UNIVERSITAT JAUME·I

ADVANCED MOTION SYSTEM USING ANIMATION AND PHYSICS

By Álvaro García Lozano
Supervised by: José Vicente Martí Avilés

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Description

This work consists of the development in Unity of a 3D scene that allows you to control a third person movement system applied to a character that moves in a realistic way and is able to move inside the scene with freedom and react to a variety of different elements.

In this project, you will be able to move with freedom and there will be obstacles to climb or jump. To achieve this, the combination of code and animation is really important. Code is needed to make detections in order to do each action and animation will be the combination of a huge animation library and some computer-generated animations a.k.a. procedural animations. Figure01 shows a preview.

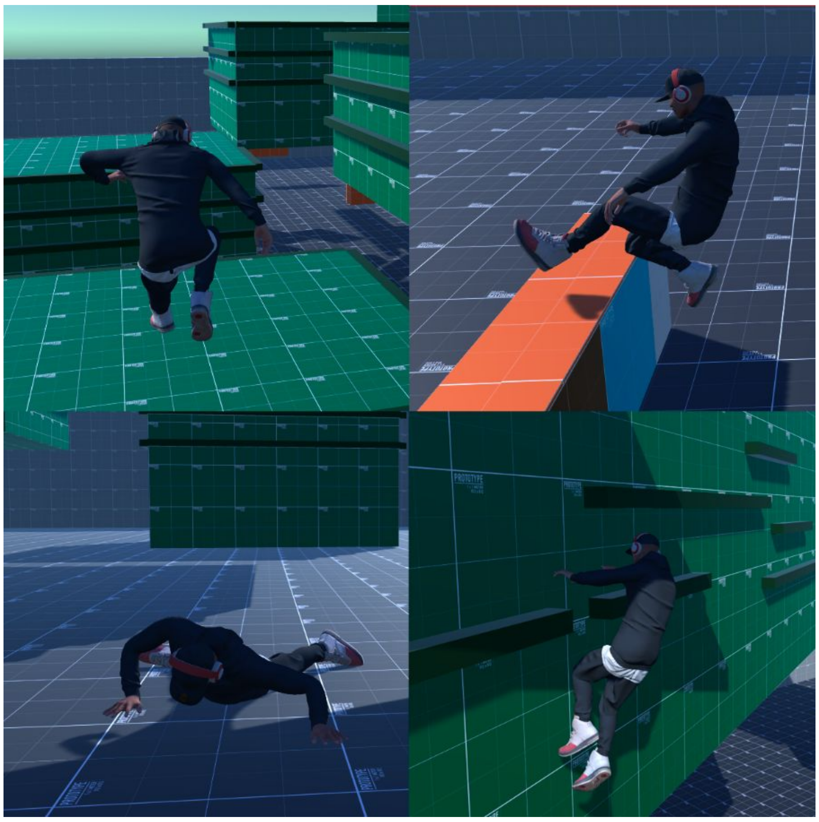


Figure01. Project movement demonstration.

Due to the verticality of the scene, the player also can fall off and get injured for that. The representation of this fall will be made by computer animation using ragdoll[1] physics.

An attempt will also be made to implement in this engine a system that combines physics and ragdoll at the same time in some simulated way by investigating different tools, This needs to be simulated since is not possible to perform as is due to internal restrictions of the Unity engine itself. This is something that no one to date has done.

At the end of the project, this research thread must have a conclusion derived from multiple tests, and the movement system must provide the freedom previously described.

1.2. Motivation

Working in 3D animation is my aspiration. After studying the subjects of the degree, I was left wanting to get deeper into this field, so this project can be the beginning of my professional future. Learning about the use and work development of procedural animation is something that, in addition to being highly demanded, always caught my attention.

Animation is very important in the industry, I'm currently working together with fellow students on a promising project that we want to carry out and it is very useful to have knowledge about the implementation of animations in a complex system.

The research section of this project stems from a GDC[2] talk[3] by Michal Mach, in this conference he talked about how Naughty Dog's[4] engine was developed for Uncharted 4[5] to perform the simultaneous combination of physics and animation in what they call Physics Animation. So my question was if something this complex and spectacular would be applicable in a free game engine.

[1] Animation process which uses a physics engine normally used to replace traditional static death animations in video games and animation films.
[2] Game Developers Conference (GDC): https://gdconf.com/
[3] Physics Animation in Uncharted 4: A Thief's End https://youtu.be/7S-_vuoKgR4
[4] Naughty Dog: https://www.naughtydog.com/
[5] Uncharted 4. Neil Druckmann, Bruce Straley [PlayStation 4] (2016). Santa Monica, California: Naughty Dog, Inc.

1.3. Objectives

- ❏ To create an animator with an extensive library of animations, giving it variety.
- ❏ The final result should look smooth and should work correctly and needs to be realistic.
- ❏ To learn about ragdoll and create one that reacts and behaves in a realistic way.
- ❏ To implement a smooth movement system that involves procedural animation, with interpolated transitions and real-time adjusted animations according to the environment.
- ❏ Discover how a complex system used in one of the most advanced games can be implemented in Unity, and if it is not possible, why.

1.4. Related subjects

- ❏ *Physics* (VJ1201) We use gravity, forces, friction, and multiple vector functions.
- ❏ *Programming* II (VJ1208) The project code has been written in C#.
- ❏ *Character Design and Animation* (VJ1226): Some theory concepts are involved, like keyframes and animation transform.
- ❏ *Game Engines* (VJ1227): The entire project has been developed in Unity engine.

2. PLANIFICATION

2.1. Requirements

The financial requirement necessary in hardware is around 400 € in current components to build a Windows-based desktop system capable of handling the Unity editor without problems. However, this is a 3D project (Unity handles 2D projects) that requires computational processing load and a system capable of handling Unity 3D software without problem has to be found.

Figure02 shows an example of a set-up with the components of a modest system able to handle this program with comfort, based on the requirements of modern versions of Unity.

Component type	*Component example*	*Orientative price (€)*
Graphics	AMD Radeon RX 570	142
CPU	AMD Ryzen 5 1600 AF	124
Motherboard	MSI B450-A-Pro	96
RAM	Corsair Vengeance 4GB DDR4	28
HDD	Seagate Barracuda 1TB	44
Power Supply	Corsair CX450M	58
Case	Cooler Master NR600	86

Figure02. Desktop build example for Unity3D.

Attending to official requirements[6], an actual system able to run Unity Editor costs around 550 €. The only software cost is the Operative System, in the example (Windows), adds another 100€ to the build. There is no extra cost to use the engine, in this case, the standard version of Unity has been used.

This project can be expanded in many ways and collaboration with more people is always possible, however, the person in charge of programming must have knowledge of the animator so that both work in synchronization, a very difficult task to coordinate if carried out by different people.

[6] System requirements for Unity 2019.4: ttps://docs.unity3d.com/Manual/system-requirements.html

2.2. Original project planification

The original planning of the project is detailed below, indicating its approximation in hours.

1. Investigation (40)
    a. Animate in Unity (10)
        i. Layers & masks
        ii. Interpolation
        iii. Animator functions
    b. Ragdoll in Unity (5)
        i. What is it
        ii. How it works
    c. Animation with Physics (25)
        i. What is it
        ii. Unity implementation

2. Preparation (15)
    a. Unity's project and character
    b. Animations download & set-up

3. Implementation (200)
    a. Character ragdoll (15)
        i. Ragdoll creation
        ii. Ragdoll adjustment
        iii. Basic animation to ragdoll system
    b. Animator (60)
        i. Procedural (30)
            1. Animated to ragdoll interpolated transition system
            2. Ragdoll to animated interpolated transition system
        ii. Animated (30)
            1. Planar movement
            2. Scene interaction
            3. Climb animations
    c. Movement (125)
        i. Planar movement (40)
        ii. Climb system (50)
        iii. Scene interaction (stairs for example) (20)
        iv. Impact system (15)

3. ANALYSIS

In this section, we talk about the study of the problems related to the realization of the project and strategy to follow to carry it out, describing the techniques and tools that could be used.

About the advanced movement simulation.

Parkour and climbing are complex systems that very few video games dare to try to implement. If they are developed in an advanced manner, could give rise to one big project each from them.

Nor is there a media that brings them all together beyond a high-end game that has a complex movement system, so all these techniques are usually satisfactorily solved in projects accessible by payment or in the video games case, projects whose code is not free to use.

There is hardly any documentation or references about these specific mechanics or an official way to carry them out since each one interprets or performs them in their own way. This is another of the main problems, since in this work, multiple functionalities are going to be adapted and everything has to be as versatile as possible without conflict. For example, the character may be required to automatically fall when approaching a cliff, but this would conflict with wanting the character to be able to jump over the edge to reach another surface.

About the Physics Animation research.

Regarding the implementation of this technique, the problem is rooted purely in the research, it is necessary to try to search and abstract from previous investigations, which are not many, the possible information in order to implement a functional system or, at least, try to explain what means are needed to implement it, and if it cannot, why.

Unity is not capable of handling physics and animation at the same time, so an implementation of this system needs to be done by deceiving the engine. A character will always be controlled by animation keyframes even if a ragdoll is activated, if you try to animate a certain part of the body, the rest do not respond to physics either, but remain motionless.

As discussed in the motivation section, the main research focus is on Uncharted 4 Physics Animation, something difficult to achieve since we are talking about a system that no other game uses, developed by one of the best companies in the world for years, and whose greatest advantage is that Naughty Dog has its own motor, which is capable of doing what they want.

Nor has anyone else (unless discovered during the research phase) managed to replicate this system in Unity.

Therefore, three major problems and / or challenges are defined throughout the project:

- Procedural animation and the mix of physics and animation.

Physics Animation gives an extra layer of realism to certain animations of an already built system and since nobody had managed to replicate it, it was not possible to determine in advance what was necessary before starting the implementation of the movement system, and hence the investigative character.
Methods and tests should be performed as new tools were discovered and applied.

It will be a matter of studying how this system could be simulated, the strategy to follow is the research and testing of the results obtained by the community, understanding its functionality and proposing new uses for them, if any, to determine how this system could be replicated .

Apart from this problem, it will be a question of implementing a system that alternates between both physics and mecanim[7] states in a realistic and natural way to make the character fall, informally called *Dynamic Ragdoll*.

The character, taking into account the movement system outlined below, will be built using RagdollWizard[8] as the base, since we have not previously worked with the use of ragdolls and any given help is a great step.

For this part, we need to investigate tools related to ragdoll handling or animation modification.
This tools that could be used to overcome this challenge are:

❏ Ragdoll Wizard, in order to set up a ragdoll base for the character.
❏ Animation Rigging[9], a new tool that allows us to real-time control certain parts of an animated character.
❏ Active Ragdolls.[10]
❏ Dynamic Ragdoll, in order to procedurally smooth transitions between states.

[7] Mecanim Animation System: https://docs.unity3d.com/462/Documentation/Manual/MecanimAnimationSystem.html
[8] Ragdoll Wizard: https://docs.unity3d.com/Manual/wizard-RagdollWizard.html
[9] Animation Rigging: https://docs.unity3d.com/Packages/com.unity.animation.rigging@0.2/index.html
[10] This system allows the ragdoll to move by applying forces to the body making it trying to stay upright or perform certain actions such as walking by an alternation of these forces on the legs.

- ### Advanced movement system

The character must be able to move around the stage naturally with a certain horizontal and vertical movement freedom. This advanced system in the project refers to the one that serves to grant freedom beyond the base movement, through the use of agility above all.

The main problem of this is the detection and resolution of all these actions in a dynamic way without conflicts between the different systems, in addition to the search for an acceptable simulation result. These systems usually make use of custom animations that adapt perfectly to what is required, but in this case, all of this will be implemented through the use of pre-configured animations.

The tools to use in this case are pure programming. All this defines a complex logic both in code and in the animator and everything must be able to work on the same character.

The strategy to follow is to decompose a functionality into its most basic operation, thinking about what we want and how we can achieve it, and from there thinking about new functionalities, adding them to the main idea.

The tools that could be used to overcome this challenge are the use of:

- ❏ Raycast[11] to detect each obstacle or special event.
- ❏ Coroutines[12] to control the code flow and perform actions that require waiting for an amount of time or a certain event.
- ❏ Interpolation[13] in order to correct values[14], positions[15] or rotations[16] in a smooth way .
- ❏ Ik System[17] in order to react to some special situations like stairs or ramps.

[11] Physics.Raycast: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html
[12] Coroutines: https://docs.unity3d.com/Manual/Coroutines.html
[13] Linear interpolation: https://en.wikipedia.org/wiki/Linear_interpolation
[14] Mathf.Lerp: - Unity. Retrieved June 10, 2020, from https://docs.unity3d.com/ScriptReference/Mathf.Lerp.html
[15] Vector3.Lerp: https://docs.unity3d.com/ScriptReference/Vector3.Lerp.html
[16] Quaternion.Lerp: https://docs.unity3d.com/ScriptReference/Quaternion.Lerp.html
[17] Inverse Kinematics: https://docs.unity3d.com/Manual/InverseKinematics.html

- Complex animation system

One of the most delicate parts of the project. For advanced movement simulation, a system with multiple states is going to be created and these have to be controlled and administered both within the game's code system and within the animator system. The Unity animator has its own parameters and transitions, each with its own conditions.

Here is an example of coordination situation:
This project has a turning aid system that allows the character to extra turn to give it more mobility. There are multiple situations in which we don't want that turning aid, otherwise the character would still turn and react to the controller when he is in ragdoll state, for example.

For an animation to happen, the animator's transition condition must be met, starting from checking controllable parameters from the character's code and whose variables are independent from the animator's own. If we want to perform a really simple action on the animator and control what happens before and after (for example to block that extra turn while we are jumping), we must track by code when it has entered and when it has left to activate / deactivate that turn, because animator as we said works on his own.

The strategy to follow is the order, the exhaustive control of the character states and the use of different methods such as animation events[18] that call a function in a certain keyframe or state behaviors[19], which we will explain in the implementation section. The annexed A1 shows an example of this complexity and the A2 an example of communication between code and animator.

The tools that could be used to overcome this challenge are the use of:

- ❏ Animator Controller[20] to handle everything.
- ❏ Animator layers[21], which can add some animations in an additive or sustitutive way and most important, can be used in certain body parts.
- ❏ Sub-State Machines[22] to tidy animator different states, also useful in order to apply a state behaviour to some animation clips.
- ❏ Animation Events.
- ❏ State Behaviors.
- ❏ Mixamo[23], Adobe’s free 3D animation gallery.

[18] Using Animation Events: https://docs.unity3d.com/540/Documentation/Manual/animeditor-AnimationEvents.html
[19] StateMachineBehaviour: https://docs.unity3d.com/ScriptReference/StateMachineBehaviour.html
[20] Animator Controller: https://docs.unity3d.com/Manual/class-AnimatorController.html
[21] Animation Layers: https://docs.unity3d.com/Manual/AnimationLayers.html
[22] Sub-State Machines: https://docs.unity3d.com/Manual/NestedStateMachines.html
[23] Mixamo: https://www.mixamo.com/

4. IMPLEMENTATION AND RESULTS

4.1. Implementation

In this section, the implementation process of the different aspects of the project will be detailed.

4.1.1. Project

- Packages

Assets used in this implementation

- ❏ Cinemachine[24] initially used, then discarded after a big issue.
- ❏ Unity Standard Assets[25] for basic movement calculations and freeview camera.
- ❏ Unity Pro Builder[26] to set the project scene and obstacles.
- ❏ Gridbox Prototype Materials[27] to texturize the scene.
- ❏ Animation Rigging as described before.

Cinemachine is a Unity package that allows you to create a virtual camera that is easy to implement and control.

In the final stretch of the project, from one day to the next cinemachine stopped working, so the entire package had to be re-imported again and the camera had to be re-created. The new camera did not stop shaking since then, so it was finally replaced by a Standard Assets camera.

To operate the camera with a controller, the MouseX and MouseY input were duplicated in the inputManager, but associated with the vertical and horizontal axes of the right joystick of the controller following this guide[28].

Probuilder and prototyping textures Asset have been used to build a suitable scenario, with exact measurements and a realistic scale ratio for project testing. With this, the ground and the boundaries, the stairs, the climbing platforms, etc. have been built, assigning different colors to the different parts and elements.

[24] Cinemachine: https://docs.unity3d.com/Packages/com.unity.cinemachine@2.1/index.html
[25] Standard Assets:
https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-for-unity-2017-3-32351
[26] ProBuilder: https://unity3d.com/unity/features/worldbuilding/probuilder
[27] Gridbox Prototype Materials:
https://assetstore.unity.com/packages/2d/textures-materials/gridbox-prototype-materials-129127
[28] Controller mapping:
https://answers.unity.com/questions/1350081/xbox-one-controller-mapping-solved.html

- Character

The character James belongs to Mixamo.

At first, unaware of the animator's operation, all the Mixamo animations were downloaded with the model itself included, (55mb per animation), which considerably increased the weight of the project. This has been drastically optimized after many tests by downloading the animations without model information. Only one animation with model has been downloaded and it is the T-Pose[29] animation. Unity works with different animation types. Those are always downloaded in generic type, and they are converted to humanoid so that it works correctly (this is not necessary in previous versions). A humanoid-type animation requires a model or avatar, which can be created from the animation itself or can be imported. Therefore, in the animations folder of James, you have the animation in T-Pose, whose avatar has been created from that animation itself.

The process to implement each one of the animations has been to import them into the project, change them to humanoid type, and then assign an external animation avatar, the avatar created from T-Pose.

After this method, James has more than 80 animations all of them optimized as shown on Figure03.

ANIMATION SPACE DIFFERENCE	**1 animation**	**50 animations**
With Skin	55,500 KB	2.775.000 KB
Without Skin	350 KB	17.500 KB

Figure03. Animation memory optimization.

Once James was on the scene, for ragdoll creation, the ragdoll wizard was used, which implies:

- ❏ Capsule colliders[30] on the extremities.
- ❏ Sphere collider for head.
- ❏ Box colliders for the torso.
- ❏ Character Joints[31] to assemble the different parts with Rigidbody[32].

This allows each part of the body to be affected by gravity and to collide and react to the environment in a realistic way.

[29] Common pose used to set bones and animations on one character, the model stands with his feet together and his arms totally open.
[30] Collider: https://docs.unity3d.com/ScriptReference/Collider.html
[31] Character Joint: https://docs.unity3d.com/Manual/class-CharacterJoint.html
[32] Rigidbody: https://docs.unity3d.com/ScriptReference/Rigidbody.html

Since the ragdoll is a primary part of this project as planned, 15 hours have been spent refining the character's ragdoll. The ragdoll Wizard had left the ragdoll incomplete and colliders were missing for the lower arms, which had to be created by hand.

Once the ragdoll was complete, the weights of each body part, the size of the colliders and the limits of each joint[33] were adjusted and tested by creating a basic character spawn system and a stage full of platforms. This allows us to determine test after test the best and most realistic configuration for the ragdoll, avoiding as much as possible that it could end up in strange, exaggerated or impossible poses.

The character has a boneRender, using the new Unity Animation Rigging, which allows us to see, as shown in Figure04**,** the character's bones and easily access those parts of the ragdoll, previously accessible through Unity's physics debug method.

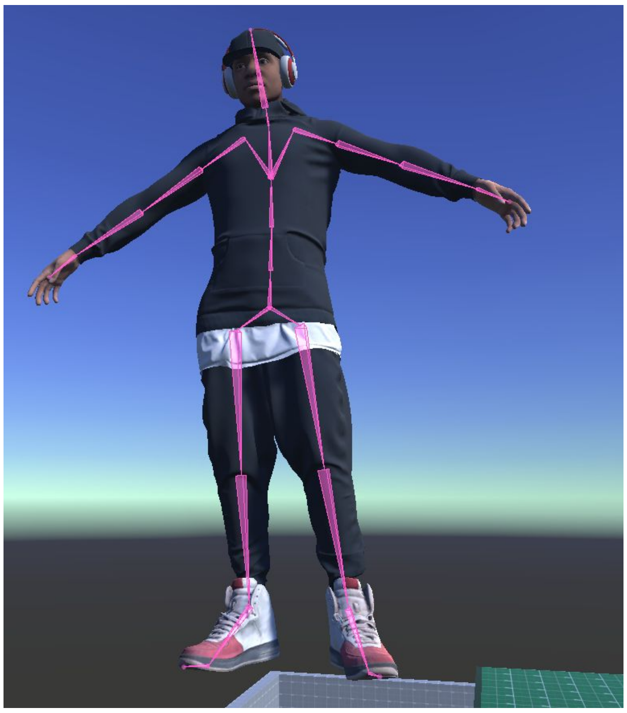


Figure04. Bone Render applied in a character.

Also, based on trial and error, a physical material[34] has been configured and assigned to certain body parts to improve behavior. This allows, with the current configuration, the lower leg to have more friction and give the sensation that the shoes break in a fall. The lower arms, on the opposite, have a different material to establish how the hands behave in a fall or the friction of these with respect to the rest of the body.

[33] Joints: https://docs.unity3d.com/Manual/Joints.html
[34] Physic Material: https://docs.unity3d.com/Manual/class-PhysicMaterial.html

4.1.2. Character Planar Movement

The type of movement conceived for this project depends on the direction of the camera. This type of control is always implemented in the same way and the Unity controller has been used as an example. Unity Standard Assets allows us, among its many functions, to access an example of free movement in the third person of a character, this movement system has been used, modifying it to our needs and adapting it to a different character with different animations.

All movement system in the Unity's example works in FixedUpdate[35] but this project has made some changes and works in Update[36]. The use of raycast is carried out in FixedUpdate, so the flow of control and detection of Inputs has been improved (input detection works better in Update) and these are transmitted to the controller by reference, speeding up detection as much as possible.
This change broke for a moment all the timers and smoothed variables, until all the values based on Time.FixedDeltaTime (for FixedUpdate) were replaced by Time.DeltaTime (for Update).

The technical part that involves calculating this direction of motion vector and a couple of small functions, including turning aid, have been implemented from this package, which is finally a small part of the entire implementation carried out. The method for calculating these values is always the same for this type of system, so it is not worth trying to implement something new or different.

The `ApplyExtraTurnRotation` function helps the character to turn faster, this has finally been determined by a static variable called `canTurn` that will be key in the entire project and which must be rigorously controlled. An inappropriate activation of this function can rotate the character while hanging in climbing or in ragdoll state for example. This function is only used when the character walks but is also controlled manually at the end of certain states to allow the player, for example, to exit from a parkour animation anticipating a turn and giving more realism and freedom to the movement.

- RootMotion

In the first initial version, all the animations were downloaded from Mixamo with "in place" mode. This means that the animation does not displace the character from the site and the character needs to move by code with instructions from the character controller. Moving by code, you have to adjust the speed by hand in order to visually correspond the speed with the rhythm of movement of the character's feet. In this project a controller will be used, making this visual correlation is something difficult with a movement by joystick, since the character will move at different speeds depending on the input.

[35] FixedUpdate: https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html
[36] MonoBehaviour.Update: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html

Following the example of the third person character of Unity Standard Assets, it was determined that the most realistic movement is the one that is directly determined by the animation, so they were all downloaded again including the movement information.

We have had to investigate how the animator works internally to use its different functions, especially the use of rootMotion[37]. ApplyRootMotion[38] is an animator option which can be controlled by code, while it is active, the root motion settings of each animation clip will be used. In this project, it is normally active, in order to allow the movement transform to be transmitted directly from the animation, this is combined with the ApplyExtraTurnRotation function to give extra spin mobility to the character at certain times. We said normally because ApplyRootMotion is disabled while the character is in the air. With that change, we can apply the position transform determined by the gravity at that moment, and not by the animation that is being carried out.

An example of how rootMotion has been set in each an animation clip[39]:
We want to make sure that, in a parkour animation, the character faces the platform and continues the movement in that direction, so the `Bake Into Pose` option of the `Root Transform Rotation` is used in these animations, preventing the clip from rotating the object. We also do not want the transform to jump with it, so, as with the rotation, the `Root Transform Position` (Y) is blocked in `Bake Into Pose`, so we will see the character jump in animation, although this jump is not really performed on the object, letting the system apply `Root Transform Position` in X and Z for the character to advance.

On the contrary, climbing applies the entire `Root Transform Position` to the body (regardless of the position lerp that is carried out for corrections), to make the character move in vertical axis (Y) or move laterally between platforms (XZ).

- ### Animator

The enormous workload and coordination between the animator and the code of this project would not have been possible without an exhaustive research of its operation and a slight prior knowledge of it. Even so, there are no general guidelines or norms to build a complex system.

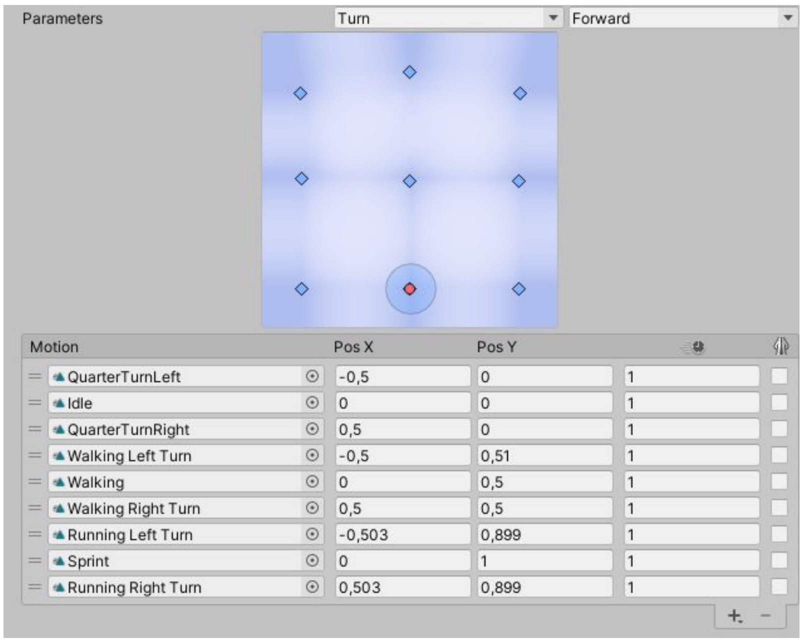
The basic movement is developed in the state called Grounded, and works through the use of a blend tree. The Unity Standard Assets movement tree was taken as a reference to understand its operation, and it has been applied to the Grounded system. The Unity example contains fifteen animations prepared for it, so it has been difficult to replicate it without animation recording media. For this project, nine animations have been implemented from diverse Mixamo animations, three for each state of movement, dividing this into Idle (y = 0), walking (y = 0.5) or running (y = 1). As shown in

[37] Root Motion: https://docs.unity3d.com/Manual/RootMotion.html
[38] Animator.applyRootMotion: https://docs.unity3d.com/ScriptReference/Animator-applyRootMotion.html
[39] AnimationClip: https://docs.unity3d.com/ScriptReference/AnimationClip.html

Figure05, each state has the standard animation and a turn animation on each side. This system, based on the forward and turn values that allow defining the desired movement direction with respect to the camera, creates a Cartesian coordinate system in which the forward value will determine the amount of movement and the turn value will do the same with the turn or tilt factor of the character.

These animations have been adapted and modified to this new system. For example, the right turn is a mirror animation of the right turn, the animation offset has had to be adjusted to square the movement of the legs, otherwise, the character would advance with both legs at the same time. This is a silly and difficult to understand error that took time to repair.

The coordinates for these animations and this movement system have also been modified.



Parameters Turn Forward

Motion	Pos X	Pos Y		
QuarterTurnLeft	-0,5	0	1	☐
Idle	0	0	1	☐
QuarterTurnRight	0,5	0	1	☐
Walking Left Turn	-0,5	0,51	1	☐
Walking	0	0,5	1	☐
Walking Right Turn	0,5	0,5	1	☐
Running Left Turn	-0,503	0,899	1	☐
Sprint	0	1	1	☐
Running Right Turn	0,503	0,899	1	☐

Figure05. Ground movement blend tree.

The forward value is decisive in many aspects of the project, its value is applied in a smooth way from the character movement script, but once it is applied, the exact control of its value is lost. It is also decisive to know when we are reproducing this movement blend tree. The movement script takes exhaustive control, asking the animator, of the forward value and when this blend tree is being performed or not. The amount of the forward value determines the amount of movement and, in the different resolutions of the hardInput, this is a determining factor.

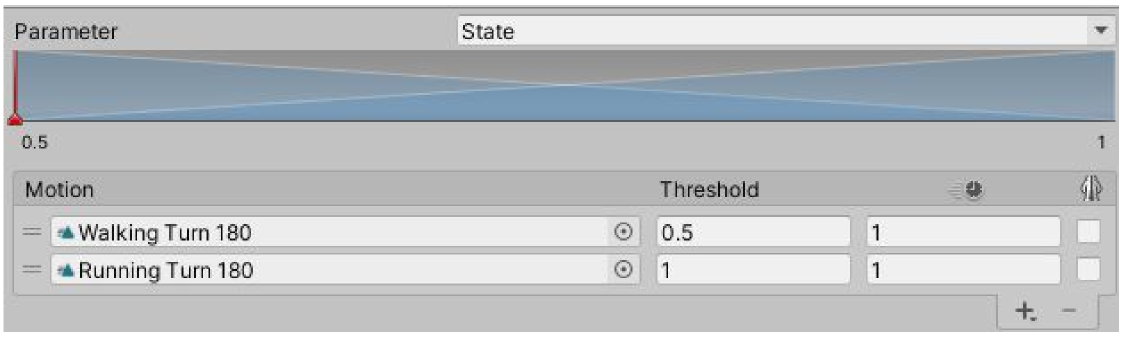
- HardInput

HardInput is the name given to the sudden movements controller, which very few professional games worry about managing. In this case, it has been optimized to respond to a sudden change of direction, for example, if we are running head-on and the player suddenly requests an input in the opposite direction.

`CheckForHardInput` is a function of the move script. This function detects sudden changes in the controller either due to a change of direction or dry braking. Checks are performed if the character's speed (taken using the animator's forward factor as previously explained) is high enough. To determine the sharp turn, we store the controller's input direction vector as received from the Input script and store the character's motion vector, determined after several tests by the character's rigidbody velocity vector. If a control input is requested in the opposite direction of movement, we will be forcing the system.

The system can present detection failures after the exit of certain states, such as climbing or parkour, since the rigidbody can present certain corrections in its velocity vector that will activate this system when it is not desired. That is why this system needs to be controlled when leaving these states.

How the spin hardInput works:
Among the previous tests, still present in the code for reference, this system worked using a coroutine, the new system is managed between the animator and the script. This system blocks the extra turn and determines if the hardInput is rough or smooth depending on the speed of the character. In the high speed event (character speed is near maximum), the animator will reproduce a running change direction animation; otherwise, the animator will perform a blend tree shown on Figure06.


Parameter: State

Motion	Threshold		
Walking Turn 180	0.5	1	
Running Turn 180	1	1	

Figure06. HardInput middle speed blend tree.

This blend tree allows the player to reproduce an interpolated middle point animation depending on speed between a soft turn animation and another one more aggressive.

We need to determine when to turn on the ExtraTurnRotation in order to be able to change the movement direction at the end of these animations, so we need to know when this state starts and ends.

Instead of a coroutine, the new system uses an Animator State Behaviour (`HardInputBehaviour.cs`). This type of script allows the user to know and control the animator in that state, so we use that to start and update a DeltaTime type timer that depends on the animation type. At the end of the counter, `hardInput` and `ExtraCanTurn` are reactivated.

HardInputDetector also detects a hard stop using the both referenced horizontal and vertical direct controller input, in order to determinate when none of this are pressed and making a reaction depending on player speed to react and quickly stop the character if he is walking toward a ledge in a dangerous situation. This is something we will see afterwards.

A lot of work has been done with the transition between movement and hardInput, as the transition from this state back to the Grounded blend tree caused the animator to stumble, interrupting the player's rate and speed of movement. Several options were implemented, first was to adjust animation exit time to fit this movement change amount, that was not accurate, not flexible, and most important, continued to make the player transition abruptly.

Less smooth amount means more direct value translation, a smoothed value forces us in a blend tree to walk between an idle and run state. After a hardInput, when we entered again in the grounded state blend, forward was often different from expected, due to his changes while the animator was on hardInput state. Normally, we want to change direction and continue our movement with the same momentum we were doing it. In this second approach, to force the grounded state to quickly adapt the player movement at the end of these animations, forward smooth amount value was temporally reduced, in order to establish quickly the movement amount, and then restore progressively the old smooth amount. This was not totally comfortable and the system continued to fail sometimes because this was so rude.

A final and more accurate approach is a new system that locks animator refresh to keep forward state value until the end of any hard input state. This, in addition to the use of animations depending on the speed as explained before, allows the mecanim to smoothly transition to and from hard input state and continue the movement linearly.

If the project was played with the "Maximize on Play" option active, some animations broke. To discard any Unity editor issue, a build was made. In the build, the problem kept happening, the character was unable to do a hard input spin, the animations were totally broken and a walking 180 degrees turn animation made the character turn less than 90 degrees. To make some tests, another version of the project was made, with just animation transform information using Root Motion, no extra turn rotation, and no other scripts on the character, just one to read a keyboard input and reproduce that animation. The animation has no offsets or bake into pose options activated. With all these ideal stats, the animator continued to reproduce just half of the spin, and this amount of rotation directly applied, depending on the screen size.

I'm not the only user with this issue and definitely it is something not new. But there is no useful answer for this problem, as founded in some forums[40] [41] [42]. Also, this issue was reproduced only by some animations, especially hand input ones.

[40] https://answers.unity.com/questions/922181/sprite-animating-backwards-and-slower-on-maximize.html
[41] https://answers.unity.com/questions/737158/animation-plays-in-preview-mode-but-not-max-screen-1.html
[42] https://answers.unity.com/questions/313468/player-animations-not-working-when-playing-scene-i.html

After several hours of testing, the solution was to activate the animator update mode[43] to physics[44]. Again, multiple tests were made, with no physics and no other scripts, just raw animation root motion and some animation clips broke depending on screen resolution, so this is something that has no sense at all.

4.1.3. Character Advanced Movement

4.1.3.1. Parkour

The parkour system allows the character to overcome certain obstacles dynamically. This system is managed by a specific character script called `ParkourController`.

As explained in the analysis section, first a basic implementation of this system needs to be proposed. The first iteration and conception of the system had to overcome an obstacle detected by raycast, rotating the character towards it to add realism. After that, the system was upgraded by adding new functionalities.

Mixamo offers many animations related to this discipline, some of these animations consist of sliding, others in jumping, so obstacles have been divided into two types, each with a different Tag[45]. Within the jump animations, these have been divided into three groups depending on the distance at which the character is from the platform. So the system currently has three slide animations, three jump animations at close range, two at mid distance and another two at long distance.

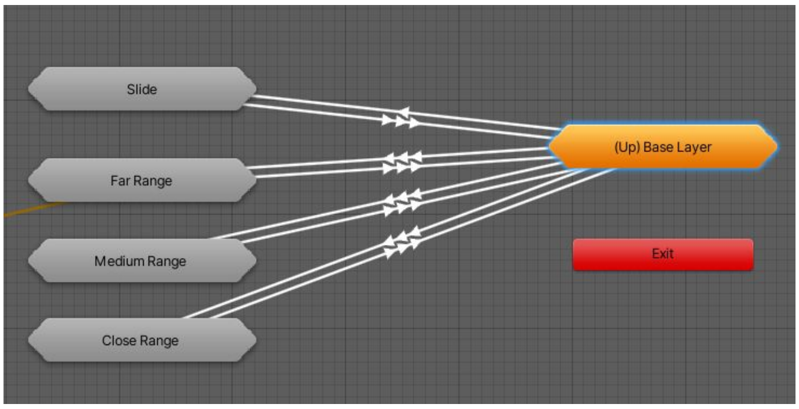


Figure07. Parkour sub-state.

[43] AnimatorUpdateMode: https://docs.unity3d.com/ScriptReference/AnimatorUpdateMode.html
[44] This option synchronizes the animator update with the physics update.
[45] Tags: https://docs.unity3d.com/Manual/Tags.html

As shown in the Figure07, sub-states have been created for each type of animation, the three corresponding to the platform jump and the sub-state Slide, for platforms of this type.

These obstacles are detected using a raycast and take the corresponding action. Jump-type obstacles are built from two equal halves.

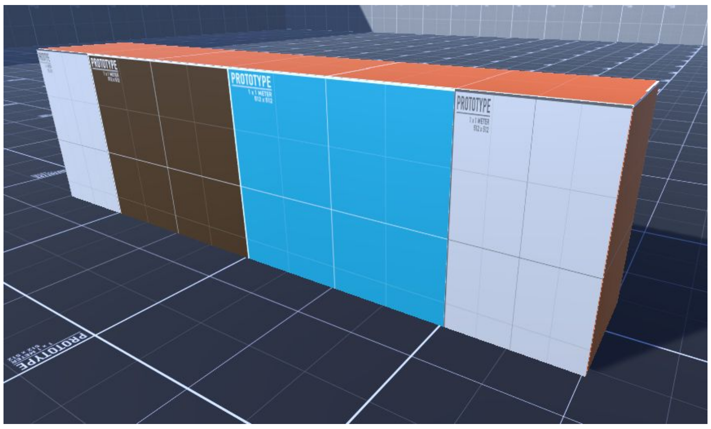


Figure08**.** Parkour jump platform.

As shown in the Figure08**,** each of them has four quads glued on its front, these are the type of objects that will have the raycast detection layer instead of the obstacle itself, the use of quads allows raycast detection to be carried out only if the we look straight ahead, so we will avoid jumping obstacles if we approach them from the side. Each half of the obstacle is made up of two quads that define that face on the left and right sides, and two small quads that define the limits of each side. Each side is used to differentiate and to support one hand or the other in the animation, so we can avoid rushing the right side of a platform for example and lean on the right hand in the air. For this, a mirror version of the animations is used in which the character supports a hand, all variations are inside the sub-state as seen in the Figure09.

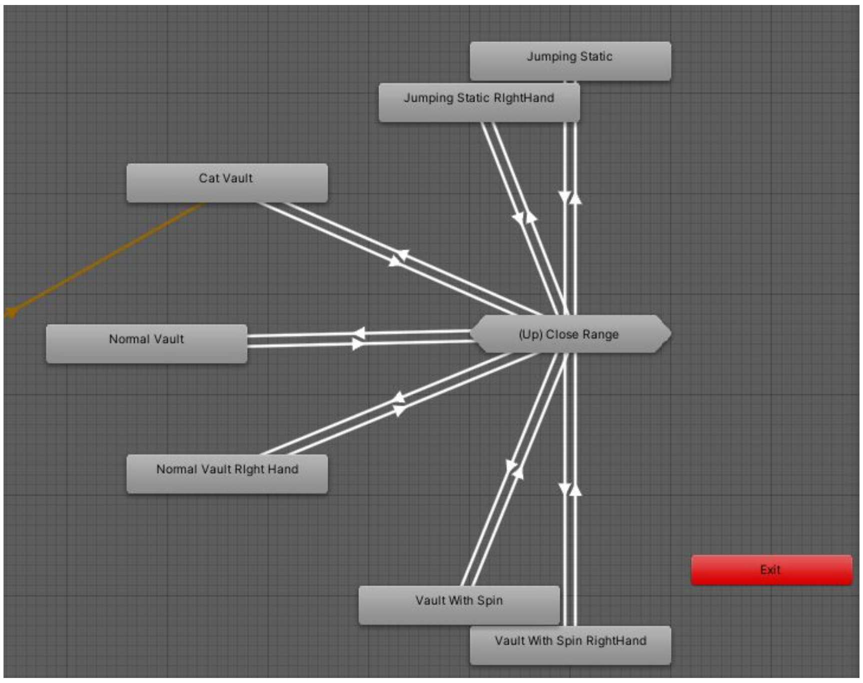


Figure09**.** How a close range sub-State looks from inside.

The idea is to rotate the character towards the platform just before the jump, since the player can approach from different angles. When rotating the character to the platform we have the exact distance to the platform, but there is a problem, the rotation must be subtle to be realistic and not a simple change in the direction of the character, so the distance must be calculated from beforehand. As the raycast is successful, the angle between the player and the platform is checked to limit the jump angle and avoid very sharp turns, the player must be more or less facing a platform to be able to jump it. The limit quads work in addition to defining the jump side, to further limit that jump angle and avoid using the raycast length to create angles that can lead to error.

The calculation of the distance is made only when the player requests the jump since it requires mathematical loading to be performed at each iteration of the raycast. The distance is calculated by trigonometry.

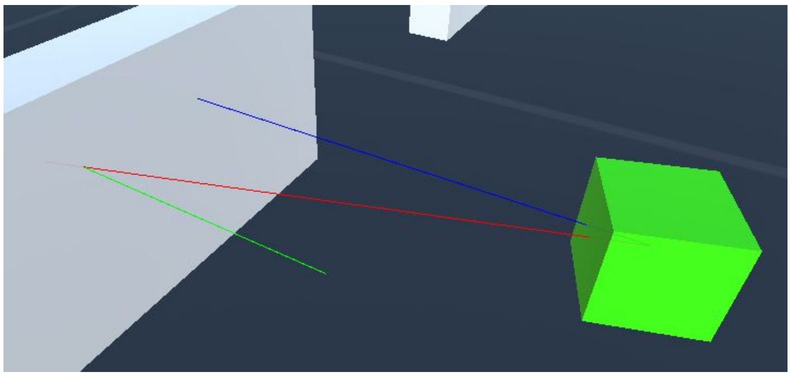


Figure10**.** Illustrative example or raycast calculation.

In Figure10, the red ray represents the frontal raycast of the character to detect these parkour events. Blue ray represents the minimum distance to get. The green ray represents the normal vector of the collision detection. We can use this normal vector to get the angle between the character raycast and the normal off the wall or set the rotation of the character in his opposite direction, to make him look at the platform before making the vault animation.

In order to get the final distance, we use the player raycast (the red ray) as hypotenuse, and the calculated angle (between red and green) to get the adjacent side (the blue ray). The blue ray represents the real distance between the platform and the player. For the necessary calculations the trigonometric functions of Unity were investigated, the cosine function is part of the Mathf[46] library, while the calculation of angles was investigated in the official documentation.[47] [48]

The animations have been indexed in the animator by using the tens and units to differentiate the distance range and animation index. Animation index 11 drifts to the first animation in distance range 1 (close), so on. This allows each animation to be indexed individually directly from the Grounded state, which allows modifying the transition by hand (input time, output time, transition time, etc.)

It was at this point that the canTurn variable became static, which determines the extra turn of the character, so that each state controls when it is activated and deactivated. In the case of parkour, the sub-state machine in which all its animations are categorized in the animator, has a script (`ParkourAnimationBehaviour`), which is responsible for reactivating `canTurn` after a certain time since this state is entered, This time varies according to the animation type, to be able to rotate the character and change the direction while the animations are running, thus giving greater mobility, always avoiding turning towards the jumped platform and trespassing it. The character's collider and gravity are also controlled to be disabled during animation to smoothly traverse the platform. Parkour and hardInput, are the only states where CanTurn is reactivated prematurely to gain mobility.

If the jump request is made while the character is walking and not running, a different animation will be reproduced in which we will safely vault the platform (if it is a jump type platform).

This animation also distinguishes the side to put your hands on and, in addition, a controlled position will be determined at a certain distance from the platform where the animation works best. This position correction works by interpolating the position at the same time as the rotation is interpolated to look to the platform. This gives more realism and more freedom with distance so that if we are completely glued to the platform the character will subtly separate just before jumping, or, if he is far to leaning properly, within the limit, it will approach and jump. In the same way as with dynamic parkour, `canTurn` is activated in just enough time to avoid turning on ourselves before reaching a safe position.

[46] Mathf: https://docs.unity3d.com/ScriptReference/Mathf.html
[47] Mathf.Deg2Rad: https://docs.unity3d.com/ScriptReference/Mathf.Deg2Rad.html
[48] Vector3.Angle: https://docs.unity3d.com/ScriptReference/Vector3.Angle.html

4.1.3.2. Jump

After many tests, the jump is performed physically, this allows the player to jump to distant places or ledges. Therefore, it is the only animation state that does not make use of RootMotion.

After testing with countless animations, the expected result was a short delay between the jump input and the jump itself, in which the character was seen to pick up a minimum of momentum. Mixamo animations start directly with the jump itself, so they have had to be downloaded with more than one animation loop, which ends with the character falling to the ground. A bigger loop allows us to adjust the animation time, giving the animator more leeway delaying the offset (instead of starting with the jump, they start with the character on the ground and then jump). The cuts and beats have been adjusted with the animator and then, by introducing an animation event, rootMotion is disabled in the moment the feet are off the ground. The detection distance of the raycast to the ground is also reduced to pass as fastest as possible to OnAir state and an impulse-type force[49]is applied to the Y axis of the character to make it jump.

In the animator, the different options have been controlled with variables, to continue running or to stop in dry braking if we do not touch any button when the character touches the ground. We are talking about ground jump, but there is a last option, that the character has jumped from a high place. In that case, we will go to a fall loop animation in which, if the fall is not big enough to activate the ragdoll, all the options are controlled in the animator when we hit the ground.

These options are:

- ❏ If the fall was slight and we are not sending a movement input, the character will fall naturally, still.
- ❏ If the fall was slight and we are sending movement input to keep the character moving forward, it will stumble.
- ❏ If the fall was large and there is no input, the character will land with more violence, ducking much more.
- ❏ If the fall was large and we are sending movement input, the character will roll on the ground when falling.

[49] ForceMode.Impulse: https://docs.unity3d.com/ScriptReference/ForceMode.Impulse.html

4.1.3.3. Climbing

Climbing is a very complex system that usually consists of many parts of development and explanation in any tutorial, which is why, like parkour, it has been carried out from the pure test.

The climbing brought with it numerous problems, including:
-The character's collider was in conflict with the wall, just as gravity wreaked havoc on the animation.
-If you jump looking at the edge of a platform, the character is hooked with one hand in the air.
-If we jump looking at the side face of a platform, the detection normal would be wrong.
-How the character is thrown onto the platform regardless of the angle of entry.

The resolution of these errors, among others, was a system that was proof of extreme situations and thought of versatility. Figure11 shows how a climbable element looks like. A climbing element or platform is defined by the platform itself and two limits on each side, which exceed the geometric limits of the platform, each containing within a position placeholder to which the character will move if it detects one of these limits. Limits and placeholders are both invisible.

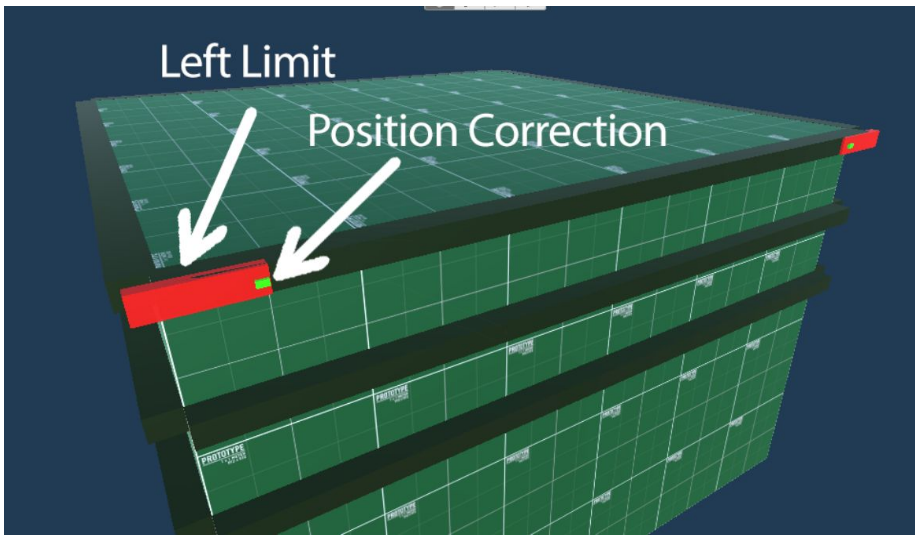


Figure11. Climbable platform example and composition.
This allows us to easily reposition the character at one end of the platform to make jumps to others, limit movement or, for example, reposition the character if he jumps from the ground towards one of the ends of the platform, preventing a hand from staying on the air as shown in Figure12**.**

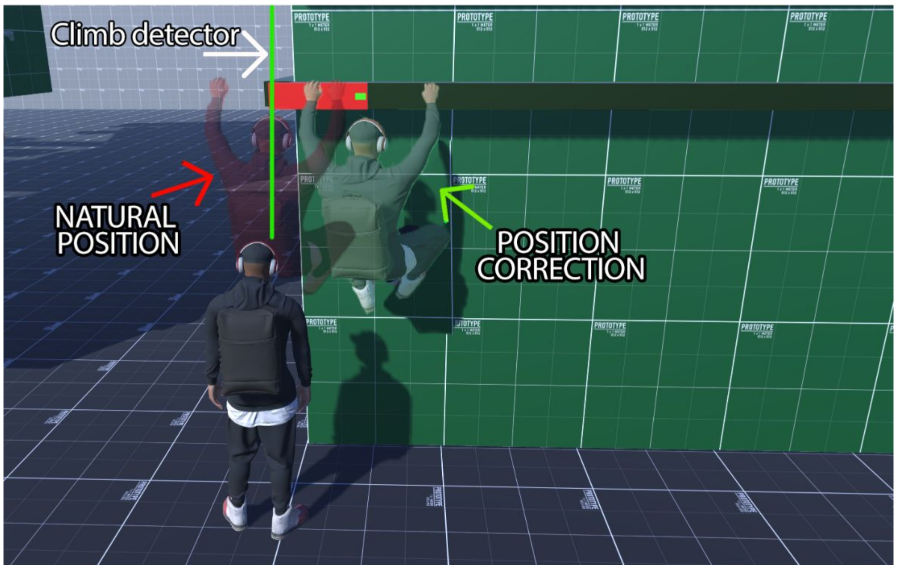


FIgure12. Position correction if one limit is detected, avoiding visual bugs.

For climbing, what we could say is a hang point, a separate object apart from the player, has been created with its own script and which contains its own object anatomy, represented in Figure13. This `HangPoint.cs` script allows the player climbing script to know which platforms are around them, to know when a jump is safe or to detect when we are heading towards a limit.

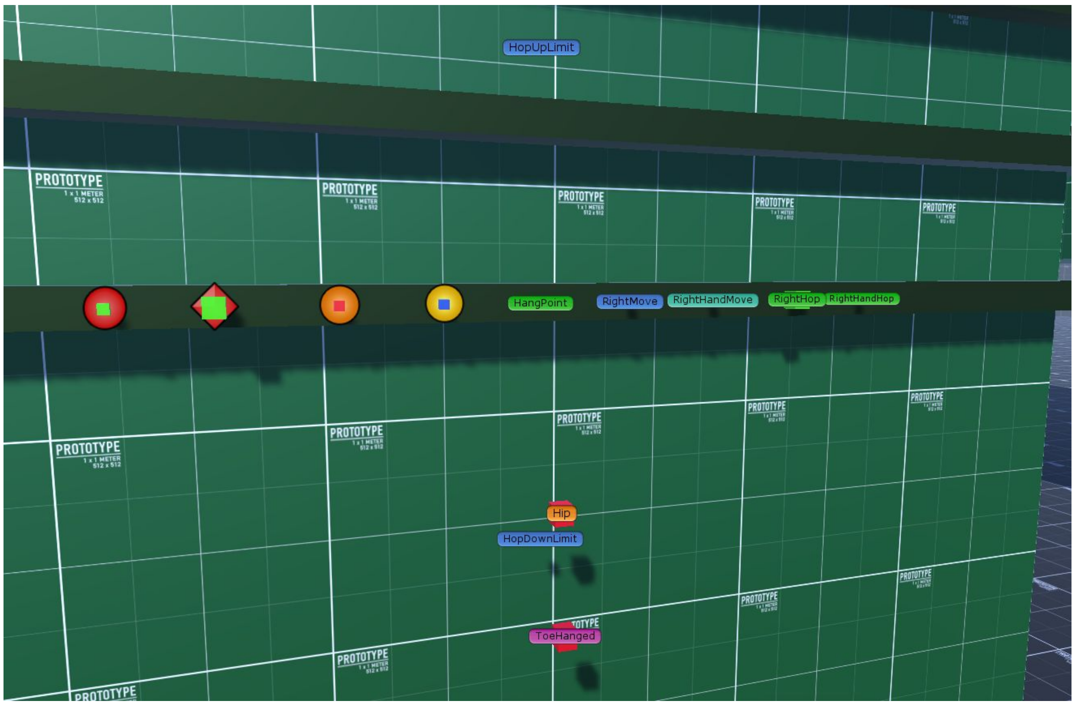


Figure13. Hang point anatomy represented.

The objects that derive from the anatomy of this hang point are position references. All important distances have been calculated, for example, in Figure13, the purple ToeHanged placeholder represents the character's transform position while he is hanging (the root position is at the bottom of the character). This anatomy contains the transforms of the position of the hands, the future position of the hands and the hang point itself in a normal movement or in a movement of either horizontally or vertically jump. This allows us to be able to know in advance if we can move or jump in depending on which direction or if our hand in that position would end up resting in the air.

The utility of these reference points is to use them to launch raycast in the forward direction and to know if we are approaching or are going to jump to a valid site. For example:
If we are leaning on a ledge and want to jump to the right, we must know with a raycast if our hand is going to end up in the air and if it is not, and the movement is valid, what is the new transform to carry out the movement up there.

All this would be much easier by using pure and hard root motion and making the character move through its animations, the big problem is that the Mixamo animations used have a recording error and the rotation of the character is modified with respect to the wall, so, after successive movements, we could see how the character enters through the wall or ends up flying in the air. After a lot of tests, this is something impossible to correct from the animation settings. This has complicated things a lot, so it is necessary to know in advance exactly where the character will end so, apart from the animation, interpolate the position of the character from where he is to where we want him to go. This is done in certain points of the animations through animation events and, as a penalty, obtained animations could look more natural in terms of rhythm. Without this correction you could not chain more than two equal animations without breaking the offset of the character.

Everything is controlled at the code level, if we can move according to which direction, when we can jump or when a movement is valid. All these permissions are controlled by animation events to, for example, make a jump to the right while we are moving the right, which implies recalculating the future final position the character will end.
The code is found in `ClimbController.cs`, in the character.
The handler's code belongs to `HangPoint.cs`, located in the GameObject of the same name.

To modify and improve the system, making it much more adaptive, controlled distance limits have been defined between which, instead of launching a single raycast at a certain height to detect a platform, a beam of them is launched. This allows you to hold onto a platform regardless of where you are, within these defined limits.

Figure14 shows the grounded raycast detection limits, detection in the air is detected by the green ray, while reds delimit the detection range in the ground.

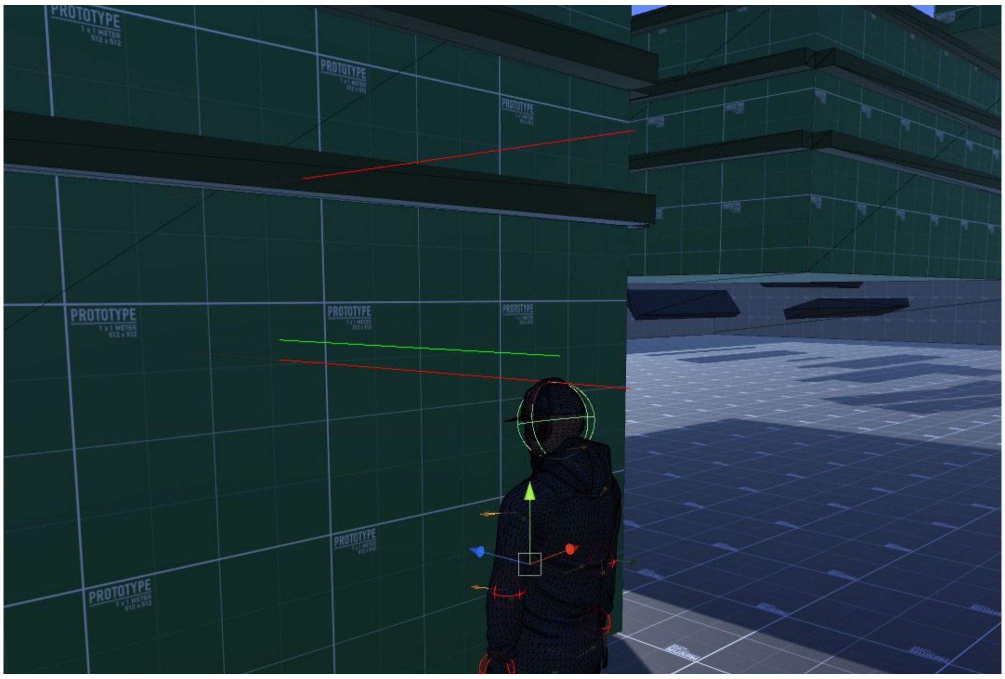


Figure14. Air Raycast and Grounded Raycast detection limits.

Figure15 shows an example of the ray beam.

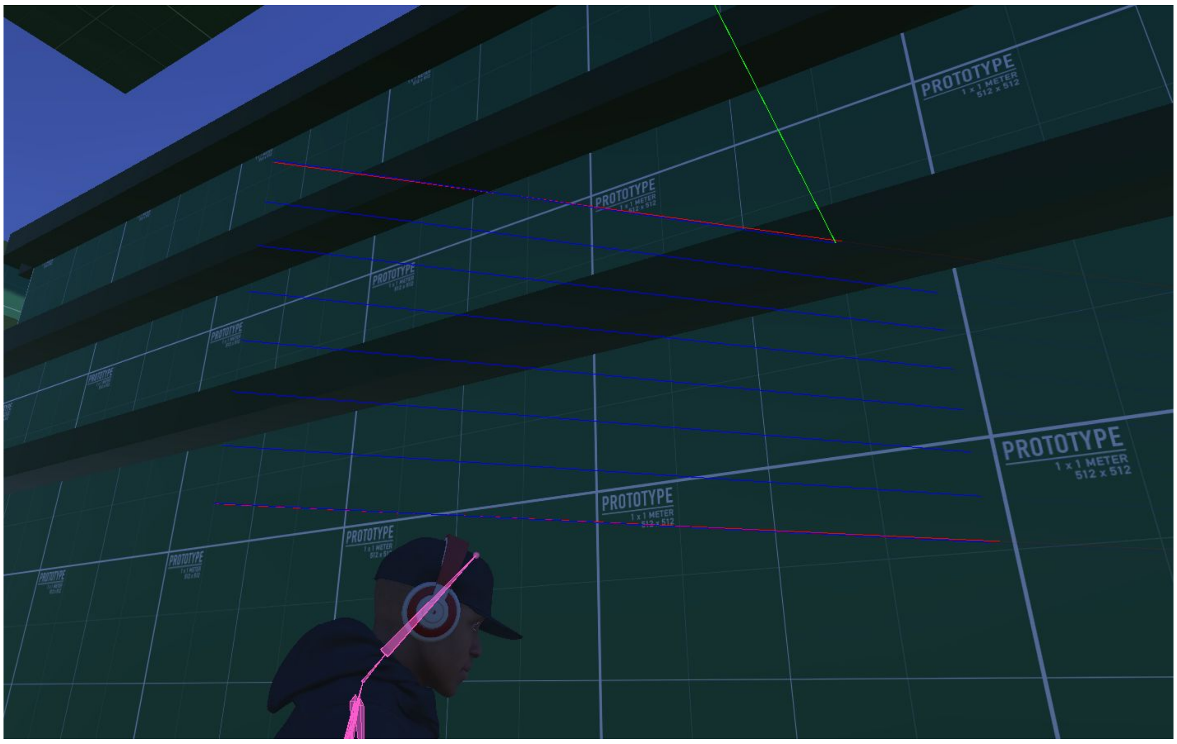


Figure15. Example of the detection beam, in which the beam coinciding with the upper limit detects the platform and its normal direction vector (green ray).

The exact point where we climb a platform does not seem to be quite natural, this is because there was no animation for it, the climbing animation has been cut from a longer animation in which we start with the legs hanging and which has been adjusted in the animator. This animation ends with the character crouched so it has to transition to an animation in which the crouched character stands up. The animator makes use of an animation behavior that will be responsible at a certain point to restore all the variables and reactivate the character's physics and collision, turned off during climbing to solve the gravity and collision problems mentioned at the beginning.

The boundary object has been carefully scaled to be detectable by the vertical raycast, this allows to detect limits in vertical jumps to allow the fluidity of the jump allowing more margin movement since we can jump to a platform that is not perfectly aligned and the position will be corrected thanks to the limit placeholder.

Finally, the fall from climbing option has also been implemented. When we let go of a platform, the character calculates the height to the ground and if it is enough to activate the ragdoll, we directly activate the ragdoll with an event of animation at a certain point in the drop animation.

4.1.3.4. Extra Details

It was decided to improve and advance the current behavior by adding a layer of complexity of animations and reactions to the character trying new things.

- Extra Turn Rotation

It has been present throughout the project and the management of this system has gone through many stages and tests until it reaches its final version.

As explained above, the spin assist system depends on the `m_CanTurn` variable, being one of the few static variables in the game. Throughout the development process, this variable was activated and deactivated by each system that interacted with it. However, when the project had so many methods, following the trace of this variable was practically impossible, so it has been left to the GameManager.

We want to disable this system when we climb, fall, jump, cross a platform or perform a specific action in which we want to limit movement (such as static jump). Therefore, GameManager has a public coroutine that is responsible for disabling this system and waiting for the animator to return to the "Grounded" animation state to reactivate it. It is possible that several equal coroutines work together at the same time, for example, if we start to climb (call the coroutine) and from climbing, we let go at a height high enough to activate the ragdoll (call the coroutine). In these cases nothing happens, both coroutines wait patiently for the same event and once we enter in Grounded animation state, all the active coroutines will be deactivated, allowing the rotation aid.

- ### Inverse Kinematics[50]

This system was implemented following a tutorial[51]. It has been implemented as a test since the intention was to apply animations to interact with stairs, but the Mixamo library does not offer competent animations for it. This system can be used on stairs or specific points with ramps, in them, the movement speed and the weight of the character's rigidbody have been controlled to prevent the character's collider from bouncing.

- ### Ledge Behaviour

This system manages when the character is on a high precipice by using two raycasts, both facing the ground, with infinite length. ExtraTurn is also blocked on this system. A dangerous fall will be detected if the raycast hit impact distance is higher than a designated value.

The first raycast is in front of the character, a little bit separated, it detects in advance when the character approaches a platform limit with a high fall in front, to perform a braking animation if we release all input while walking.

The second raycast is nearly to the character and serves to detect when we are on a ledge about to fall. If this is the case, the character enters in balance mode. This detection can be chained with the previous braking, causing the character to hard brake and, if the edge is hurried enough, the character will transition to balance.

We have two options If the character is about to fall and is balancing. If we move in the opposite direction to the fall, the character will regain balance, going to normal state, if we move in the direction of the fall, the character will lose balance and fall, activating the ragdoll at a certain point in the animation with an event.

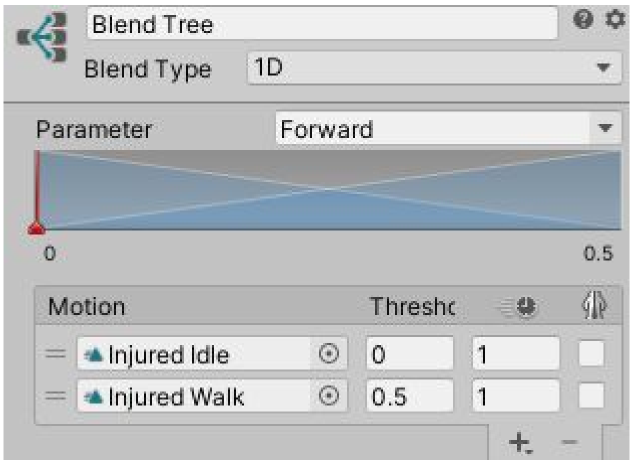
- ### Wounded State

This state is managed by the GameManager. The ragdoll impact system defined in the next section is used to determine the amount of damage the character takes in a fall. If this amount is enough, specific animations are played to stand up, then, the character is stunned and the Injured animation layer will be activated, which works in an override way, so we will not be able to see the Base Layer.

This state is managed through the use of a timer in the GameManager, and the main difficulty is to adapt the entire code system, since we do not want the character to climb or run. All systems are disabled except the static jump, which has its own animation and the system is configured so that if it detects a fall, it automatically goes to ragdoll, resetting the system.

[50] Inverse Kinematics allows characters to procedurally adapt, for example, their feet height to match with the terrain under them.
[51] IK tutorial: https://www.youtube.com/watch?v=MonxKdgxi2w

Originally the movement blend tree consisted of another 2d Freeform Cartesian system, but the construction of the spin animations by mirroring the existing ones caused the character to limp on a different leg depending on the turn. Finally, Figure16 shows the final movement, done through a blend tree of two animations. Character will turn with the help of ExtraTurnRotation.


Blend Tree
Blend Type 1D
Parameter Forward

Motion	Thresho		
Injured Idle	0	1	
Injured Walk	0.5	1	

Figure16**.** Injured movement Blend Tree

- Extra Animations

A multitude of extra animations have been implemented, to add life and realism to the character. This system works in two additional animator layers and the dynamic one works in additive mode. This layer has been assigned an avatar mask so that only the upper body is modified, this way we can see how the arms and head reproduce animations while the legs move according to the grounded system.

This system is managed through the GameManager, running every so often, as long as we are in the grounded state to avoid extrange animations during climbing for example.

When activated, the animator distinguishes whether the character is standing or walking (the system does not react if we are running). Within one of the two states, a blend tree has been created with all the animations available in that state. These two states were created because the static animations only work well when we are stopped. The GameManager knows how many variations are available in each state (by some variables) and assigns a random state of the corresponding blend tree. These blend trees are used to organize the system, not to interpolate animations.

The animator manages the speed of the character at all times, so if we are in the static state and we start walking, they will be deactivated smoothly. If the state, on the other hand, is a moving animation, the animator will take care of deactivating it if the character starts running or completely stops.

4.1.4. Advanced Ragdoll

4.1.4.1. Physics Animation

The most difficult aspect of the research was Unity's simultaneous animation & physics handling capacity. We will talk about the practical and implementation tests.

The idea was to be able to implement a climbing system where the character holds on only by his hands after a jump and use this system to simulate natural physic movement in the rest of the body.

In the Uncharted exhibition you can see how the Naughty Dog's engine is able to manage the use of physics in certain parts of the body (Figure17) while the rest continue in animation, applying real-time physics with different weight in different parts of the body in witch they call Physics Animation, unlike Unity, which is not prepared for it.

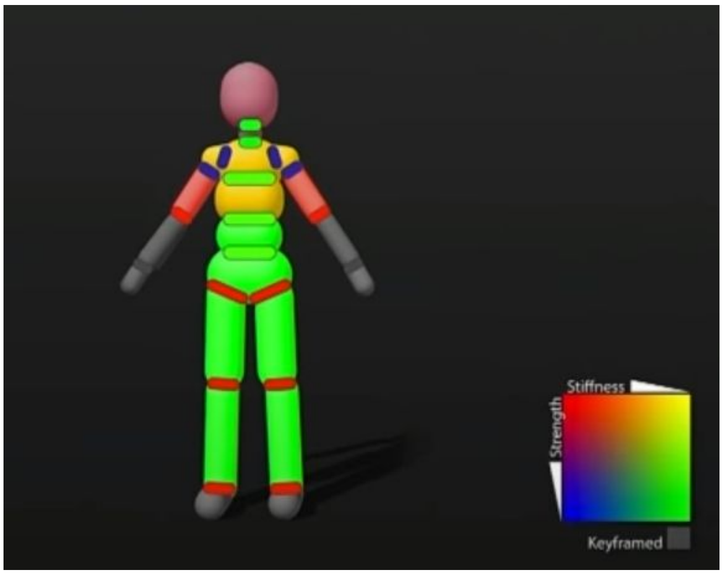


Figure17. Naughty Dog's ragdoll and physics influence[52].

As defined in the project's analysis, it was not possible to define in advance what type of character or system would be necessary to replicate Physics Animation. The different tests had to be implemented on a base system, it was necessary to first create the movement and climbing mechanics and the basic transition between animation and ragdoll. These methods and tests had to be carried out with the risk of having to restructure the project or remake the character so that they work correctly and cannot be implemented due to time constraints.

[52] This was shown on 34:52, in the Uncharted GDC video linked in reference [3]. Here is the bibliography.

Therefore, it was necessary to determine the type of movement, the use of one or more characters, and the type of ragdoll. The type of movement has been described above, based on Root Motion instead of code movement for more realism. The AnimFollow reference system is used for the dynamic ragdoll part and uses two characters; in it, the character only knows how to crash and fall, the project does not go further. For this reason, the use of this system based on these two characters was ruled out at the beginning of the project since it would have made the project more complex compared to other advanced movement mechanics. Finally, the ragdoll was created in the easiest and most common way possible due to the inexperience in this section and the complexity of it. Ragdoll WIzard was used, which uses character joints in its joints, as described in the implementation section of the character.

Once the system was fully set up and movement basics were working, Physics Animation tried to be implemented following three different research lines:

- Animation rigging package

This research was quickly concluded because this package allows the mecanim system to adapt certain poses by adjusting the character's body, however, no physics can be applied to it as discovered during the investigation[53]. Body parts follow a target depending on a weight value, so this package is really useful to obtain several different animation poses with a huge memory save, by only storing the position and weight of these targets, decreasing memory consumption compared to a completely different animation.

- Active Ragdoll

The common use of this system is not useful due to various reasons. The character's joints are usually in the vast majority of cases of type hinge joints[54]. This deforms the character's mesh so the characters that use active ragdoll are usually non-anatomical characters and the movement is not natural, although it allows more freedom in other cases such as free arm movement to grasp things, for example.

Games of this style have a common visual type of movement, for example: Gang Beast[55] and Human: Fall Flat[56], both developed with Unity.

In order to try to apply this system new joints of type "Hinge Joint" were created in the arms of the character. This joint type connects two rigidbodies to make them move like they are connected by a hinge, useful to make pendulums for example. The idea was, with the character in total ragdoll, grab his hands to specific points located on the platform from which we want the character to hang by using these hinge joints.

[53] Physics with animation rigging: https://forum.unity.com/threads/animation-rigging-physic-constraint.678160/
[54] Hinge Joint: https://docs.unity3d.com/Manual/class-HingeJoint.html
[55] Gang Beast [Pc, Xbox One, Ps4] (2014). Sheffield, UK: Boneloaf Limited
[56] Human: Fall Flat [All] (2016). Los Cristianos, Santa Cruz de Tenerife: No Brakes Games

In this way the body could hang side by side and by using counter-movement forces, the lower body could be stabilized until a neutral pose was reached, where we could transition to the hanging idle animation.

The problem with these hinge joints is that they have to be connected to a rigidbody and cannot be deactivated, in case a rigidbody is not assigned, the connector grabs the world, a point in space.
An active connector implies that as soon as the character changes to the ragdoll state, it remains hooked, hanging on an invisible point, so two GameObjects were created with RigidBody which these new hinge connectors were hooked, we will refer to them as handlers. The only way to prevent hands from grabbing onto these objects hooking the character was to assign these objects as children of the hand itself, as shown in Figure18, making the connector and the connected object move at the same time allowing the ragdoll to move naturally.

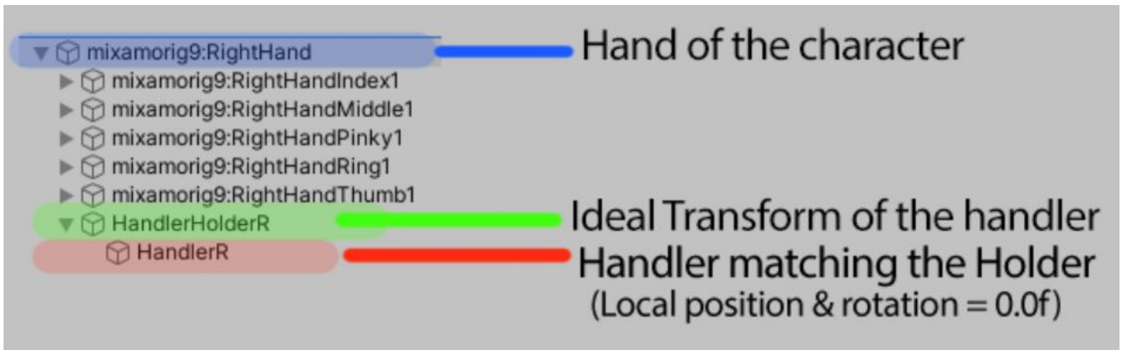

mixamorig9:RightHand
mixamorig9:RightHandIndex1
mixamorig9:RightHandMiddle1
mixamorig9:RightHandPinky1
mixamorig9:RightHandRing1
mixamorig9:RightHandThumb1
HandlerHolderR
HandlerR
Hand of the character
Ideal Transform of the handler
Handler matching the Holder
(Local position & rotation = 0.0f)

Figure18. Handler and holder of the new connector restructured in hierarchy.

The idea was, when we jump onto a platform, to assign an animation point where the character hooks onto the platform, unparenting those handlers from the hand and attaching them to the platform at certain points to force the ragdoll to grab onto the platform.

A GameObject "holder" was created in each hand to store the initial transform of his handler, this allows the handler to regain his transform after being detached from the hand to be placed on the grasping platform.

We need to maintain an ideal transform of the handler to ensure hands rotation will be ideal when we unparent the handlers and try to simulate the player is hanging on a platform.

As will be explained later, the dynamic ragdoll system allows the character to go from the physical state to animation modifying and rotating different parts of the body in procedural animation. This system needs to differentiate these holders in order to avoid rotating them and break the transform ideal reference point for the handlers. Certain tags were used in order to avoid a determined Rigid Body being affected by the Dynamic Ragdoll system.

A script was created, which is still in the project although it is no longer in use, it is called `RagdollController`.cs. It restores the handlers to match the holders transform and would be responsible for force management to the rigidbody to stabilize it.

At this point, the system could transition to ragdoll allowing it to give the feeling of grabbing onto a ledge and by forcibly transitioning back to animation state was able to restore those handlers in order to repeat the process.

The movement was not natural due to axis limit restriction, so another try was made using character joints instead, allowing us to configure and restrict each angle limit. Result was a little bit disappointing, the character hung unnaturally and, like the hinge joint system, if this system was activated in a fall with a certain speed, the grip broke the ragdoll. Visually, it was just a dead body hanging from his hands, a system without any animation has a huge visual difference with Uncharted, thanks to animation we obtain small visual clues that give realism to the system, such as the head looking where the hands are grasped, the arms making force to avoid falling or the effort of trying to correct the legs by hip bumps, visual feedback that a dead body doesn't give.

Also, having another joint in the arms broke the free ragdoll. By having this new connector or handler related to the object with which it moves, it should not be affected by anything else, but the fact of activating the physics of these body parts made a reaction in both joins (character body parts joint and new climb hanging joint) so animation to ragdoll transition caused the arms to break for a moment making this transition very noticeable, which is intended to be subtle. Since the use of these connectors cannot be disabled and the fact of trying to deceive them causes system failures, it was decided to finally discard this method.

- AnimFollow

It is a system invented by a developer[57] that until not long ago was paid. This system serves to realistically transition between animation and ragdoll, which in this project we designate as one of the two sections of dynamic ragdoll. The operation is based on deceiving Unity through the use of two characters. One of these characters will always be animated and the other in ragdoll. The animated character is not visible to the player and serves as a reference guide for the ragdoll, which replicates it. With this system we can, in a collision event or fall, for example, stop replicating this system, and the visible transition between a character lively and the ragdoll is as natural as possible since in practice there is not.

The code has an extraordinary complexity that was beyond the knowledge acquired even after research and part of the fundamental implementation already developed. The code uses these configurable joints to apply a calculated torque to the different components of the body so that it does not fall and perfectly imitates the animated system. All these body parts are internally in full use of physics and gravity, the complexity of calculating and using these different forces is enormous.

During the project implementation, after having carried out research and tests with the other systems described, this code was analyzed, trying to understand its structure and operation, this gave rise to an interesting theory by which this system could be used to recreate Physics Animation.

[57] AnimFollow creator: http://www.kavorka-racing.com/

It was decided to take some extra investigation time trying to replicate this system in a simple way. Our system uses a single character that is the one that alternates between animation and physics and is in charge of managing and controlling the other systems, such as parkour or climbing. AnimFollow has a master, the animated character, and a slave, the character with a ragdoll, so an attempt was made to replicate the system differently and easily. The system tested consisted of just the opposite, a second character was created that would be the slave which has an animator who imitates our character ones (This second character is still present in the project).

In this test, the main animator, which is in the master character (unlike AnimFollow), continues to work as scheduled, and the slave system is an animator that mimics the main one. This system is shown in Figure19.

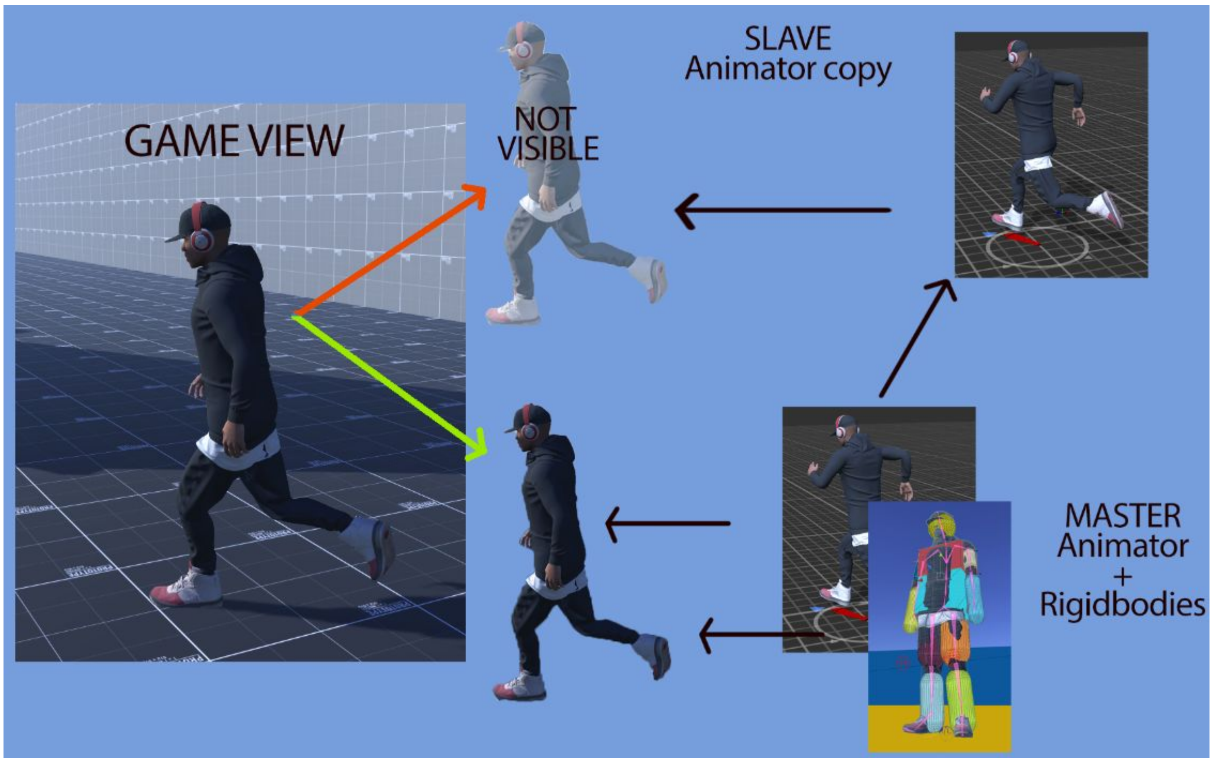


Figure19. Simplified two characters system implementation.

The idea is for both characters and animators to work synchronously, so if the main character goes into ragdoll state, the second animator will invisibly continue to make the animation. Both animators have the same structure and animations, but by code we are only modifying the parameters of one of them, that of the master. The `AnimatorMimic.cs` script is responsible for copying these parameters from one animator to another, as can be seen in Figure20.

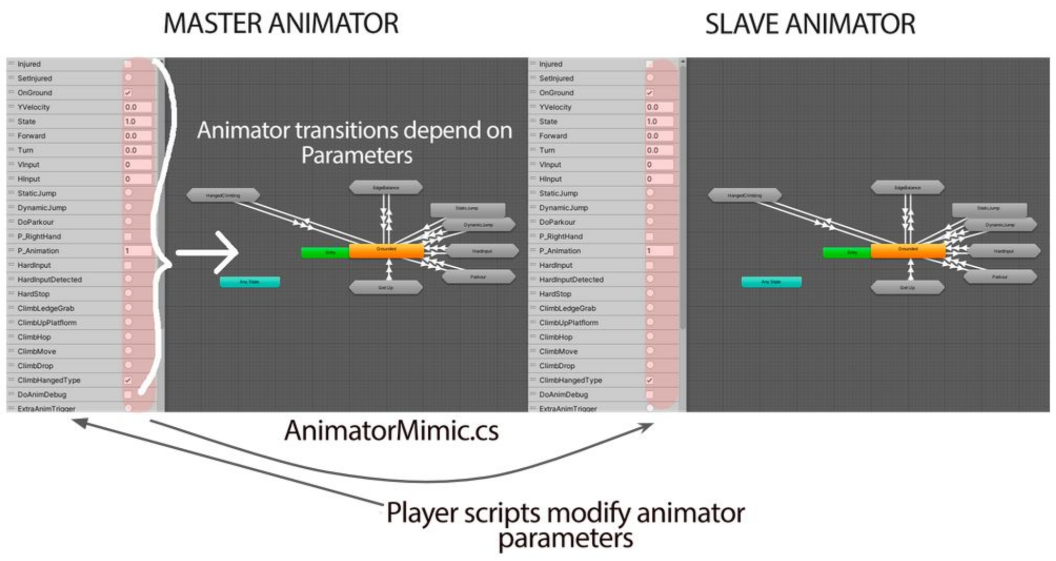

MASTER ANIMATOR
SLAVE ANIMATOR
Animator transitions depend on Parameters
AnimatorMimic.cs
Player scripts modify animator parameters

Figure20. AnimatorMimic example function.

For triggers, their call had to be located in order to call the same trigger consecutively in the second animator. This whole system was just a test, since in reality we would only need the second animator to do certain animations to be able to replicate with ragdoll, not all at all times.

The idea behind this system was to store the orientation and position transforms of the animated body parts and apply them to the body parts with ragdoll by using a lerp of these values with a weight factor in which zero means pure ragdoll and one means complete imitation. Due to the character's gravity and character joint union, the different parts of the body deformed with values less than one, and this system only managed to replicate the animator by brute force and rewriting the position values, so it was not going to nowhere.

After covering the section on handling both at the same time, it is time to talk about how the transition between these two systems has been implemented in a realistic and natural way through the use of procedural animation.

4.1.4.2. Dynamic Ragdoll

The non-procedural transition system between states (mecanim-> ragdoll & ragdoll-> mecanim) had been created, following research in different forums. The operation of these transitions has been called dynamic ragdoll. The most common use of ragdoll and animation in Unity is to deactivate the animator and activate the character's physics, removing its kinematic[58] state ragdoll components. When you want to return to animation, rigidbodies need to be deactivated by turning on their kinematic state and the animator.

- Ragdoll to Animation

At this point the basic transition system of change has been created, which alternates suddenly. All the sources and lines of research to improve this transition lead to the same site, an open source from 2013 in which the creator Perttu Hämäläinen wrote a blog[59] in which he showed his new system, this has been the point of Unity reference for all projects and solutions found during the research process.

It took approximately five hours to understand this complex code. This system allows by means of procedural animation to create a transition between a final position of the ragdoll and the corresponding animation to be raised. The explanation of this complex system has been commented and explained in the DynamicRagdoll script itself throughout the project, it will not be defined in detail in this report since it is not a code made by the student and would require a lot of space and time, but it will be explained in broad strokes.

At the point where this blending is activated, the system stores the position information of the ragdoll, calculates whether the character has to stand up from his back or his belly and activates the animator with this animation. The system is divided into two parts, one in which we will calculate the new root of the transform and another in which we will perform the blending of the ragdoll to the mechanism.

When we are in ragdoll, the position of the body moves away and the transform is separated from the character since it is physically independent as shown in Figure21.

[58] Kinematic: https://docs.unity3d.com/ScriptReference/Rigidbody-isKinematic.html
[59] Dynamic Ragdoll: http://perttuh.blogspot.com/2013/10/unity-mecanim-and-ragdolls.html

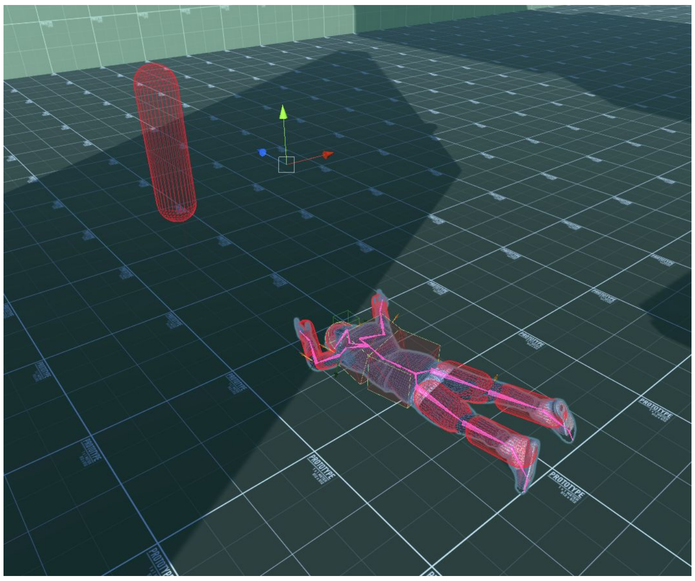


Figure21**.** Transform displacement in ragdoll state example.

This transform cannot be adjusted again to the character's mesh simply because it is the father, and any movement modification that is made will move the child with it. The system is in charge of making these modifications in the character's transformation without affecting the body's transformation thanks to an interpolation system. In addition, the ground clearance and body rotation are calculated and the new transform is placed in the best way possible. Adjusting the height with the ground serves to match the animation in the best way when getting up. While this calculation happens, in the first part of this blending, the animator is active, performing the beginning of the animation internally, so the animator tries to overwrite the ragdoll, but it stays in position ignoring the animator thanks to this interpolation. This blending system ignores the animator and parent modifications because he rewrites the position and rotation of the ragdoll at each iteration to coincide with the one stored when starting the blend. Once the transform has been recalculated, we move on to the second part, blending between poses. This blending is constantly active in a mathematically intelligent way to solve the problems that we have just explained. It smoothly transitions by using Lerp between the position of the bones where the ragdoll is located and the current position of the same bones in the animation that is internally happening.

This system has been modified and adapted for the project, working together with the rest of the mechanics and controlling the turning aid or the state of the character for example. An intermediate state has been defined between the ragdoll and the blending called `CharacterState.waitingForStablePosition`, in this state, the script will calculate when the ragdoll stops, and it will start a small timer before going to transition state. This automatically lifts the character after a fall making sure the character has come to a complete stop. For the determination of this movement, the speed of the rigidbody of the central hip bone of the character (on which all depend) is used.

As described in the HardInput section, an engine error caused the animation system to have to work on Physics Update, the only error in this new type of update in which the animator and the physics system update at the same time. was the blending between ragdoll and mecanim, performed in a LateUpdate and with visible stumbles with this update system. To correct that, the animator's update status changes to normal while Dynamic Ragdoll performs this blending, and when it ends, the animator is reconfigured again in Physics Update mode.

- ### Animation to Ragdoll

Having a character in an animated state implies that the physics are not calculating, if we left the components of the body as static they would accumulate a physical force due to gravity that would cause the moment we activate the ragdoll state the character will behave as if it has been falling for a long time. On the contrary, keeping the body parts kinematic implies that at the moment of transitioning to ragdoll the body will begin to drop from zero, regardless of the movement it is making. This, for example, causes a character in free fall with its respective animation, to activate the ragdoll state and then seeing how her body suddenly behaves as if activated from the ground.

As previously explained, it was decided not to follow the AnimFollow example and try to implement an acceptable transition system from a single character. Tests were carried out with the slave animator with the idea of making a transition using that influence weight. The ragdoll is supposed to imitate the transforms of the animated character as previously defined causing that weight to smoothly decrease, gradually transitioning from animation to ragdoll, but as already explained, this system deformed the character's mesh and was not realistic at all.

The speed of the character was calculated just before the transition to ragdoll, the idea was to use it to apply it to the character once in ragdoll. It is worth mentioning that there was no defined system for ragdoll activation, so most of the project tests carried out at first have been through the use of commands and keys to control states or force situations.

The first attempt was, once the ragdoll was activated, to apply this thrust speed to the different parts of the body for a short period of time by using AddForce[60] to the Rigidbody of the components. The result was similar to a push, nothing was visually good since continuity of movement was sought.

The second approach was, instead of applying this force for a time, directly assigning the speed at which the character moved to the different rigid bodies, so that the limbs would move, but the result would not affect the movement and it would be much more continuous than in the previous case. This result was indeed much more realistic than the previous one. The basic difference of this system to the AnimFollow's complex is that we calculate the speed of the character and apply it to the body when it goes to ragdoll, while AnimFollow calculates the rotation and speed of each animation bone and gradually applies it to the different ragdoll parts, transmitting the movement in a more linear way in the extremities or in certain poses.

[60] AddForce: https://docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html

The next thing was to determine when this system was intended to be used. The ragdoll will be activated automatically when a large drop occurs, this is verified in the FixedUpdate by activating the ragdoll using `SetRagdollForImpact`() if the Y speed of the character's rigidbody is greater than a certain value.

At a very advanced stage of development it was discovered that in the asset store there was a transition system from animation to ragdoll that was not indexed at any time by google during the research. Its code was implemented to compare procedures.

This system uses character joints, and a coroutine that reconfigures the anchor points in a strange way. Applied to our system, the transition was a little more linear and realistic, but when it came to reconfiguring the connectors, the boundaries remained broken, and the legs were bent or the arms were twisted, possibly due to using another character whose hierarchical structure is very different. In the project of this asset store package, we see how the transition to ragdoll is not remarkable but these limits are also broken. causing the limbs to unrealistically bend, perhaps due to the same problem or because it is not properly configured.

The use of force is the same, and this has been used as a small touch-up. In the final version of our system, speed was assigned directly as an immediate driving force (ForceType.Impulse), this project assigns the RigidBody the body speed directly as speed and not by force. The result is not remarkable, but it may require less physical cost.

This coroutine has not been implemented since it breaks the limits of the model despite trying to adjust it, nor has its logical system beyond the change in force system and some name of functions such as `GetUp`, `ActivateRagdollParts` or the status of ragdoll awaiting detection, that they were more precise or convenient than the previously designated names.

4.1.4.3. Impact System

This script is based on Perttu Hämäläinen's ragdoll system, in which a script is assigned to each part of the ragdoll during project start-up. The Perttu Hämäläinen’s code applies a force with the mouse cursor to launch the character, calculating its damage from impacts and showing it on the screen. The impact part has been reused, assigning a maximum damage threshold.

Each part of the body calculates its impact on a hit when we fall, adding the impact information to a counter controlled by the `Impact Ragdoll script`.
If the damage is greater than a given limit, he activates a boolean that will activate wounded up animations, these animations will activate the wounded layer when they finish using a state animator behavior.

This layer is controlled by GameManager and, when time comes out, it decreases the weight amount of the layer gradually, in order to visually recreate the recovery of the character.

4.2. Results

As described at the beginning of the ragdoll investigation section, it was decided more convenient to implement the use of a single character. For this reason, animFollow remains as a discarded reference source, and his use of the dynamic ragdoll will be tried to replicate by another method, although it would be less realistic. Physics Animation is complex and serves to give detail and realism to certain animations on an already built system, so when the potential use of AnimFollow to replicate this system was discovered, the state of the project implied a great risk of its implementation, compromising all the planning, since it escapes from the hours established as a limit and the priority above all was to have an advanced movement system that had to be finished. Also, this requires a lot of hours of extra engine documentation since most ragdolls use character joints and this system uses the complex configurable joints. This type of joint is completely adjustable and the other joints can be made with this. These joints, as indicated by the author of the package, require a lot of adjustment time, since each character requires a custom adjustment of values for each part of the body. AnimFollow is built over the Ethan Character from Unity's Standard Assets Package.

As for the research section, this new parallel hands-on research was limited to approximately fifteen more hours. Even so, numerous systems and theories have been discarded and a possible solution has been found for future work. After the revision of the code and internal operation of AnimFollow, although it has not been fully understood, this system could be used to adjust the distribution of forces so that we can leave a specific part of the body to the free use of physics and the joint can be configured in a very precise way to prevent the mesh from suffering deformations. In other words, it is determined that AnimFollow, with an adjustment of the code, its functionality and its parameters (according to the character in which it is applied), would be capable of replicating Physics Animation in the Unity engine.

It has been a bit daunting to come across issues during the project caused by the engine itself. During the development, Unity constantly broke, hindering the development but without considerably affecting the project thanks to the recurring saving and different backups both in Unity Collab and in Github, even so, it has been infuriating on certain occasions. In addition, different problems have arisen such as the last-minute malfunction of Cinemachine or the bug that varied the rootMotion of certain animations depending on the size of the screen.

The initial planning clearly differentiated the development of animation and code, the final planning is more similar to that provided in the design document, it has diversified in functionality since the animation system is something to be developed in parallel throughout the project.

Parkour, jumping and climbing have been developed in the advanced movement section (which we could call agility), since they affect gameplay much more directly, and the use of hardInput or ground movement of the character is considered basic movement., since it does not imply agility.

The development of the project can follow these sections in an orderly way, but it is convenient that the ragdoll section is developed jointly during the project. It was intended that hard input also reacts to walls, an extra option in order to stop the character if we were running toward one in determined situations, but there was no time for that due to the rootMotion's Unity problem that forced us to redistribute time.

-Research(65)
-SetUp(6)
-Implementation (180)

1. Ragdoll Set Up: (15)
    a. Configure
    b. Testing
    c. Basic mecanim transition system
2. Basic Behaviour (40)
    a. Standard movement (10)
    b. Hard Input (30)
        i. Detection
        ii. Reaction
3. Advanced Movement (90)
    a. Agility System (65)
        i. Parkour (25)
        ii. Jump system (10)
        iii. Climbing (30)
    b. Extra Detail (25)
        i. Ledge detection
        ii. Inverse Kinematics
        iii. Animator extra layers
        iv. Injured state
4. Advanced Ragdoll (35)
    a. Impact System (5)
    b. Dynamic ragdoll (30)
        i. Ragdoll to Mecanim
        ii. Mecanim to Ragdoll

This would be a diagram of the final planning structure of the project, highlighting in green the sections in which less hours have been invested than expected and in red in which it has been invested more. In it you can see how the basic behavior suffers from a greater demand for hours, they were destined to fix the conflict described with the use of RootMotion in the state of HardInput.

Due to the final simplicity of the IK system to react to the stairs since there were no suitable animations for it, this system took just one hour of implementation and another hour of adjustment, much less than expected, so it has been added as a detail minor on the scene. Edge detection is without a doubt the most complex aspect within this extra level of detail, even so, the sum of these mechanics allowed us to free up some time.

The setUp section suffered another unforeseen event when redoing the camera, leaving aside the problematic cinemachine. This was quickly solved using a Standard Assets, which after its configuration is even better.

The second major charge is awarded for the fifteen extra hours of research. As you can see, this slightly affects the development of the agility system, although it is very little given that the impact system turns out to be easier than expected as well as not as much workload is needed in the extra details system. All those hours have been invested avoiding damaging the agility system as little as possible.

Finally, the objectives set apart from the research have been met, there are slight planning adjustments given the final simplicity of some and the complexity of others, but the most important thing has been the restructuring of planning into more concrete parts and more conveniently grouped.

A complex movement system has been created, which can be considered advanced given the complexity of the code, the number of control scripts working at the same time in a coordinated manner, the internal structure of the animator (including sub state machines, state behaviors and different layers) and number of animations and functionality. In addition, it has come up with an organic simulation system to be proud of, and above all, versatile, in which we can see how the character latches onto a nearby platform regardless of its height, performs different animations depending on the distance to an obstacle or support the hand that suits him best depending on the side he is approaching.

However, I am saddened by the result of certain elements such as movement in climbing or animation of climb up a platform due to the limitation of animations and the poor recording condition of the most crucial, although the result is positive to have been built from public animations.

5. CONCLUSIONS AND FUTURE WORK

5.1. Conclusions

It is not worth implementing a system like Physics Animation in Unity as long as you have a limited time, unless you have previous knowledge, since it is still a level of detail that, although it surely takes much more advantage of a system with custom animations, would end up being applied in a simulated way by constantly having to avoid the limitation of this engine.

AnimFollow, which was not preconceived for this purpose, is a system that requires a lot of workload and an exhaustive knowledge of the physics and configuration of connectors but, with a good custom animation system, is the only tool capable of giving results very similar to what was seen in Uncharted 4.

It is a shame to have stayed so relatively close to having managed to replicate this system, but this was an investigative section, and I am happy to have discovered the way to cheat such an important engine as Unity, opening the possibility of implementing one of the world's most advanced procedural animation systems to date.

As for the rest of the project, much has been learned and discovered about the implementation of different systems and how to build a complex animation system using very useful animator tools.

Personally, I am comforted to have recreated, from practice and originality, two movement systems such as the use of parkour and climbing, since they are very complex systems to elaborate that many great games have tried to carry out awkwardly. I have learned so much about the animation implementation in Unity, having a stable animator made up of approximately eighty animations coordinated by a system made up of so many control scripts.

I have always been curious about procedural animation, and this project has given me in-depth knowledge about this type of animation and the difficulty of implementing a complex animation system, apart from significantly improving my ability of programming in Unity, going on to dominate tools that, until now, tried to avoid, such as the use of Raycast, interpolation of position or rotation by Lerping, or the use of Joints.

5.2. Future work

This work can be continued in multiple ways. Parkour and climbing are two systems on which to add infinite layers of mobility and complexity.

The readjustment of hours caused by AnimFollow's research did not significantly harm the climbing system since it was quite advanced and did not have a defined functionality goal, the intention was to improve it as much as possible within hours. All this functionality can be applied in the future development of the project since only with climbing can side detectors be improved just like vertical climbing, making a character drop and grab the ledge at his feet, scale walls with different inclination, place your feet, get tired, take corners, etc. The possibilities are endless

Without the burden of time, the character can be remade by using configurable joints and animFollow can be used to replicate Naughty Dog's Physics Animation, although as long as you don't have custom animations it won't be able to look the way the original does. .

Animation Rigging is a new tool that, although it has not found its usefulness in this project, has a lot of future potential and, according to the Unity forums, the developers are working to give this new system physics support, who knows that new discoveries can be born from there if it is carried out. The problem with Animation Rigging is that it hardly has documentation and there is almost no information, there is an advanced tutorial for its use but it is found in Unity Pro, the paid version of the platform. I hope that, in the future, more people will learn to use this tool , sharing his experience with others.

The Mixamo library contains thousands of animations that can be used in some way to add extra functionality to the character.

6. BIBLIOGRAPHY

INTRODUCTION

2. (n.d.). Game Developers Conference (GDC). Retrieved June 10, 2020, from https://gdconf.com/
3. Physics Animation in Uncharted 4: A Thief's End: https://youtu.be/7S-_vuoKgR4
4. (n.d.). Naughty Dog. Retrieved June 10, 2020, from https://www.naughtydog.com/
5. Uncharted 4. Neil Druckmann, Bruce Straley [PlayStation 4] (2016). Santa Monica, California: Naughty Dog, Inc.

PLANIFICATION

6. (n.d.). System requirements for Unity 2019.4 - Unity - Manual. Retrieved June 10, 2020, from https://docs.unity3d.com/Manual/system-requirements.html

ANALYSIS

7. (n.d.). Mecanim Animation System - Unity - Manual. Retrieved June 10, 2020, from https://docs.unity3d.com/462/Documentation/Manual/MecanimAnimationSystem.html
8. (n.d.). Ragdoll Wizard - Unity - Manual. Retrieved June 10, 2020, from https://docs.unity3d.com/Manual/wizard-RagdollWizard.html
9. (n.d.). Animation Rigging | Animation Rigging | 0.2.6-preview - Unity .... Retrieved June 10, 2020, from https://docs.unity3d.com/Packages/com.unity.animation.rigging@0.2/index.html
11. (n.d.). Scripting API: Physics.Raycast - Unity. Retrieved June 10, 2020, from https://docs.unity3d.com/ScriptReference/Physics.Raycast.html
12. (n.d.). Unity - Manual: Coroutines. Retrieved June 10, 2020, from https://docs.unity3d.com/Manual/Coroutines.html
13. (n.d.). Linear interpolation - Wikipedia. Retrieved June 10, 2020, from https://en.wikipedia.org/wiki/Linear_interpolation
14. (n.d.). Scripting API: Mathf.Lerp - Unity. Retrieved June 10, 2020, from https://docs.unity3d.com/ScriptReference/Mathf.Lerp.html
15. (n.d.). Scripting API: Vector3.Lerp - Unity. Retrieved June 14, 2020, from https://docs.unity3d.com/ScriptReference/Vector3.Lerp.html
16. (n.d.). Scripting API: Quaternion.Lerp - Unity. Retrieved June 10, 2020, from https://docs.unity3d.com/ScriptReference/Quaternion.Lerp.html
17. (n.d.). Inverse Kinematics - Unity - Manual. Retrieved June 10, 2020, from https://docs.unity3d.com/Manual/InverseKinematics.html
18. (n.d.). Using Animation Events - Unity - Manual. Retrieved June 10, 2020, from https://docs.unity3d.com/540/Documentation/Manual/animeditor-AnimationEvents.html
19. (n.d.). Scripting API: StateMachineBehaviour - Unity. Retrieved June 10, 2020, from https://docs.unity3d.com/ScriptReference/StateMachineBehaviour.html
20. (n.d.). Animator Controller - Unity - Manual. Retrieved June 10, 2020, from https://docs.unity3d.com/Manual/class-AnimatorController.html
21. (n.d.). Unity - Manual: Animation Layers. Retrieved June 10, 2020, from https://docs.unity3d.com/Manual/AnimationLayers.html
22. (n.d.). Sub-State Machines - Unity - Manual. Retrieved June 10, 2020, from https://docs.unity3d.com/Manual/NestedStateMachines.html
23. (n.d.). Mixamo. Retrieved June 10, 2020, from https://www.mixamo.com/

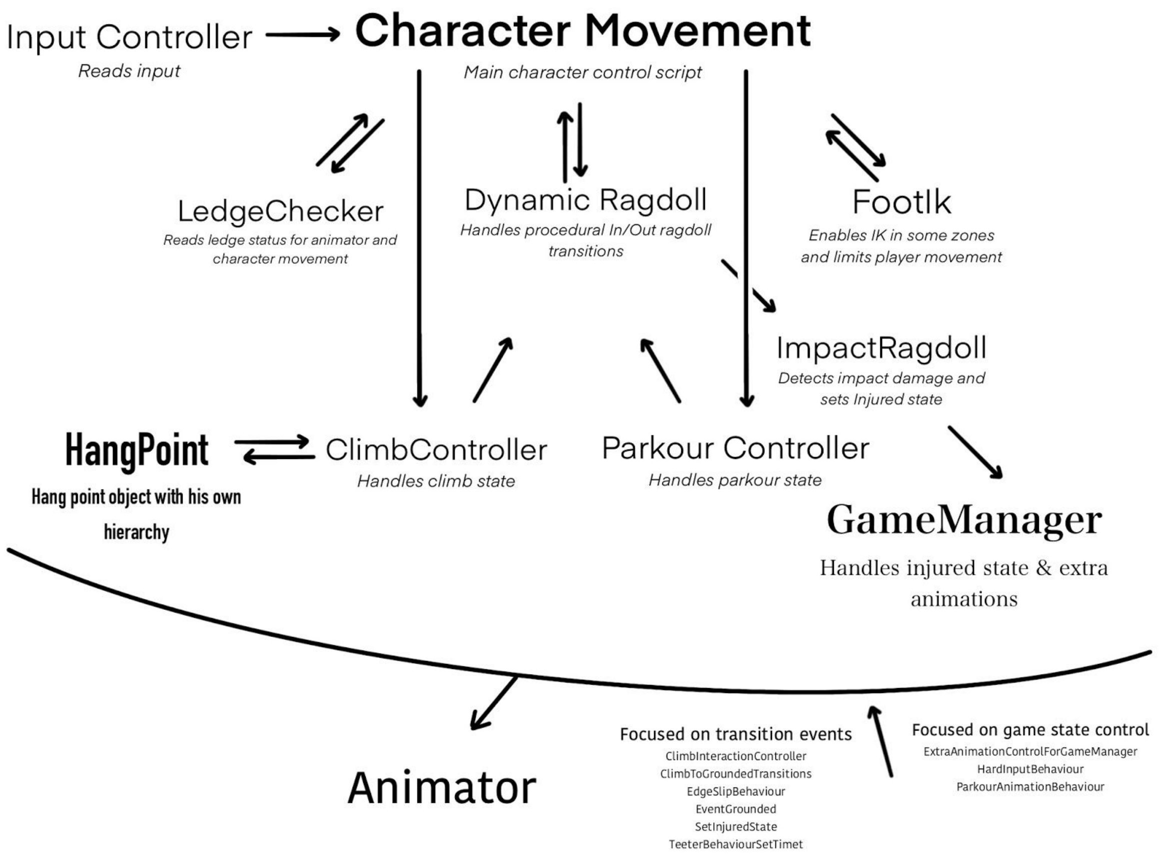
IMPLEMENTATION

24. (n.d.). About Cinemachine | Package Manager UI website. Retrieved June 10, 2020, from https://docs.unity3d.com/Packages/com.unity.cinemachine@2.1/index.html
25. (n.d.). Standard Assets (for Unity 2017.3) | Asset Packs | Unity Asset Retrieved June 10, 2020, from https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-for-unity-2017-3-32351
26. (n.d.). ProBuilder - Unity. Retrieved June 10, 2020, from https://unity3d.com/unity/features/worldbuilding/probuilder
27. (2018, October 3). Gridbox Prototype Materials | 2D Textures & Materials | Unity .... Retrieved June 10, 2020, from https://assetstore.unity.com/packages/2d/textures-materials/gridbox-prototype-materials-129127
28. https://answers.unity.com/questions/1350081/xbox-one-controller-mapping-solved.html
30. (n.d.). Scripting API: Collider - Unity. Retrieved June 10, 2020, from https://docs.unity3d.com/ScriptReference/Collider.html
31. (n.d.). Character Joint - Unity - Manual. Retrieved June 10, 2020, from https://docs.unity3d.com/Manual/class-CharacterJoint.html
32. (n.d.). Scripting API: Rigidbody - Unity. Retrieved June 10, 2020, from https://docs.unity3d.com/ScriptReference/Rigidbody.html
33. (n.d.). Unity - Manual: Joints. Retrieved June 10, 2020, from https://docs.unity3d.com/Manual/Joints.html
34. (n.d.). Physic Material - Unity - Manual. Retrieved June 10, 2020, from https://docs.unity3d.com/Manual/class-PhysicMaterial.html
35. (n.d.). MonoBehaviour.FixedUpdate() - Unity - Manual. Retrieved June 12, 2020, from https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html
36. (n.d.). Scripting API: MonoBehaviour.Update() - Unity. Retrieved June 12, 2020, from https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html
37. (n.d.). Root Motion - how it works - Unity - Manual. Retrieved June 12, 2020, from https://docs.unity3d.com/Manual/RootMotion.html
38. (n.d.). Scripting API: Animator.applyRootMotion - Unity. Retrieved June 12, 2020, from https://docs.unity3d.com/ScriptReference/Animator-applyRootMotion.html
39. (n.d.). Scripting API: AnimationClip - Unity. Retrieved June 12, 2020, from https://docs.unity3d.com/ScriptReference/AnimationClip.html
40. RootMotion problems: https://answers.unity.com/questions/922181/sprite-animating-backwards-and-slower-on-maximize.html
41. https://answers.unity.com/questions/737158/animation-plays-in-preview-mode-but-not-max-screen-1.html
42. https://answers.unity.com/questions/313468/player-animations-not-working-when-playing-scene-i.html
43. (n.d.). Scripting API: AnimatorUpdateMode - Unity. Retrieved June 13, 2020, from https://docs.unity3d.com/ScriptReference/AnimatorUpdateMode.html
45. (n.d.). Tags - Unity - Manual. Retrieved June 13, 2020, from https://docs.unity3d.com/Manual/Tags.html
46. (n.d.). Scripting API: Mathf - Unity. Retrieved June 13, 2020, from https://docs.unity3d.com/ScriptReference/Mathf.html
47. (n.d.). Scripting API: Mathf.Deg2Rad - Unity. Retrieved June 13, 2020, from https://docs.unity3d.com/ScriptReference/Mathf.Deg2Rad.html
48. (n.d.). Scripting API: Vector3.Angle - Unity. Retrieved June 13, 2020, from https://docs.unity3d.com/ScriptReference/Vector3.Angle.html
49. (n.d.). Scripting API: ForceMode.Impulse - Unity. Retrieved June 13, 2020, from https://docs.unity3d.com/ScriptReference/ForceMode.Impulse.html
51. https://www.youtube.com/watch?v=MonxKdgxi2w
53. https://forum.unity.com/threads/animation-rigging-physic-constraint.678160/

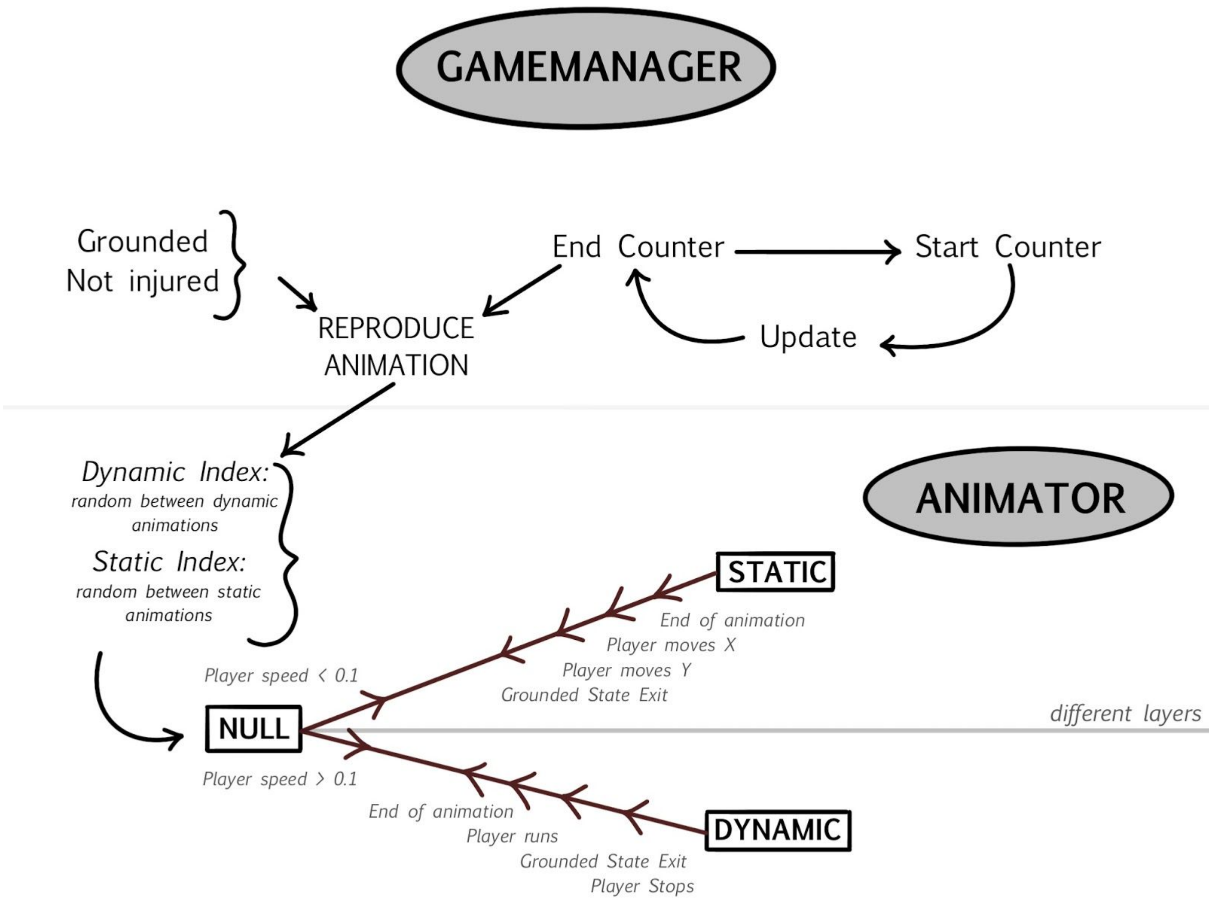
54. (n.d.). Hinge Joint - Unity - Manual. Retrieved June 13, 2020, from https://docs.unity3d.com/Manual/class-HingeJoint.html
55. Gang Beast [Pc, Xbox One, Ps4] (2014). Sheffield, UK: Boneloaf Limited
56. Human: Fall Flat [All] (2016). Los Cristianos, Santa Cruz de Tenerife: No Brakes Games
57. AnimFollow creator: http://www.kavorka-racing.com/
58. (n.d.). Scripting API: Rigidbody.isKinematic - Unity. Retrieved June 13, 2020, from https://docs.unity3d.com/ScriptReference/Rigidbody-isKinematic.html
59. http://perttuh.blogspot.com/2013/10/unity-mecanim-and-ragdolls.html
60. (n.d.). Scripting API: Rigidbody.AddForce - Unity. Retrieved June 13, 2020, from https://docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html

7. ANNEXED

Link to Github Repository: https://github.com/al361691/TFGAdvancedMovementSystem


Input Controller
Reads input
Character Movement
Main character control script
LedgeChecker
Reads ledge status for animator and character movement
Dynamic Ragdoll
Handles procedural In/Out ragdoll transitions
FootIk
Enables IK in some zones and limits player movement
ImpactRagdoll
Detects impact damage and sets Injured state
HangPoint
Hang point object with his own hierarchy
ClimbController
Handles climb state
Parkour Controller
Handles parkour state
GameManager
Handles injured state & extra animations
Animator
Focused on transition events
ClimbInteractionController
ClimbToGroundedTransitions
EdgeSlipBehaviour
EventGrounded
SetInjuredState
TeeterBehaviourSetTimet
Focused on game state control
ExtraAnimationControlForGameManager
HardInputBehaviour
ParkourAnimationBehaviour

A1. Different scripts on the project and an overall preview about how they communicate between them and how everything is connected to the animator.



A2. Script & Animator communication workflow basic example: Setting an extra animation from code using conditions and a time counter and evaluating the rest of conditions directly from animator, including which state we reproduce and when we come back to null state (not extra animation is reproduced).