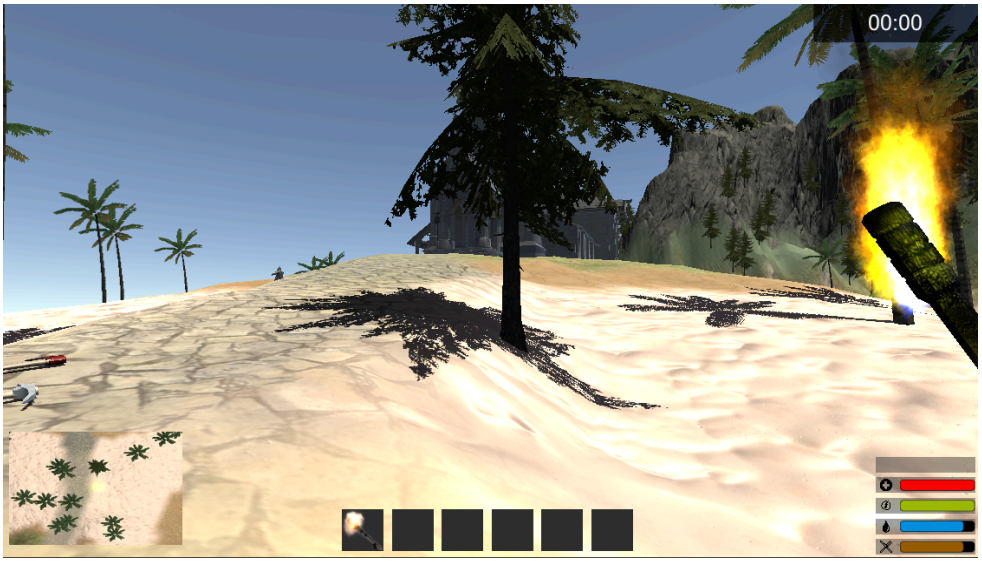

Figure 2.12 The player holding a torch

In the below screenshots, there are the most relevant items from the particle system that I have used to recreate the fire effect:

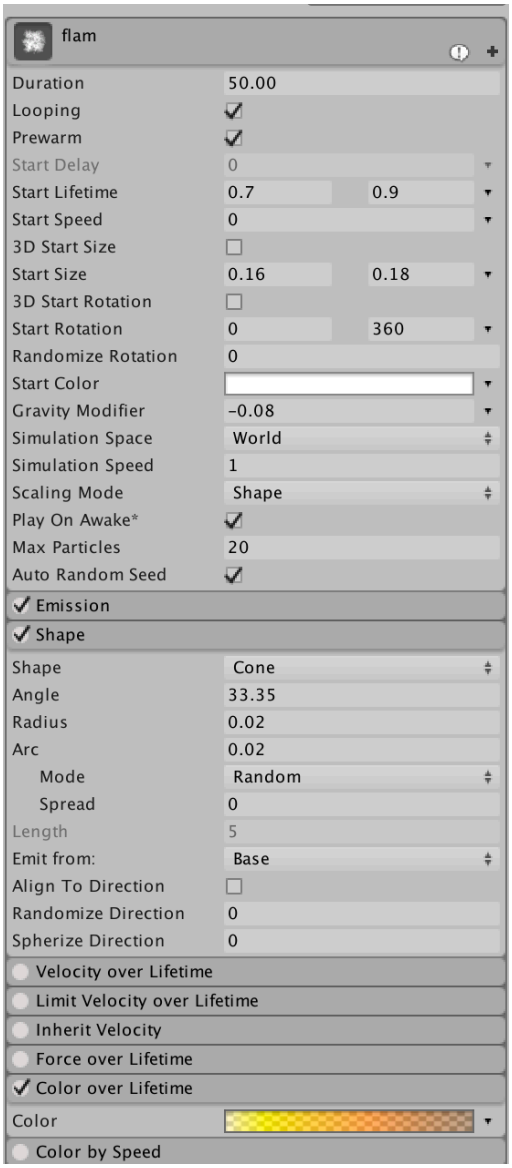

Figure 2.13 Flame particle system

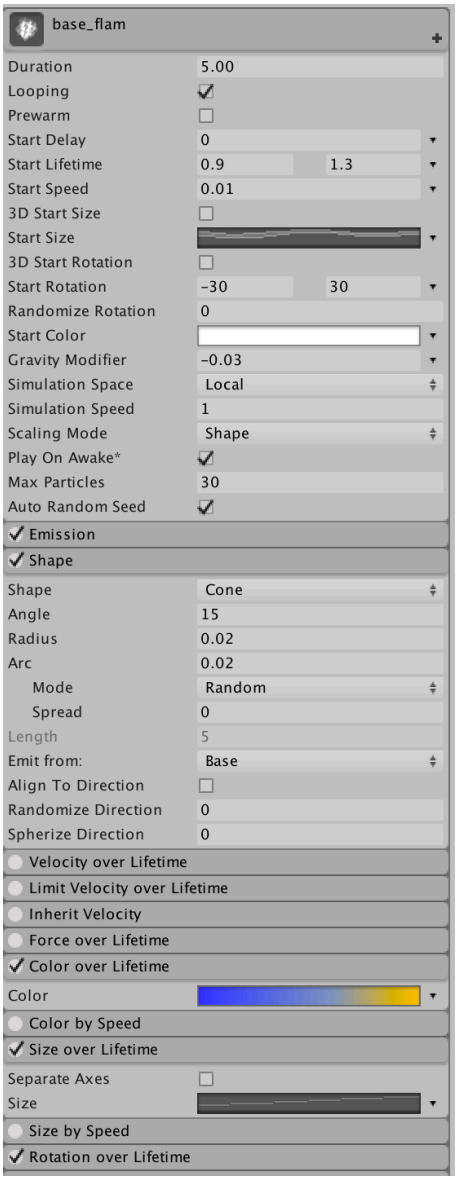

Figure 2.14 Base of the flame particle system

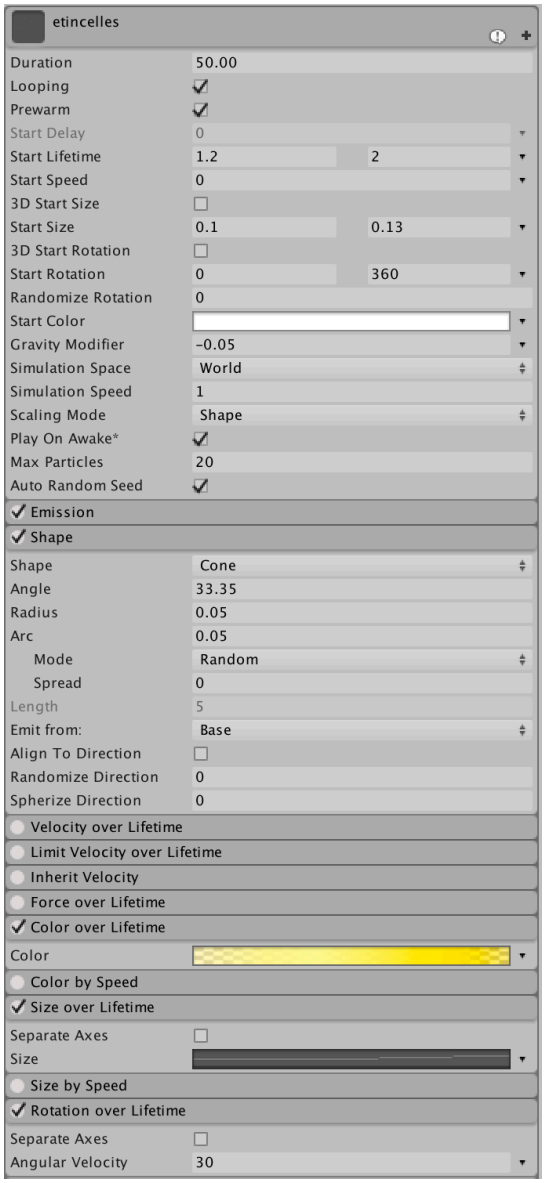

Figure 2.15 Sparks particle system

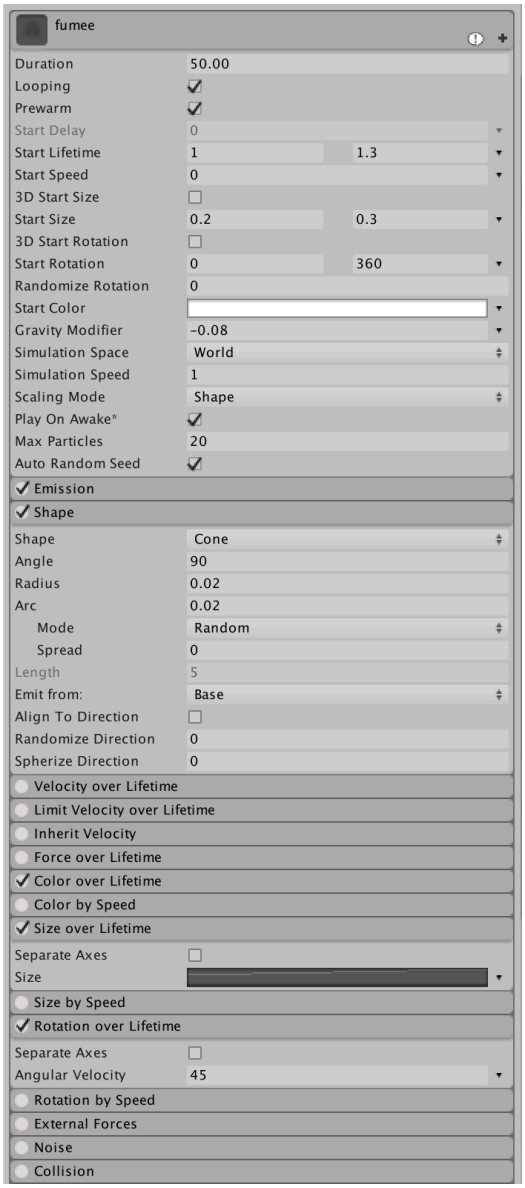

Figure 2.16 Smoke particle system

All these four effects have the same conical shape and all are looping. Combined all together they generate the effect of fire.

2.5.2.2 Hit effects

The hit effects are almost the same, the only difference between them is the aspect they have, as the default one is brown and is bigger while the other one is red and smaller (to simulate the effect of blood).

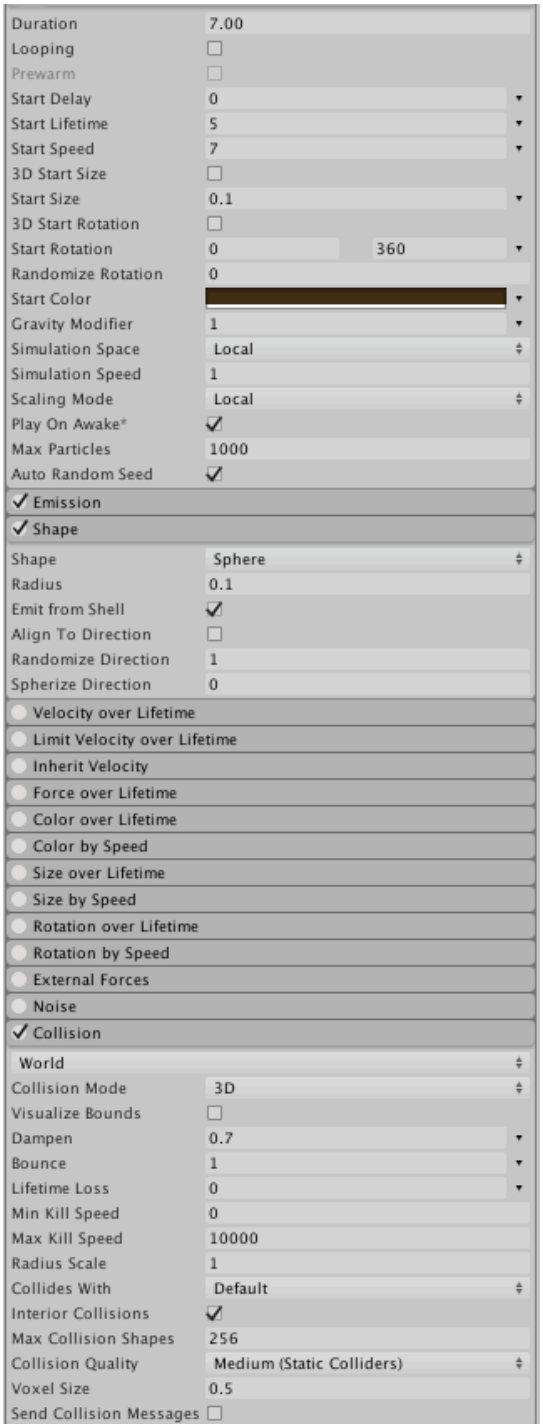

Figure 2.17 Hit effect particle system

2.5.2.3 Muzzle flash

The third effect is the muzzle flash the gun produces when it is fired. This effect adds realism to shooting a rifle or a gun. The most important thing about this effect is the colour of the effect, as it should be white at the beginning, orange/yellow at the middle and white again at the end. Some defined parameters are the following:

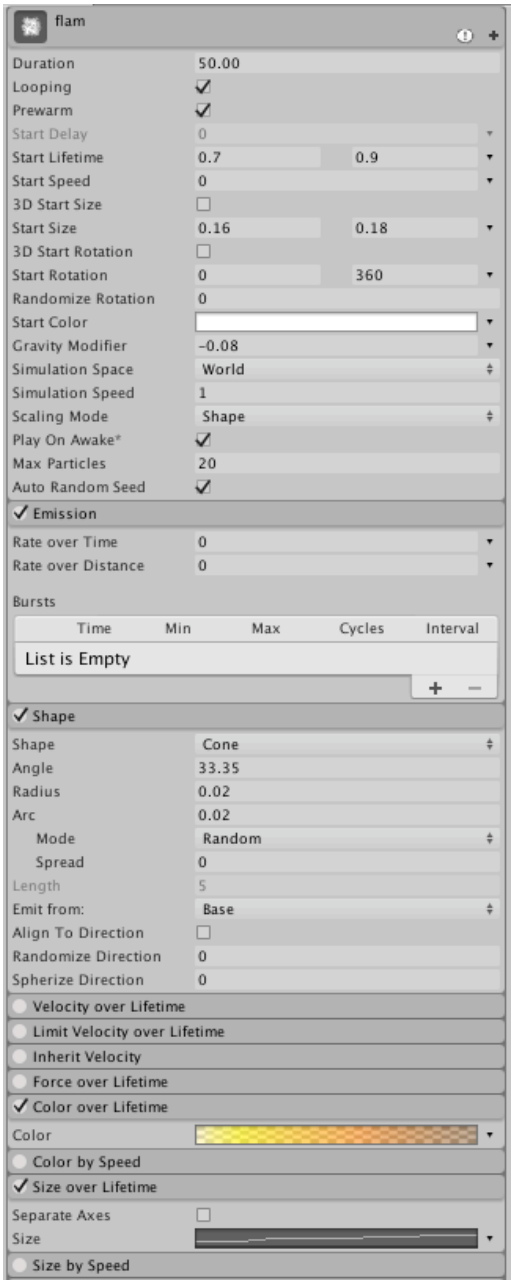

Figure 2.18 Muzzle flash particle system

2.5.2.4 Poisonous area

I thought that a good way to make the game a little bit more difficult would be to include some kind of poisonous zone that is hazardous to the player's health should they walk through it.

Figure 2.19 Poisonous area

After looking for a way to achieve this effect, I decided to create another particle system and enable the collision module from the particle system. With it enabled, the particle system can send messages to the objects that collide with its particles, thereby minimising the player's health.

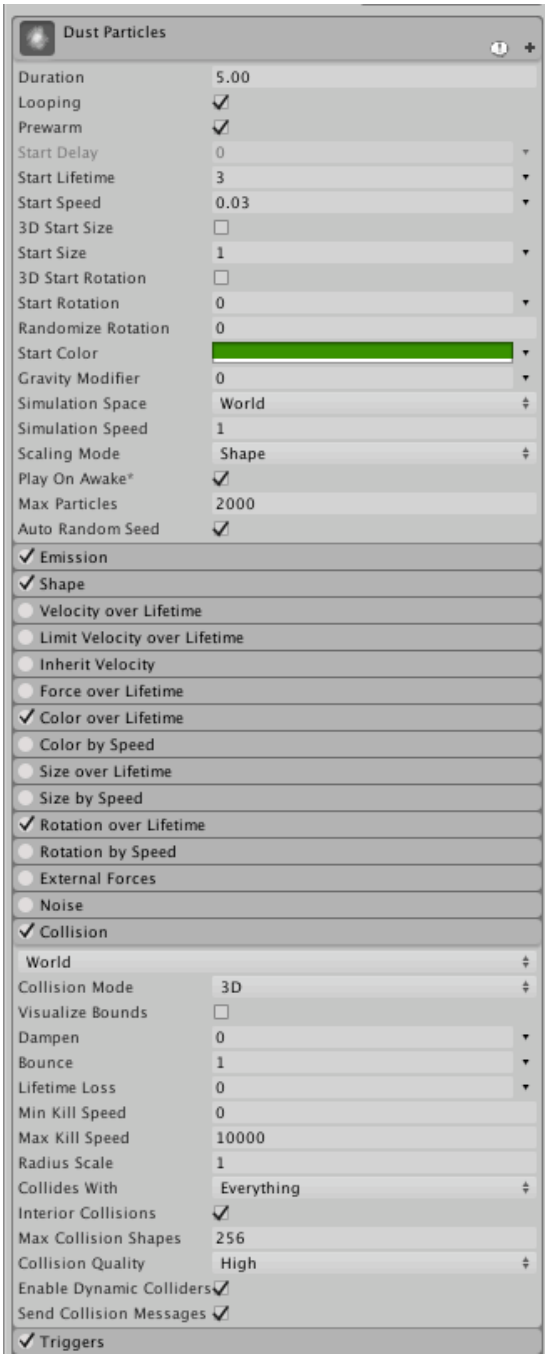

Figure 2.20 Poisonous area particle system

2.6 Animations

The animations in a game are what make the game smooth and appear realistic. To create realistic smooth animations requires a lot of time. As my game was a first-person game, I considered that animations were not one of the most important things but it would be nice to have some.

Animations like swinging an axe, a pick-axe were my main goal. With that in mind, I started looking for ways to do the animations. Unity has a kind of animation engine to animate gameobjects. Unfortunately, as I wanted to create more complex animations, I had to look to Blender and MakeHuman [20] to achieve these animations.

The first step was to create a human mesh from MakeHuman to Blender. Once in Blender, I just deleted the whole body leaving just the right arm. Then, I just had to import the tool/weapon I wanted to animate and make the animation. Finally, the only thing I had to do was to import the animations to Unity.

Some screenshots of the process:

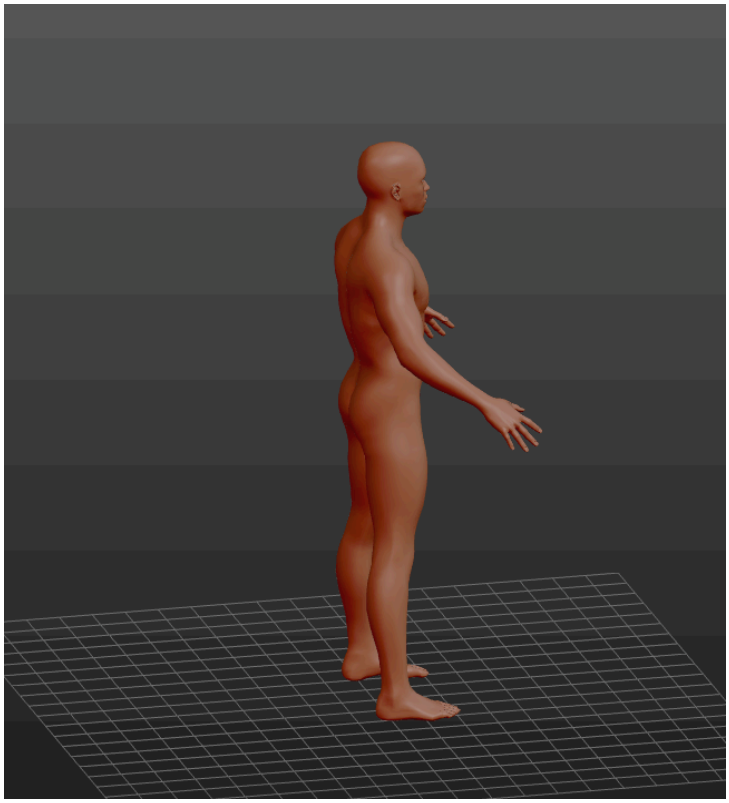

Figure 2.21 MakeHuman human mesh

Figure 2.22 Mesh from MakeHuman cleaned, with bones and with the pickaxe being animated

Figure 2.23 In-game screenshot with the pickaxe playing the swinging animation

Items that have these kinds of animations are: axe, pickaxe, torch and rock. Then, with Unity, I created basic animations for items like guns and rifles, and some other UI elements.

Because just surviving can be quite boring, I decided to create enemies that would try to kill the player. From the beginning, my goal was to include animals in the game, but as I could not find animated animal models, I had to turn down the idea. Instead, I imported a model from mixamo.com with some animations: idle, run, fire and hit.

The animator works as follow:

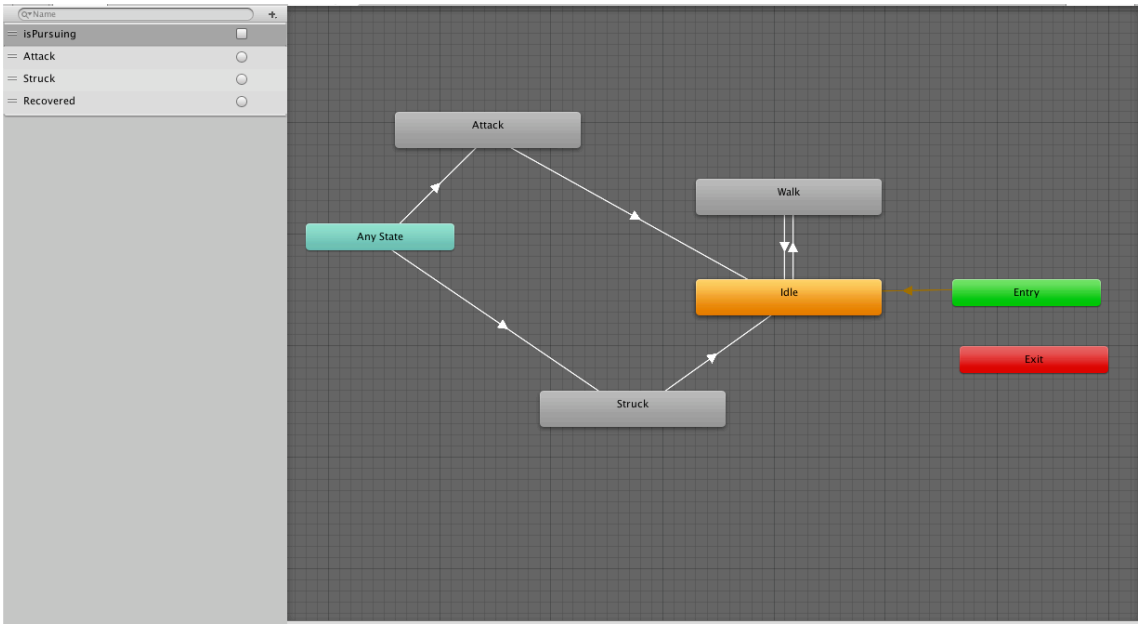

Figure 2.24 Enemy animator

With these four states, the enemy is able to idle, walk/run, to follow you, strike when it gets hit and attack you.

2.7 AI

As stated above, I wanted to add some difficulty to the game.

While the animation part was trivial, the scripting part was a little bit more complicated. As I had read that the best way to create a simple AI was to implement a finite state machine, I decided it would be a good idea to have my own one as well. Moreover, I wanted to create a kind of gang, or at least to let the player have allies. Finally, I just implemented the enemy part, so enemies can have allies that will be informed if one of them sees something suspicious.

The state machine consists of eight states:

- Alert:

  This state checks if the enemy sees our player, and if this is the case, it informs its nearby allies and they go together to the last seen position.
- Flee:

  This state checks if the enemy's health is under a particular parameter, in which case, the enemy would escape.
- Follow:

  This state controls the enemy movement when following someone, either the player or an enemy
- Investigate harm:

  This state controls the enemy when it is investigating the last seen position of the player.
- Melee attack:

  This state performs the melee attack on the player.
- Patrol:

  This state performs a patrol operation, the enemy walks in no particular direction.
- Pursue:

  This state makes the enemy pursue you to attack.
- Range attack:

  This state performs the range attack on the player.

The finite state machine diagram is shown below:

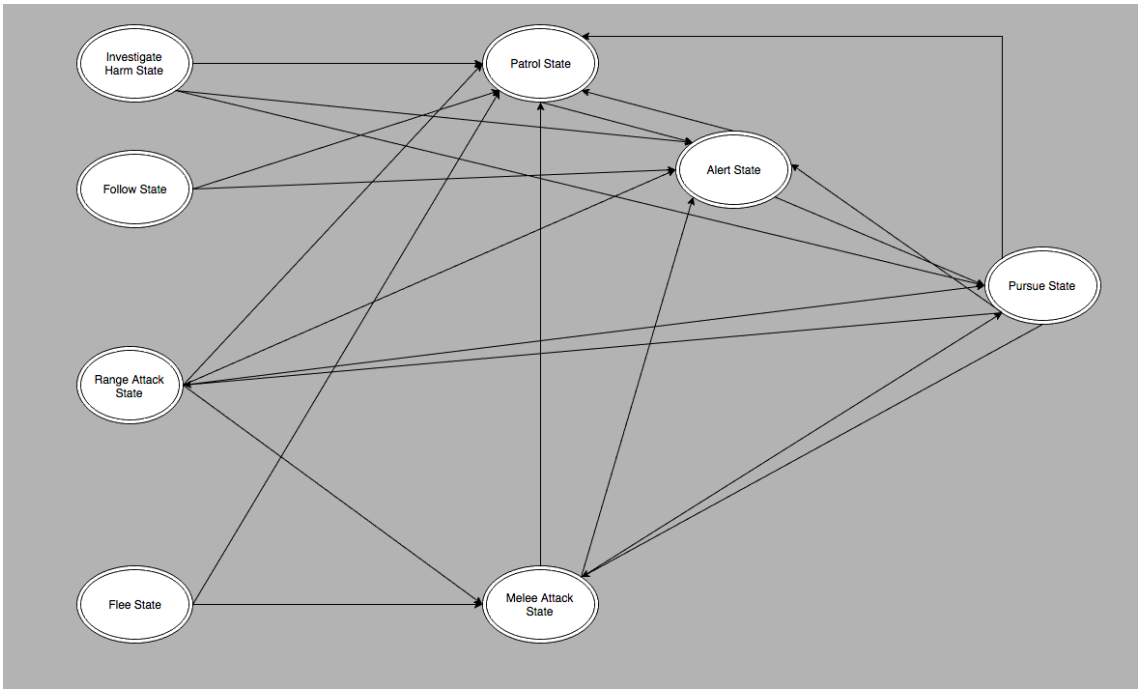

Figure 2.25 AI State Machine

In order to move the NPC (Non-player Character), I used the built-in NavMeshAgent from Unity. The NavMeshAgent is a component that every NPC must have that controls its movement through the map, allowing it to go to a position, stop moving, resume moving, consulting the remaining distance to cover, etc.

Using this powerful component, I built the AI above it, so every time the UI had to move to somewhere, I just had to update its destination so the NPC would go there.

One example of this behaviour can be seen in the below code snippets.

```
void MoveTo(Vector3 targetPosition)
{
    if (Vector3.Distance(npc.transform.position, targetPosition) >
npc.myNavMeshAgent.stoppingDistance + 1)
    {
        npc.myNavMeshAgent.SetDestination(targetPosition);
        KeepWalking();
    }
}

void KeepWalking()
{
    npc.myNavMeshAgent.isStopped = false;
    npc.npcMaster.CallEventNpcWalkAnim();

}
```

```
void Patrol()
{
    if (npc.myFollowTarget != null)
    {
        npc.currentState = npc.followState;
    }
    if (!npc.myNavMeshAgent.enabled)
        return;

    if (npc.waypoints.Length > 0)
    {
        MoveTo(npc.waypoints[nextWayPoint].position);
        if (HaveIReachedDestination())
        {
            nextWayPoint = (nextWayPoint + 1) % npc.waypoints.Length;
        }
    }
    else
    {
        if (HaveIReachedDestination())
        {
            StopWalking();
            if (RandomWanderTarget(npc.transform.position,
npc.sightLayer, out npc.wanderTarget))
            {
                MoveTo(npc.wanderTarget);
            }
        }
    }
}
```

We can see here is the patrol action, and two needed actions to perform this task. In MoveTo, it checks if the distance between the NPC and the destination point is greater than the stopping distance (NavMeshAgent parameter), it sets a new destination point and starts moving the character as well as plays the walking animation.

With patrol state, the NPC can walk to random destination as well, and once it gets there, it calculates a new point and if it is possible, it sets its new destination point there.

Another feature of the NavMeshAgent is the possibility to disable zones and make them non-walkable. I have used this feature to make the sea non-walkable so NPCs would not be able to go to water zones, because if not, NPCs could walk to water zones.

Below, we can see the walkable mesh in cyan and the non-walkable mesh not painted, so it means it is unreachable.

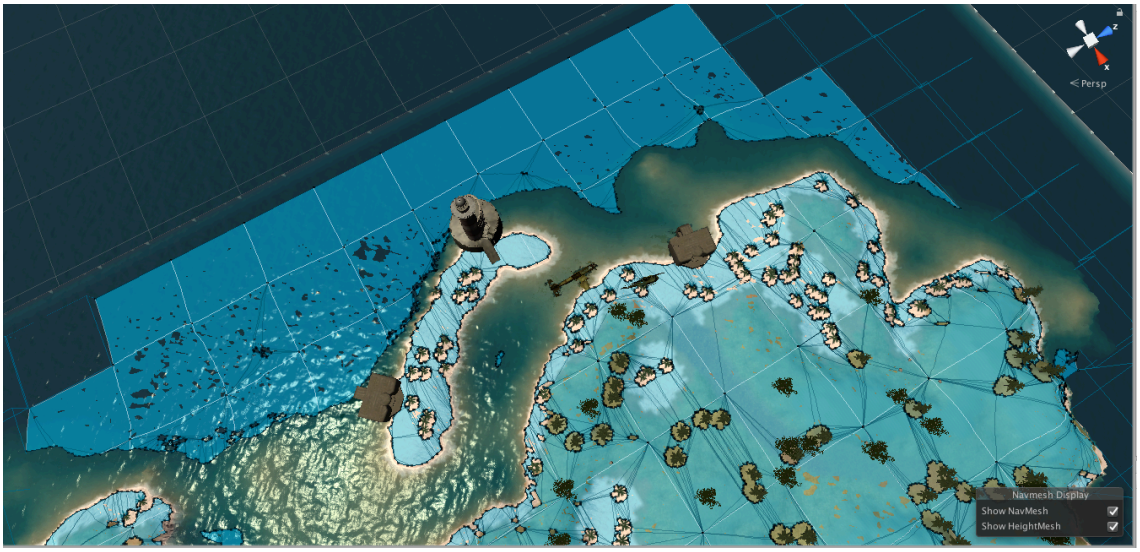

Figure 2.26 Example of walkable and non-walkable zone

2.8 Sea and underwater effect

As the game occurs on an island, the terrain is surrounded by water. I thought it would be a good thing to recreate an underwater world as well. Unfortunately, I did not have time to that so I had to find an easy way around it.

The chosen solution was a light version of a third-party asset called AQUAS [16]. What AQUAS does is to add a blue fog effect to the camera as well as adding a bubble effect and a kind of wet effect when the player emerges from the water.

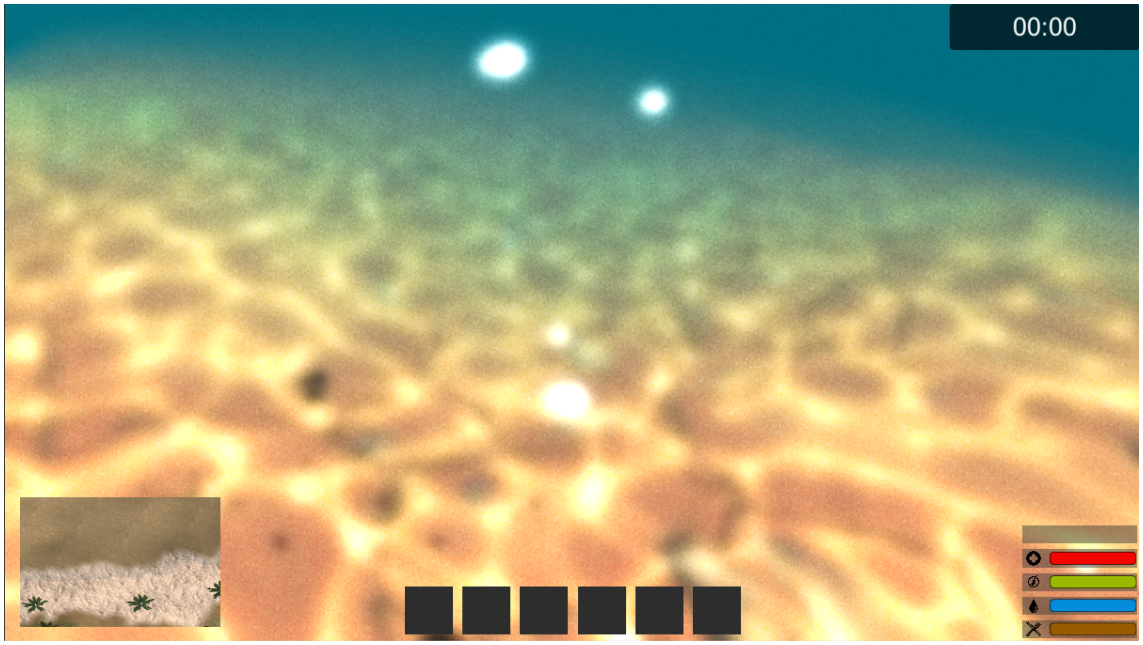

Figure 2.27 Underwater effect when the player is underwater

Figure 2.28 Effect when the player emerges from water

2.9 Post-processing effect

Apart from the standard shaders and effects, the developers can write their own shaders to add realism to the game. After looking for information on how to write shaders, I decided to write a simple shader that would mix the texture from the camera with a bloody canvas to simulate and notify the player when the character gets hurt.

In the picture below we can see the effect of the shader when applied to the camera.

Figure 2.29 Hurt canvas applied when the character gets hurt

To achieve this effect, I had to write a vertex and fragment shader by interpolating the image that comes from the camera with the one that comes from the applied texture. As the texture is transparent in the middle, I can recreate this effect by controlling the alpha channel of the texture.

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    fixed4 blood = tex2D(_BloodTex, i.uv);
    fixed4 result = lerp(col,blood,blood.a*_Multiplier);
    return result;
}
```

Figure 2.30 Code snippet from the custom hurt shader

3. Testing and performance

The workflow I followed to test the game was a kind of Unit-testing. Every time I finished the implementation of some component, I spent more or less 2 hours testing it separately from the game, trying and assuring myself that everything was working and that, once the integration with the whole system was done, it would not crash but continue working. And this point is important because I spent a lot of time refactoring the inventory because once I was about to integrate it with the system, it crashed or it did not work as I was expecting.

Another important number of hours went into checking that all the inner features of the inventory worked as expected. The inner management of the inventory was the most difficult part to test given that I had to check the content of every item inside the array one by one and that the inventory was updating them well.

Once I had done that, and at the very end of the development, I made a little set to test that everything was working well, animations were running properly and smoothly.

During these tests, I found little mistakes such as colliders that were not working as expected, **Rigidbodies** with either too low or too high weight, so this part was about adjusting all these parameters to make the game run smoothly.

Another important factor, if not the most important, is the performance of the game. During the development, I tried to keep the FPS (frames per second) as high as possible, so having an optimized game was on my list, despite knowing I would not be sure if I could achieve that.

To check the performance of the game, I activated the stats tab and used the profiler window to have more information about the performance.

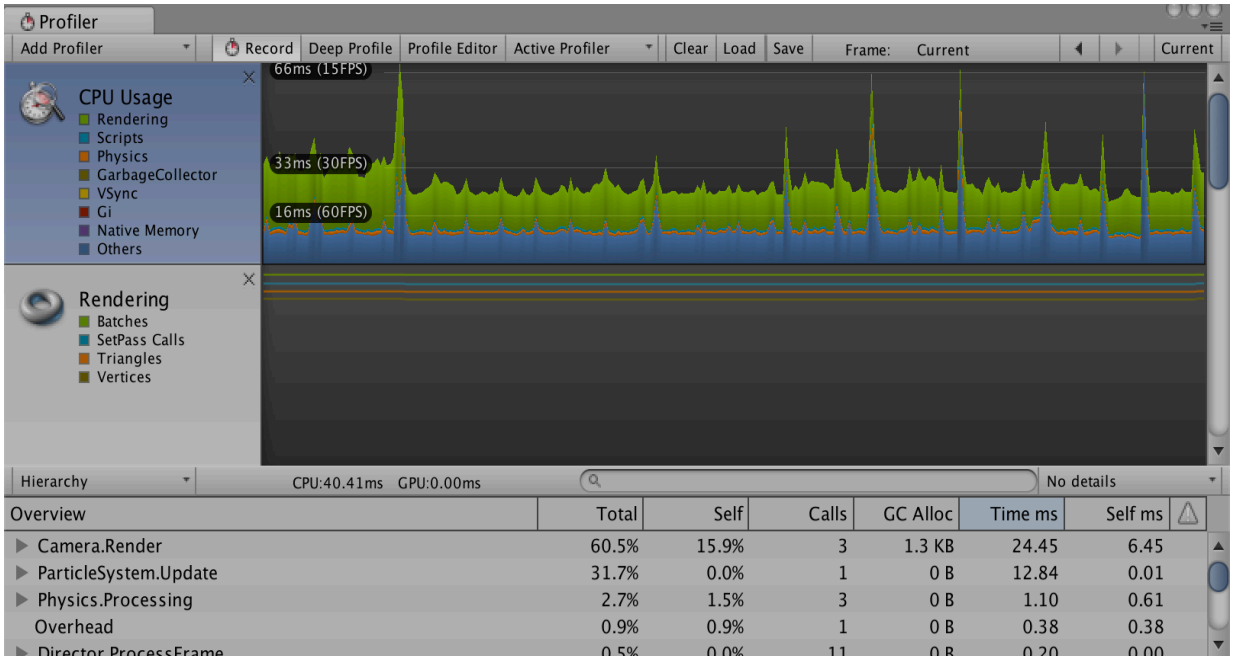

Figure 3.1 Profiler showing the performance

As we can see, sometimes there is a pronounced decrease of fps. After trying to figure out why this was happening, I found that it happened only when the character started to move in a zone with a lot of meshes, like in the forest, where there are a lot of trees.

After testing a little bit more, I discovered that placing a lot of objects in the scene decreased the FPS so I had to find a workaround. What I did was, with a World Creator [15] tool, I placed the objects I wanted to be interactive on the map, and once done that, I removed half of them randomly to adjust the performance. As interactive items are trees, rocks, bushes and mushrooms, I did it with all of them.

4. Challenges

The first thing I did before starting to develop the video game was to think about what I wanted to achieve. With all these goals in mind, I made a list to discard those that did not seem feasible.

After this process, what I had was:

- Player Inventory
- Crafting system
- Chopping and mining system
- Realistic island terrain with water
- Hunting system
- Animations
- Enemies

The first problem I faced was my lack of skills in both Unity and video game development. To solve this problem, I spent 1 week watching tutorials and playing with Unity and trying things in order to get used to it. Once I did that, it was time to start the video game.

Secondly, I had to choose between designing a first-person game or a third-person game. My first option was to do a third-person game with a first-person camera allowing the player to change the view. Finally, after not finding a proper model and animations, I decided to discard this option to use a first-person approach.

Since the very beginning, I was obsessed with the scalability of the video game. I wanted to improve the game and add new features once I had more knowledge and I didn't want to change everything that I had done. And to do that, I would have had to implement a system that allows scripts to interact with each other without increasing the level of complexity.

With that in mind, I implemented an event system, so if I had to call on five scripts at the same time, with the event system I was able to subscribe all these five scripts to an event, so that when the event was fire, all these scripts would be called on.

Another challenge I had to face was the communication and the exchange of information between the inventory and the real world.

This was important because I had to take into account all the different situations that could happen and how these would be handled.

The first approach was a basic one, with only one item per slot. Once done that, the next step was to implement the drag and drop mechanism, so it would increase the usability and the ease of the inventory. After that, a host of problems appeared.

I had to rethink how the inventory would work several times until finally found an easy one. With the crafting system, the same thing happened. I had to reduce the crafting system because it would have meant spending more time on the inventory part.

To develop a good game, a lot of features need to be done, and animations are no exception.

One of my goals was that the character could use tools and weapons, like axes or guns, so I wanted the create the animations for these actions.

My main problem here was the lack of free assets to use, as I had found some animations but for the whole character, not only the arms, and not the ones that I needed.

So, what I did to solve that problem was use the MakeHuman software, I exported a human model and with Blender, I was able to cut the mesh to leave just one arm.

Afterwards, all I had to do was to animate the arm depending on which tool the character could have.

5. Improvements

Like in every game, there is always room for improvement. And this thesis is no exception.

5.1 Player improvements

In the player field, there are a lot of things to improve.

Firstly, we have the change from a first-person view to a third-person view, along with all that involves, such as changing some scripts to adapt the behaviour to a third-person view. Another thing that would need to be done is the animation, as it would require not only animating an arm but the whole character for actions like swinging an axe or holding a gun.

This work would have exponentially increased the time I would have had to dedicate to learn and use it, and I saw it was out of my scope.

Secondly, the inventory could be improved by adding some features and changing the existing ones.

One change that could be done is in the crafting system, adding a timer when crafting could be an option, or adding new ways to craft items, such as a cooking system to cook meat. Another one could be the addition of more item management features, such as allowing the player to split the item in case there is more than one.

5.2 Scripting improvements

This is maybe the field where most improvements can and should be made. As it is my first game and I lack experience, I have made a lot of mistakes coding the scripts.

Despite having implemented the event system, more changes could be applied. One of them could be the implementation of a pooling system for items that will be used many times, such as bullets or particle systems.

Another improvement that could be done is the pickup system, as I'm not happy at all with the final result. It works as expected, but the logic behind it, could be approached differently.

The main problem here is that to pick up an item, two calls to the event system need to be made. The first one is to add the item to the inventory and the second one to perform all tasks that are performed when the item is picked up.

5.3 Visual improvements

A video game always requires of visual effects to look realistic, and this game definitely requires some of them.

Although I have used the ones that come with Unity, I have just been able to write two shaders, one that applies a blood texture to the image that comes from the camera so that it looks like the character is hurt. The other one is to simulate the effect when the character is underwater. This effect adds a global fog and deforms the image a bit.

I would have liked to have written more shaders, because with the few I have tried and seen, you can achieve great visual effects if you can master them.

5.4 Terrain improvements

The terrain is where it all happens. And in a survival game, the terrain is of great importance.

In this case, the terrain could be improved by making all trees to be choppables. I really wanted to include that feature in my game, but I saw it was impossible due to the amount of work that it would require. Terrain trees are not placed as GameObject, but as submeshes inside the terrain mesh. So, in order to access to the tree, the terrain mesh has to be modified as well as there is a large list of steps to be checked before accessing it. Another improvement could be the addition of more details, applying more detailed textures or making the terrain look more realistic. Populating the terrain with realistic vegetation objects, or with buildings could help.

5.5 Expanding the game

As this is an open world game, there are endless opportunities to expand the game.

One way to expand the game could be by adding new items for the player to use, like weapons, for example javelins. Another one could be the addition of activities like fishing, hunting or farming: These activities, if added to the game, would boost the interactivity with the world. Additionally, with the addition of these activities, it would

add new ways for the player to gather resources and, as a consequence, increase the chance of survival.

6. Conclusions

When I started thinking about what to do for my BSc thesis, I decided that it should be something that I would really love because otherwise, I would not give it my best. After thinking about it for days, I decided that I wanted to develop a video game, because as a very active video game player, I was interested in discovering how they are made, not only the appearance and what a player can see, but inside them. And I think that my BSc thesis has thrown some light on it.

In the early stage of the game, I did not know what I would be able to do, so lots of features and ideas came to my mind. And after finishing it, I can say that I have almost met my expectations. From the long list that I had at the beginning, I have only not been able to implement a few of them, in part because my lack of experience and in part because I did not have enough time to develop them despite having the knowledge and experience to put it into practice.

During the development, and as a consequence of the constant learning process, I spent a lot of time refactoring the code, as I wanted to do each part as simple as possible in case I wanted to expand them later. Looking back, I see that it was a good decision, as it has saved me lots of problems afterwards.

At the end of development, while checking that everything was working as expected, I realized that I would have liked to implement more features to make the game more realistic. And as a result of this process, when I saw how complicated it is to develop a video game, because despite the fact that as a player you expect incredible graphics, great playability, with an immersive plot that gives you goose bumps, you do not pay attention to those details that really makes the difference between a good game, and a really great game.

With this thesis, apart from enjoying the process a lot, I have learned a set of new skills that will most definitely help me, but I would like to highlight only two: First, I have improved my programming skills a lot, as I have learnt new methodologies and useful tricks. Second, I have seen the importance of setting realistic goals and proper time management, as it will help you to check if you are heading in the right direction or not.

Developing this game has opened me up to a new world of possibilities, and it is a world that I like. I have learned how to work with Unity and its environment, I have been able to experience the whole creative process, from having the idea until its final stage, of having the game completed.

7. References

1. Unity - Learn - Modules [Internet]. Unity. [accessed 23 February 2017]. Available from: https://unity3d.com/es/learn/tutorials

2. Unity (game engine) [Internet]. En.wikipedia.org. [accessed 15 May 2017]. Available from: https://en.wikipedia.org/wiki/Unity_(game_engine)

3. Unity 5 Advanced FPS Tutorials - GTGD S3 How To Make A Game - YouTube [Internet]. YouTube. [accessed 11 April 2017]. Available from: https://www.youtube.com/playlist?list=PLwyZdDTyvucwjwqucleVQB7U12H2JPvg5

4. Mechanim & Mixamo in Unity 5: The Basic Basics! [Internet]. YouTube. [accessed 13 March 2017]. Available from: https://www.youtube.com/watch?v=BEIaakl9vJE&t=1132s

5. Creating a GTA-like Minimap with Unity 5 [Internet]. YouTube. [accessed 6 May 2017]. Available from: https://www.youtube.com/watch?v=ktTegOCaffw&t=2906s

6. [Unity 5] Tutorial: Day And Night Cycle - sun rotation, intensity and the time of the day [Internet]. YouTube. [accessed 26 March 2017]. Available from: https://www.youtube.com/watch?v=y6TCQfFB2xg

7. Survival Game Tutorials in Unity 5 - YouTube [Internet]. YouTube. [accessed 10 February 2017]. Available from: https://www.youtube.com/playlist?list=PLb34wPRpZdVevgA60IIJ5pQi4RCR264FU

8. Unity Tutorial: A Practical Intro to Shaders - Part 1 [Internet]. YouTube. [accessed 11 May 2017]. Available from: https://www.youtube.com/watch?v=C0uJ4sZelio

9. Unite 2015 - Writing Shaders [Internet]. YouTube. [accessed 6 May 2017]. Available from: https://www.youtube.com/watch?v=epixwRw80MM

10. (Unity 5) Let's Make Rust! - YouTube [Internet]. YouTube. 2017 [accessed 14 March 2017]. Available from: https://www.youtube.com/playlist?list=PLXzbvVfPPpBmWBfdNT7JqmZevisTL4iyR

11. Technologies U. Unity - Manual: Unity User Manual (5.6) [Internet]. Docs.unity3d.com. [accessed 17 February 2017]. Available from: https://docs.unity3d.com/Manual/index.html

12. Unite Europe 2016 - A Crash Course to Writing Custom Unity Shaders! [Internet]. YouTube. [accessed 6 May 2017]. Available from: https://www.youtube.com/watch?v=3penhrrKCYg

13. Mixamo: Quality 3D character animation in minutes [Internet]. Mixamo.com. [accessed 21 February 2017]. Available from: https://www.mixamo.com/

14. 3D Models for Free - TF3DM | Free3D.com [Internet]. Free3d.com. [accessed 17 March 2017]. Available from: https://free3d.com/

15. World Creator. BiteTheBytes UG; www.world-creator.com/.

16. AQUAS. DOGMATIC; https://www.assetstore.unity3d.com/#!/content/52103.

17. Asset Store [Internet]. Assetstore.unity3d.com. [accessed 6 February 2017]. Available from: https://www.assetstore.unity3d.com/

18. Blender Fracture Modifier Build. Blender; http://df-vfx.de/fracturemodifier/.

19. Technologies U. Unity - Manual: State Machine Basics [Internet]. Docs.unity3d.com. 2017 [accessed 10 April 2017]. Available from: https://docs.unity3d.com/Manual/StateMachineBasics.html

20. MakeHuman. MakeHuman team; www.makehuman.org.

21. Event Aggregator [Internet]. Martin Fowler. [accessed 23 May 2017]. Available from: https://martinfowler.com/eaaDev/EventAggregator.html