# GAME AUDIO AND MUSIC

A Complete Guide to
Creation, Design, and Integration


INCLUDES A COMPLETE
DOWNLOADABLE GAME PROJECT
*THE WANDERING TERROR*
BUILT WITH UNITY AND FMOD

ANTONIO VILLA

E-BOOK
PDF • EPUB

## How This Book Came to Be?

If someone had told me when I was seven years old, sitting in front of an Amstrad CPC 464 while waiting what felt like an eternity for a game to load from a cassette tape, that one day I would write a book about audio and music implementation in video games, I probably would not have believed them.

That computer came with a manual that sparked an unexpected curiosity in me: discovering how just a few lines of BASIC code could bring a program to life.

Not long afterwards, I began taking private piano lessons—the first instrument I ever played. Looking back, that decision would have a far greater impact on my life than I could possibly have imagined at the time. Over the years, music gradually eclipsed my other passions—gaming, programming, sports, archaeology, and many others—ultimately leading me to pursue it professionally. I went on to earn a degree in Music, a teaching degree in Music Education, two master's degrees (in Historical Musicology and Music Composition), and a Higher National Diploma in Sound for Audiovisual Productions and Live Events.

Even so, amidst all that musical training, and almost by chance, I enrolled in a web programming course one summer. It rekindled my fascination with the world of ones and zeros, and before long I found myself devouring languages such as HTML, JavaScript, PHP, Java, and CSS. It reminded me that I never wanted to leave this extraordinary world—a world where a simple keyboard could be used to create an endless variety of tools, applications, and experiences.

As the years passed, I balanced my work as a composer with my career in education, always hoping that one day I would be able to combine two of my greatest passions in a single project: programming and music.

The turning point came when, several years ago, one of my students asked me how music is implemented in video games. To my embarrassment, I had to admit that I had no idea. I knew how to compose music for games, but I had no understanding of the processes that allow that music to become an integral part of the game itself.

My curiosity was immediately awakened, and I began researching the subject from different perspectives. I did not want to focus solely on audio and music implementation in isolation because I knew that, if I ever decided to write about it, I would first need a solid understanding of how video games are actually created. My goal was to gain a broad overview of the game development process before concentrating on the implementation of audio and music.

And that is exactly what I did.

Little by little, I explored the inner workings of game development: game engines, concepts such as Components and GameObjects, asset management, and, of course, programming. Along the way, I discovered C#, a language that immediately felt both intuitive and familiar thanks to my previous experience with Java.

After several months of intensive study, I had developed a much clearer understanding of modern game development. It was then that I chose the technologies on which this project would be built: Unity as the game engine and FMOD as the middleware for creating and managing audio and music events.

The reasons were compelling.

Both are exceptionally powerful tools and, for small-scale projects, available free of charge. They boast impressive professional portfolios—Unity has been used to develop titles such as Pokémon GO and Hearthstone, while FMOD powers games created by leading companies including Activision, Microsoft, and Capcom. Just as importantly, both are widely adopted throughout the industry and remarkably intuitive to learn.

Accessibility, versatility, and professional quality therefore became the guiding principles behind my choice of software. My objective was simple: to ensure that you, the reader, could not only acquire as much knowledge as possible, but also apply it confidently to your own game development projects.

The next obvious question was the million-dollar one: What exactly did I want to write about?

At first, a question of that magnitude can seem almost impossible to answer. It proved much easier, however, to decide what I didn't want to write.

I had no interest in producing a dense, overly academic treatise with little or no practical application—one of those books that inevitably ends up gathering dust on a shelf or propping up a wobbly table. From the very beginning, I knew this book had to be practical. Readers needed to start building something real as early as possible, even if that "something" was initially simple.

The question was... what?

To answer that question, I imagined a real-world scenario.

Suppose you have just been hired by a game development studio and are responsible for implementing all the audio and music systems in a new game. However, this is your first time facing such a challenge, and you lack the fundamental knowledge required for the job. Deadlines are approaching quickly, and before you find yourself drowning in a sea of anxiety—or developing a serious case of impostor syndrome—you decide to take a step back and ask yourself a few essential questions:

What audio and music systems are common to almost every video game?

Which gameplay events require sound or music?

What do I need to do to create those systems and implement them in Unity?

Answering these questions within the context of that hypothetical scenario not only helped me deepen my research, but also shaped the teaching methodology that would eventually become the foundation of this book. As the goblins in *World of Warcraft* famously say, *"Time is money, friend."* Every section therefore had to be clear, practical, and straightforward, designed to solve a specific problem that a newcomer might encounter while helping them gradually develop the skills and knowledge needed to build a career in game development.

After several months of brainstorming, I began outlining what would eventually become the structure of the book. Each chapter was conceived as a self-contained learning unit focused on one or more challenges related to audio and music implementation in video games, together with the techniques required to solve them. As my research progressed, those initial ideas continued to grow in both number and complexity until they ultimately evolved into fourteen complete chapters.

Each chapter follows the same instructional structure. It begins with a brief introduction to a specific problem or requirement, continues by presenting the tools we will use to solve it, guides the reader through the creation of the solution, and concludes with its implementation inside the game through C# scripting.

Consider a simple example: adding sound effects to player actions.

We begin with an introduction to **one-shot events**, the type of event most commonly used for player actions, NPC interactions, and many other gameplay situations. Next, we explore the FMOD tools required to create one-shot events—including event types, instruments, and other essential concepts—before building the event itself and finally implementing it in Unity through code.

At that point, the destination was much clearer and everything was ready to begin. As with any creative endeavour, however, the journey was far from free of obstacles.

Professional programmers face challenges every day, whether related to code, optimisation, or countless other technical issues. As a composer and teacher venturing into a field that was largely new to me, I naturally encountered even more.

Fortunately, every mistake became an opportunity to learn.

For that reason, throughout this book I have included not only the problems I encountered, but also the solutions that worked for me. My hope is that, should you stumble across the same obstacles, you will be be able to overcome them quickly and continue moving forward without unnecessary frustration.

That said, it is important to remember that there is rarely a single correct solution to any given problem. While writing this book, I often developed several different approaches before selecting the one I believed to be the clearest, most practical, and most effective from a teaching perspective.

By then, my ideas had become much clearer, and I had already covered a considerable part of the journey. Even so, I still felt that something was missing. I knew which problems I wanted to solve, which knowledge I wanted to share, and which teaching methodology I wanted to follow... but there was one crucial element still absent.

A video game.

After all, how could this be a truly practical learning experience without a game in which to test everything we had learned?

The Wandering Terror Is Born

I immediately got to work. With the outline of the book's fourteen chapters in hand, I began designing the Game Design Document for the video game that would serve as the practical foundation of the project. Its purpose would be to allow readers to apply the concepts presented throughout the book in a simple, intuitive, and progressive way. Every chapter would have a direct reflection inside the game.

Keep in mind that this is not a book about Unity or FMOD. It is a book about audio and music implementation in video games, which makes the game itself an essential part of the learning experience.

That is how, in collaboration with a professional game programmer, **The Wandering Terror** was born: a simple RPG[1] specifically designed to explore, step by step, the audio and music requirements that arise during game development. We begin with the fundamentals and gradually move towards more advanced techniques and workflows.

Throughout the book, we will build a complete interactive audio system for this game. Every new concept you learn will be immediately applied within the project, allowing you to see how it works in a real development environment.

By now, you may already be halfway convinced to buy this book—or, if you already have, to keep reading. The only question that may still remain is whether this book is the right fit for you.

The answer is a resounding **yes**.

Because of its multidisciplinary approach, this book has been written for a wide range of professional profiles.

---

[1] Role Play Game.

If you are a programmer or game developer looking to expand your expertise into sound design, audio implementation, or music composition, this book provides a comprehensive and practical guide to the techniques and workflows used to create professional-quality game audio.

If you are a composer, you will gain valuable insight into the game development process while learning sound design techniques, event creation in FMOD, and the C# code required to implement your work inside a game.

If you are a sound designer, in addition to discovering creative audio processing techniques, you will gain a deeper understanding of video game music composition, event design, and the programming required to integrate interactive audio systems successfully.

Finally, if you have no previous experience in any of these disciplines but would like to enter the fascinating world of game audio, this book is also for you. Its practical and educational approach assumes no prior knowledge and will help you develop, step by step, the skills required to tackle many of the tasks involved in professional game audio implementation.

My hope is that this book will do more than teach you how to implement audio and music in video games. I hope it inspires the same curiosity that was awakened in me by a student's question several years ago.

If, by the time you reach the final page, you are able to look at a video game through the eyes of a sound designer, a composer, and a game developer simultaneously, then this book will have fulfilled its purpose.

## About the Author

Antonio Villa is a music and philosophy teacher, composer, audio designer, and performer with more than twenty years of experience in education, music composition, and live performance.

His academic background combines expertise in music, education, and audio technology. He holds a Bachelor's Degree in Music, a Bachelor's Degree in Music Education, a Higher Technician Diploma in Sound for Audiovisual Productions and Live Events, a Master's Degree in Music Composition with New Technologies, and a Master's Degree in Historical Musicology. He is currently pursuing a PhD in Music and Music Science and Technology, where his research focuses on the application of artificial intelligence, semiotics, and the psychology of emotion to music composition.

Throughout his career, Antonio has combined teaching with music composition and audio design while maintaining a strong interest in the relationship between music and technology. This unique combination of artistic, technical, and educational experience has enabled him to develop a practical, project-based teaching methodology that helps learners understand complex concepts through real-world applications.

In *Game Audio and Music: A Complete Guide to Creation, Design, and Integration*, Antonio brings together two of his greatest passions: music and programming. The result is a hands-on guide that takes readers through the complete game audio workflow—from music composition and sound design to the implementation of interactive audio systems in a video game developed specifically for educational purposes.

**Website:** www.antoniovillamusic.com

**Contact:** info@antoniovillamusic.com

## 3.5 Connecting FMOD and Unity

Up to this point, we've been working with FMOD and Unity as separate applications. It's now time to connect them so that the audio events created in FMOD can be played directly within Unity.

To do this, open: **FMOD → Edit Settings**

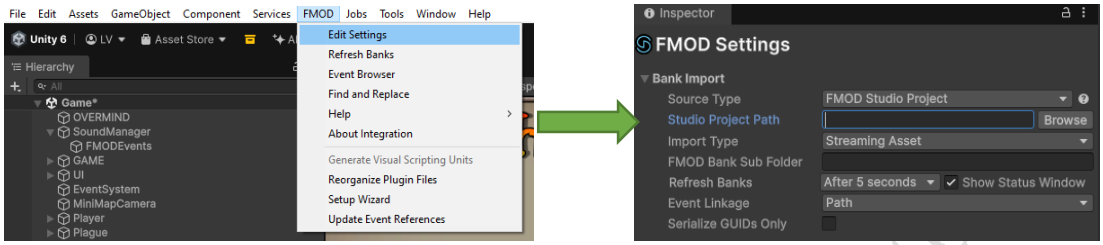


Figure 54. Accessing the FMOD settings in Unity.

Since we'll be building the entire project (`FMOD Studio Project`), locate the **Studio Project Path** field and select the folder containing your FMOD Studio project.

To verify that Unity is correctly connected to FMOD, we'll use the **Event Browser** included with the FMOD Unity integration.

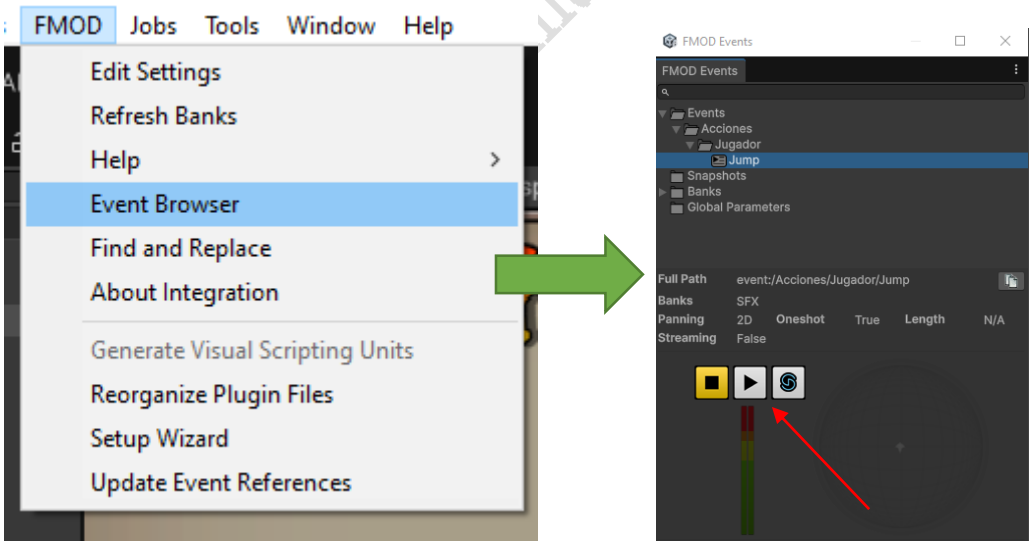


Figure 55. FMOD Event Browser in Unity.

### Debug.Log()

This is the most commonly used debugging method. It prints messages to the Console, allowing you to inspect variable values or follow the program's execution flow.

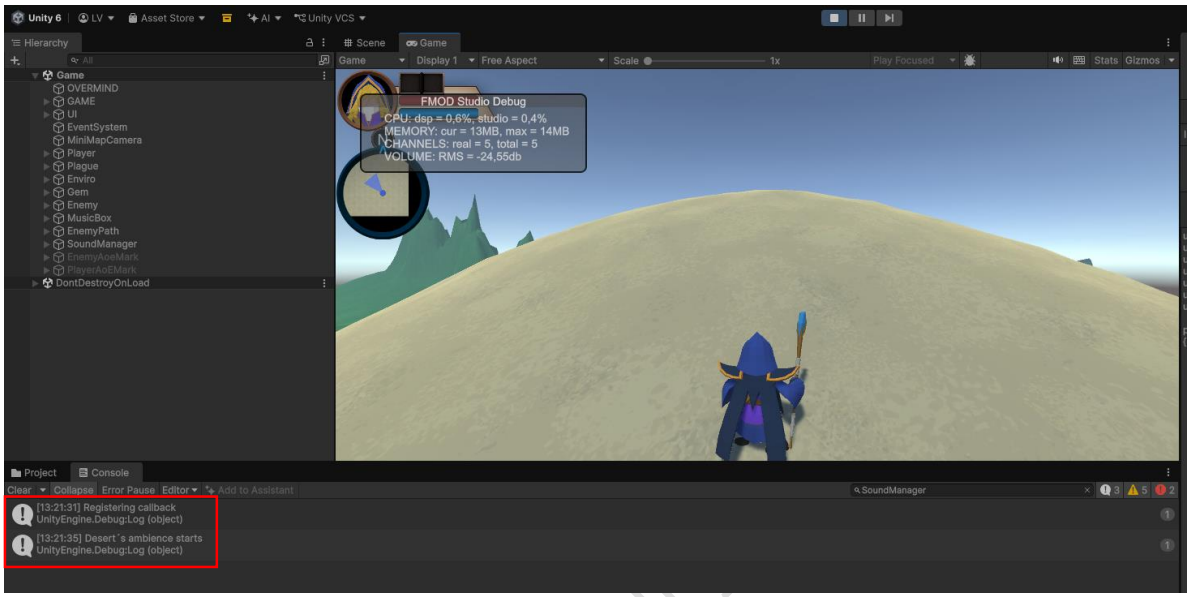


Figure 65. Example of Debug.Log messages in the Unity Console.

### Debug.LogWarning()

Similar to `Debug.Log()`, but intended for warnings. Messages appear in **yellow**, indicating that something is not necessarily wrong but could cause problems later.

### Debug.LogError()

Displays **error messages** in the Console. These messages appear in **red** and usually indicate critical problems in your code.

If **Error Pause** is enabled in Unity, the Editor automatically pauses whenever a `Debug.LogError()` message is generated.

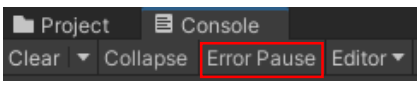


Figure 66. Unity Console window.

**Unity will not allow the project to run if compilation errors are present**, making the Console an excellent tool for verifying that your code is correct.

### *Playing the ambience*

1. Returning to the integration of the **desert ambience** into our project, the code responsible for playback is located in the **`PlayMusic(EMusic music)`** method inside the **`SoundManager.cs`** script. For efficiency and convenience, this code will manage both **music and ambience playback** throughout the game using **FMOD**. Whenever the player leaves a specific area (desert or forest), the game detects the change and, through code, stops or starts the appropriate music and ambience.
2. The `PlayMusic(EMusic music)` method requires several variables. If we do not declare them first, the code will produce an error. We will declare the `EMusic` variable (an **enum** type) and the `currentMusic` variable, which stores the **currently active** value of that enumeration.

```
public enum EMusic { Menu, Desert, Forest, BossApproach, Boss, }

public EMusic currentMusic;
```

3. Now that both have been declared, we can write the method. Make sure that all instances are correctly referenced; otherwise, an error will occur. **Visual Studio** usually indicates errors with a **red wavy underline** beneath the incorrect word or character. In the following example, the code responsible for stopping the desert ambience is incorrect.

```
public void PlayMusic(EMusic music)
{
    switch (currentMusic)
    {
        case EMusic.Desert:
            //stop area music

            break;

        case EMusic.Forest:
            //stop area music
             break;
    }
     currentMusic = music;
    switch (music)
    {
        case EMusic.Desert:
            //play area music

            //play area ambience
            DesertAmbience.start();
            Debug.Log("Starting desert´s ambience");
            break;
        case EMusic.Forest:
            //play area music
            DesertoAmbience.stop(FMOD.Studio.STOP_MODE.ALLOWFADEOUT);
            Debug.Log("Stopping desert´s ambience");
            //play area ambience
            break;
    }
}
```

*Creation of methods to develop the logic of the player´s heartbeats*

17. created, locate the code responsible for managing the player's health. This can be found in **Player.cs**.
18. Inside **Player.cs**, locate the section that controls the player's health logic. Once found, add the heartbeat methods using the following conditional logic.

```
public override void ReceiveDamage(float damage)
{
    if (Game.MusicBox.IsPickedUp && Game.Gem.IsPickedUp)
    {
        damage *= 0.25f;
    }
    if (Game.Enemy.IsEnraged)
    {
        damage *= Game.Enemy.enrageDamageModifier;
    }
    base.ReceiveDamage(damage);
    Game.UI.RefreshPlayerHealth(currentHealth / health);
    int IntHealthPlayer = Mathf.RoundToInt(currentHealth);
    // Change
    if (IntHealthPlayer <= 50)
    {
        SoundManager.Instance.HeartbeatsPlay();
        SoundManager.Instance.HeartbeatsUpdate(IntHealthPlayer);
        //Debug.Log("Player Health is " + IntHealthPlayer);

        if (IntHealthPlayer == 0)
        {
            SoundManager.Instance.HeartbeatsStop();
            // Stop the heartbeat
        }
    }
    else
    {
        SoundManager.Instance.HeartbeatsStop();
    }
}
```

Damage modifiers

**Converting the `currentHealth` variable to the integer variable `IntHealth`**

If the player's health is less than or equal to 50, the instance will be started and updated. Otherwise, it will be stopped using a *fade-out*

**If the player dies (health = 0), the instance is stopped immediately.**



## Code explanation

First, declare the integer variable **IntHealthPlayer**, which is obtained by converting the floating-point variable **currentHealth** using **Mathf.RoundToInt(currentHealth)**. This variable stores the player's current health.

The first condition:

```
IntHealthPlayer <= 50
```

Checks whether the player's health is **50% or lower**. If the condition is met, the code inside the braces is executed.

In other words, we would have four VCA groups whose respective *faders* would allow us to control the level of all dialogue, all SFX, all music tracks, and all ambience through their respective *VCA faders.*

Our menu, using *sliders*, will allow us to modify the levels of our VCA groups.


*Illustration 204. Unity menu view and VCA configuration in FMOD.*

1. Now that our project is configured, we will go to the mixer (*Mixer*, Ctrl + 2) and perform the mix on the groups and VCAs. We will leave the VCAs at 0 dB.
2. Once everything has been assigned, we will export all the banks (F7).
3. We then go to Unity and run the game. At the same time, we will **synchronize** FMOD with Unity to check that the mixer is working correctly. We will do this using *Live Update* and selecting *Localhost.*

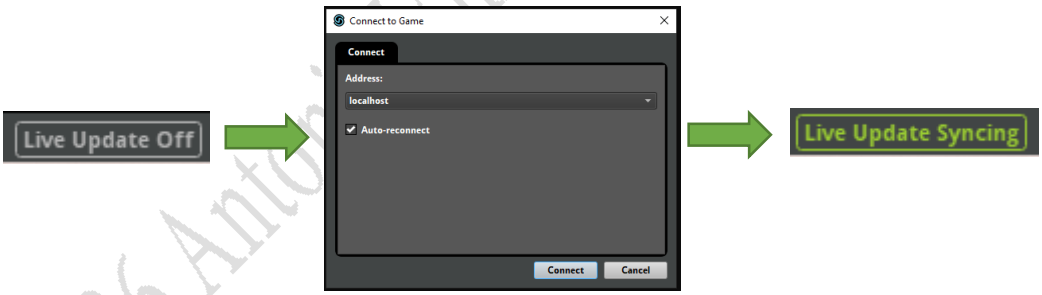


*Illustration 205. Synchronization between FMOD and Unity.*

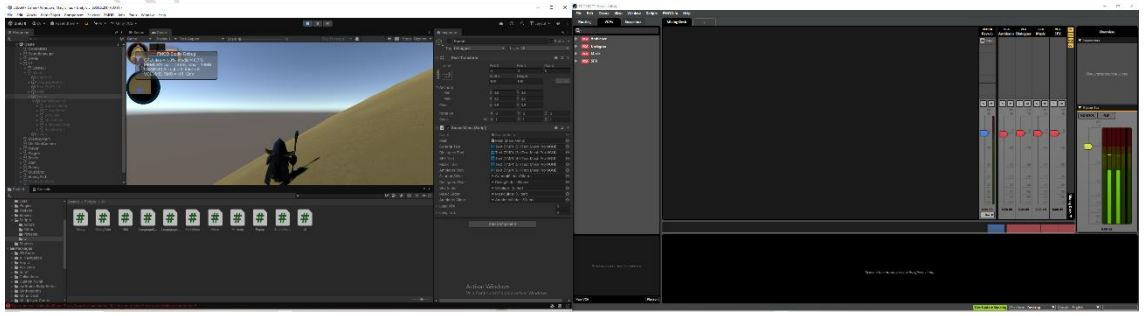

***Illustration 206. Synchronization between FMOD and Unity.***

1. Now, in the bar 17, we will add the **final destination marker**: Death.

The result should look similar to the following:

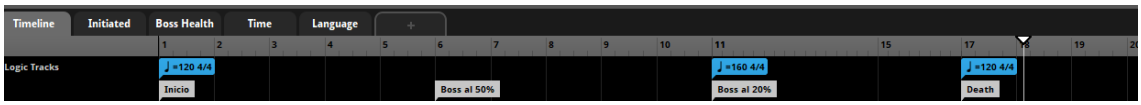


*Figure 172. Adding tempo markers.*

### *Creating two logic levels: Markers and Tempo*

Remember that, to prevent regions and transitions from overlapping, we can select a marker or region and drag it up or down to create different logic levels:

**Creating Two Logic Levels: Markers and Tempo**


*Figure 173. Assigning markers in FMOD.*

2. Once the markers have been created, we will **create the first transition region** (*Add Transition Region*). We will place this transition region on the second level[2] (above the markers[3] and below the *tempo*). We will configure it as follows:
   - Destination: Boss at 50%.
   - We will leave the remaining options unchanged.
3. To **assign the region to which we want to transition**, simply select the region, right-click, and choose **Set Destination to > Markers >** *the marker of our choice* (in this case, *Boss at 50%*).

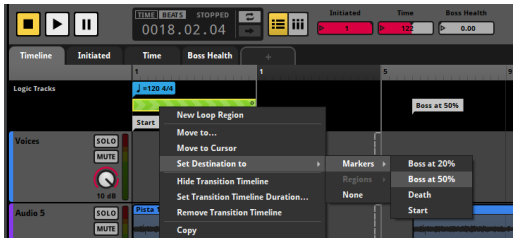


*Figure 174. Selecting the marker to which a transition region will lead.*

[2] Remember that the higher an element is positioned on the timeline, the higher its priority.
[3] To make room for the transition region between the marker and the tempo marker, simply drag the marker slightly downward, taking care not to create more than one additional level.

4. We will now **adjust the length of the transition regions**. To adjust a transition region, first select it and place the cursor over the edge of the region you want to modify until the cursor changes. Once it has changed, we can adjust the length of the region.
5. The transition conditions for the different regions are shown in the following tables:

*Table 21. Transition regions, parameters, and transition conditions.*

| Transition Regions | Boss Health | AND / OR | Time | Initiated | Event State |
|---|---|---|---|---|---|
| **Boss at 50%** | 50–20.1 | AND | 0–119 sec | 1 | Not Stopping |
| **Boss at 20%** | 4.1–20 | AND | 150–500 sec | 1 | Stopping |
| **Death** | 0–4 | AND | — | 1 | Not Stopping |

*Table 22. Control regions, parameters, and transition conditions.*

| Control Regions | Boss Health | AND / OR | Time | Initiated | Event State | Bar |
|---|---|---|---|---|---|---|
| To Start | 50.1–100 | AND | 0–119 sec | 1 | Not Stopping | 6 |
| To 50% | 20.1–50 | AND | 0–119 sec | 1 | Not Stopping | 11 |
| To 20% | 4.1–20 | OR | 150–200 sec | 1 | Not Stopping | 16 |
| To 20% | 0 | AND | — | 1 | — | 17 |

The control regions act like a typical **nightclub bouncer**, who decides whether or not to let the player proceed to the next block. Therefore, remember to close them without including any audio file; otherwise, you will create an awkward silence that interrupts the looped playback of the section currently being played.

The logic is simple: **if the event does not meet the required conditions, it cannot move to the next block and will continue playing in a loop until those conditions are met.**

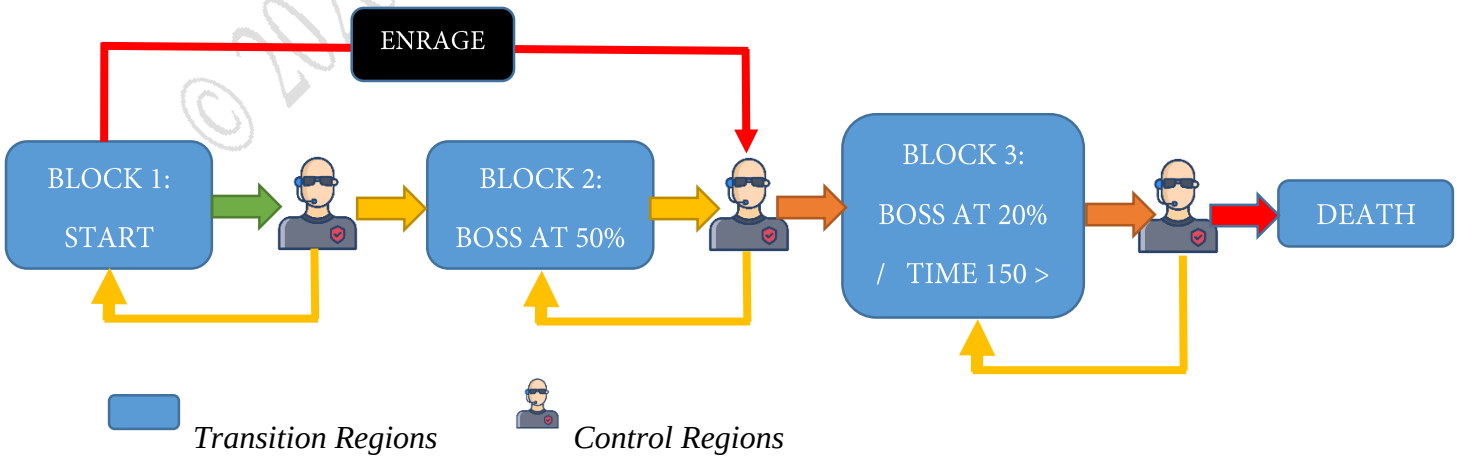


*Figure 177. Logic of the BossEncounter event in FMOD.*

 *Boss Voices*

6. We will configure the boss's voices so that they are triggered according to four parameters:
    - **Language:** This will depend on the user's language selection. If the player chooses **English (0)** in the game, the **Voices ENG** track will be used; otherwise, **Voices SPA** will be used. Both tracks contain the same instruments and activation logic, but obviously in their respective languages. The **Language** parameter is therefore added as an additional condition to distinguish between them and ensure that the correct voice is played.
    - **Initiated:** When the **Initiated** parameter has a value of **1**, a boss voice file will be played.
    - **Boss Health:** As the boss's health decreases, we will place different audio files at appropriate points. It is important to leave sufficient separation between them to prevent the voice lines from overlapping.
    - **Time:** When its value reaches **150 seconds**, we will trigger an audio file to accompany the transition to the *enraged* block.



The setup for the **English voices** will be as follows:

*Table 24. Voice implementation scheme for the Boss Encounter event in English.*

| File | Language | Boss Health | Initiated | Time |
|---|---|---|---|---|
| *BossTalkGreet* | *English (AND activation condition)* | | *1* | |
| *BossTalkEnrage* | *English (AND activation condition)* | | | *150* |
| *BossTalk60_30* | *English (AND activation condition)* | *57–60 and 27–30* | | |
| *DarkBomb_ENG* | *English (AND activation condition)* | *47–50, 17–20 and 7–10* | | |

The setup for the **Spanish voices** will be as follows:

*Table 25. Voice implementation scheme for the Boss Encounter event in Spanish.*

| File | Language | Boss Health | Initiated | Time |
|---|---|---|---|---|
| *Greet_SPA* | *Spanish (AND activation condition)* | | *1* | |
| *Enrage_SPA* | *Spanish (AND activation condition)* | | | *150* |
| *Rival_SPA* | *Spanish (AND activation condition)* | *57–60 and 27–30* | | |
| *Bomba_Oscura* | *Spanish (AND activation condition)* | *47–50, 17–20 and 7–10* | | |

## Adjusting Boss Health Values for Voice Playback

If you want to change the **Boss Health** values at which the audio files are triggered, go to the *Enemy.cs* script and modify the values in the following code:

```
//Moments when the Boss speaks according to its health
    public float[] talkHealthPoints = new float[] { 0.6f, 0.3f };
    bool[] wasTalkOnHealthPoints;

    public float[] spell2HealthPoints = new float[] { 0.5f, 0.2f, 0.1f };
    bool[] wasSpell2OnHealthPoints;
    //-------------------------------------------------------------
```

Alternatively, you can modify them directly in the **Unity Inspector**, in the *TalkHealthPoints* and *Spell2HealthPoints* fields.

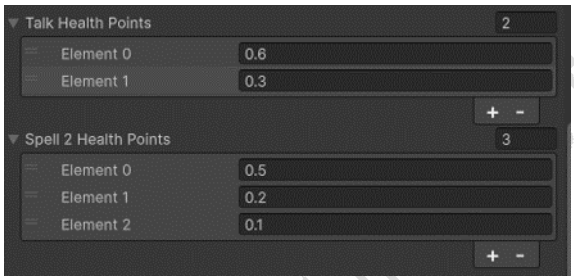


*Figure 184. Modifying the values used for voice playback.*

We have now configured the voices. Let's insert them into the event.

7. We will begin with the voices that introduce the encounter with the *Boss*. To do this, we will go to the **Initiated** parameter sheet and place the two files at value **1**, each on its respective track. We can either drag the audio files directly from *Assets* or create *single instruments*. Once they have been created, we must add the **Language** condition so that the appropriate voice is played according to the selected language.

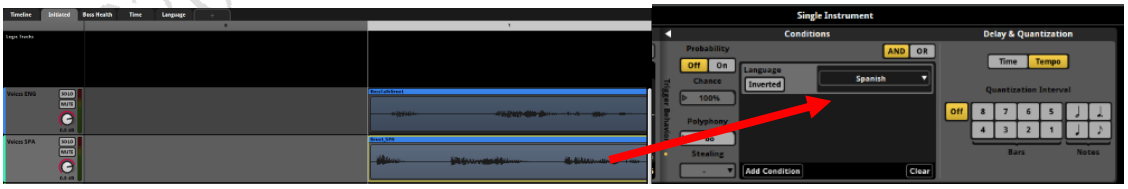


*Figure 185. Adding a track in the Initiated parameter sheet and a language condition to the track.*

8. We will continue with the **Boss Health** parameter sheet, where we will place the *single instruments* at the previously specified boss health values.

# MASTER GAME AUDIO AND MUSIC. FROM IDEA TO IMPLEMENTATION.

This comprehensive guide takes you from the foundations of sound and music to the integration of advanced audio systems in Unity using FMOD.

Through clear explanations, practical examples, and a complete game project, you'll learn how to design interactive music, implement dynamic audio, work with dialogues, and optimize your game's sound for a professional result.

- Music composition and interactive systems
- FMOD from basics to advanced implementation
- Complete Unity integration with C#
- Dialogues, localization, and multilingual support
- Optimization, profiling, and performance

## INCLUDES A COMPLETE GAME PROJECT

Explore a complete Unity project (The Wandering Terror) and learn by doing.

Modify every audio system, experiment, and see the results in-game.

## ANTONIO VILLA

Composer, sound designer, and educator with over 20 years of experience in music, audio, and teaching.

Passionate about game audio and interactive music, he combines academic rigor with practical experience to help you turn your ideas into immersive soundscapes.

DOWNLOAD THE PROJECT

UNITY PRACTICAL

FMOD INTEGRATION

PROFESSIONAL TECHNIQUES