**SHAMANIC**

**SYNESTHESIA**

**Audio Implentation**

Audio For Games 2022

Shamanic Synaesthesia is a novel, audio-focused environment in which a natural ecosystem deteriorates or blooms as the result of player-driven actions. The theme of the game is a synthesis of Neanderthalic, Shamanic and Biblical themes. The graphical style utilises warped graphics to evoke and embrace certain distinct imperfections of early-1990’s 3D, while not strictly adhering to their limitations. The audio is also inspired by this aesthetic.

The world contains a variety of trees and plants with individual growth cycles of life and death. The world also contains interactable NPC’s each with their own distinct audio design and implementation method.

**Opening the project:**

Recommended Versions:

Unity: **2022.2.5f1**

Unity FMOD Integration: **2.02.11**

FMOD Studio: **2.02.11**

When you open the project, you will receive the following warning regarding the Input System:

Graphical user interface, text

Description automatically generated

Select **‘No’.** The project uses the old input system.

**Warnings**

**Known Errors/Bugs:**

* **In the editor, the audio sometimes loads every other time the game is loaded.**
* **Using the flute inside the cave can cause the player movement to break.**
* **Generative music may sometimes fail to update the musical mode and get stuck one mode.**
* **Because of the unique camera type, selecting things can be unreliable as the frustrum dilates as the player rotates the mouse wheel, which affects the scripts responsible for selecting items.**
* **The camera may fly underground when picking a mushroom (Returns to normal upon exiting the behaviour)**

The full development of this game spanned approximately 7 months. Its development brought about many technical challenges and learning curves that required extensive research in all manner of areas of game development.

**INDEX**

**Project Hierarchy**

The hierarchy has been organised through means of coloured, titled strips using [Pretty Objects](https://assetstore.unity.com/packages/tools/gui/pretty-objects-201639) (Amit Klein, 2021). This was essential in maintaining some level of sanity when developing as the complexity and interdepency of the project grew. Typically, the hierarchy is typically exclusively grey, which can make it difficult to quickly navigate the project via the hierarchy.

**== AUDIO ==** contains the Music Manager script, under the ‘Music’ Transform. This script is responsible for controlling various musical parameters at runtime. This includes the arrangement, playback, and modulation.

Graphical user interface

Description automatically generated

**== MAP==** contains the **Map Object Generator** transform. This contains several lists containing the prefabs for that are generated at runtime.The MapObjGen script attached to this object

**== SOUND EMITTERS ==** contains two empty Transforms that act as containers for runtime generated audio emitters that are generated elsewhere.

Water Emitters store the generated Studio Event Emitters that are generated in the MapObjGen script (Found in Map Object Generator).

The ‘Flower Pool’ Transform stores a dynamic list of flowers, which is instantiated for each raindrop that hits the ground while the Player Faith parameter is greater than or equal to 50.

**== WEATHER ==** contains the Weather Manager, which scripts that control dynamic rain and wind strength.

**== PLAYER ==** contains the Animator component, and thus a convenient portal to the Animation Events and the Player Sound Effects script. The Player script contains a number of important components:

**FMODListener** – Contains the FMOD Studio Listener, and a dormant script that can manages the position of the main FMOD attenuation object at runtime, parenting it to other objects dynamically.

**== AREA MANAGER ==** strip contains the Animator component, and thus the Animation Events and the Player Sound Effects script. The Player script contains a number of important components:

**FMODListener** – Contains the FMOD Studio Listener, and a dormant script that can manages the position of the main FMOD attenuation object at runtime. Currently not used.

**Controls**

**The game requires mouse interaction only.**

When the game loads, you are in the title screen. You can scroll around, but nothing more. Click to continue to the game.



**The Title Screen that appears on start-up**

**Movement**: click where you would like to go, your player should move where you expect. Your player will look where your mouse points. You have 360-degree scroll centerer around the player. An ideal mode of travel is to hold the mouse where you would like to go and scroll around, you should change direction quite smoothly.

The speed you run is relative to the distance between the player and the cursor, so the further you extend the cursor, the faster you run.



**The Deer is selected, and the ‘Talk’ option is avaliable.**

**Interaction**: Right click on objects to open the selected objects radial menu, which will present you with options.

Hover over an element and release while hovering to select the option. To cancel and hide the menu, release the mouse anywhere outside of the button.

**Flute Mode**

**The Player is selected, revealing the Player menu. Each item is labelled on hover.**



**‘PRAY’**

**‘REFLECT’**

**‘DANCE’**

**‘LOOK’**

**‘PLAY MUSIC’**

**QUIT GAME**

If you successfully select the ‘Play Music’ option (denoted by the flute UI), on the players menu, you should enter **Flute Mode.**

While in flute mode, you may press or hold down the left mouse button, anywhere on the screen to play the flute for as long as you wish. Right click to exit flute mode.

The musical mode corresponding to the active mode is mapped from the center of the screen to the edges, which yields notes that are in the current musical scale. This mode changes independently, and when the mode changes, your current note may also change if it’s not in the scale.

**Corruption/Karma System**

The world is designed as always being influenced by constantly modulating player stats that swing between distinct states, particularly influenced by the **Faith** parameter, which is represented with the **blue** tip of the UI triangle that appears above the player. This state is constantly updating depending on player input. The value is clamped to 0-100.

IStats is the interface used to store references to the various functions that control player stats. These stats affect gameplay and audio-visual feedback across the codebase.

Player parameters are updated in the **Character Class** script. The Player Class script is attached to the player GameObject in the Hierarchy. The Player Class inherits from the Character Class and thus contains its properties.

Text

Description automatically generated

**Path: Scripts/Assets/Player**

Text

Description automatically generated

**Path: Scripts/Assets/IStats**

**Path: Scripts/Assets/CharacterClass**

Text

Description automatically generated

The ***OnFaithChanged*** event (line 334), is invoked with the current faith value, minStat, and maxStat values. This event is accessed through a variety of other scripts and used for other purposes.

Text

Description automatically generated

**Path: Scripts/WeatherControl**

Take the wind for instance. This code subscribes to a new function called WindStrength. The purpose of this function, and all other occurrences of this code pattern, is to continuously remap (or, normalize) the range of minFaith (0) and maxFaith (100) into a new range of our choosing to animate any manner of other parameters, in this case, the variables in line 171-173 which set various parameters related to the trees.

The reason these systems are interconnected is so that parameter changes affect all things concurrently, and so visual modifications are reliably interwoven with audio modifications as they are being modified by the same parameter, but for different ranges.

The way this was achieved was to use a custom remap function, comprised of a lerp function and an inverse lerp function. (Freya Holmer [- The Simple Yet Powerful Math We Don't Talk Abou](https://youtu.be/NzjF1pdlK7Y?t=1472)t). This is code pattern that is found throughout the code.

**Generative Modal Music System**

The system works by triggering samples via a programmer instrument in the FMOD Studio Project. Like the rest of this project, it makes extensive use of coroutines and loops.

Much like the map, the composition is very much rooted in indeterminism as a core concept, as well as randomness. The unpredictable and emergent quality reliably generates a composition while maintaining novelty. This emergent novelty is at the heart of what the game strives to cultivate.

A picture containing graphical user interface

Description automatically generatedA enum called Modes contains the *names* of various musical modes (Major, Aeolian, Dorian, Phrygian, etc.) and associates each mode with an array of string values representing musical notes. These references are passed into a function that passes these notes to the programmer instrument instance, retrieving the relevant note.

The benefit of this method is that it’s hugely expandable and flexible.

You can quite easily add new instruments, modes, new rules for shuffling notes, various time modification at different points in the code etc...

Each time a note is played there is a small-time buffer which is the result of a random range between two exposed variables that represent min and max buffer before a note is played.

The Play Note function primarily determines the instrument sound and the polyphony (how many voices are played at once). There is a third parameter, which determines whether a note pitch is to be set manually, or not. If a pitch name is not given (null) then the pitch will be determined by the subsequent processes.

Each time the note is played, an array of notes corresponding to the current mode is shuffled. A Dictionary object called ModeInfo is then created to map each mode to its corresponding array of notes.

You can determine the note length, that is, how long the sample is looped for before releasing/stopping. This occurs in a Coroutine, ‘NoteDuration’.

By default, the duration is a random range between a min and max duration. The coroutine loops while the note hasn’t ended. This is determined by the note instance’s timeline position, which starts from 0 as the instance is triggered.

If it’s value is equal to or greater than the provided duration parameter, then it will stop and release the note.

**Harmonic Stability**

**Harmonic stability** is a global FMOD parameter that controls the automation curve controlling the maximum random pitch deviation of the programmer instrument of which the generative music system is output from. The further this value is away from 0, the less harmonic and dissonant the input material becomes.

The Harmonic Stability Parameter is adjusted in a loop in the **RainControl** script.

The RainControl script controls a particle system that generates flowers when rain particles collide with the ground. The script manages an object pool of flower objects and uses a method to generate flowers at the collision point of the rain particles. The flowers grow and then shrink, returning to the pool when they are fully grown (As determined by the LerpScale script, the same as the trees. An FMOD sound is triggered when rain particles collide with the ground. The script uses a global parameter trigger to control a harmonic stability parameter that is determined by the player's faith level. The harmonic stability value is used to modulate the sound of the rain, creating a more peaceful or chaotic atmosphere.

There is often a clear tonal center , that is while the harmony is not being shifted to the more enharmonic degree of random, by means of the ‘Harmonic Stability’ Global FMOD parameter. So long as the appropriate range of min and max values are applied, there is a good balance of repetition and progression, stability and disharmony.

A Fisher-Yates method for shuffling was used to shuffle the notes as this is supposedly the least biased method of shuffling. Essentially, it’s an index walking through the array of elements and randomising each element to some other remaining index in the array. The result of this method that each element in the array has an equal probability of being in any position, which is truly random, and relevant to the core principle of the game: randomness.

**Tree Growth**

The tree growth is one of the games primary features. They all grow in random succession, growing, living, dying then repeating.

The tree SFX was designed with a granular approach. Using a scatterer instrument containing a large amount of very short crackling wood samples, where a small looping segment of the timeline produces a continuous flow of granular ‘bits’ that create a complete tonal sound image that compliments the texture in question. In this case, wood under stress as it strains from the ground.

FMOD Studio – TreeGrow Event

**Graphical user interface

Description automatically generated**

It’s important to not loop this too narrowly as it may cause performance issues.

This is also especially true of the ‘Spawn Rate’ parameter and ‘Polyphony’.

The benefit of this granular method is that it’s highly unrepetitive and rich in variation, particularly when compounded with additional pitch randomisation automation. One drawback is that is quite CPU intensive. However, this is remedied by setting up Virtualized voices, which helps marshal the number of active instances by dynamically disabling the least audible instances.

Pitch, Spawn Rate and Release Time have been automated, such that they conform to the growth curve of the tree. The growth time of a given tree is a unique, local variable, determined independently. It is expressed as a percentage value derived from the LerpScale component, which is attached to each tree. This script holds a reference to a normalized range of 0 to 1, achieved using a remap function. 0 represents the minimum level of growth while 1 is max.

There is an initial, dramatic exponential curve downward in pitch, which intends to reflect the thickening density of the tree as it sprouts from sapling to fully grown.

**Time of Day**

There is an overarching time of day parameter which lerps continuously through a range of 0-24, representing the game hours. This script was an expansion of the ‘Day, Night, and Light Controller’ published by ‘Pleebie Jeebies’ (available on the [Unity Asset Store](https://assetstore.unity.com/packages/tools/particles-effects/day-night-and-light-controller-3d-201611)). The TimeOfDay parameter corresponds to an FMOD parameter of the same name and is updated in conjunction.

The EQ curves of this time-of-day effect are intended to reflect the brightness and darkness of the scene as dictated by the sunlight and its intensity depending on the time of day.

The EQ contour that the time-of-day parameter modules is best illustrated in the FMOD Studio project.

Graphical user interface

Description automatically generated with medium confidence

FMOD Studio ‘Music’ event, TimeOfDay parameter.

Each curve represents an EQ automation over TimeOfDay. This spans on a continuous scale from 0-24. When TimeOfDay reaches 24, it loops back around. This effect is particularly present when using the ‘Reflect’ behaviour in-game.

**Procedurally generated emitters**

The procedural generation of the terrain allows for the implementation of random terrain generation. Adaptability for this case where the terrain is different every time the game is played is important. The initial position of generated emitters should vary naturally with the dynamic and changing mesh landscape. To ensure that water is generated only in appropriate areas, there needs to be a condition that adapts to the terrain and generates water only in areas where it is meant to be present.

**Water Emitters**

Graphical user interface, application, Teams

Description automatically generatedThe method that I used to solve the issue of unpredictable occurrence of water was to instantiate the emitters for each vertex on the mesh of a plane. And then for each of these points, RayCast down from a height offset above this point and check whether the ray hit water or not. If not, the emitter on that point is destroyed. This method reliably works across different terrain types. There is also a sample density control, which is essentially saying ‘generate a point for every ‘x’ number of steps’, which means the density of emitters may be controlled as the generation index is taking larger steps between vertices before generating a water sound emitter.

Using ProBuilder (Available in the package manger), a subdivision modifier can be applied to a plane mesh. This increases the vertex density of the plane, which enables the emitter generation script to access more vertices from the mesh’s list of vertices. The higher the density, the more points that can be generated, but it becomes more CPU intensive as the emitter generation script uses a RayCast for every vertex.

Text

Description automatically generatedGraphical user interface

Description automatically generated with medium confidence

This script (GenerateEmitterCheckers) generates the emitter checkers, checking if they’re generating in a water-safe zone. If not, they get destroyed.

Path: Scripts/MapObjGen – Lines 216 to 280 (For the next two figures)

A screenshot of a video game

Description automatically generated

Early screenshot of water emitter success in an earlier build of the game

**Dynamic Wind Emitters and Object Pooling**

Object pooling is used to generate a variable (and adjustable) amount of wind emitters within a random radius around the player. Each emitter has a lifetime of which it’s active for, before it’s GameObject is deactivated, and returned to the object pool.

**A picture containing text

Description automatically generated**

Scene View of the wind emitters during run-time. Each blue orb is a **WindZone3D** prefab. All generated emitters move sporadically in random directions. These orbs continuously generate around a random radius relative to the player, as they are parented to the player transform.

Each **WindZone3D** prefab contains a **NavMeshAgent** component, which is the precursor to determining its random position and direction on the Navmesh.

A benefit of using a NavMeshAgent to control a wind zone is that it will always conform to the same terrain mesh contours as the player, while smoothly interpolating from one random position to the next.

Object pooling is much cheaper on CPU performance than continuously instantiating and destroying Wind Zone prefabs due to bypassing the need for garbage collection by instantiating all of the required prefabs just once, and setting them as active/inactive. It’s well documented that this saves a good chunk of CPU. ([Unity has a resource on Object Pooling](https://learn.unity.com/tutorial/introduction-to-object-pooling))

Each active Wind Zone is connected to the weather system, so each instance continuously updates the ‘WindStrength’ FMOD parameter based on the same parameter that controls the shaking of the tree leaves.

**Text

Description automatically generated**

**Dialogue**

The dialogue system is an expansion of [BMo’s tutorial on creating a simple dialogue system](https://www.youtube.com/watch?v=8oTYabhj248). When the player interacts with the NPC (Selecting ‘Talk’) – dialogue is started. The dialogue is skippable and terminates when the final index in the dialogue ‘lines’ array is reached.

The player may freely walk away a break the conversation from any interaction they begin with an NPC, so distance attenuation has been implemented to highlight this effect to the player.

Debugging and Profiling

**Asset Design**

When it comes to asset design, a useful approach to take is to imagine you’re contributing to the larger sound image.

**Conclusions**

I am only regretful that the project is not more developed and organised than it is. There are quite a few features and variables that are unimplemented, and there are some clear organisation issues. Though this is one of the inevitable challenges of managing a project as a solo developer. The project will inevitably that balloons in size and complexity and inter-dependency over time.

SFX Samples used:

**Didgeridoos:**

<https://freesound.org/people/bman001/sounds/257448/>

<https://freesound.org/people/boobaloo/sounds/318641/>

https://freesound.org/people/borralbi/sounds/351980/

<https://freesound.org/people/dethrok/sounds/272165/>

<https://freesound.org/people/visionari1/sounds/61505/>

<https://freesound.org/people/qubodup/sounds/173923/>

<https://freesound.org/people/BaDoink/sounds/538754/>

https://freesound.org/people/chiptraxxx/sounds/431178/

<https://freesound.org/people/olliehahn12/sounds/268788/>

https://freesound.org/people/nemaavla/sounds/240916/

https://freesound.org/people/JappeHallunken/sounds/501296/

**Nature Ambience:**

Wood Creaks: <https://freesound.org/people/6polnic/sounds/231438/>

<https://freesound.org/people/mickdow/sounds/253327/>

<https://freesound.org/people/jergonda/sounds/254735/>

Water: <https://freesound.org/people/ceich93/sounds/318064/>

<https://freesound.org/people/kvgarlic/sounds/326097/>

Bird Chirp: <https://freesound.org/people/Lizardhood/sounds/427040/>

Crow Caw: https://freesound.org/people/straget/sounds/404687/

Crickets: <https://freesound.org/people/audible-edge/sounds/78810/>

**Character:**

Vomit Sounds: <https://freesound.org/people/D.jones/sounds/528790/>

Vocal Chants: <https://freesound.org/people/Falsalama/sounds/64603/>

<https://freesound.org/people/boobaloo/sounds/318638/>

<https://freesound.org/people/borralbi/sounds/351980/>

<https://freesound.org/people/djgriffin/sounds/21221/>

Bibliography:

*Fisher-Yates Shuffle algorithm – implanted in ‘MusicManager’ script -*

<https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle>

https://alessandrofama.com/tutorials/fmod/unity/shuffle-playlist