Implementation of Video Game Character Visual Effects Final Degree Work Report

Miguel Ferrer Carrasco

Final Degree Work
Bachelor's Degree in
Video Game Design and Development

Universitat Jaume I

May 25, 2022

Supervised by: Carlos Marín Lora

To David Salinas, Víctor, Pablo, David Aguilar, Toño, Irene, Adri, Issa, Gonzalo and Juanan. The ones that made me feel loved and appreciated, which supported me through the toughest moments and help become a better person every day.

Acknowledgments

I would like to thank my Final Degree Work supervisor, Carlos Marín, for supporting me and trusting in my work, motivating me to keep up. His guidance has been essential from the conception to the outcome of this project

Abstract

This document contains the report of the Video Game Design and Development Final Degree Project by Miguel Ferrer Carrasco. It consists of the development and implementation of six different stylized video game visual effects (VFX) in the game engine Unity 3D by using both procedural and handmade techniques. The effects implemented in this work are intended to serve as working assets for an action-RPG or fighting 3D video game, being reproducible and parameterized for their use in other projects.

Keywords

VFX, Shader, Visual Effect, Game Development, Game Asset, Unity

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

Contents

In this chapter, the motivations and purpose of this project are displayed, and where did the idea come from.

1.1 Work Motivation

Visual effects (VFX) are an essential element for video games, as they provide visual feedback to the player, connecting actions with results in a smooth way, and also upgrade the visual appeal of the game, whether making it more realistic or adding an extra stylized look to the art style.

While VFX in filmmaking refers to any computer-generated graphics integrated with live-action footage, in the video game industry it is a term used to describe anything in-game that moves that isn't a character or an object (e.g. dust, explosions, rain, fire, etc).

So, as *Screen Skills* puts it down in *What's a VFX artist good at?* [4], a VFX artist employs art, since they need to understand composition, color, texture, and light, physics, as they have to recreate how elements from nature like fluids, light, or particles behave, and computer science, to program the effects, optimize rendering and manage the software environments.

The goal of this project is to show the workflow behind the creation of visual effects, and how the elements described above are used in the process.

1.2 Objectives

These are the objectives set for the project based on the motivations of the work:

- Understand the different tools used in Unity for VFX.
- Create shaders by using Unity Shader Graph.
- Create visual effects by using both Particle System and VFX Graph.
- Produce professional-looking Visual Effects, which could actually be included in a video game.
- Use both procedural and traditional ways to create patterns for the VFX.
- Develop an interactive demo showcase featuring the results.

1.3 Related Subjects

A list of the subjects that are related to this project:

- VJ1221 - Computer Graphics
- VJ1223 - Video Game Art
- VJ1226 - Video Game Characters and Animation
- VJ1227 - Game Engines

1.4 Environment and Initial State

The interesting thing in visual effects is that belongs both in art and programming. In the Game Design & Development degree, students learn about many aspects of video game development, but the industry often requires specific profiles of developers. Even within general fields such as art, scripting, or design, there are several specializations, for instance, artists can be modelers, concept artists, UI designers, and so on.

It is not easy to choose one field, if none of them are especially easy for you, but there is a role in game development that requires to manage both art and programming, which is called technical artist. A person that creates visual effects by combining his own art with computer graphics techniques, developing shaders, animations, and understanding the game engine environment to produce wonderful visuals and make a huge impact on the liveliness of a video game.

So this project is meant to be a starting point as a technical artist and shall serve as a portfolio. The structure of this project took shape after the first meetings with the supervisor, Carlos, who, after explaining the motivations, suggested to approach this project as if creating the VFX were created for an existing game in development, taking supposed characters and giving them abilities that required adding effects. And the final idea was to use, not characters as such, but typical RPG classes found in many games, which made it easier for to find references and have more versatility for the design.

Also, these VFX were designed around animations from mixamo, which were chosen as the ones being closer to the proposal.

Chapter 2

Planning and resources evaluation

Contents

This chapter displays the plannification of the work and the resources and tools used throughout the project.

2.1 Planning

The planning for this project was adjusted to the graphic shown in Figure 2.1 below, dividing the 300-hour span by 8 stages of the development. This stages feature:

1. **40 hours:** Researching sources for the creation of the visual effects and looking for art references.

2. **5 hours:** Setting up the Unity project, and importing the necessary packages and assets.

3. **20 hours:** Designing the 6 different visual effects over the character animations from *mixamo*.

4. **120 hours:** Implementing the actual VFX in Unity and other software tool used for the project.

5. **30 hours:** Assembling the animations in different scenes for the final demo application.

6. **50 hours:** Reporting the workflow.
7. **25 hours:** Writing and fitting the report in this paper.
8. **10 hours:** Preparing the presentation and video demonstration for the defense of the work.

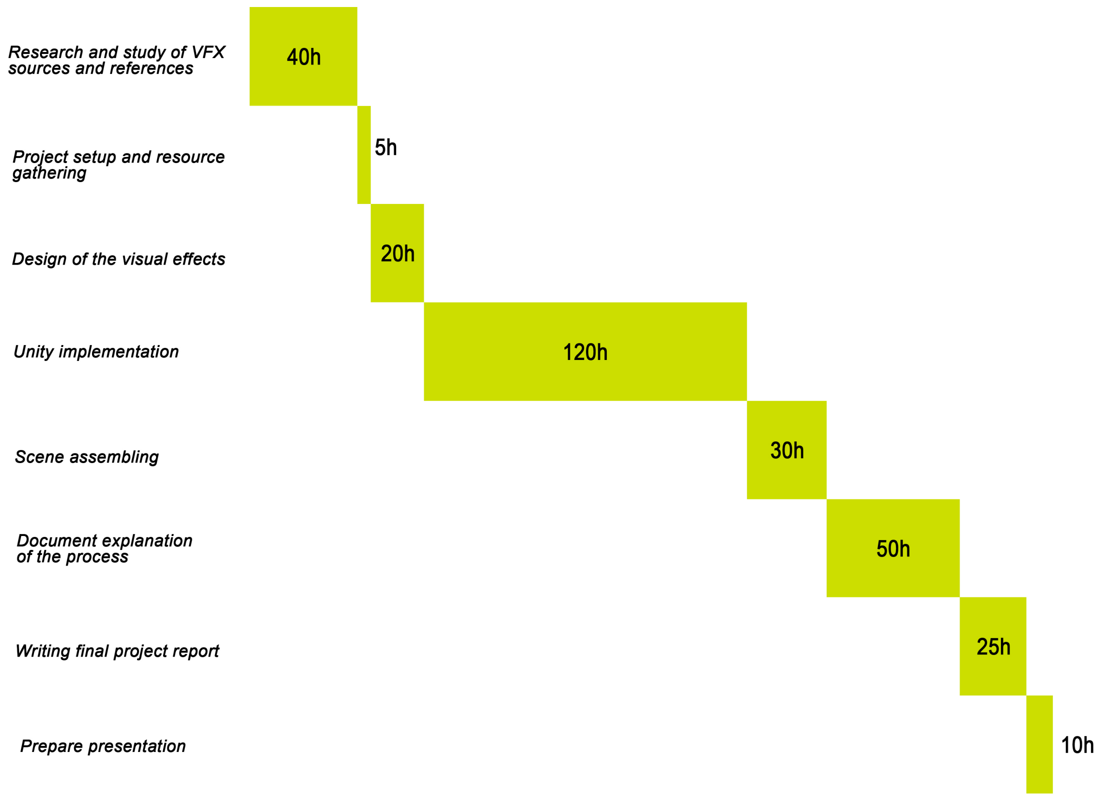


Figure 2.1: Final Degree Work hour planning

2.2 Resource Evaluation

This section contains a list of the hardware and software used for making this project:

- **Desktop Computer**: The PC used during the project
  - **Processor**: AMD Ryzen 7 5800X
  - **GPU**: GeForce RTX 3070 8GB
  - **RAM**: 32GB 3200Mhz

  - **OS**: Windows 10 Pro

- **Gaomon PD1560 Drawing Display**: The graphic tablet used to create the custom textures for the visual effects
- **Unity 3D**: Game engine used for the project. Version 2020.3.32f1
- **Visual Studio 2022**: Scripting tool that can be attached to Unity and facilitates programming with C#, the coding language used for this project.
- **Blender**: Free modeling software used to create meshes and textures.
- **Adobe Photoshop Portable**: Picture editing tool used to create textures.

Chapter 3

System Analysis and Design

Contents

This chapter presents the requirements analysis, design and architecture of the proposed work, as well as the early artistic and flow designs for the visuals.

3.1 Requirement Analysis

This is the preliminary requirements analysis of the final demo application. In this section the functional and non-functional requirements of the presented work are listed.

3.1.1 Functional Requirements

A functional requirement defines a function of the system that is going to be developed.

- **R1.** The user can choose between two scenes for each character.
- **R2.** The user can go back to the main menu and quit the application.
- **R3.** Effects can be triggered both with mouse and keyboard.
- **R4.** The camera view can be controlled with the mouse.
- **R5.** The user can zoom in and out.

- **R6.** VFX suit the animations.
- **R7.** Animations must not be triggered simultaneously.

3.1.2 Non-functional Requirements

Non-functional requirements impose conditions on the design or implementation.

- **R8.** The UI must be simple to not distract the user from the VFX.
- **R9.** Controls and UI must be understandable and intuitive to the user.
- **R10.** The camera must move smoothly.
- **R11.** The art for the VFX shall have a stylized aesthetic.
- **R12.** The showcase scene must be clean and simple to keep the VFX as the main element.

3.2 System Design

This section presents the logical and operational design of the system to be carried out. To do the logical design is shown with a use case diagram (see Figure 3.1).

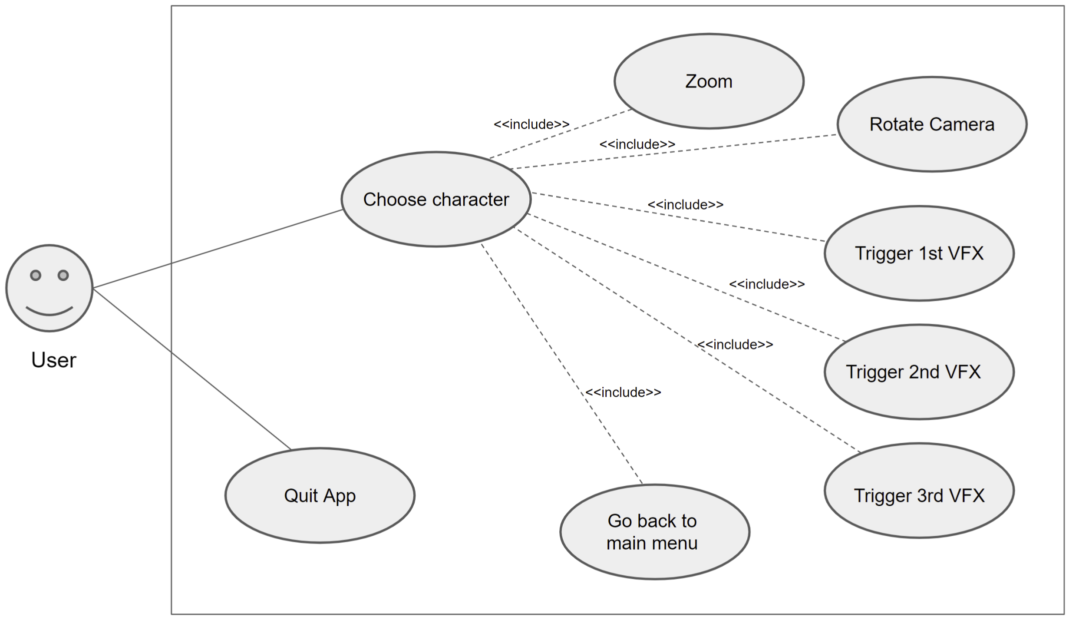


Figure 3.1: Use case diagram of the demo application

And the descriptions of these cases are here:

Requirements:	R1
Actor:	User
Description:	The user can click on the model of the mage or the warrior to select one and access each character scenes
Preconditions:	The user must be on the main menu
Steps normal sequence:	1. The user hover the mouse over one of the characters on the screen 2. The user right clicks on the character of choice
Alternative sequence:	None

Table 3.1: Functional requirement: Case of use «CU01. Choose character»

Requirements:	R2
Actor:	User
Description:	The user can close the application with the escape key
Preconditions:	The user must be on the main menu
Steps normal sequence:	The user presses the escape key
Alternative sequence:	None

Table 3.2: Functional requirement: Case of use «CU02. Quit App»

Requirements:	R5
Actor:	User
Description:	The user can zoom in and out on the character
Preconditions:	The user must be on one of the character scenes
Steps normal sequence:	The user scrolls the mouse wheel in the desire direction to zoom in or out
Alternative sequence:	None

Table 3.3: Functional requirement: Case of use «CU03. Zoom»

Requirements:	R4
Actor:	User
Description:	The user can rotate the camera view around the character
Preconditions:	The user must be on one of the character scenes
Steps normal sequence:	The user move the mouse in the direction desired to rotate around the character
Alternative sequence:	None

Table 3.4: Functional requirement: Case of use «CU04. Rotate Camera»

Requirements:	R3 & R7
Actor:	User
Description:	The user triggers the first animation
Preconditions:	1. The user must be on one of the character scenes 2. No other animation is showing
Steps normal sequence:	The user clicks the left mouse button
Alternative sequence:	The user presses the A key

Table 3.5: Functional requirement: Case of use «CU05. Trigger 1st VFX»

Requirements:	R3 & R7
Actor:	User
Description:	The user triggers the second animation
Preconditions:	1. The user must be on one of the character scenes 2. No other animation is showing
Steps normal sequence:	The user clicks the right mouse button
Alternative sequence:	The user presses the S key

Table 3.6: Functional requirement: Case of use «CU05. Trigger 2nd VFX»

Requirements:	R3 & R7
Actor:	User
Description:	The user triggers the third animation
Preconditions:	1. The user must be on one of the character scenes 2. No other animation is showing
Steps normal sequence:	The user presses the mouse wheel
Alternative sequence:	The user presses the D key

Table 3.7: Functional requirement: Case of use «CU05. Trigger 3rd VFX»

Requirements:	R2
Actor:	User
Description:	The user get back to the main menu screen from the character scenes
Preconditions:	The user must be on one of the character scenes
Steps normal sequence:	The user presses the escape key
Alternative sequence:	None

Table 3.8: Functional requirement: Case of use «CU08. Go back to main menu»

3.3 Artistic Designs

In this section, the initial designs for the VFX will be explained and illustrated over animation frames.

3.3.1 Warrior Abilities

Slash

A simple close-range attack consisting of a sword slash that leaves a flashing trail. This is a common VFX seen in video games with melee combat, as it represents the movement of a weapon when making a basic attack, specifically the blur that the human eye perceives with fast motions (see Figure 3.2).

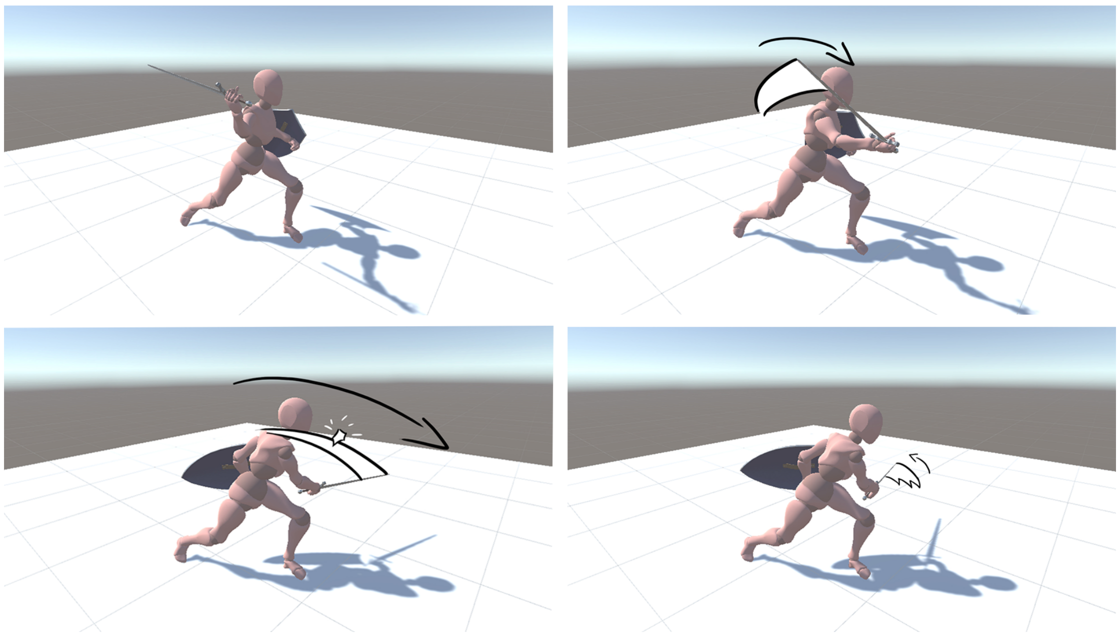


Figure 3.2: Slash concept over animation frames

There are many visual representations of sword trails in media, depending on the intensity and size of the slash. The main reference for a stylized result in this project is the slash seen in Japanese *manga*, a common technique to display dynamic sword combat scenes represented in a static format.

So, to represent a blade blur for a basic attack, using warm colors will be avoided, keeping the trail white with blue or light gray shading, matching the tone of the sword.

Shield Projection

For this ability, the warrior will use the shield for protection, casting a light projection of a shield, twice the size of the original, staying a few seconds right in front of the character, and finally going forward to vanish fading away (see Figure 3.3). This VFX is a graphic exaggeration for blocking with the shield and gives the player clear feedback on the character's defensive action and the range of effect.

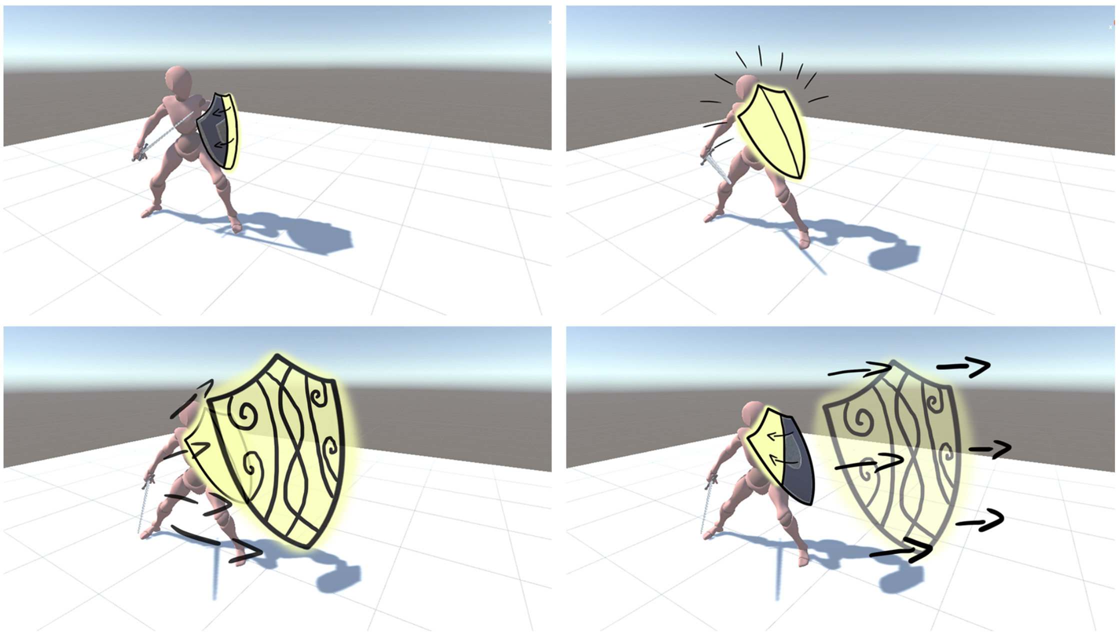


Figure 3.3: Shield Projection concept over animation frames

This VFX is less common than the previous one. In this case, the inspiration comes directly from the MMORPG *World of Warcraft*, specifically the paladin class skill *'Shield of the Righteous'* [5] where a pale light shield is cast in front of the character to attack.

Since this goes for a defensive approach, the shield should be accentuated, making it bigger, more outstanding, and more persistent, so it represents a barrier that protects the character while it lasts.

However, for the color scheme, the warm yellow spectral style from the reference will be maintained, as it reinforces the feeling of holy magic protection, with light as the source.

Ground Slam

A combination of three actions to make one impactful attack. First, a heavily charged sword slams against the floor, secondly, a dust cloud moving forward, and finally, rocks emerge from the cracked ground as if the impact created an earthquake (see Figure 3.4). This VFX represents a slow but powerful and heavy attack, so it is important to properly portray the force used when slamming the floor.

As for the rocks, the spawning has to be very well timed with the crack of the ground, because a sword producing this effect is completely unrealistic and the only way to make the correlation between these two events believable is to synchronize them with precision.

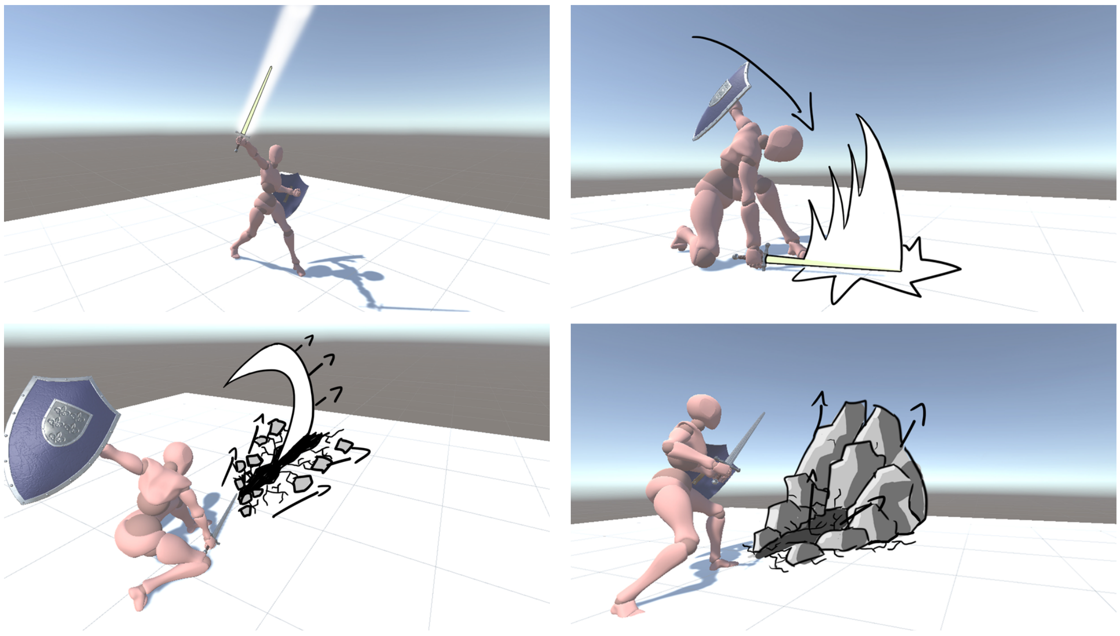


Figure 3.4: Ground Slam concept over animation frames

Heavy slams and ground cracking are a very widespread VFX in video games separately, but not really common as a single combo attack. The main reference for this sword strike comes from *Reinhardt*'s *Earthshatter* attack from *Overwatch*, even though he uses a hammer or a battle-ax, the motion and ground cracking work the same, even though it doesn't lift rocks from the ground.

In order to represent the tectonic nature of the attack, the colors used to match heated earth will vary from orange to brown, with whites for highlights and black/dark gray for contrast.

Let's breakdown the different parts of the VFX to set the artstyle:

1. Sword charging: An orange aura will cover the sword as a white ray lightens the blade like a beacon.

2. Floor Slam: Sword trail will maintain the color of the blade, while the impact on the ground will create a blast of color from white to yellow and orange.

3. Ground cracking: The cracks in the ground will appear in darker shades while the center glows with warmer tones, to represent the heat of the impact

4. Rocks aftermath: The rocks will have a pale low saturated brown tone, close to gray, to keep the color correlation

3.3.2 Mage Abilities

Fireball

A small fire projectile shot from the left hand that finishes with an explosion. It's meant to be a far ranged attack that follows a straight line and vanishes after hitting a target blasting in flames for visual feedback (see Figure 3.5).

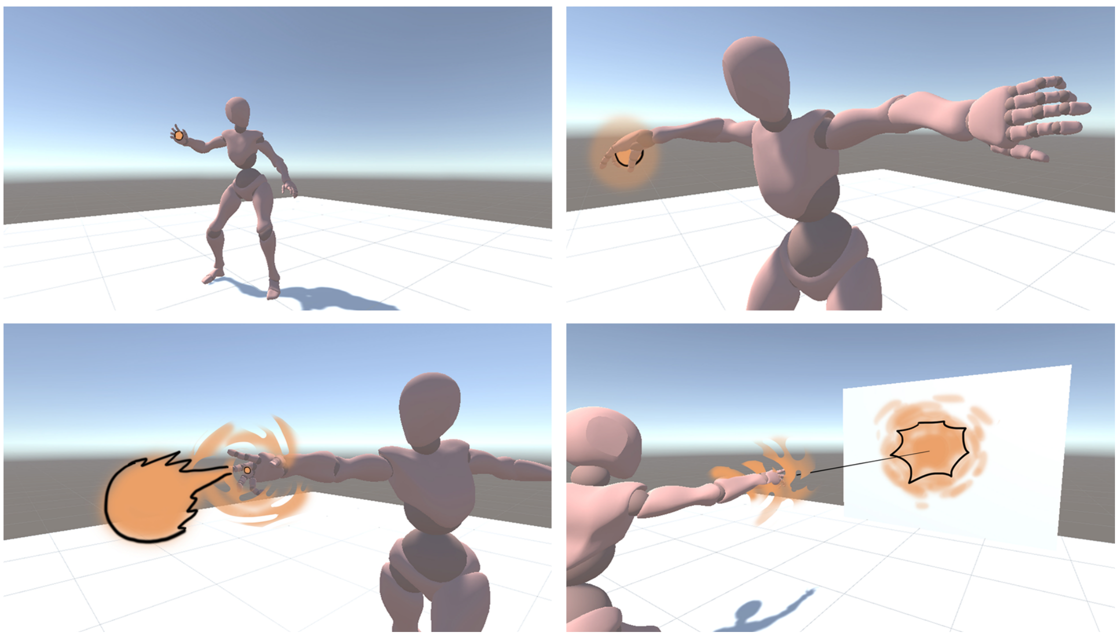


Figure 3.5: Fireball concept over animation frames

The fireball is a common VFX used as an elemental fire attack in fantasy video games, it has even existed in fantasy role games that pre-dated *Dungeons & Dragons.* Nowadays it is an essential must for magic combat in games. As a popular reference, *Nintendo*'s widely famous character, *Mario*, has been throwing fireballs since *Super Mario Bros.* and has become an iconic feature in the franchise.

For the art style, in order to represent fire, the colors used will be warm, specifically from red to yellow tones.

Lightning Cast

The lightning discharge casting is a medium-range offensive skill that affects an area in front of the character in an imprecise way (see Figure 3.6). Since electric rays move seemingly randomly, the user will only notice hitting a target if it shows any visual feedback like being electrocuted.

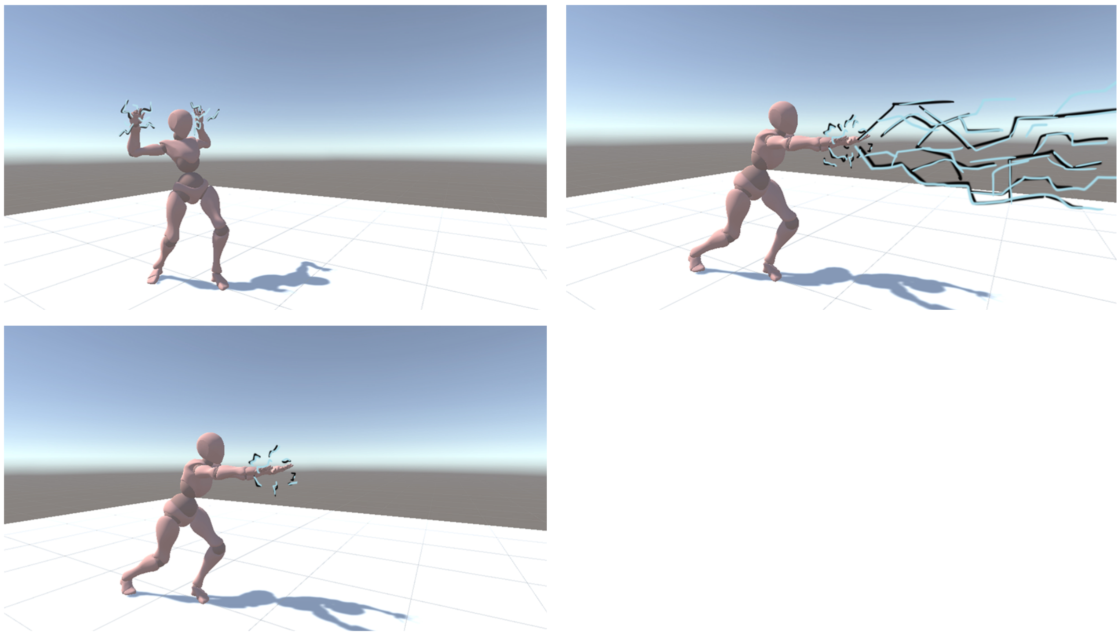


Figure 3.6: Lightning Cast concept over animation frames

Lightning is a common VFX for any kind of visual media, it is flashy, very eye-catching, and not very difficult to simulate with computer graphics.

The choice of color shall be a display of intense blue tones for the base color and white for highlights making a stylized and appealing result, keeping a cohesive look for all the VFX in this project.

Magic Circle

The following ability consists of a compound VFX, first the aura that covers the mage, and secondly, the circle pattern that automatically appears below, creating a beautiful scene (see Figure 3.7).

This skill is not meant to attack or defend, but to power up the character in some way. So, in terms of design, what matters is that the effect is attached to the user and changes the character visually while it lasts in order to properly communicate the change in the stats, furthermore, the circle serves as an area of effect indicator, letting the user know that the ability is limited by its perimeter.

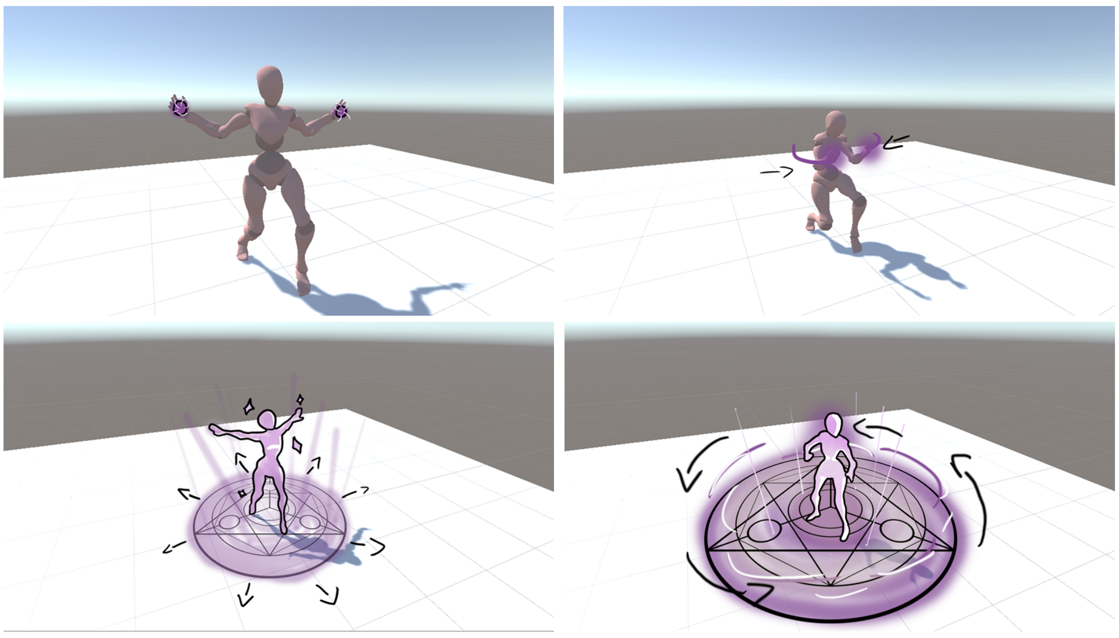


Figure 3.7: Magic Circle concept over animation frames

This type of circle is a well-known concept in modern culture, they are often inspired by pagan practices like alchemy or witchcraft, and are heavily associated with magic in fantasy settings. The complex geometric patterns that are drawn by themselves on the floor, indicate the magic user's dexterity and serve as a demonstration of power.

The design for the circle itself is astrology-inspired, but avoiding too much complexity so other elements of the scene like the lights and the character aura may stand out as well.

Chapter 4

Work Development and Results

Contents

This chapter is a report of all the development from this project, detailing the process behind it and the decisions taken. It also includes a section for the results obtained afterward, where the changes in the original design are explained.

4.1 Work Development

Each VFX is broken down into the different components that result in the final effect, as they are created using several methods that are explained in detail throughout the following sections.

4.1.1 Warrior Abilities

Slash

- Blade Glow Shader:

In order to emulate the slash coming from the blade, the sword shall glow with a color matching the slash, making the player associate cause and effect. The material that provides this glow effect is made with a custom shader using Unity's Shader Graph.

Start by creating a lit and opaque blank shadergraph, and recreate the original PBR material from the sword, so the material can be changed by script without the player noticing since the current shader is more simple and won't look exactly the same.

This is easily made by using the same base color, metallic/smoothness, and normal map textures from the source material and setting them up the same way, as seen in Figure 4.1.

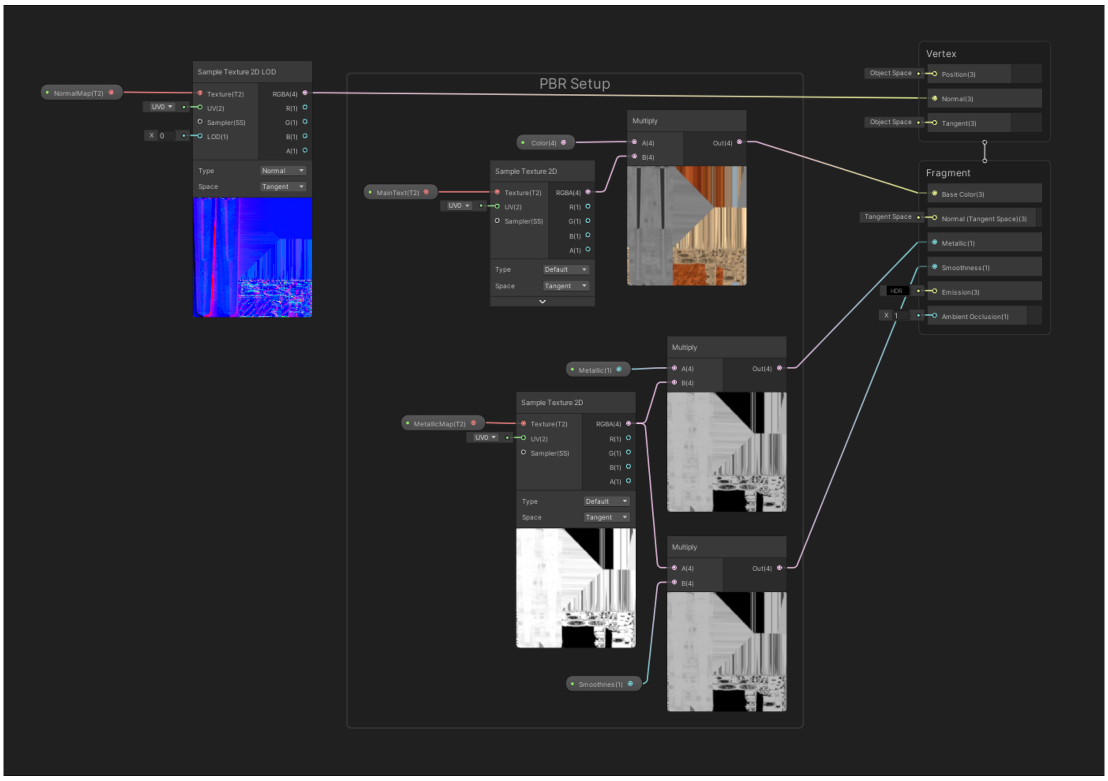


Figure 4.1: PBR set up matching the original material

Now, the objective is to create a glowing layer covering the blade, from the edge to the inside, and make it fade in and away so it only glows when slashing.

First, the glow effect is obtained by adding a Fresnel property based on the physical observation of objects becoming more reflective at sharp angles, as described in Microsoft Azure Docs [1].

After adding a Fresnel, variables for customizing the color and the power of the effect (when closer to 0, the more it covers the surface) are created. The color is multiplied by the fresnel and then, since only the blade shall be glowing, a mask texture is used to specify in the UVs of the mesh where the fresnel applies (see Figure 4.2). This mask (see Figure 4.3) is simply made by editing the metallic texture of the model, leaving only the unwrapped segments of the blade.

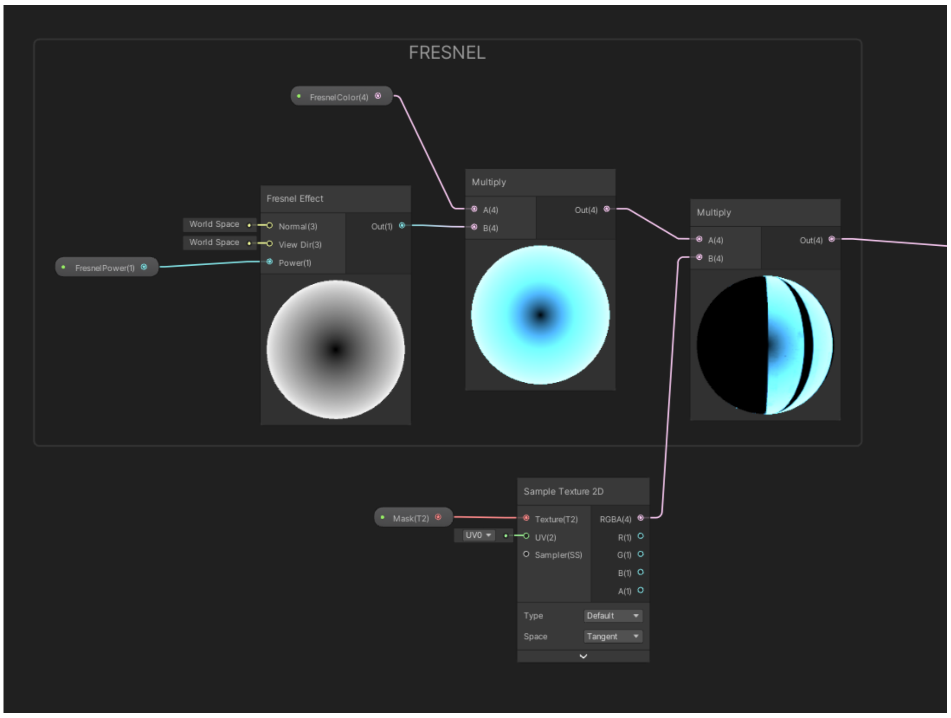


Figure 4.2: Set up for the fresnel effect with mask texture

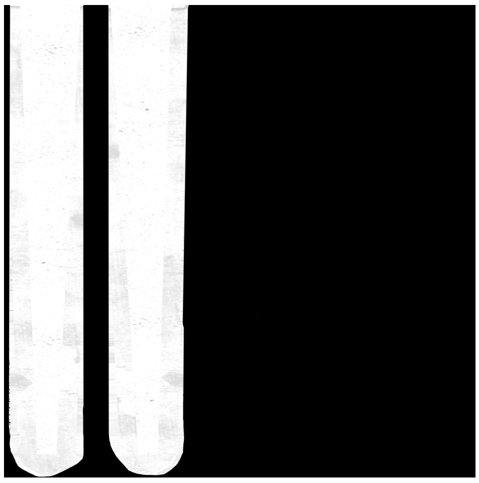


Figure 4.3: Mask texture from the sword model leaving only the blade surface

Applying this setup to the emission will grant the desired result, but it will make the blade stay glowing as long as it has the material attached to it. And changing the material by script will produce an abrupt and harsh change.

To make it change smoothly, a linear interpolation operator is added, so it progressively fades from the PBR setup to the fresnel effect. Then, by interpolating between the fresnel and a 0 value, outputting it to the emission channel, and adding a float variable to control the fading, the shader is finished.

- Slash Composition:

Since the trajectory of the slash follows a circular path, it is most convenient to create a mesh that leads the trail of the sword.

Making a plane seemed like a good idea, but in the character animation, the movement of the cut is not a perfect curve, so trying to match the mesh with the animation was hard and inaccurate. So the first attempt at the slash as a plane mesh was discarded for this effect, but later used as a component on the Ground Slam (see *Ground Slam, Slash VFX Graph.* Within this chapter).

The next approach was to create a model with volume, so there's more range for the slash to look like it's following the blade. In Blender, a simple disk shape (see Figure 4.4) with a hole in the middle for the character is modeled and then imported into Unity as fbx format.

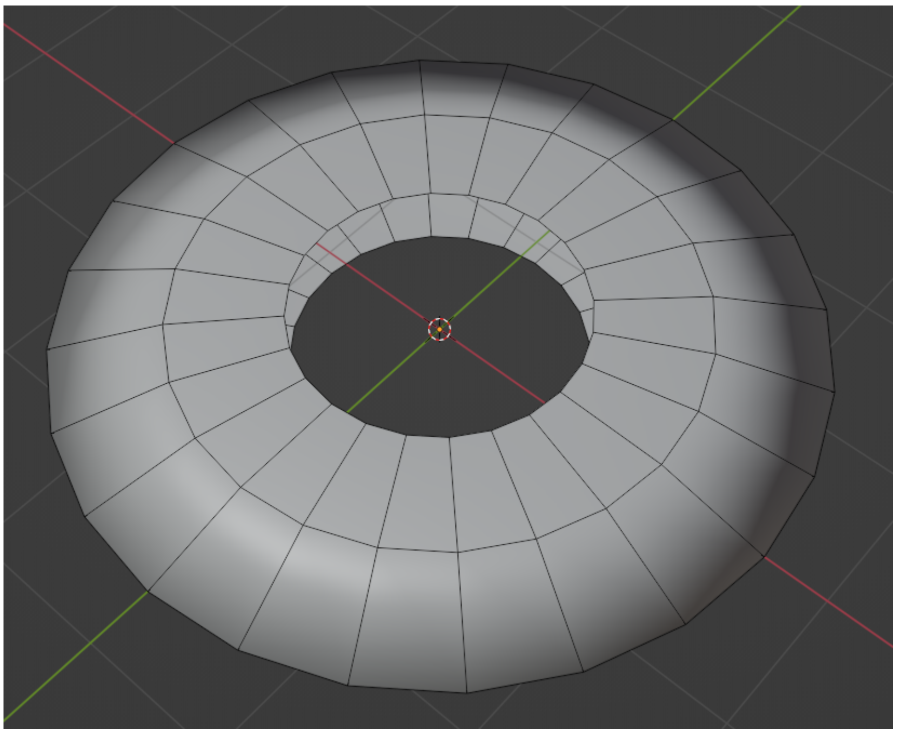


Figure 4.4: Slash mesh from blender

In Unity, the mesh needs a custom shader which starts with an unlit blank Shader Graph, to prevent the effect to appear all around the mesh, a mask has to be applied, in this case, since a radial mask is used to only render the shader on the outer edges, a Polar coordinate node is added, which will convert the input UV of the model to polar coordinates [1], split it on the RGBA red channel and invert and the power. To avoid graphic errors for values below 0, they are clamped between 0 and 1.

Then the actual texture will be using a voronoi pattern, a type of procedural noise that generates random cell-like shapes, granting a dissolved look for the slash. By customizing the power and scale, and animating it by assigning the angle offset of the voronoi to a time node, the procedural texture is finished.

The only thing left for the shader is to apply color and transparency. To do so, the mask and the animated voronoi are multiplied to a custom color value and connected to the base color channel. The transparency will be obtained by multiplying the color's split alpha channel to the masked voronoi and connecting the output to the alpha channel of the shader. See full shader setup in Figure 4.5.

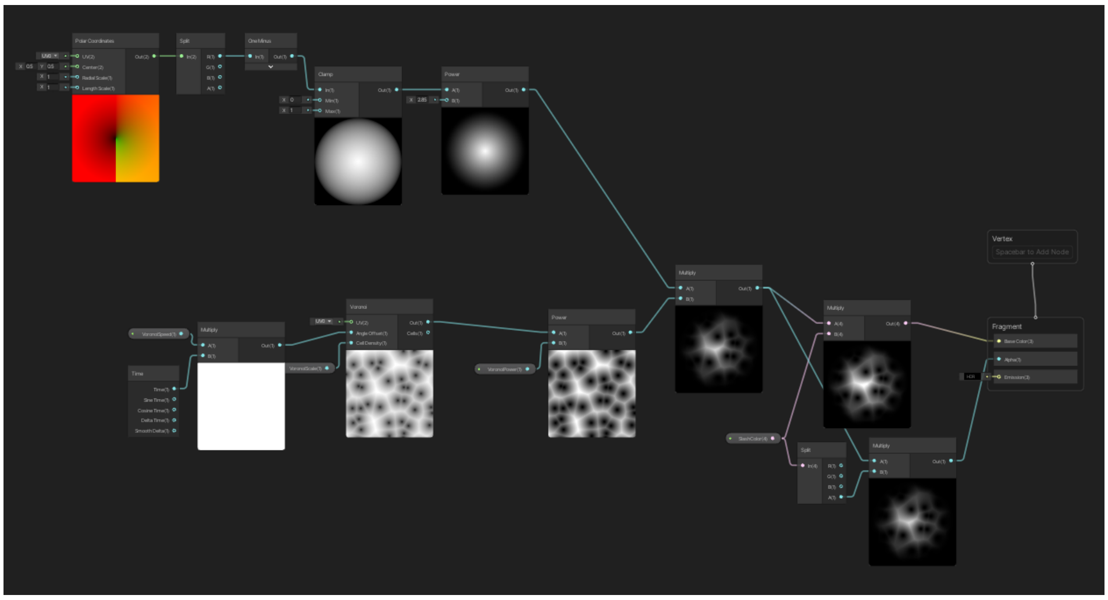


Figure 4.5: Slash shader setup

Now for the VFX Graph, where the behavior of the slash is defined, starting by initializing the VFX with a single burst and a lifetime of 0.3 seconds, since the slash is a unique and quick instance. Then, in the update section, the particle is rotated on the Z axis, so it spins around the character.

The acceleration over time is controlled by a downward curve that makes the spin smoothly lose speed. And for the output, the model and the previously explained shader are assigned. Also, by controlling the variables of the shader, a bright light blue color with intensity, matching the sword glow, is added to the particle, the scale of the voronoi is set to 10, the speed to 5, and the power is controlled by another curve with an upward path, since the higher is the power of the voronoi, the more it looks like it's dissolved.

Then a copy of the setup is made to create another slash, darker and smaller, that renders before, so it stays behind the first slash, to add contrast to the effect (see Figure 4.6).

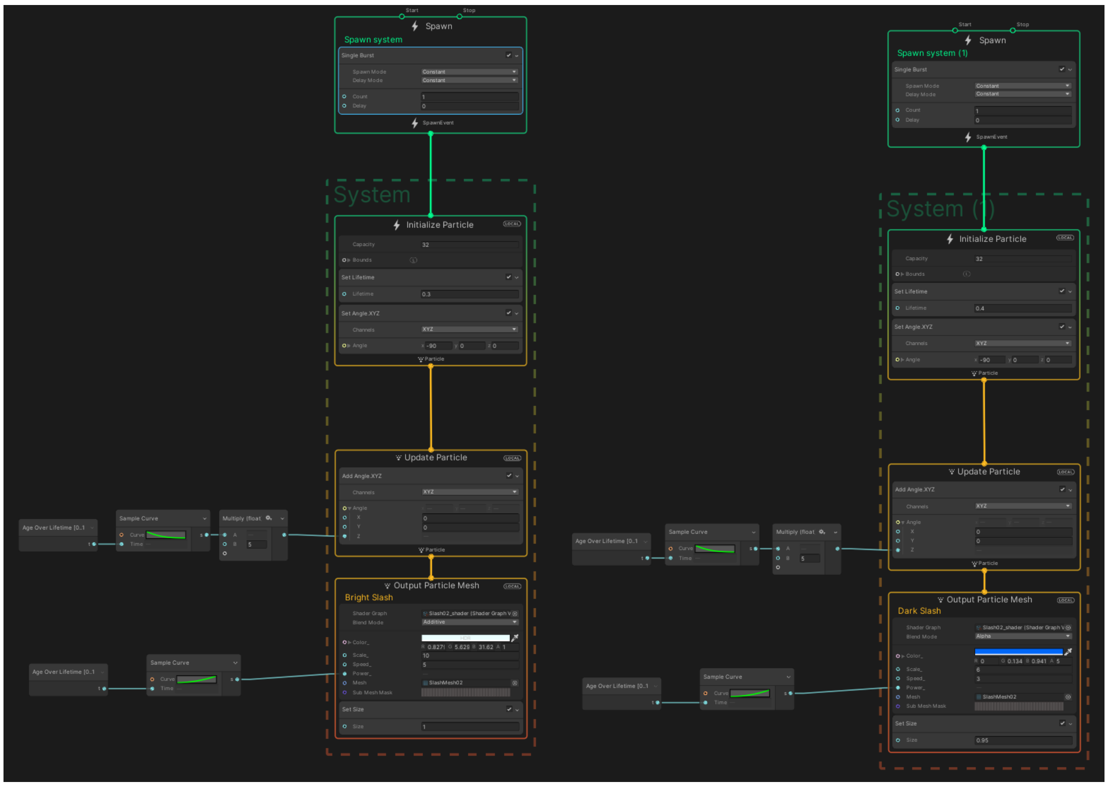


Figure 4.6: Slash VFX Graph set up

Shield Projection

- Shield Glow Shader:

The initial part of the VFX is the shield getting a glowing layer that starts covering the mesh from right to left. This effect is created with a shader that lerps the emission the following way:

First of all, the textures of the original material have to be inputted into the blank shader.

Next step is to control the material by a vertical threshold, this is possible by splitting the position and getting the red channel. Which will be subtracted to a custom variable that will let the amount of emission interpolated be controlled.

To make the threshold of the lerp sharper, the result is divided by 0.2 and clamped between 0 and 1. Then interpolating the desired color and a 0 value with the position operator made with the lerp node (see Figure 4.7) and outputting the result into the emission channel, the shader is complete.

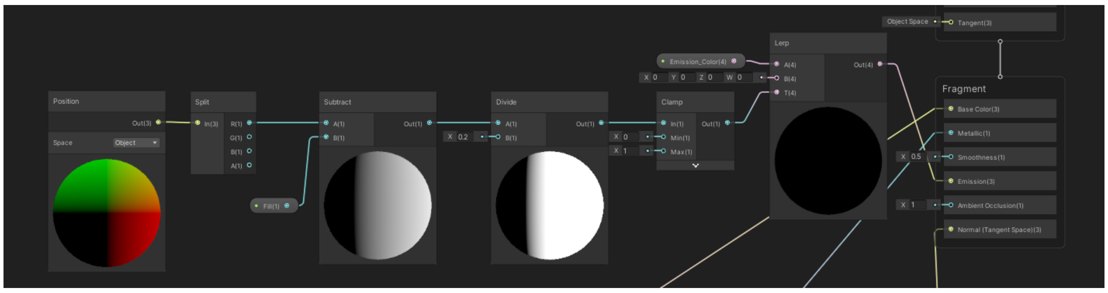


Figure 4.7: Position gradient lerping shader

- Shield Cast Shader and Particle System:

The shield that is projected in front of the character uses a mesh, most conveniently the same mesh as the shield asset to keep coherence between cause and effect, but the model has an arm grip that won't look good on the VFX, so by editing the model in Blender the mesh gets rid of them (see Figure 4.8).

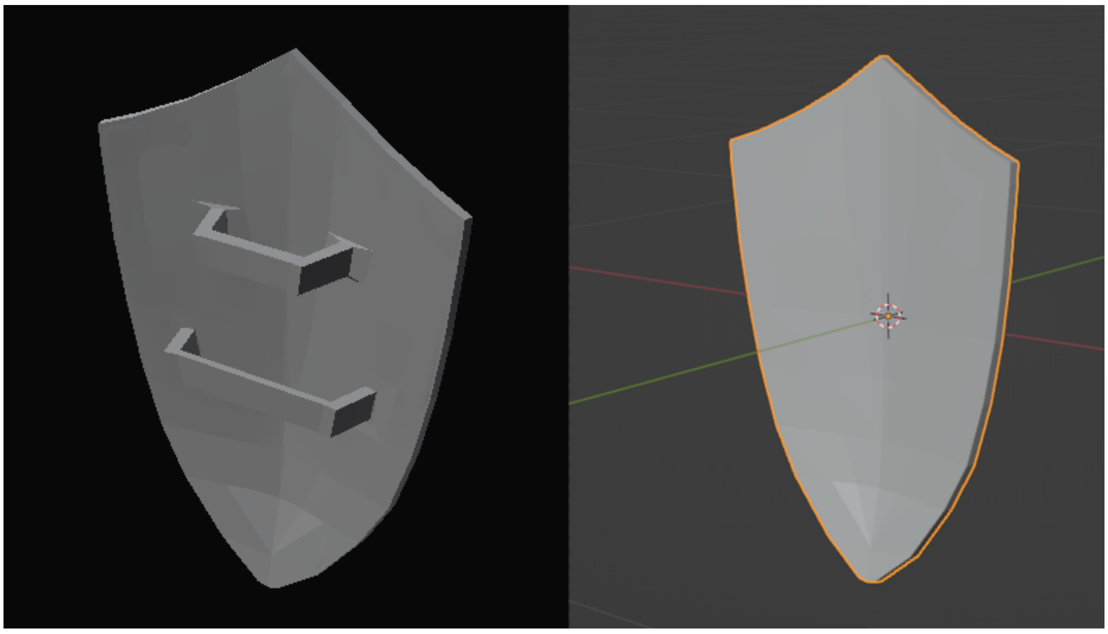


Figure 4.8: Original and edited shield meshes

For the particle, the look must be like a spectral glowing shield, and that can be obtained with a not-so-complex shader involving a fresnel effect, which provides glow and transparency, and an animated noise map to make it dynamic.

The noise map is controlled by multiplying time by a speed variable and then adding it to the UV coordinates of the noise map scroll automatically over time.

Then a fresnel effect with a power of -0.3 and a color variable is multiplied to the noise map and connected to the base color, alpha, alpha threshold, and emission channel, with this the shader is finished and applied to a material for the shield projection (see Figure 4.9).

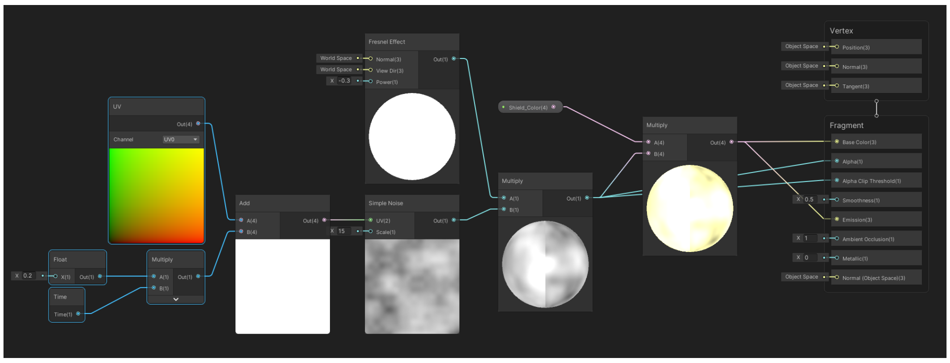


Figure 4.9: Projected shield shader setup

With the particle system, the spawn and movement of the projection are defined. It has to appear to increase in size from the original shield and move forward slowly until it disappears mid air.

The duration and lifetime are set to 10 seconds since the shield has to stay for a little while and a delay is set to match the animation starting point. Then the size is set to be considerably bigger than the shield model and slightly wider. The rotation is set at -88 degrees on the X-axis to tilt it a little hence covering the character better. The emission is set to a single burst since the VFX has to show only once when activated and no looping. The velocity and size over time are set with curves which define not only the general motion but also the specific behavior of the particle at the start point (see Figure 4.10).

The velocity value is very high at the beginning, decelerating drastically, and then, at less than 20% of the lifetime, it switches to a slight downward curve decelerating slowly. The size follows the same timing but with opposite values.

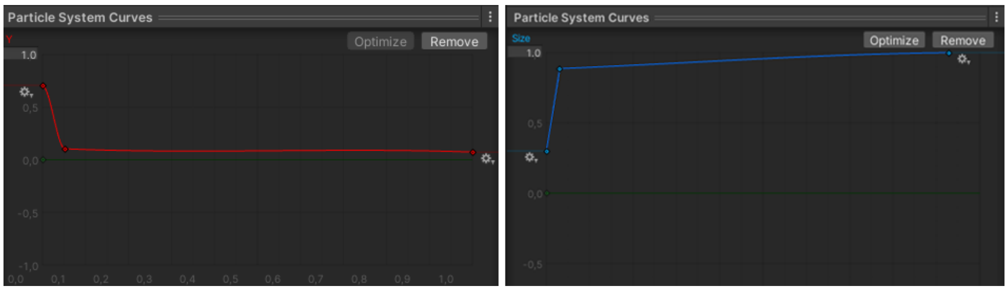


Figure 4.10: Particle System curves for velocity (left) and size (right)

Ground Slam

- LightRay Particle System:

This VFX starts with a ray of light coming from the sky, lighting the sword when the character lifts it up and giving way to the shader on the blade. The particle system for this part is pretty simple, it consists of a cylinder mesh stretched in the Y-axis to be elongated as a ray with a default particle material half-transparent, making a single burst and having 0.7 seconds of lifetime. By controlling the alpha with a color gradient the start and end are a fast fade.

- Blade Shader:

The light ray is followed by an instant change of material on the blade of the sword. This material is made with a custom shader. First, the shader recreates the original PBR material by applying the same textures.

Then a fresnel set up with yellow color emission and using a mask to only cover the blade in the sword mesh is applied. And, to make it dynamic, an animated gradient noise map, scrolling over the UVs, is also applied to the emission channel of the shader.

- Slash VFX Graph:

As the sword goes down, a slash trailquickly appears behind. This is the first version made for the slash VFX, that was discarded for the lack of precision, but as an addition to a more complex VFX like this, contributes a lot to the whole composition, and since the movement of the sword is straighter, it does not look off. This effect is made with a curved plane modeled in Blender, with the UVs oriented in order to make the slash texture come from the top (see Figure 4.11).

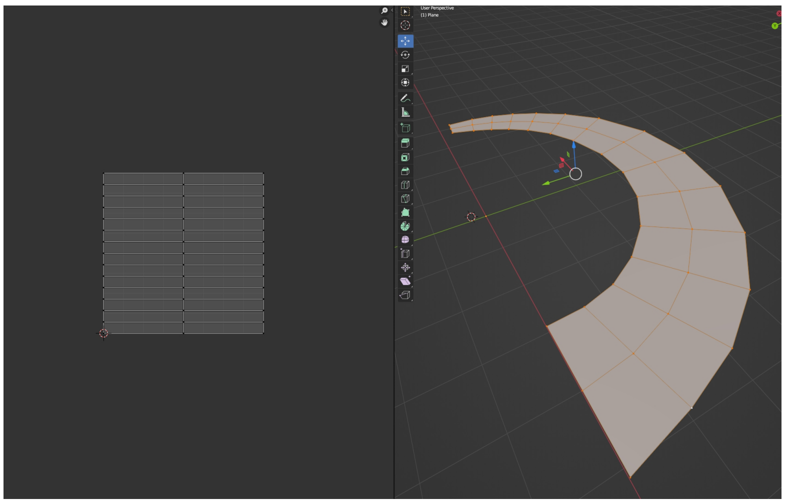


Figure 4.11: Slash mesh plane with UV unwrap from Blender

This texture, made by hand in Photoshop, will scroll and stretch through the plane, leaving a black line at the top for transparency (see Figure 4.12).

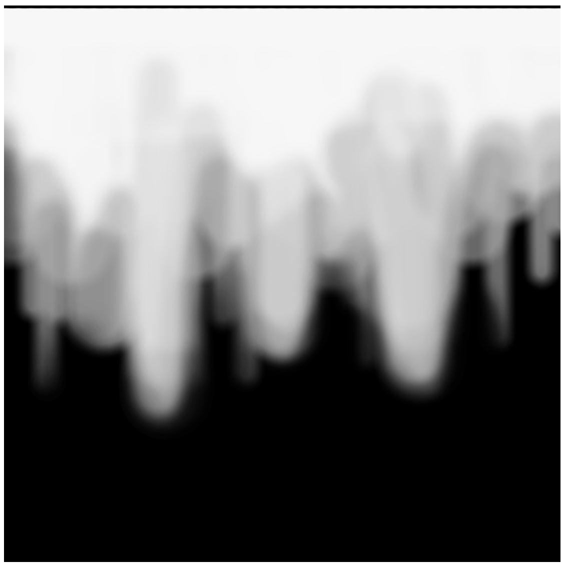


Figure 4.12: Slash texture made in Photoshop

The shader for the material will make it scroll by controlling the offset in the Y-axis from the UV coordinates, clamped to avoid texture repetition.

Now, to control the acceleration and the colors of the slash, a property from the particle system called custom vertex will be used, and to do so, the UV node of the shader is split and connected to the offset, and a color gradient with the color variables.

Then the texture transparency is applied to the alpha channel and the base color is obtained by multiplying the texture with the color gradient (see Figure 4.13)

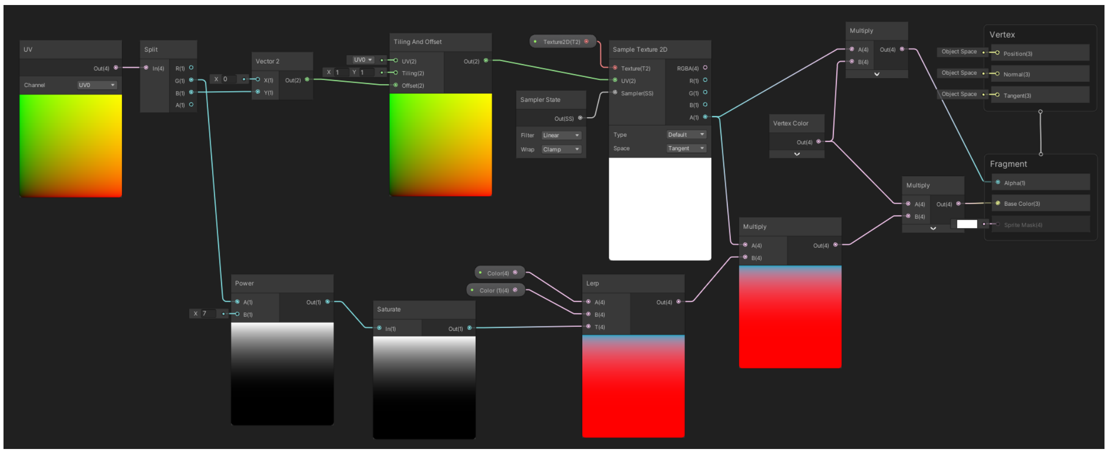


Figure 4.13: Scrolling slash shader setup

For the particle system, the following properties are set up to get the right behavior: The duration is set to 2 seconds, even though it doesn't really matter, the actual duration is determined by the scrolling speed; no looping; lifetime matching the duration, it doesn't make a difference either; an emission of 1 burst and no rate over time since it is a single instance that occurs following the sword and no shape emitter. The rest of the values are left as default. The important part is the custom data, which were configured on the shader to control acceleration and color gradient. The X controls scroll speed over time, and with a soft downward curve the slash will smoothly decelerate, and the Y curve controls the gradient, making the second color cover more surface, as seen in Figure 4.14.

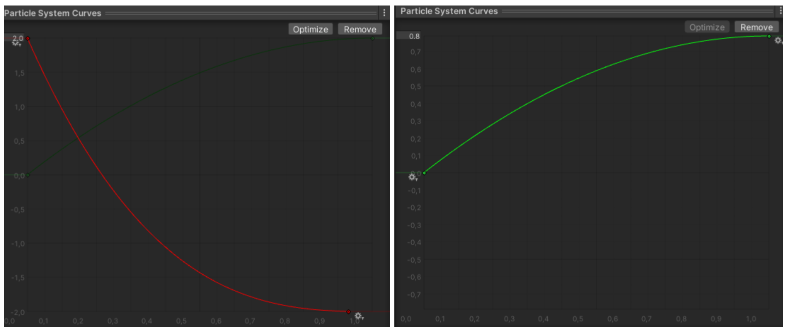


Figure 4.14: Speed over time curve and gradient curve, respectively

The VFX is then placed and synchronized with the animation to get the final result.

- Crack Decal Particle System:

When hitting the ground with the sword, the floor has to visibly break and that is made with a particle with a crack decal texture. The texture, Figure 4.15, edited with Photoshop, uses grayscale for transparency.

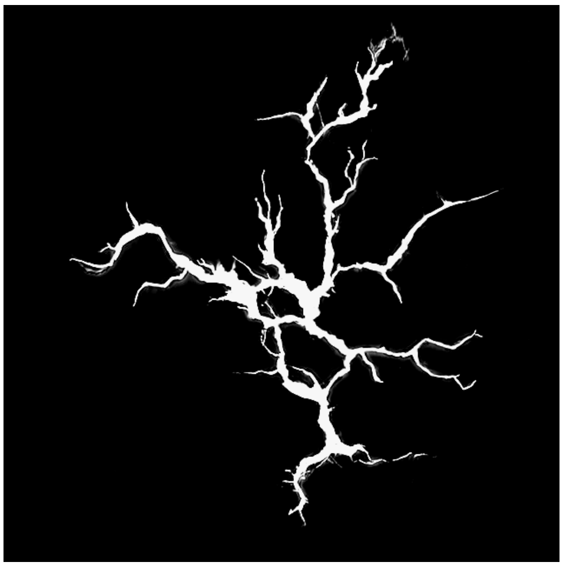
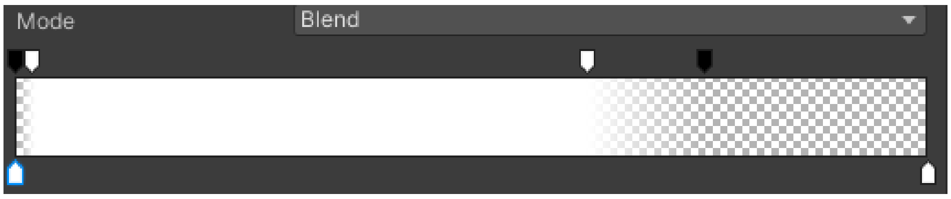


Figure 4.15: Ground crack decal texture

Then the effect is made by applying a Mobile Particle Multiply material with the texture to a particle system as an horizontal billboard, positioning the transform at 0.01 in the Y axis to avoid colliding with the ground. Rescaling to make it cover enough surface and then giving the particle 5 seconds of lifetime and 1.8 of delay for syncing the animation.

To make it fade in and out the color over time is controlled by a gradient with transparency at the start and the end shown in Figure 4.16 below.



Mode: Blend

Figure 4.16: Transparency gradient

- Dust Particle System:

To recreate the dust, a material with a custom texture is needed, in this case, the approach is to make the particles align with the view, so it is always seen the same way. Because it is a low detail texture, as shown in Figure 4.17 to the right, and it will be surrounded by other particles, some being the same and others being the rest of the components of this VFX, the visuals stay solid and the user won't notice.

Figure 4.17: Dust cloud texture

So, this part of the effect is made with two Particle Systems to recreate close and dark dust, and farther and light dust.

The first one is the light dust cloud, which starts with low transparency and high speed, reaching a considerable distance, spreading to the sides and lifting a little, gaining opacity over time, this cloud lingers for a few seconds and then fades away. The following parameters are needed to create this behavior:

For the duration, 3 will be more than enough (for a VFX), since the goal is to make it linger for a bit, then a delay is applied to sync the animation, and a random lifetime between 1.5 and 3 seconds, dust clouds are uneven in reality and this will apply to several aspects of this particle system.

The color is a dark tone of brown, because the material created for this particle system is additive, keeping a light tone because even assigning said color, the original texture is very clear and won't get dark as if it were multiplied.

For the emission, only a single burst is needed, but the emission shape shall be a cone on the ground right in front of the character with the shape that shows Figure 4.18 below:

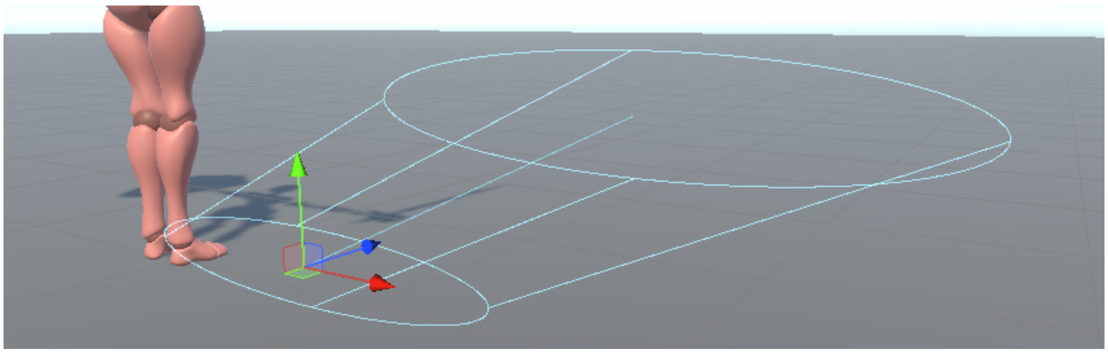
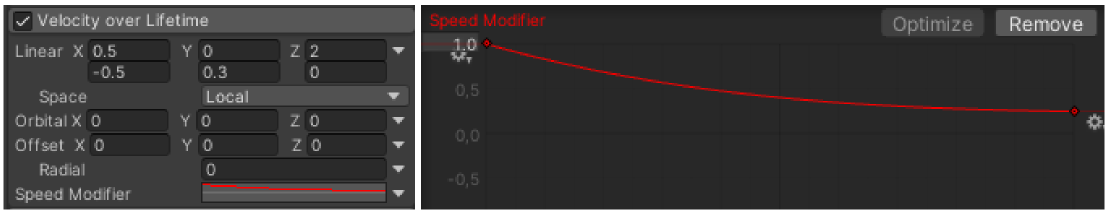


Figure 4.18: Emission cone shape for the dust

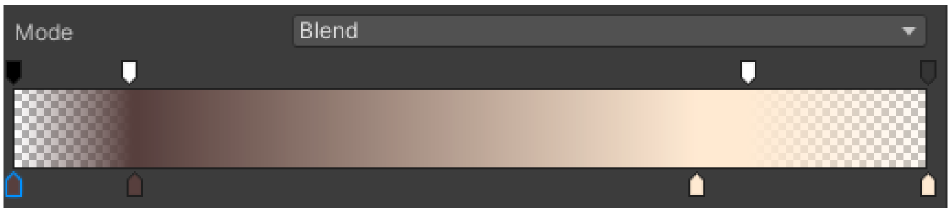
That covers the starting point of the particles, then there's the velocity over lifetime, which has to be very low, as set in the linear velocity like Figure 4.19 and decrease over time to slowly decelerate without coming to a full stop, which is controlled by the curve of the speed modifier.



```
✓ Velocity over Lifetime
Linear   X 0.5     Y 0     Z 2
           -0.5      0.3     0
   Space           Local
Orbital X 0        Y 0     Z 0
Offset  X 0        Y 0     Z 0
   Radial          0
Speed Modifier
```

Figure 4.19: Dust velocity over time setup

Another important aspect is the color over lifetime, which controls the fade in and out, gaining opacity rapidly at the beginning and slowly becoming transparent until it finally fades, changing its color to a lighter tone, Figure 4.20.



Mode: Blend

Figure 4.20: Dust color over time gradient

Simultaneously, the size over lifetime is controlled by the curve shown in Figure 4.21, slightly increasing over the duration and reaching the peak at the end. The second dust cloud is a darker set of particles due to having a multiplying material that remains closer to the origin, not reaching as far as the first cloud. So the main difference besides the color is the velocity over time, which starts with higher values for the X-axis, to spread more than the first cloud, but with the curve in the speed modifier, the velocity drops faster and stays closer to the character.

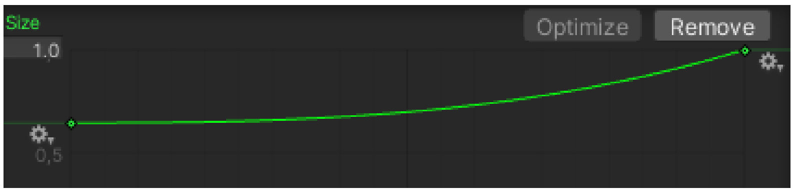


Figure 4.21: Dust size over time curve

- Rocks VFX Graph Composition:

This component is one of the most elaborated of the entire project, it involves modeling, texture painting, Shader Grap, and VFXGraph.

To summarize what is to expect from this part, it consists of three rows of rocks and rubble that emerge from the incandescent ground, each row larger and spreading wider and then fading away shrinking in the following order, rocks first, then the rubble, and last the glowing particles on the surface of the ground.

So the first thing needed are 3D models of the rocks and the rubble base, which, in Blender, by modeling the basic shape and decimating it with a modifier turns out like Figure 4.22 below:

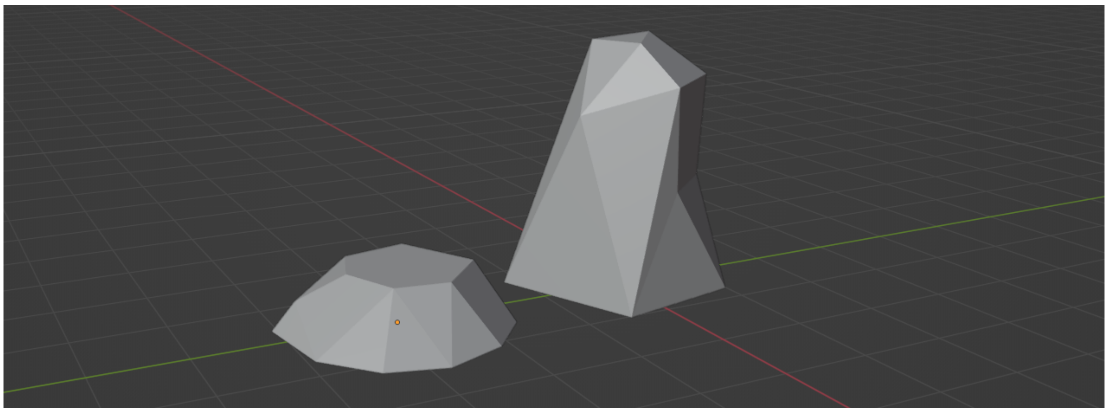


Figure 4.22: Rubble and rock meshes modeled in Blender

After testing a basic color material, the results were a bit off, since the rocks are the most highlighted aspect of the VFX, keeping it visibly low poly was discordant to the art style, so a custom texture, the one in Figure 4.23, was handpainted in Blender, achieveing a much more appealing result.

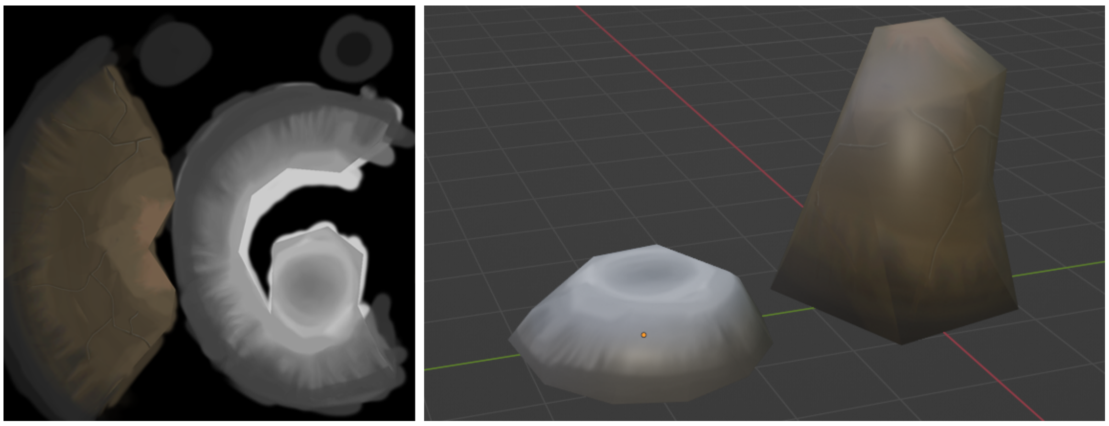


Figure 4.23: Rock and rubble texture and models with the texture applied

In Unity, this time creating a material won't be necessary, the rocks use a shader inputted directly to the VFX Graph, which both applies the texture and enables the model to dissolve away.

This unlit shader starts by entering the texture map and multiplying it by a color variable and a constant value of 2 to make it brighter. For the dissolve effect, the shader uses a simple noise map, which by adding to a dissolve amount slider from 0 to 1 will let have control over the amount of noise applied, but the values won't reach a full blackout (which means full transparent) unless it is remapped to make a range between 0.5 and 1.3 and inverted with the one minus node.

Then the result is also remapped between and clamped to get that contrast of blacks and whites. This already serves as the transparent region, so it is outputted to the alpha channel. Besides, it is multiplied by the texture for later getting the base color.

One more thing left. A threshold that will make the dissolution look more like burning or disintegration. This is achieved by stepping the clamped noise a bit, inverting it, and then subtracting the previous clamped noise, like in Figure 4.24, to leave a threshold between dark and white areas which will be multiplied by a color variable.

This is then clamped and added to the texture with the noise applied and outputted to the base color channel. After having the previous elements ready, it's time to assemble the VFX Graph.

To keep track of the structure and avoid redundancy through the explanation, the effect consists of three rows of three particles, in the first row there are 5 particles, in the second, 7 particles, and in the third, there are 9 particles. This applies to every set

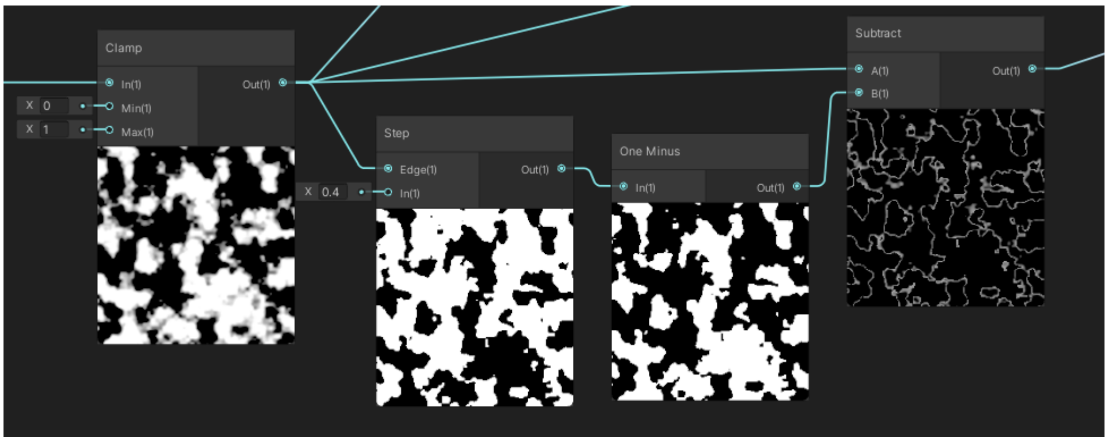


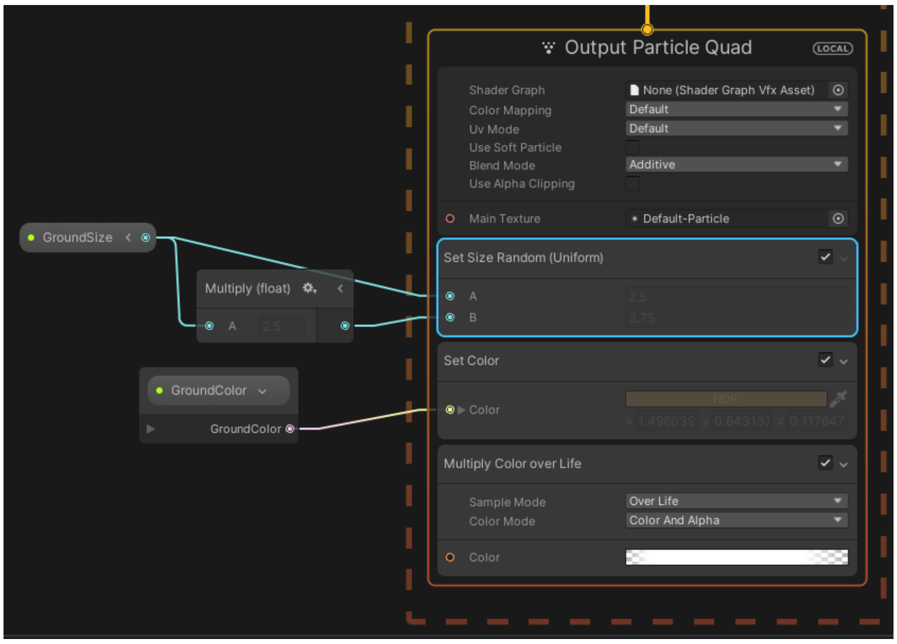
Figure 4.24: Dissolution threshold setup

of particles making only a single burst, and neither of them requires update functions, these are only meant to spawn and disappear.

The first ones are the ground particles, glowing orange decals underneath the rocks as if the floor is igniting. The particle has to initialize inside the surface a plane, which is obtained by scaling a box shape by the Y-axis to 0, so it doesn't have any height. The size of the plane is controlled by a variable radius for the X and Z-axis. After creating the plane shape, the particle has to be rotated 90 degrees on the X-axis to match the surface horizontally.

This is the particle that has to remain longer, the lifetime is set to 3.5 seconds of duration and since the particle is flat, an Output Particle Quad will render it properly. There's no need for a custom shader, and the default particle texture is enough to work a glowing circle. Also, to give variety to the particles, they are initialized with random size.

Finally, the color is multiplied over time to fade in and out at the start and the end of the effect. This setup for the output can be seen in Figure 4.25 down below.



Shader Graph: None (Shader Graph Vfx Asset)
Color Mapping: Default
Uv Mode: Default
Use Soft Particle
Blend Mode: Additive
Use Alpha Clipping
Main Texture: Default-Particle

Set Size Random (Uniform)
A: 2.5
B: 3.75

Set Color
Color: HDR, x 1.498039 y 0.643137 z 0.117647

Multiply Color over Life
Sample Mode: Over Life
Color Mode: Color And Alpha
Color

Multiply (float): A 2.5

Figure 4.25: Ground particles output

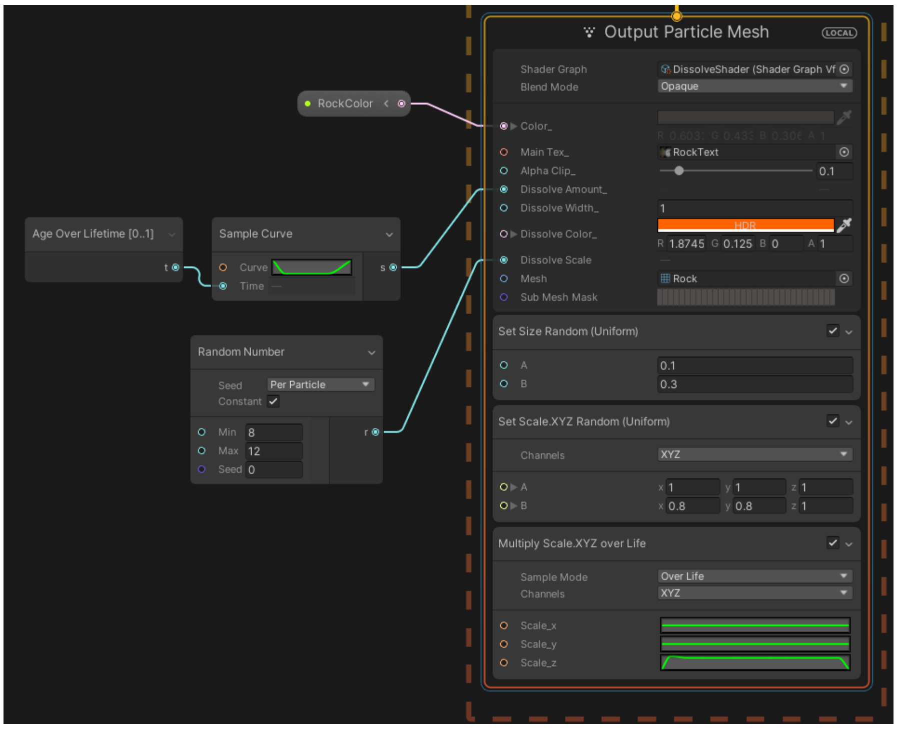
The next two rows keep the same properties but several values are increased to make them appear further and bigger, and a delay is added to make the rows spawn sequentially.

Said values are a distance variable for the set position, a multiplier for the shape's radius to make the particle's surface bigger, and another multiplier for both random values of the set size on the output.

The next set of particles are the actual rocks that emerge from the ground. The set position is the same as the previous particles since the aim is to make the correlative, spawning in the same place, so the values of the plane shape remain the same, using the same radius variable. Lifetime is random with a slight difference in values to avoid having a uniform effect.

For the position, first, the models have to be pivoted with an offset of -0.5 because the anchor of the model is positioned in the center of the mesh. Also, to pretend variety, the particles are randomly rotated in the Z-axis and slightly tilted on the Y and X-axis. Unlike the ground particles, the rock particles are rendered as 3D models and require an Output Particle Mesh, Figure 4.26, where the dissolve effect shader and the rock model are applied.

The shader is then set up with a color variable, the hand-painted texture, a curve over lifetime to control the dissolve-in and out of the particles, a color for the dissolve effect, and a random range between 8 and 12 to add variety when dissolving.



Random Number
Seed: Per Particle
Constant: ✓
Min: 8
Max: 12
Seed: 0

Output Particle Mesh
Shader Graph: DissolveShader (Shader Graph Vf
Blend Mode: Opaque
Color_: R 0.603 G 0.433 B 0.306 A 1
Main Tex_: RockText
Alpha Clip_: 0.1
Dissolve Amount_
Dissolve Width_: 1
Dissolve Color_: HDR R 1.8745 G 0.125 B 0 A 1
Dissolve Scale
Mesh: Rock
Sub Mesh Mask

Set Size Random (Uniform) ✓
A: 0.1
B: 0.3

Set Scale.XYZ Random (Uniform) ✓
Channels: XYZ
A: x 1 y 1 z 1
B: x 0.8 y 0.8 z 1

Multiply Scale.XYZ over Life ✓
Sample Mode: Over Life
Channels: XYZ
Scale_x
Scale_y
Scale_z

Figure 4.26: Rock particles output

The initial size of each particle will be a random number between 0.1 and 0.3, but also will be randomly scaled on the X and Y-axis to make some rocks wider than others. When the particles appear, it has to look like they emerge from the ground. To do so, the scale in the Z-axis must increase exponentially at the beginning and, to make them come back to the ground, the inverse procedure is applied to the end. Also, to add a bit of dynamism to the animation, a slight peak is added to the curve after reaching the main height to make the model perform a slight bounce. The curve that controls this behavior is the one shown in Figure 4.27.

Following rows of rocks have the same setup with the variables and multipliers used in the ground particles segment to appear further and larger. Now the final part of this VFX is the rubble particles, which serve as a base for the rocks previously configured.

The initialization is the same as the rocks with the only two differences, aside from the actual mesh, being the lack of tilt in the rotation, to make the particle a straight base and a longer lifetime than the rocks, since it is the second particle to disappear, after the rocks and before the ground particles, as seen in Figure 4.28.

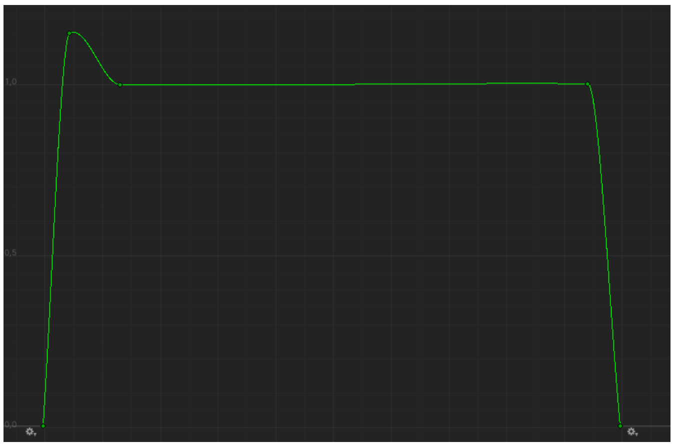


Figure 4.27: Rock VFX scale over time curve

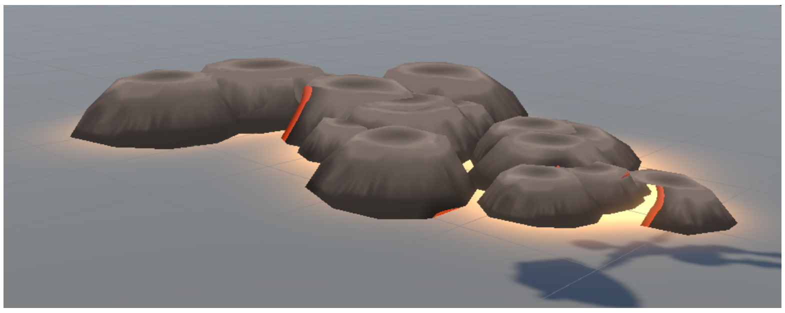


Figure 4.28: Rubble VFX dissolving

4.1.2 Mage Abilities

Fireball

- Charge Particle System:

First of all, a simple particle system, with little emission, a default particle texture map, and a short trail, is added to the hand of the character to represent the cast charging. It behaves as a single burst glowing ball with increasing size and fades over time.

- Projectile Composition:

For the projectile, a tear-like shaped mesh is modeled in Blender and imported to unity as an FBX, with the UVs shaped in a certain way to make the fire texture more easily, Figure 4.29.

Said texture is made in Photoshop, painting with white over a black background, to get the alpha transparency from the grayscale, meaning black pixels are transparent and white are fully opaque

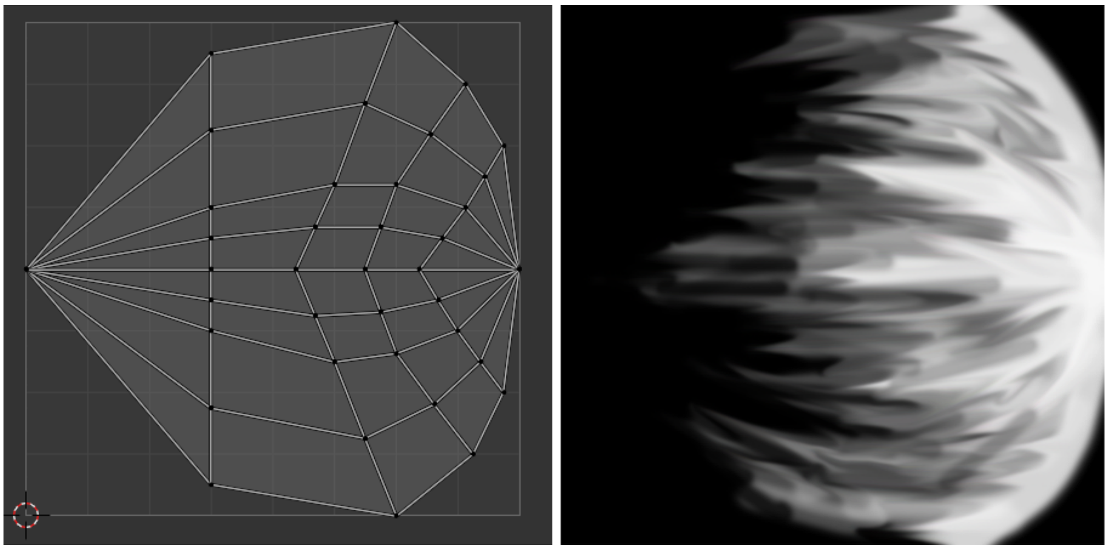


Figure 4.29: UV unwrap in Blender (left) and VFX texture (right)

In Unity, the material for the projectile is Simple Lit, with a base red color and an intense orange for the emission.

Then, after creating a Particle System and assigning the model and the material, the particle starts defining by setting looping and prewarming, because the projectile has to spawn right in action and keep the same movement while it is active. To make it reset every second, the lifetime is set to 1, then to correct the model's direction the rotation in the Y-axis is set to 90 degrees and the size of the particle will vary in width and height between 10 and 12, but length will remain the same.

The speed will be implemented by script since the projectile has to interact with other elements in the scene. The color of the particle is already defined in the material so the start color in the particle system does not affect the output.

Then the particle has to spin around itself in a fast pace, this makes up for the model being low-poly and creates a sense of dynamism and fire force. To do this, the rotation over lifetime is set to 720 degrees; this allows the particle to make two full rotations each second, which is the duration of its lifetime, but since it is looping, the particle will keep rotating over and over.

This settles the mesh particle, as previewed in Figure 4.30, then another particle is added to the fireball game object to pulsate in the interior of the mesh with a warm red color, to look like a heated core.

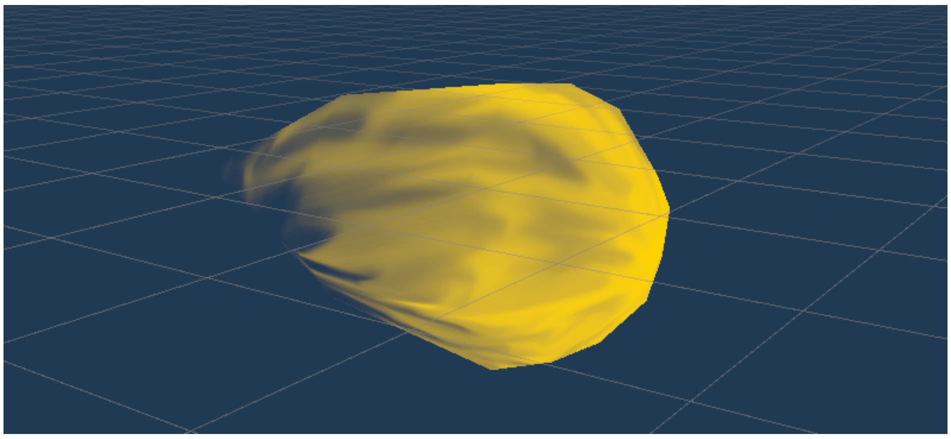


Figure 4.30: Fireball mesh particle preview

The particle has the default material from Unity Particles Unlit and almost the same properties as the previous one but without the varying size. What makes it change intermittently is the color over lifetime, which is set as a gradient (see Figure 4.31) that makes it smoothly transparent every half second.

Then as the last part of the fireball, a trail renderer that follows the projectile. First of all the trail has to softly decrease its width which is controlled by a descending curve. Also along with the width, the transparency has to smoothly drop down until it's fully transparent with an orange gradient to alpha 0.

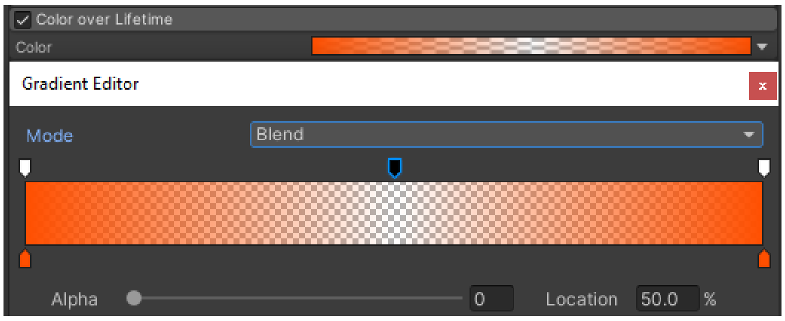

Color over Lifetime
Color
Gradient Editor
Mode Blend
Alpha 0 Location 50.0 %

Figure 4.31: Fireball core color gradient

- Blast VFX Graph:

The blast is a flashing explosion that leaves a dark circle on the surface it hits and consists of a VFX Graph with four different parts.

First is the flash, a light that quickly shows and disappears with a single burst. To add variety, the lifetime is set to a random value between 0.1 and 0.2 seconds, not much since it has to be an instant. Then for the output, the default particle texture and red color will suffice as intense light, the size is set to 4 times its original size and will decrease linearly over life by setting a straight downward ramp from 1 to 0 as a curve multiplier.

It is important to set the orientation as Face Camera Plane, to get the same result independently of the camera angle. Finally, the color is set to decrease the opacity starting at 10% of its lifetime.

The second block of the setup is a cross to stylize the explosion. The setting is similar to the flash, the approach consists in stretching the flash particle to make a cross-section, duplicating it, and giving each section a different angle.

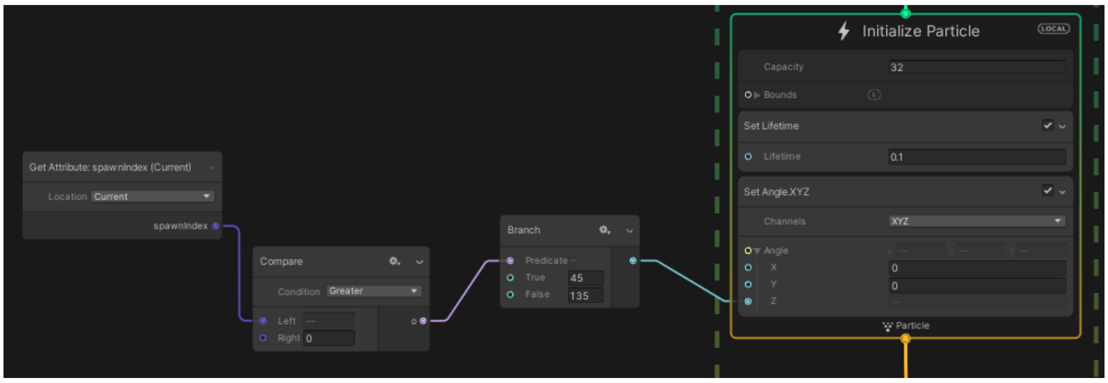


Figure 4.32: Cross-sections obtained by index

First, the particle count is set to 2, for the two sections, then, to control the angle separately, a spawnIndex node is used to modify each particle by index and compare if it is greater than 0, because the particle index is [0,1] the angle is set on the Z-axis by branching the result of the comparison, applying a 90-degree difference between the sections of the cross (see Figure 4.32). For the output, the setting is the same as the flash, with the only difference being the particle is scaled 0.1 on the X-axis to stretch it down. The final result for the flash, Figure 4.33, looks fine, even when the user won't be able to appreciate it, since it only lasts 0.1 seconds.

The third part of the blast is the sparks, a burst of several sparks shot in every direction that stretch until fading. This effect takes a single burst of 20 particles which

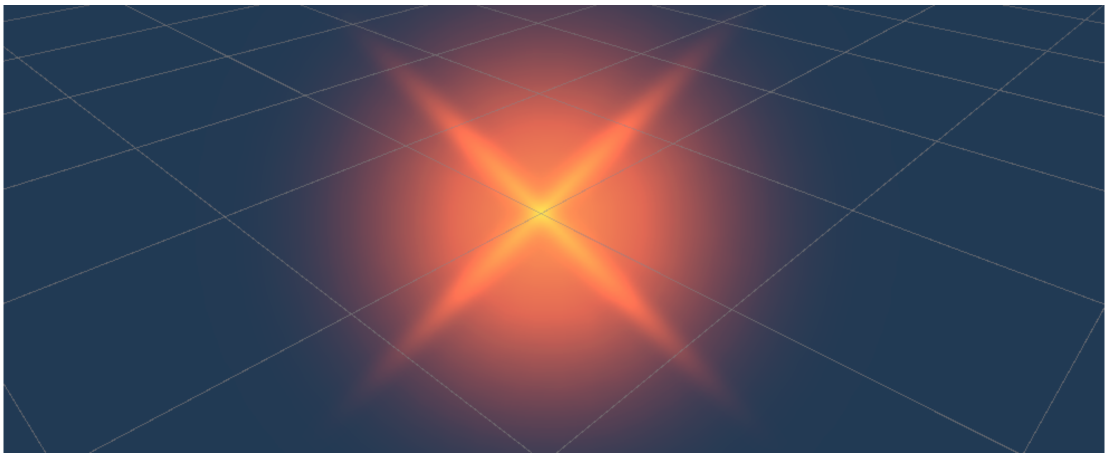


Figure 4.33: Flash VFX Preview

will be positioned over the surface of a sphere with a radius of 0.1, and a random lifetime between 0.1 and 0.4 seconds.

Then for the output, it also uses the default particle texture and is oriented along with velocity, so the perspective is forced to the origin of the movement which is the center of the sphere.

The size is random between 0.1 and 1 but also stretched in the X-axis to make the sparks thin and then controlled over lifetime to decrease with a curve so it shrinks until it disappears.

The last part of the blast is the decal, a burnt surface that the explosion leaves behind. This is also made with a single burst with 1 particle, and it has a considerably longer lifetime than the rest of the parts with 3 seconds of duration. The output is also different, in this case, since it is not just a particle, but it affects the surfaces of other objects, and this is possible with an *Output Particle Forward Decal.*

Which is set with a default particle texture, a full black color, and multiplying a gradient over time to make it fade away.

Lightning Cast

- Static Electricity Particle System:

This part serves as anticipation for the casting and to further emphasize the electric discharge. The effect uses a handmade animation sheet for the bolts with a blur filter, to further help the bloom in the game engine (See Figure 4.34). Then a universal render pipeline lit material is created using the sheet texture, with an intense light blue color for the emission.

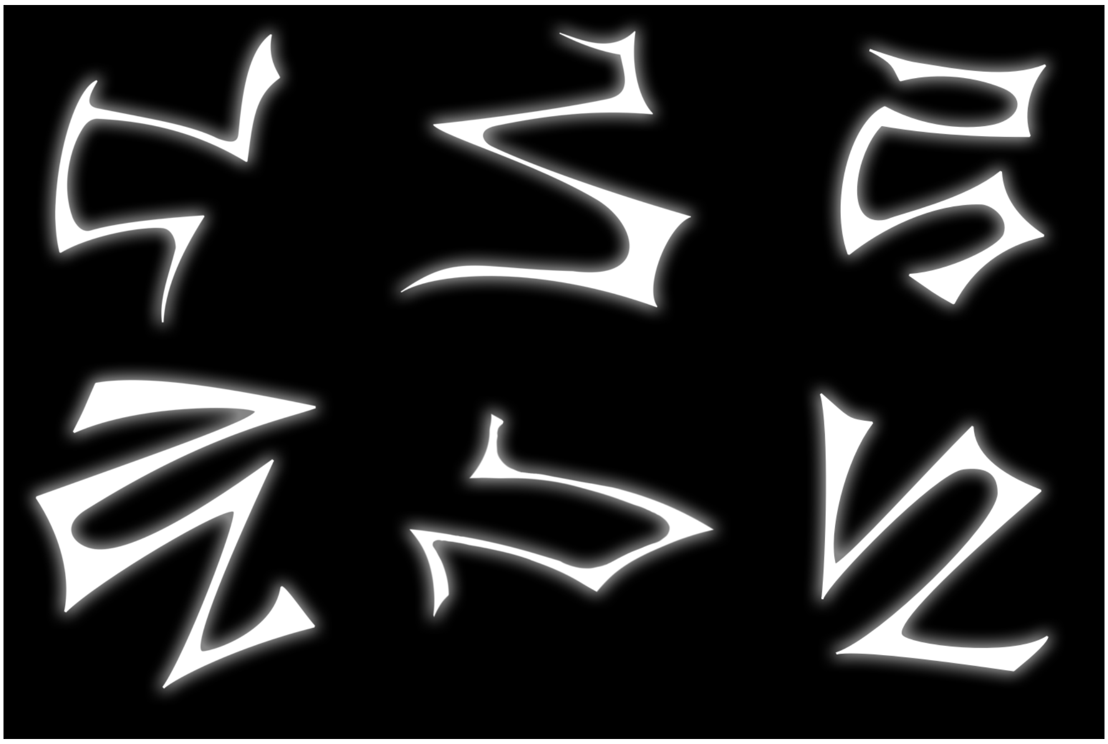


Figure 4.34: Electricity animation sheet with a blur filter

In the particle system, the duration doesn't matter, since it is going to be looping while active. Otherwise, the lifetime matters, as the bolts have to rapidly show and disappear, swapping through frames in the sheet. So a random number between 0.05 and 0.1 will suit, leaving some margin to make the electricity more natural and wild, this will be applied in most of the properties of this VFX from now on.

The size and rotation must vary too to make it organic, so the start size is set between 0.2 and 0.7 and the rotation to any angle of a full circle. Starting color remains the same as it was already set in the material and for the emission, 10 particles over time will be enough. Then the emitting shape is defined as a sphere of radius 0.12, so it emits from the hand in all directions.

Two copies of the VFX are placed in both hands of the character to fill them with bolts and a third and larger copy, with a cone shape to follow along with the lightning casting are also placed in the fingers.

- Lightning Cast Particle System:

The main part of this VFX is the lightning discharge. This is fully made procedurally with Particle System.

Starting off with a Universal Render Pipeline Lit material with a light blue intense color for the emission and the Default Particle Unity's texture for the base map, granting a fade touch on the edge, the material is assigned to the rendering of the particle system. Then, the following properties are established for the initial state of the particles:

Duration is not important, since the effect will be looping; lifetime random between 0.5 and 0.7, speed with a value of 15, size random between 0.05 and 0.1, and the color still a light blue.

Then the emitting shape is set to a cone stretched in the Y axis to fit around the fingers of the character (see Figure 4.35).

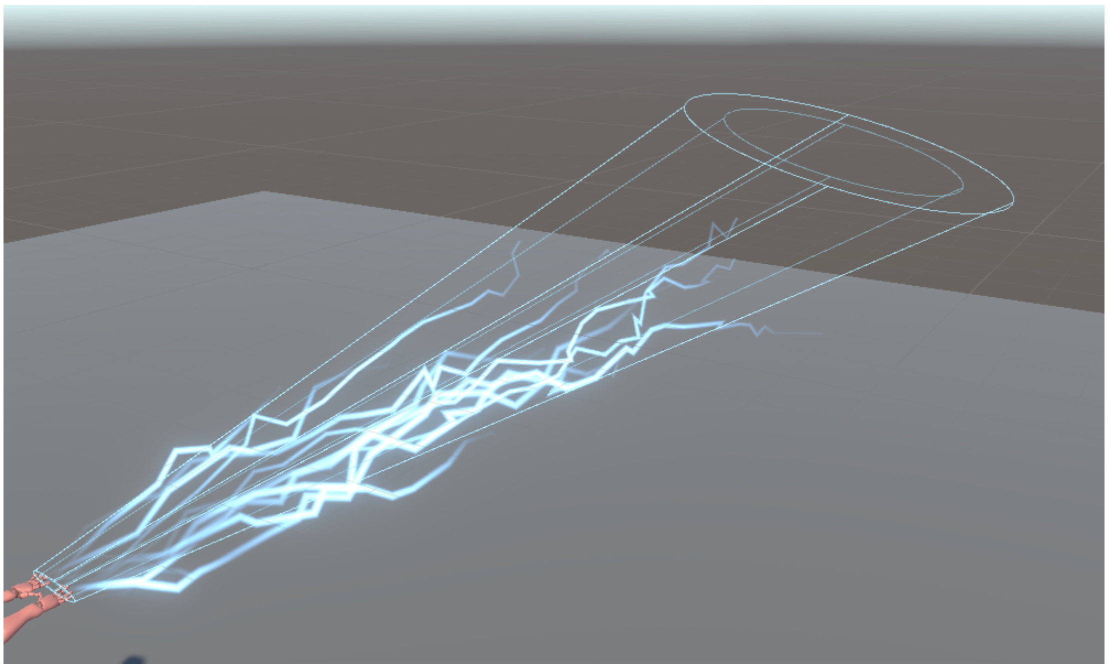
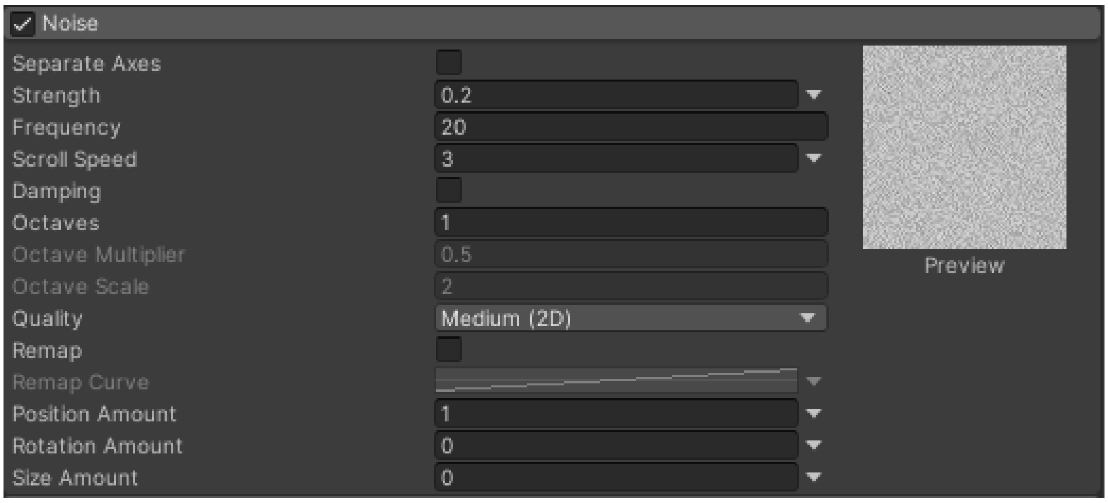


Figure 4.35: Lightning emission cone shape with a preview of the VFX

After that, the trails of the particles are set on to get the elongated shapes of the lightning, and to make them turn randomly, an animated noise map, Figure 4.36, is applied with the following properties:



✓ Noise	
Separate Axes	☐
Strength	0.2
Frequency	20
Scroll Speed	3
Damping	☐
Octaves	1
Octave Multiplier	0.5
Octave Scale	2
Quality	Medium (2D)
Remap	☐
Remap Curve	
Position Amount	1
Rotation Amount	0
Size Amount	0

Figure 4.36: Noise block from Unity's Particle system with lightning setup

This affects the path of the trails and twists them using a noise map that scrolls, creating many different combinations at a fast pace, and finally, the particle system is positioned in front of the tips of the character's fingers, aiming in the same direction.

Magic Circle

- Charge Particle System:

The particle system for the spell charge is the same previously explained for the Fireball, a single burst glowing ball with an increasing size that fades over time, but with a violet color and placed in both hands of the character.

- Character Glow Shader:

Since this glowing layer had to cover up the character's model, it had to be showy and bold, so the final choice was to make it with a galaxy photo [2] as a texture, which had to be edited in Photoshop to make the borders of the image match, to make it seamless.

Starting with a blank shader Graph, the texture is converted to a cube map, and the view is set to be based on the camera, meaning the texture map will orient along with the user's perspective, this gives an illusion of depth, Figure 4.37.

But to make it glow and make it dynamic, an animated fresnel effect is added to the shader, scrolling a simple noise texture through the fresnel over time. The speed, power of the fresnel, and color of the fresnel are controlled through public variables to customize the effect.

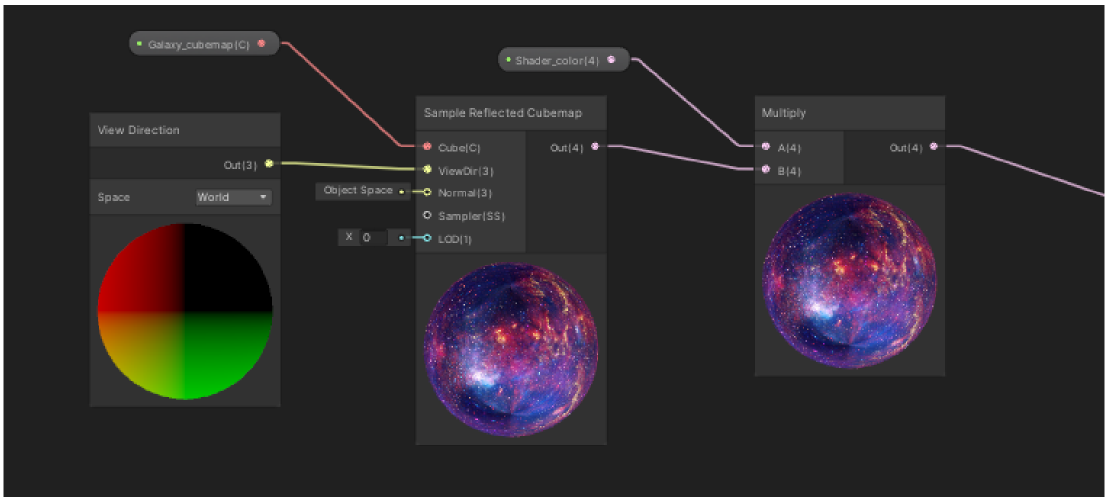


Figure 4.37: Perspective oriented cube map in Shader Graph

Then, to create the transition from the plain pink material of the character to the shader, a dissolution effect is interpolated with a basic color. More complex characters with materials more elaborated would need a recreation of their materials, but in this case, the basic color is enough.

The transition lerp is made by taking a simple noise texture, remapping it between -10 and 10 to get a black and white contrast, adding it to a variable that will be used to control the amount of shade is dissolved, clamping the result to avoid weird values and finally connecting it as the interpolation operator (see Figure 4.38). The emission will also be obtained by multiplying the result of the interpolation with the dissolution, so the base color doesn't get emission.

- Character Particle System:

These particles spin close and slowly around the character, and there are two types. First, the sparks, looping lights with random but durable lifetimes, between 2 and 8 seconds, and varying size, color, and speed. The shape of emission is a wide cone with an intern radius that avoids the character, and for the particles to orbit around decelerating, their orbital velocity on the Z-axis is defined by a downward curve, the same applies to their size which decreases over time (Figure 4.39).

- Circle and Glow Decal Particle System:

The main part of this VFX is the magic circle itself that appears on the ground.After creating the design in Adobe Photoshop, a blur filter is applied to recreate a bloom effect coming from the circle (see Figure 4.40)

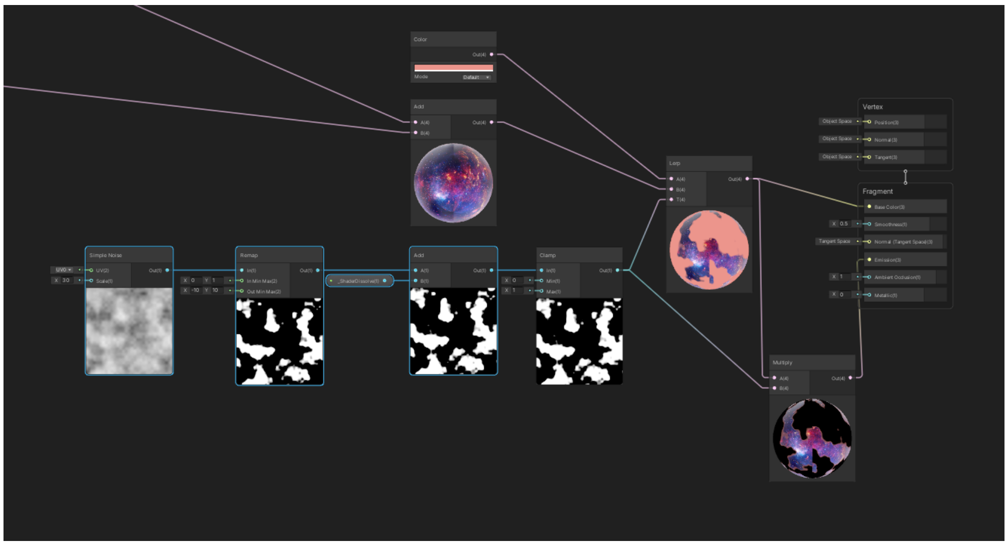


Figure 4.38: Noise texture setup for the dissolving galaxy shader

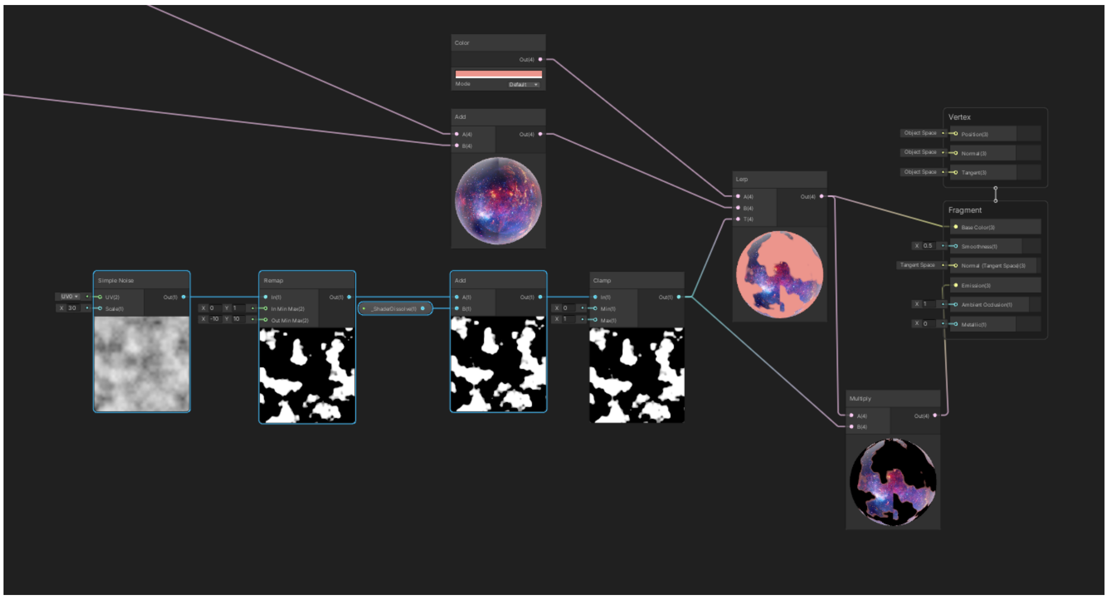


Figure 4.39: Emission cone shape for the character particles

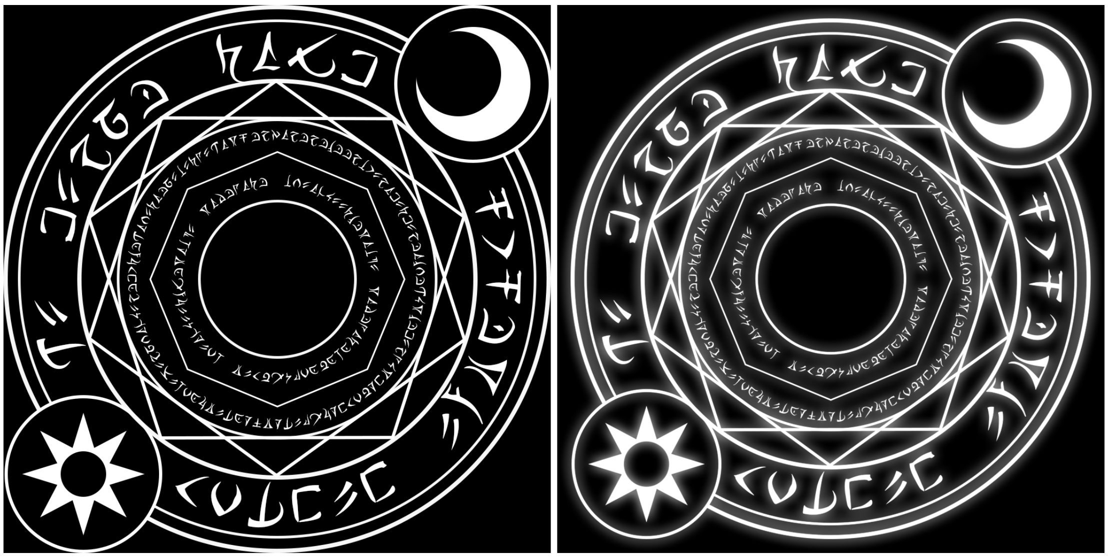


Figure 4.40: Magic Circle texture both with and without blur effect

The texture is imported into unity using Alpha Grayscale for transparency and an additive particle material is created with it. For the particle system, the initial size is set to 4 and the simulation speed to 0.06, to make it move slowly. Then by setting the color over lifetime as a gradient from transparent to opaque, the size increases rapidly right at the start and the rotation to keep making full 360 rotations over lifetime, the animation where the circle fades in growing and spinning is created (see Figure 4.41).

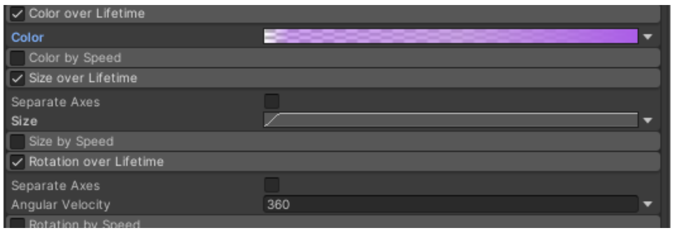

✓ Color over Lifetime
Color
Color by Speed
✓ Size over Lifetime
Separate Axes
Size
Size by Speed
✓ Rotation over Lifetime
Separate Axes
Angular Velocity 360
Rotation by Speed

Figure 4.41: Color, size, and rotation over time for the magic circle

The renderer must be set as a Horizontal Billboard, so the particle shows on the ground. Another horizontal billboard is created, with the default particle material to make the center of the circle glow.

- Trail Ring Particle System:

The trail ring is a particle system that spins around the character like a disk.
Starting by setting the size random between 0.05 and 0.2, a half-transparent color and 10 particles emitted per second. The shape shall be set as a circle around the

character model, and then the orbital speed in the Z-axis is set to 1.17, to make them move following the circle. The next step is to make the particles grow and shrink over time, to do so, the size over lifetime has to be set as a curve like the following shown in Figure 4.42.

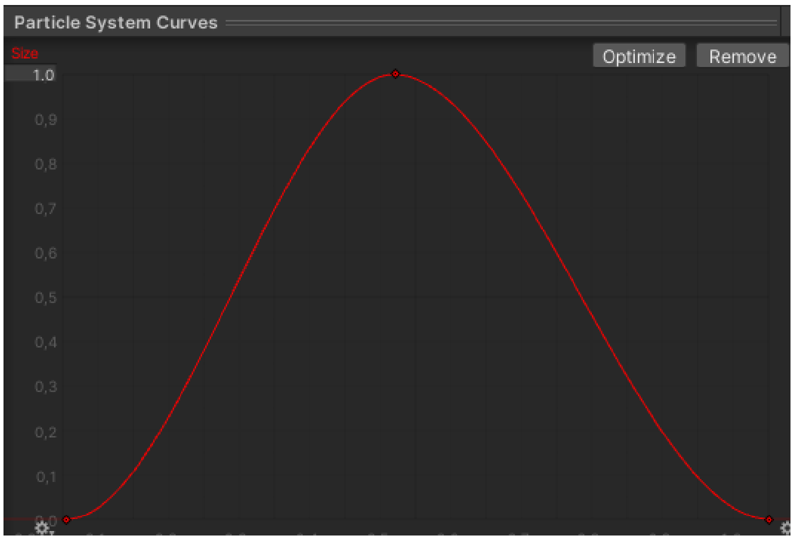


Figure 4.42: Size over time curve for the particle ring

Trails following the particles are set with 0.06 minimum vertex distance, to make them curve smoothly and make the size of the particle affect the width. The color will vary between pink and blue to match the rest of the VFX.

Then a copy of the particle system is created with the orbital speed negative, to make the particles spin in the opposite direction.

4.1.3 Demo Application

The Unity app for displaying the VFX animations consists of three scenes: the main menu, where the user can choose between the two classes, and two other similar scenes, one for each character, where the view can be zoomed and rotated around, and the effects are triggered when the keys shown in the UI are pressed. For the main menu, the character models act as buttons, looping their idle animation and showing a yellow outline when hovering the mouse over as Figure 4.43 shows.

Since the project focuses on action VFX and not UI effects, the outline shader has been obtained from the Unity Asset Store, 'Quick Outline' by PlayHarbor [3].

To make the mouse detect the model, a raycast is projected from the camera, and when it hits one of the character colliders, it enables the Outline component from the game object. When the user clicks the left mouse button during the raycast hit detection, it changes the scene respectively (Figure 4.44). Also, the app quits when pressing the escape key.

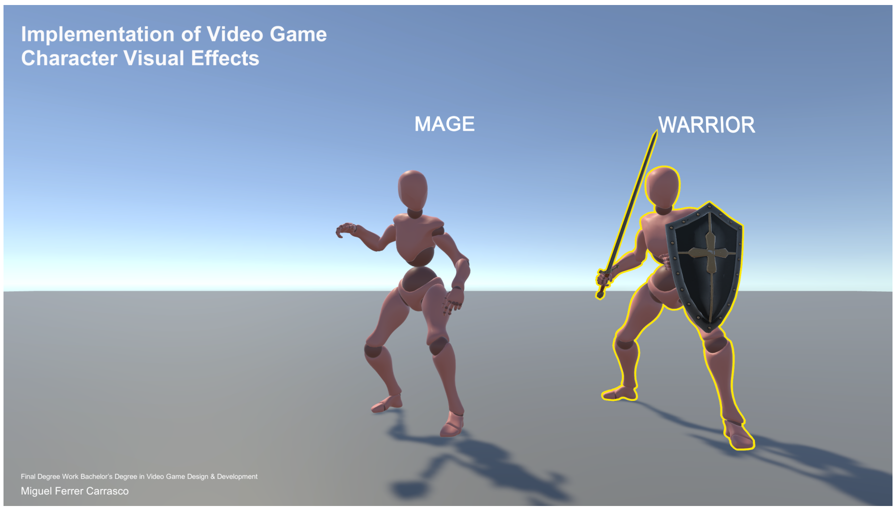
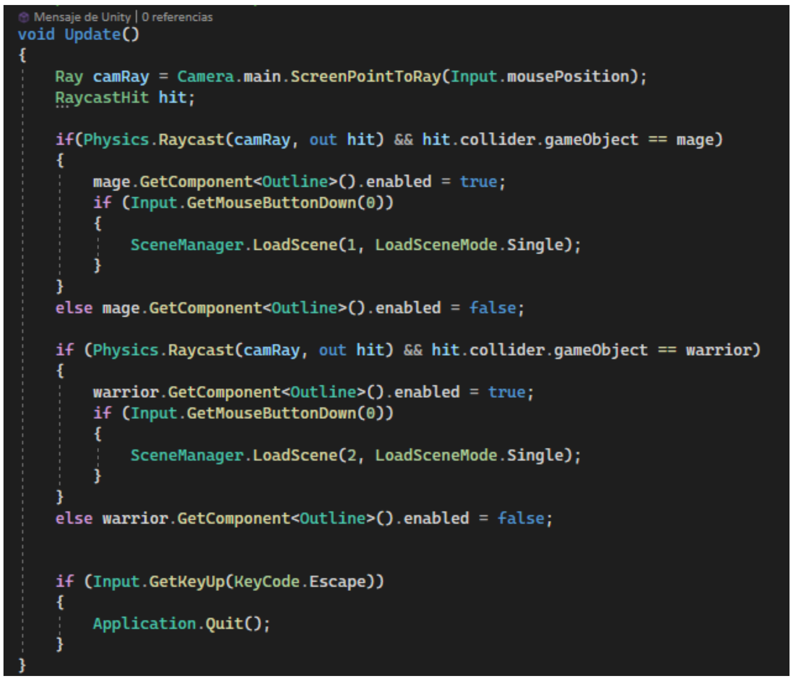

Implementation of Video Game
Character Visual Effects
MAGE
WARRIOR
Final Degree Work Bachelor's Degree in Video Game Design & Development
Miguel Ferrer Carrasco

Figure 4.43: Demo Main Menu with mouse over the warrior



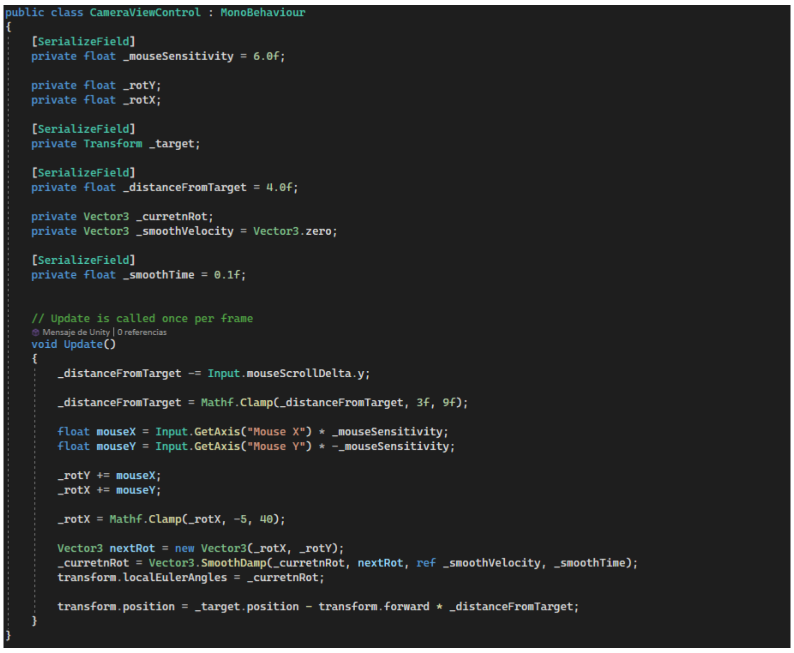
```
Mensaje de Unity | 0 referencias
void Update()
{
    Ray camRay = Camera.main.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;

    if(Physics.Raycast(camRay, out hit) && hit.collider.gameObject == mage)
    {
        mage.GetComponent<Outline>().enabled = true;
        if (Input.GetMouseButtonDown(0))
        {
            SceneManager.LoadScene(1, LoadSceneMode.Single);
        }
    }
    else mage.GetComponent<Outline>().enabled = false;

    if (Physics.Raycast(camRay, out hit) && hit.collider.gameObject == warrior)
    {
        warrior.GetComponent<Outline>().enabled = true;
        if (Input.GetMouseButtonDown(0))
        {
            SceneManager.LoadScene(2, LoadSceneMode.Single);
        }
    }
    else warrior.GetComponent<Outline>().enabled = false;


    if (Input.GetKeyUp(KeyCode.Escape))
    {
        Application.Quit();
    }
}
```

Figure 4.44: Main Menu control setup

The other scenes, which share the same structure, are controlled by a script that manages the triggering of VFX, or character abilities for this matter, and a script for the camera.

The controls are horizontal and vertical mouse axis for rotating around the target, which is the character, clamping the X-axis to avoid the vertical movement making the camera go through the floor or over the character, and the mouse wheel to increase or decrease the zoom (Figure 4.45).

These changes in values of the position and rotation of the camera are smooth to make the control feel more natural to the user.



```
public class CameraViewControl : MonoBehaviour
{
    [SerializeField]
    private float _mouseSensitivity = 6.0f;

    private float _rotY;
    private float _rotX;

    [SerializeField]
    private Transform _target;

    [SerializeField]
    private float _distanceFromTarget = 4.0f;

    private Vector3 _curretnRot;
    private Vector3 _smoothVelocity = Vector3.zero;

    [SerializeField]
    private float _smoothTime = 0.1f;


    // Update is called once per frame
    Mensaje de Unity | 0 referencias
    void Update()
    {
        _distanceFromTarget -= Input.mouseScrollDelta.y;

        _distanceFromTarget = Mathf.Clamp(_distanceFromTarget, 3f, 9f);

        float mouseX = Input.GetAxis("Mouse X") * _mouseSensitivity;
        float mouseY = Input.GetAxis("Mouse Y") * -_mouseSensitivity;

        _rotY += mouseX;
        _rotX += mouseY;

        _rotX = Mathf.Clamp(_rotX, -5, 40);

        Vector3 nextRot = new Vector3(_rotX, _rotY);
        _curretnRot = Vector3.SmoothDamp(_curretnRot, nextRot, ref _smoothVelocity, _smoothTime);
        transform.localEulerAngles = _curretnRot;

        transform.position = _target.position - transform.forward * _distanceFromTarget;
    }
}
```

Figure 4.45: Camera control setup

4.2 Results

The results turned out to be satisfactory, even with some differences from the initial designs, the VFX created are clear and show what they were meant to do. All the objectives established in point 1.2 have been completed.

To test the Demo App and see the results, a Unity application is available to download:

Final build (Google Drive):
`https://drive.google.com/file/d/1V3IOnv4SL2WBJQz5lWyXxId1UVBVXZ3z/`

Also, to check on the development itself, the Unity project can be fully downloaded from the following link:

Unity Project (Google Drive):
`https://drive.google.com/file/d/1pY3wI0CWBPwTYyXCYj9tghuqLK-h7dsg/`

For convenience, there is also a video showcase where all the VFX can be seen in detail and motion without having to download the app from the following link:

Demo Reel Video (Youtube):
`https://www.youtube.com/watch?v=2yhQLxMl-ak&ab_channel=MiguelFerrer`

4.2.1 Warrior Abilities

Slash

Making the blade glow was not part of the design, but watching how the blade and the slash lacked some sort of correlation, it seemed that the best way to make it match was to have the blade glowing in a similar color (See Figure 4.46). This added much more sense to the VFX and visually is more satisfying.

Changing the slash to a bigger and more complex version was indeed an enhancement, since the slash was the main element, it just was not enough with the early version that was finally used in the Ground Slam. For the application, the animation was changed to one with a sequence of three slashes, to better show off the VFX, since the first animation only makes one cut, which made the effect difficult to appreciate.

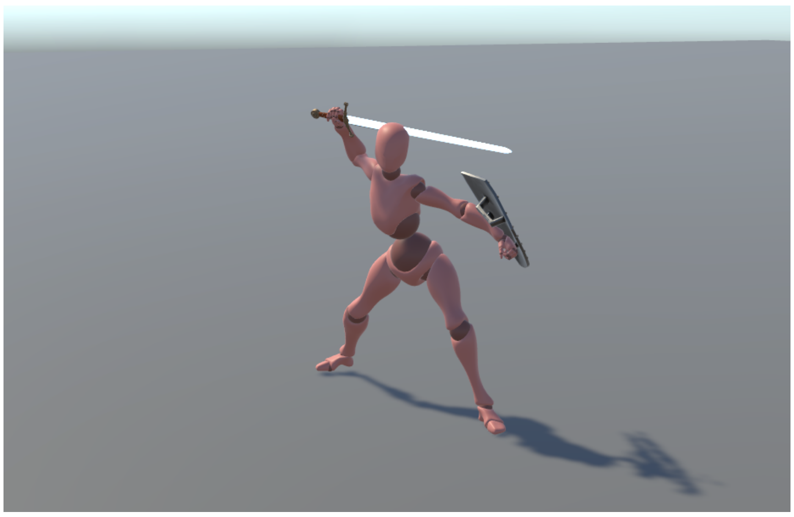


Figure 4.46: Sword model with shader applied

Shield Projection

The transition of the shield shader turned out exactly how the initial design intended it to be. And was even fairly simple to make, interpolating smoothly between the two states.

The projection shield was more difficult to make, there were not many references so this particular effect was entirely made from scratch. The first attempts had many graphic errors which, by investigating the causes, finally were solved, and the outcome ended up nice and smooth (See Figure 4.47).

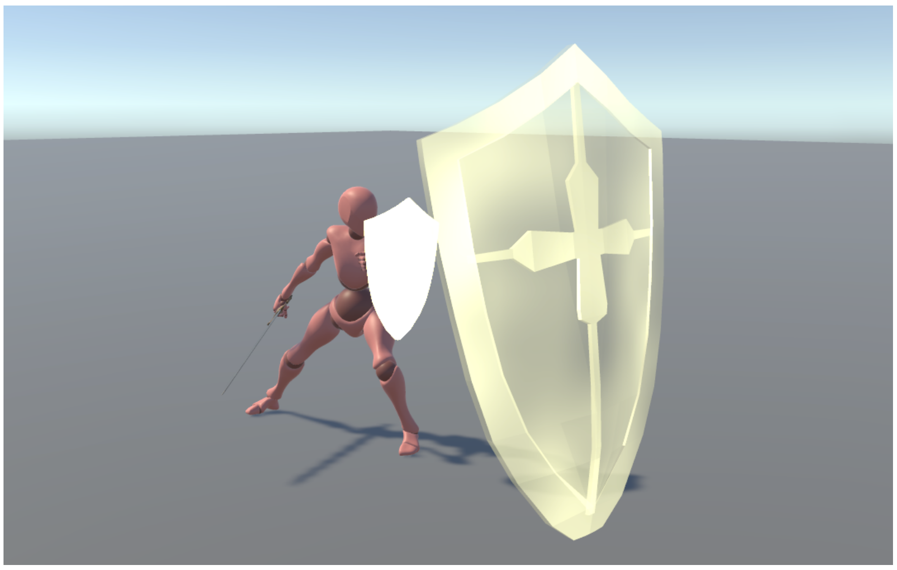


Figure 4.47: Shield projection particle

Ground Slam

The sword shader activation seemed clunky at first because it didn't have any fading transition, but with the addition of the light ray coming from the sky, it just made sense to make the blade change instantly, like it was struck by lightning (See Figure 4.48).

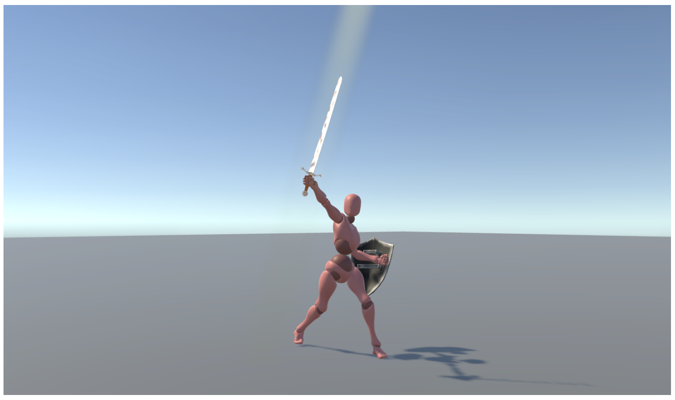


Figure 4.48: Ground Slam sword shader applied

For the slamming part, it loses impact without the many effects that have been applied. Having the slash, the shader, the crack decal, and the dust creates a meaningful composition that expresses forcefulness.

The rocks coming from the ground are one of the particles I am proudest of, the result is very appealing and the models and textures are completely handmade. It is the only effect from this project that has a base color texture and keeps the stylized look that was established in the designs (See Figure 4.49).

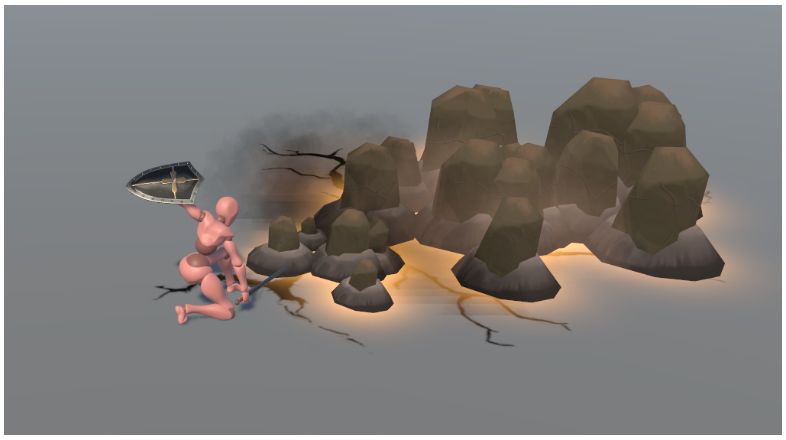


Figure 4.49: Rock particles coming from the ground

4.2.2 Mage Abilities

Fireball

The charging particle was really simple but it adds depth to the effect. Creating anticipation for the VFX makes them more satisfying to see.

The actual fire projectile resulted in a complex effect with many layers and using every technique seen in this document. The outcome was satisfactory, and it follows properly the design and intentions (See Figure 4.50). Even though the details are not seen, being so fast, I think the fire texture could be enhanced.

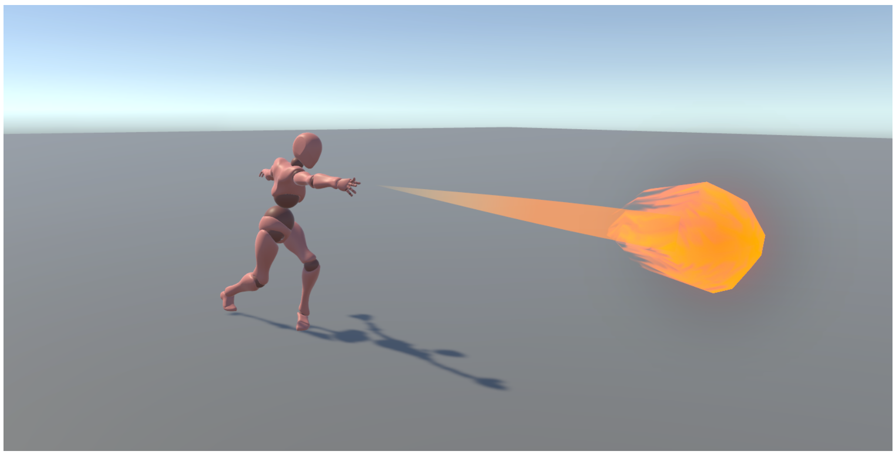


Figure 4.50: Fireball launch

The fireball blast is probably the quickest effect, it is very difficult to appreciate it properly, but that is fine, because as short as it is it properly displays the effect of the fireball.

Lightning Cast

As said in the previous section, anticipation is important for a VFX, it helps the user to expect what's coming and obtain satisfaction when watching a result according to their expectations.

The static electricity made for this effect turn out well, but it could be better if the animation sheet had more frames, and the shapes had more variety, nevertheless, the result is what was expected to be (See Figure 4.51).

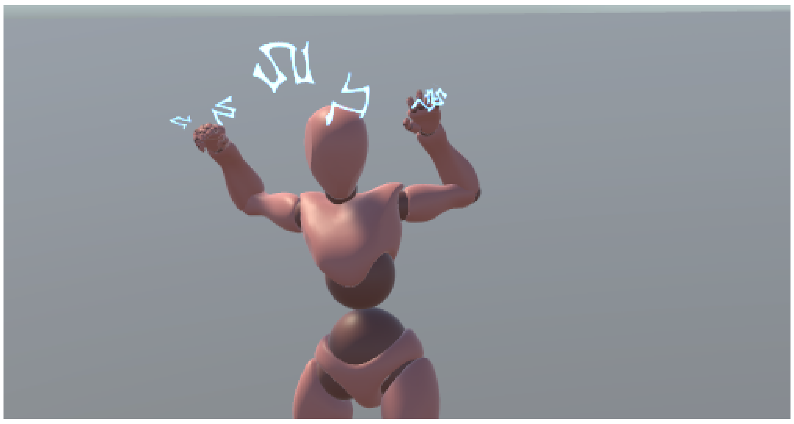


Figure 4.51: Spark charging before casting

The lightning cast was originally supposed to be made manually with animations, but doing it procedurally allowed to control much more of the outcome, and it really went well, the result is appealing and actually random, which makes it more natural for an electric discharge.

Magic Circle

These charging particles proved to be very useful, even so, that for this effect the particles remained on the hands of the character during the lifetime of the VFX.

The transition of this VFX is the most appealing of them all, every component comes into the scene smoothly gradually gaining their actual size and color, with the particles dancing around the character. It really pays off the effort, the result is above all expectations (See Figure 4.52).

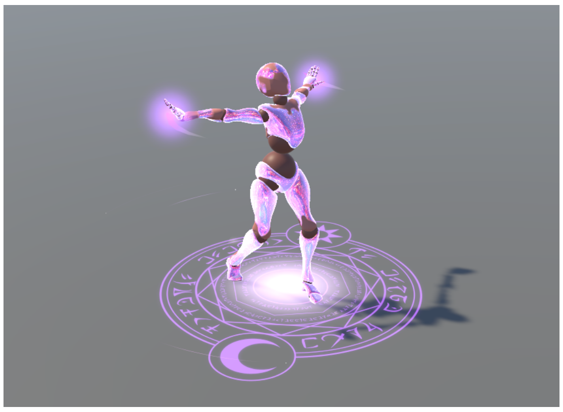


Figure 4.52: Releasing the circle and shader lerping

Watching all the effects linger is very satisfying, the shader of the character is so glossy and brilliant that it stands out overall, even when the main element of the VFX is the circle below, all the components end up orbiting the character, which makes the greatest impression (See Figure 4.53).

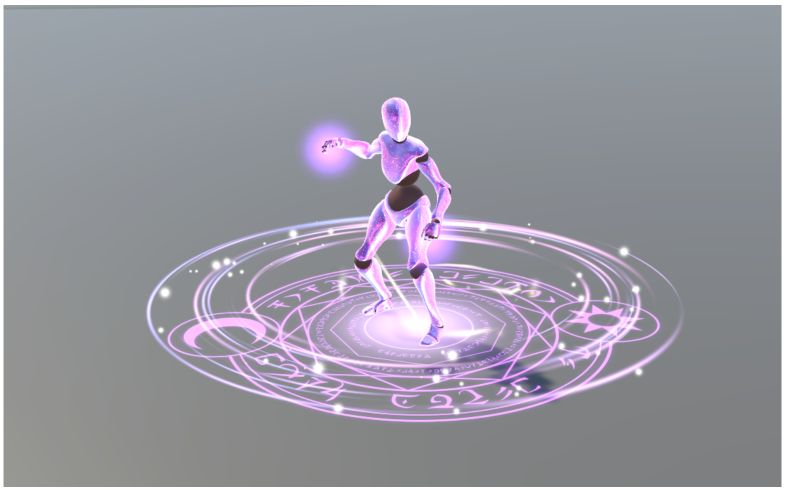


Figure 4.53: Magic circle with all of the particles

CHAPTER 5

Conclusions and Future Work

Contents

This chapter contains a conclusion for the project results and what can be expected as future plans related to this work.

5.1 Conclusions

Creating VFX is a hard and time-taking task that is often overlooked due to be seemingly 'simple' or just because most of the effects made for a composition often last less than a second, which means you could be spending a lot of time working on a particle and making sure it is perfectly synced with the action but the final visual is only seen for an actual instant. This paper shall provide a glimpse of the workflow behind the production of this kind of visual for video games, showing some of the methods that technical artists use to create expressive and dynamic effects.

Nevertheless seeing the complete composition of a VFX really pays the effort, you can create amazing visuals, and engines like Unity provide many tools to get what you are looking for your effects. Even after some design and technique changes, the final result is very close to the initial concept, and all the objectives established at the proposal have been cleared.

5.2 Future work

Although the final results have been satisfactory and fulfilled all the objectives, there is still place for enhancements. Some more than other, but these VFX can still be upgraded by adding more particles to accentuate certain elements and remaking textures with more detail and even more resolution.

So the work hasn't ended yet, this is the star for the next stages of polishing and obtaining results beyond what has been achieved in this project, now without the time constriction that this project initially had.

Besides form still working on this effects, this project has opened a window to apply this methods and procedures to other game engines like *Unreal Engine*, even though the tools are different from the ones that Unity provide, the theory behind is still the same. From the beginning, this project was intended to explain the professional workflow for VFX production, which shall be applied in any environment given.

Bibliography

[1] Microsoft Azure. Fresnel effect definition. https://docs.microsoft.com/en-us/azure/remote-rendering/overview/features/fresnel-effect.

[2] NASA. Milky way center region photography. https://www.nasa.gov/topics/universe/features/milkyway_heart.html.

[3] Chris Nolet. Quick outline. unity asset store. https://assetstore.unity.com/packages/tools/particles-effects/quick-outline-115488description.

[4] Screen Skills. Visual effects artist in the game industry. https://www.screenskills.com/job-profiles/browse/games/technical-art/visual-effects-vfx-artist/.

[5] Wowhead. World of warcraft, shield of the righteous. https://www.wowhead.com/spell=53600/shield-of-the-righteous.

Appendix A

Source code

This appendix contains a total of 9 C# scripts made for each VFX. They're function is to control if the animations can be triggered, when are they triggered and the timing of the majority of the components of the visual effect through coroutines, changing materials or enabling GameObjects.

It also contains the scripts for the camera view control and the scene management in the main menu, where the UI character buttons are also scripted.

Warrior Slash script

```
public class Slash_Script : MonoBehaviour
{
    public Animator anim;
    public GameObject sword;
    public List<Slash> slashList;
    public List<Material> swordMaterials;
    public ReelSceneController controller;

    private bool attacking;
    private bool endAttack;
    private float glowFade;

    void Start()
    {
        attacking = false;
        endAttack = false;
        DisableSlashes();
        glowFade = 1;
    }

    void Update()
    {
        if (!controller.abilityActive && Input.GetButtonDown("Fire1") && !attacking)
        {
            sword.gameObject.GetComponent<MeshRenderer>().material = swordMaterials[1];
            anim.SetTrigger("attack");
            StartCoroutine(SlashAttack());
            attacking = true;
            controller.abilityActive = true;
        }
        if (attacking && !endAttack)
        {
            if (glowFade > 0)
            {
                glowFade -= Time.deltaTime * 0.8f;
                sword.gameObject.GetComponent<MeshRenderer>().material.SetFloat("Fade_", glowFade);
            }
            else glowFade = 0;
        }
        if (endAttack)
        {
            if (glowFade <= 1)
            {
                glowFade += Time.deltaTime * 0.8f;
                sword.gameObject.GetComponent<MeshRenderer>().material.SetFloat("Fade_", glowFade);
            }
            else
            {
                sword.gameObject.GetComponent<MeshRenderer>().material = swordMaterials[0];
                glowFade = 1;
                endAttack = false;
            }
        }

    }

    IEnumerator SlashAttack()
    {
        for(int i=0; i < slashList.Count; i++)
        {
            yield return new WaitForSeconds(slashList[i].delay);
            slashList[i].slashObj.SetActive(true);
        }
        endAttack = true;
        yield return new WaitForSeconds(1);
        DisableSlashes();
        attacking = false;
        controller.abilityActive = false;
    }

    void DisableSlashes()
    {
        for (int i = 0; i < slashList.Count; i++) slashList[i].slashObj.SetActive(false);
    }
}
[System.Serializable]
public class Slash
{
    public GameObject slashObj;
    public float delay;
}
```

Warrior Shield Projection script

```
public class Shield_cast : MonoBehaviour
{
    public float delay;
    public Animator anim;
    public GameObject shieldVFX;
    public GameObject shield;
    public ReelSceneController controller;

    private float glowFade;
    private bool startGlow;
    private bool endGlow;
    private bool casting;

    void Start()
    {
        casting = false;
        glowFade = -1;
    }

    void Update()
    {
        if (!controller.abilityActive && Input.GetButtonDown("Fire2") && !casting)
        {
            anim.SetBool("blocking", true);
            anim.SetTrigger("shield");

            StartCoroutine(StartShield());
            casting = true;
            controller.abilityActive = true;
        }
        if (endGlow && glowFade > -1)
        {
            glowFade -= Time.deltaTime * 2;
            shield.gameObject.GetComponent<MeshRenderer>().material.SetFloat("Fill_", glowFade);
        }
        if (startGlow && glowFade < 1)
        {
            glowFade += Time.deltaTime * 2;
            shield.gameObject.GetComponent<MeshRenderer>().material.SetFloat("Fill_", glowFade);
        }
    }

    IEnumerator StartShield()
    {
        yield return new WaitForSeconds(delay);
        startGlow = true;
        shieldVFX.SetActive(true);

        yield return new WaitForSeconds(7);
        startGlow = false;
        endGlow = true;

        yield return new WaitForSeconds(1);
        casting = false;
        anim.SetBool("blocking", false);
        shieldVFX.SetActive(false);
        endGlow = false;
        controller.abilityActive = false;
    }
}
```

Warrior Ground Slam script

```
public class Slam_Sword_attack : MonoBehaviour
{
    public float swordDelay;
    public float slashDelay;
    public float rocksDelay;
    public Animator anim;
    public GameObject sword;
    public GameObject slash;
    public GameObject crack;
    public GameObject dust;
    public GameObject rocksVFX;
    public GameObject lightRay;
    public List<Material> swordMaterials;
    public ReelSceneController controller;

    private bool attacking;

    // Start is called before the first frame update
    void Start()
    {
        sword.gameObject.GetComponent<MeshRenderer>().material = swordMaterials[0];
    }

    // Update is called once per frame
    void Update()
    {
        if (!controller.abilityActive && Input.GetButtonDown("Fire3") && !attacking)
        {
            StartCoroutine(StartSlam());
            anim.SetTrigger("slam");
            crack.SetActive(true);
            dust.SetActive(true);
            lightRay.SetActive(true);
            controller.abilityActive = true;
        }
    }

    IEnumerator StartSlam()
    {
        yield return new WaitForSeconds(swordDelay);
        sword.gameObject.GetComponent<MeshRenderer>().material = swordMaterials[1];

        yield return new WaitForSeconds(slashDelay - swordDelay);
        slash.SetActive(true);

        yield return new WaitForSeconds(rocksDelay - slashDelay - swordDelay);
        rocksVFX.SetActive(true);
        sword.gameObject.GetComponent<MeshRenderer>().material = swordMaterials[0];
        slash.SetActive(false);

        yield return new WaitForSeconds(4);
        rocksVFX.SetActive(false);
        crack.SetActive(false);
        dust.SetActive(false);
        lightRay.SetActive(false);
        controller.abilityActive = false;
    }

}
```

Mage Fireball casting script

```
public class Spawn_projectile : MonoBehaviour
{
    public GameObject firePoint;
    public List<GameObject> projectiles = new List<GameObject>();
    public float animationDelay;
    public GameObject spellChargeVFX;
    public float chargeDelay;
    public Animator anim;
    public ReelSceneController controller;

    private GameObject effectToSpawn;
    private bool coolingdown;

    void Start()
    {
        effectToSpawn = projectiles[0];
    }

    void Update()
    {
        if (!controller.abilityActive && Input.GetButtonDown("Fire1") && !coolingdown)
        {
            anim.SetTrigger("fire");
            StartCoroutine(SpawnVFX());
            coolingdown = true;
            controller.abilityActive = true;
        }
    }

    IEnumerator SpawnVFX()
    {
        GameObject vfx;

        if(firePoint != null)
        {
            yield return new WaitForSeconds(chargeDelay);
            spellChargeVFX.SetActive(true);

            yield return new WaitForSeconds(animationDelay);
            vfx = Instantiate(effectToSpawn, firePoint.transform.position, firePoint.transform.rotation);
            spellChargeVFX.SetActive(false);
        }
        else
        {
            Debug.Log("No␣hay␣punto␣de␣spawn␣para␣el␣proyectil");
        }
        yield return new WaitForSeconds(2);
        coolingdown = false;
        controller.abilityActive = false;
    }
}
```

Mage Fireball movement script

```
public class Projectile_Move : MonoBehaviour
{
    public float speed;
    public GameObject impactVFX;

    void Update()
    {
        if (speed != 0)
        {
            transform.position += Vector3.forward * speed * Time.deltaTime;
        }
        else Debug.Log("El␣proyectil␣no␣tiene␣velocidad");

    }

    private void OnTriggerEnter(Collider other)
    {
        Instantiate(impactVFX, transform.position - transform.forward * (0.1f), transform.rotation);
        speed = 0;
        Destroy(this.gameObject);
    }
}
```

Mage Lightning script

```
public class Lightning_Cast : MonoBehaviour
{
    public float sparksDelay;
    public float lightningDelay;
    public Animator anim;
    public GameObject lightningVFX;
    public List<GameObject> staticSparks;
    public List<GameObject> lightningSparks;
    public ReelSceneController controller;

    private bool casting;
    private float castingTimer;
    private float stopTime = 3.2f;

    void Start()
    {
        casting = false;
    }

    void Update()
    {
        if (!controller.abilityActive && Input.GetButtonDown("Fire2") && !casting)
        {
            anim.SetTrigger("lightning");
            foreach(GameObject spark in staticSparks) spark.SetActive(true);

            StartCoroutine(SpawnLightning());
            casting = true;
            castingTimer = 0;
            controller.abilityActive = true;
        }

        if (casting) castingTimer += Time.deltaTime;

        if (castingTimer > stopTime)
        {
            lightningVFX.SetActive(false);
        }

    }

    IEnumerator SpawnLightning()
    {
        yield return new WaitForSeconds(sparksDelay);
        foreach (GameObject spark in lightningSparks) spark.SetActive(true);

        yield return new WaitForSeconds(lightningDelay);
        lightningVFX.SetActive(true);

        yield return new WaitForSeconds(3);
        foreach (GameObject spark in lightningSparks) spark.SetActive(false);
        foreach (GameObject spark in staticSparks) spark.SetActive(false);
        casting = false;
        controller.abilityActive = false;
    }
}
```

Mage Magic Circle script

```
public class Circle_Cast : MonoBehaviour
{

    public float delay;
    public Animator anim;
    public GameObject circleVFX;
    public GameObject spellChargeR;
    public GameObject spellChargeL;
    public Material materialGlow;
    public ReelSceneController controller;

    private float dissolve;
    private bool switchShader;
    private bool casting;

    void Start()
    {
        dissolve = -10f;
        casting = false;
        switchShader = false;
    }

    void Update()
    {
        if (!controller.abilityActive && Input.GetButtonDown("Fire3") && !casting)
        {
            dissolve = -10f;
            anim.SetTrigger("circle");
            spellChargeL.SetActive(true);
            spellChargeR.SetActive(true);
            StartCoroutine(StartCircle());
            casting = true;
            controller.abilityActive = true;
        }
        if (casting && switchShader)
        {
            if (dissolve < 10) dissolve += Time.deltaTime * 30;
        }
        if (!casting)
        {
            if(dissolve > -10) dissolve -= Time.deltaTime * 30;
            else dissolve = -10;
        }
        materialGlow.SetFloat("_ShaderDissolve", dissolve);
    }

    IEnumerator StartCircle()
    {
        yield return new WaitForSeconds(delay);
        switchShader = true;
        circleVFX.SetActive(true);

        yield return new WaitForSeconds(10);
        casting = false;
        switchShader = false;
        circleVFX.SetActive(false);
        spellChargeL.SetActive(false);
        spellChargeR.SetActive(false);
        controller.abilityActive = false;
    }
}
```

Main menu controller script

```
public class SceneController : MonoBehaviour
{
    public GameObject warrior;
    public GameObject mage;

    void Update()
    {
        Ray camRay = Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;

        if(Physics.Raycast(camRay, out hit) && hit.collider.gameObject == mage)
        {
            mage.GetComponent<Outline>().enabled = true;
            if (Input.GetMouseButtonDown(0))
            {
                SceneManager.LoadScene(1, LoadSceneMode.Single);
            }
        }
        else mage.GetComponent<Outline>().enabled = false;

        if (Physics.Raycast(camRay, out hit) && hit.collider.gameObject == warrior)
        {
            warrior.GetComponent<Outline>().enabled = true;
            if (Input.GetMouseButtonDown(0))
            {
                SceneManager.LoadScene(2, LoadSceneMode.Single);
            }
        }
        else warrior.GetComponent<Outline>().enabled = false;


        if (Input.GetKeyUp(KeyCode.Escape))
        {
            Application.Quit();
        }
    }
}
```

Camera movement script

```
public class CameraViewControl : MonoBehaviour
{
    [SerializeField]
    private float _mouseSensitivity = 6.0f;

    private float _rotY;
    private float _rotX;

    [SerializeField]
    private Transform _target;

    [SerializeField]
    private float _distanceFromTarget = 4.0f;

    private Vector3 _curretnRot;
    private Vector3 _smoothVelocity = Vector3.zero;

    [SerializeField]
    private float _smoothTime = 0.1f;


    // Update is called once per frame
    void Update()
    {
        _distanceFromTarget -= Input.mouseScrollDelta.y;

        _distanceFromTarget = Mathf.Clamp(_distanceFromTarget, 3f, 9f);

        float mouseX = Input.GetAxis("Mouse␣X") * _mouseSensitivity;
        float mouseY = Input.GetAxis("Mouse␣Y") * -_mouseSensitivity;

        _rotY += mouseX;
        _rotX += mouseY;

        _rotX = Mathf.Clamp(_rotX, -5, 40);

        Vector3 nextRot = new Vector3(_rotX, _rotY);
        _curretnRot = Vector3.SmoothDamp(_curretnRot, nextRot, ref _smoothVelocity, _smoothTime);
        transform.localEulerAngles = _curretnRot;

        transform.position = _target.position - transform.forward * _distanceFromTarget;
    }
}
```