

Conception and Evaluation of a Procedural Music System
for Engagement with Player Emotion and Memory
in Digital Games

PROJECT DOCUMENTATION

by

Vadim Nickel

-
-

submitted to obtain the degree of

MASTER OF ARTS (M.A.)

at

TH KÖLN UNIVERSITY OF APPLIED SCIENCES
COLOGNE GAME LAB

Course of Studies
DIGITAL GAMES

First supervisor: Prof. Dr. Cécile LE PRADO
TH Köln University of Applied Sciences

Second supervisor: Prof. Dr. Sonia FIZEK
TH Köln University of Applied Sciences

Written in the Summer Term 2021
Submitted on September 17th, 2021

Table of Contents

1. Development.....	3
1.1. Background	3
1.2. Proof-of-concept.....	3
1.3. Implementation	4
1.4. Prototyping	4
1.5. Technology	5
2. Control components.....	7
2.1. ProcMu Master.....	7
2.2. Interpolator	8
2.3. ProcMu Guidance	9
2.4. Music Zone.....	10
3. Music configuration.....	11
3.1. General settings	11
3.2. Euclidean Rhythms	12
3.3. Chord Player	13
3.4. Sample And Hold Melody / Sample and Hold Bass.....	14
3.5. Synthesizer configuration.....	15
3.6. Scale.....	16
4. Technical implementation of experiment.....	17
4.1. Experiment Starter.....	17
4.2. Experiment Conductor	17
4.3. Game Action	17
4.4. Survey	18
4.5. Azure	18
4.6. Data collection and processing.....	18
Declaration of Authorship	19

1. Development

1.1. Background

In my Bachelor thesis “Generative Music in Digital Games – Application and Evaluation of Procedural Content Generation Principles”, I had first researched the concept of generative music and its application in digital games. For the practical project of that thesis, I had developed a generative music system using Unity and ChuckK, which could generate multi-layered music based on a given configuration. Using triggers placed in the game environment, changes in the generative parameters of the musical output could be initiated.

However, this system required extensive effort in configuring the music generation parameters and would rely on additional program logic to provide a dynamic outcome. Further, it was sample-based and did not allow for sound synthesis and therefore more granular sound manipulation in real-time.

At the time, I had already been working on a personal project called Zolar, a first-person puzzle adventure game. One of the rewarding experiences in puzzle games occurs when the player manages to overcome an obstacle via logical reasoning, i.e., solving a puzzle. However, the solution is not always clear, and players may end up wandering the game environment, not knowing what to do. I was looking for ways to provide clues to the player without interfering with their own reasoning, and thus possibly diminishing the rewarding experience. Being aware of the emotional impact music can have, I decided to develop an approach which uses music to guide the player: a location-based approach. This means that each area in the game environment should have its own music style, a musical signature. Over time, the player should internalize the respective musical signatures, and may therefore be lured towards an area by playing back the music of the target area.

1.2. Proof-of-concept

I first implemented a rudimentary approach to location-based procedural music in Zolar. Here, a constant musical output was generated by selecting from various audio sample collections in random intervals. The minimum and maximum possible interval could be set manually. Multiple layers allowed for overlapping sample playback, creating a dense musical structure. Samples could be assigned to different areas in the game environment. Based on the player’s proximity to these areas, the likelihood

of triggering a specific sample increased with proximity of the player to the respective area. Therefore, as the player moves through the environment, the music would gradually change from one style to another.

After conducting the research that is presented in the thesis, I realized that this location-based approach could be enhanced to involve the player's emotions. By additionally modifying various parameters in the music generation, the player's behavior should be influenced. If the player takes too long to proceed in the game, the music should become more intense, suggesting a sense of urgency, with the intention of making the player find their goal faster.

1.3. Implementation

Having in mind the complex structure of the generative music system that I created for my bachelor thesis, my approach for this new system was to find a way to unify musical expressivity with good usability. For the new system, I chose a modular approach: various modules that are dedicated to a specific purpose could be combined to create more complex music structures. They needed to be configurable with a minimum number of parameters. The new system would be called ProcMu – procedural music. Its functionality is presented below.

1.4. Prototyping

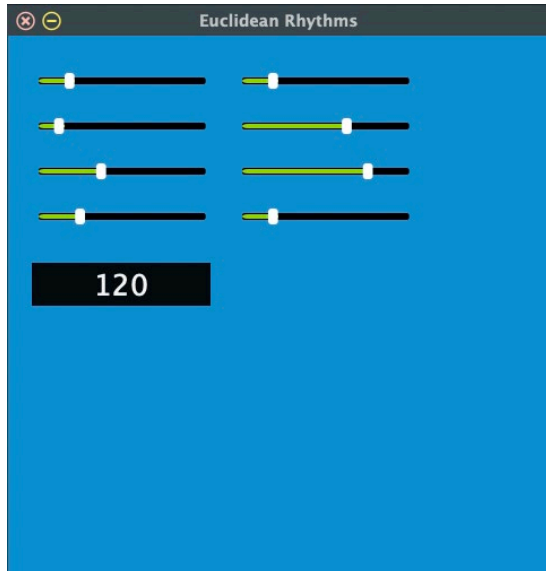
Individual modules were first prototyped in a virtual modular system setup using Bitwig Grid¹. Below is an example for the prototype of the Sample and Hold Melody module:



¹ Modular systems are comprised of various individual units with different functions, such as oscillators or filters. Synthesizers are a fixed combination of such units with a pre-determined signal path. A modular system allows to rearrange and connect the units as desired, changing the various characteristics of the ensuing sound. There are many software implementations of such systems, such as Bitwig Grid or VCV Rack.

Next, the modules were recreated in Csound. Before they were integrated with Unity, they worked as standalone units with their own user interface for further development and debugging.

Below is the interface for the prototype of the Euclidean Rhythm module:



After initial development of the modules, they were combined with a sequencer that controls the speed of the music playback and timing of individual notes. Then, using CsoundUnity, an interface between the game engine and the Csound component was created.

1.5. Technology

Csound²

Csound is an audio programming language. It was originally created by Barry Vercoe in 1985 at the MIT Media Lab. It is based on MUSIC-N, which was originally written by Max Mathews in 1957 at Bell Labs. Csound is still being actively developed by a dedicated community.

This framework has been chosen because it can be easily integrated into the Unity engine via CsoundUnity (see below).

Furthermore, it is compatible across all common operating systems and architectures, allowing the use of ProcMu beyond the scenario presented in the theoretical part of this thesis.

² Available on: <https://csound.com/download>, last checked on 15.09.21

Such additional scenarios include but are not limited to: Integration of the system into other engines, e.g., Unreal Engine, Godot; integration into embedded systems, e.g., Raspberry Pi, Arduino, taking parameter values from external sensors, creating hardware controllers...

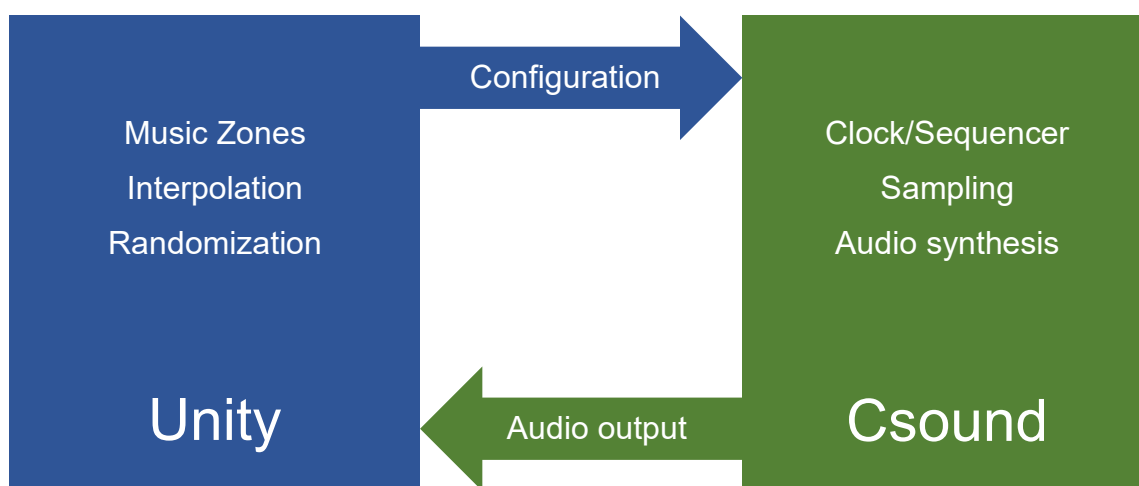
Unity³

Unity is a 3d game engine, which was originally released by Unity Technologies in 2005. It is used to provide the environment for the experiment which is part of the thesis. For the technical implementation of the experiment, see 7.

CsoundUnity⁴

CsoundUnity is an interface plugin that allows to send data from Unity to Csound and receive audio from Csound in Unity. It was originally released in 2015 and is still in active development by Rory Walsh.

The schematic below shows the functional structure of ProcMu:



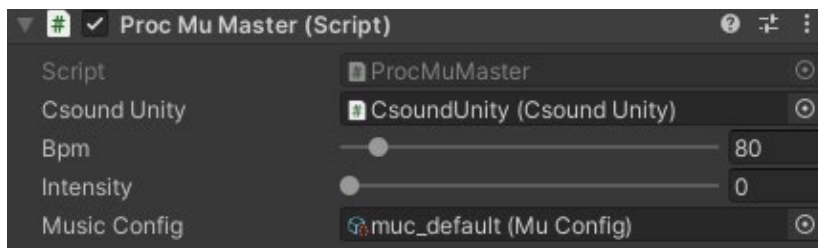
Configuration information is passed from Unity to Csound. The configuration is determined by the music configuration object (see 2). According to the configuration received from Unity, Csound will generate timed audio in real-time, through playback of samples and real-time synthesis. The resulting audio is passed back to Unity.

³ Available on: <https://store.unity.com/front-page>, last checked on 15.09.21

⁴ Available on: <https://rorywalsh.github.io/CsoundUnity>, last checked on 15.09.21

2. Control components

2.1. ProcMu Master



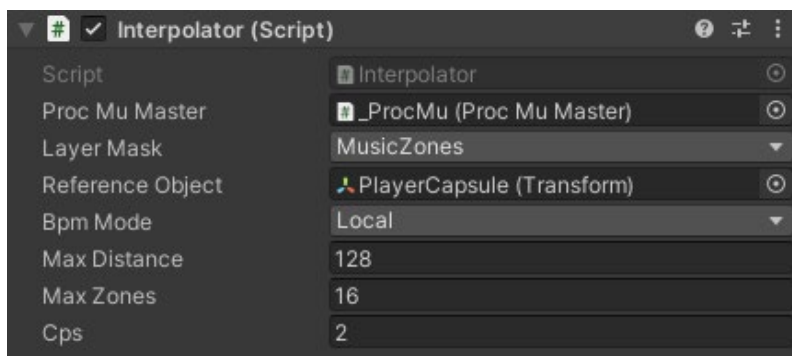
Description

Responsible for transferring the global musical configuration to Csound.

Configuration parameters

- *Csound Unity*: Unity to Csound interface object.
- *BPM*: Current tempo of music.
- *Intensity*: Current intensity. Determines which parts of a musical configuration are considered for music generation. Each music configuration provides generative parameters for both the lowest possible intensity (0) and the highest possible intensity (1). Settings are interpolated, based on the current intensity value, e.g., if the current intensity is 0.5, then the settings will be an even mixture of the configurations for the lowest and highest intensity value (see 4.3).
- *Music Config*: Global music configuration. Can either remain static, or changed by external components, such as the interpolator.

2.2. Interpolator



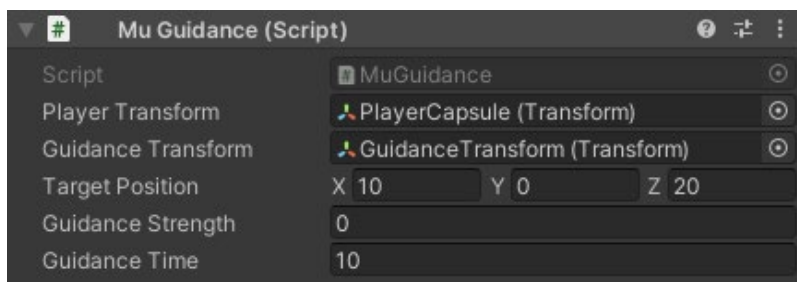
Description

Creates a music configuration dynamically, based on the position of the reference object. The parameters of the configuration are a result of interpolating between the values set in the music configurations of the music zones in proximity of the reference object. Interpolation is weighted according to the distance of the surrounding music zones to the reference object. The closer it is to a specific music zone, the more its value contributes to the final result. If a value cannot be interpolated, e.g., the musical scale, the setting of the closest music zone is used.

Configuration parameters

- *ProcMu Master*: The ProcMu Master object that is controlled by the interpolator.
- *Layer Mask*: The layer that the music zones are assigned to.
- *Reference Object*: The object whose position is referenced for interpolation. This can be either the player or any other object.
- *BPM Mode*: Chooses how music playback speed is determined. Global: The bpm value set in the ProcMu master component is used and remains unchanged. Local: The bpm value is adjusted automatically by interpolating between bpm values assigned to music zones in vicinity, changing dynamically with the reference position.
- *Max Distance*: Maximum distance of music zone center to the reference position.
- *Max Zones*: Maximum number of music zones within distance considered for interpolation.
- *Cps*: Checks per second, i.e., how often interpolation is performed. Higher values allow for smoother interpolation but may also have a larger performance impact.

2.3. ProcMu Guidance



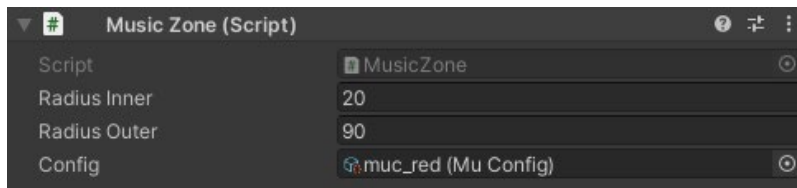
Description

Provides a reference position for the interpolator that will move between player and target position. The guidance transform must be associated to the reference object in the interpolator if guidance is to be used. This is to guide the player using music, as it will over time resemble the style associated with the target position area.

Configuration parameters

- *Player transform*: Player reference object.
- *Guidance transform*: Guidance reference object that is moved between player and target position.
- *Target position*: Position to which the player is intended to be lured by the guidance system.
- *Guidance strength*: Defines where the reference position lies between player position (strength = 0) and target position (strength = 1).
- *Guidance time*: The time in seconds it takes the guidance system to go from 0 to 1 strength once guidance process is initiated.

2.4. Music Zone



Description

A music zone defines the area in which its configuration is used for music generation.

Configuration parameters

- *Inner radius*: When the reference position is within the inner radius of a music zone, the interpolator will exclusively use the configuration of the respective zone.
- *Outer radius*: When the reference position is within the outer radius of multiple music zones, the musical output will be a result of interpolating the values of all music zones within a defined range around the reference point. The likelihood of using the configuration of a specific music zone depends on the proximity of the reference point to each respective zone: The closer it is to the center of a zone, the more likely its configuration is used for music generation.
- *Config*: The music configuration that is associated with the music zone.

3. Music configuration

The procedural music output is comprised of multiple layers. Each layer uses a different module to generate a musical element. The configuration of modules is defined in the music configuration. There are modules for percussive, as well as melodic output, which are presented in the following. All modules hold two configuration states, one for minimum (0) and one for maximum (1) intensity.

3.1. General settings



Description

The general settings control the operation of all modules.

Configuration parameters

- *BPM*: The tempo associated to the musical configuration. This is used to define a specific tempo per music zone.
- *Scale*: Holds all possible notes that can be played by the sound modules, depending on their respective set octave ranges.
- *Active bars*: The sequencer constantly iterates through sets of four bars with 16 steps each. This option allows to activate or deactivate playback of individual modules for the duration of one bar each. If the given intensity value is < 0.5 the left block is used, if > 0.5 the right block is used.

3.2. Euclidean Rhythms

Euclidean Rhythm settings				
	Intensity = 0 settings		Intensity = 1 settings	
Sample/trigger	min pulses	max pulses	min pulses	max pulses
CLAV1	1	2	2	3
TABLA2	1	2	2	3
TABLA3	2	4	4	6
METAL1	2	3	8	12

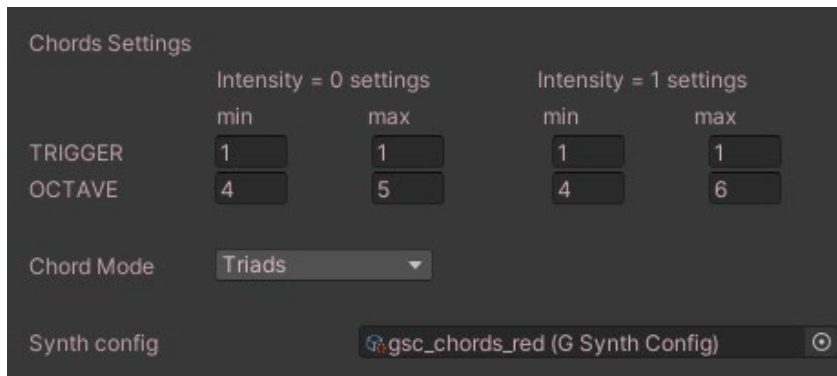
Description

This module is a sampler for up to four percussive sounds. Samples must first be added to the sample database (see 4.1). The module contains a sequencer that repeatedly iterates through a cycle of 16 steps. The sequencer moves to the next step every time it receives a trigger signal from the clock. If the last step (16) has been reached, the cycle is repeated, going back to the first step. Pulses, i.e., signals to play a percussive sound, are automatically distributed as equally as possible across all 16 steps, according to the Euclidean rhythm algorithm by Toussaint (see thesis for information). The number of pulses can be varied automatically and randomly, according to the configuration of the module.

Configuration parameters

- *Sample*: One audio sample per layer can be assigned, defining the sound that is heard when a percussion is triggered. Samples can be added using the sample manager utility that comes with ProcMu.
- *Min/max pulses*: Defines how often the respective percussive sample is triggered per layer. On each cycle, a number within the min/max range is randomly determined.

3.3. Chord Player



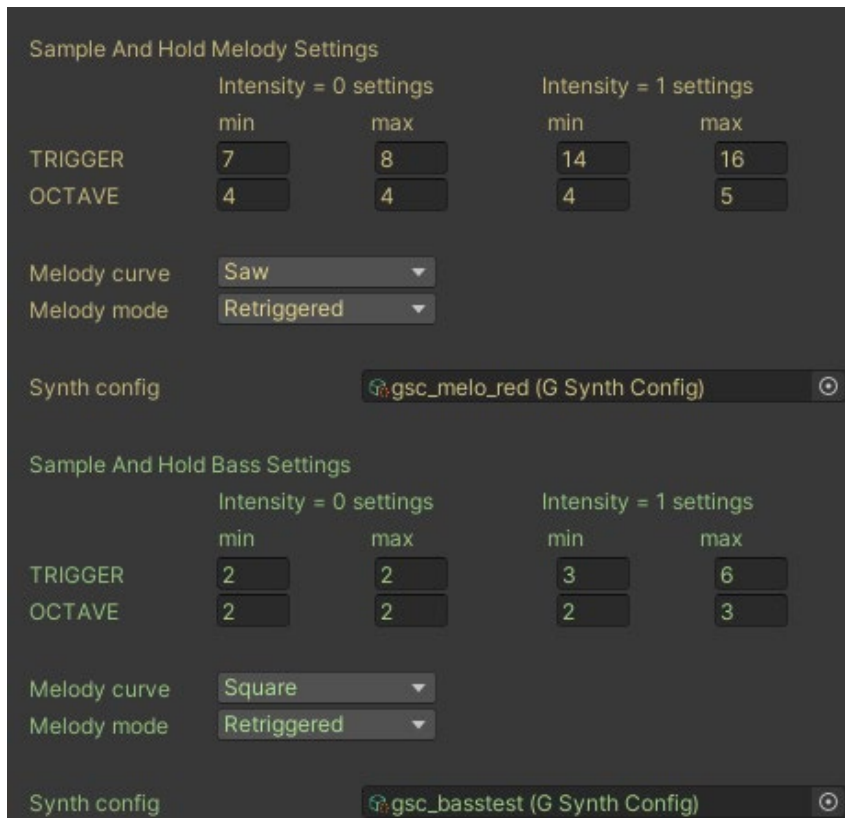
Description

Plays multiple notes simultaneously when triggered, forming a chord.

Configuration parameters

- *Trigger*: Defines how often a chord is played, per cycle of 16 steps. On each cycle, a number within the min/max range is randomly determined. The distribution of triggers works in the same way as for the Euclidean rhythm module.
- *Octave*: Defines which part of the scale is used to determine the chord notes.
- *Chord mode*: The algorithm used for generating chords based on the given scale and octave setting. Currently, two modes are supported: Triads will generate a chord from three notes, triads oct will do the same, adding another note one octave lower than the root note of the chord.
- *Synth config*: The synthesizer parameters used by the module, defining the properties of the ensuing sound.

3.4. Sample And Hold Melody / Sample and Hold Bass



Description

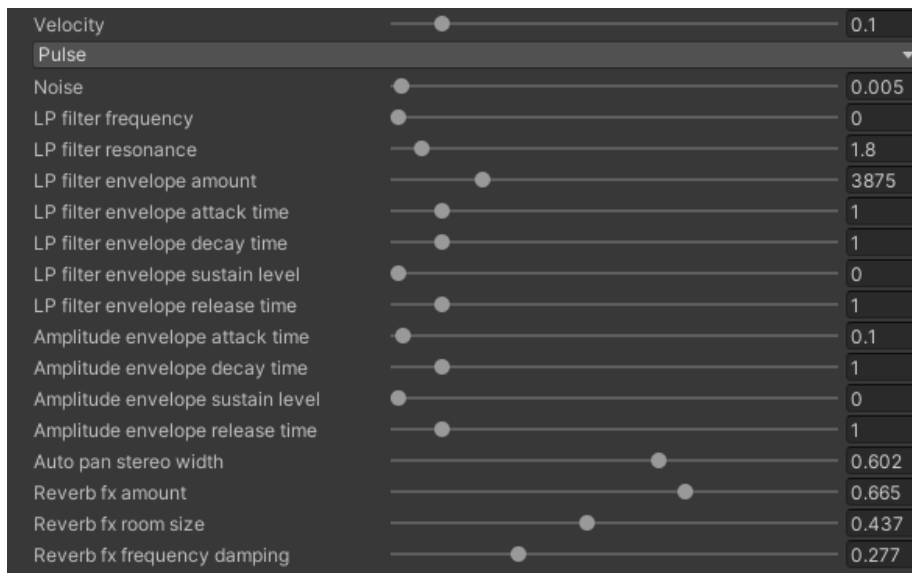
Plays a monophonic melody according to given parameters. Currently, the modules for bass and melody are identical, meant to support low-pitch and high-pitch melodies playing simultaneously and independently of each other.

Configuration parameters

- **Trigger:** Defines how often individual notes of the melody are played, per cycle of 16 steps. On each cycle, a number within the min/max range is randomly determined. The distribution of triggers works in the same way as for the Euclidean rhythm module.
- **Octave:** Defines which part of the scale is used to determine the melody notes.
- **Melody curve:** Defines the movement of the melody by sampling fundamental waveforms. Saw generates a descending melody that starts at the highest possible pitch in the scale within the octave range after reaching the lowest possible pitch. Sine smoothly moves up and down the scale. Square alternates between the lowest and highest possible notes. Pulse plays the highest possible note, followed by a series of lowest notes.

- *Melody mode*: Retriggered mode triggers individual notes, according to the trigger parameters. Further, a continuous mode is planned, using a continuous sound that changes in pitch, fading in and out according to an occurrence parameter.
- *Synth config*: Same as in chord player.

3.5. Synthesizer configuration



Velocity	0.1
Pulse	
Noise	0.005
LP filter frequency	0
LP filter resonance	1.8
LP filter envelope amount	3875
LP filter envelope attack time	1
LP filter envelope decay time	1
LP filter envelope sustain level	0
LP filter envelope release time	1
Amplitude envelope attack time	0.1
Amplitude envelope decay time	1
Amplitude envelope sustain level	0
Amplitude envelope release time	1
Auto pan stereo width	0.602
Reverb fx amount	0.665
Reverb fx room size	0.437
Reverb fx frequency damping	0.277

Description

The synthesizer configuration defines the sound characteristics of the virtual synthesizer that is used by the chords and sample and hold melody/bass modules. The parameters, which can be seen in the picture above, correspond to a typical synthesizer design. The signal path is based on that of a Dave Smith Instruments Prophet '08 hardware synthesizer, albeit in a reduced manner and with an additional reverb feature.

3.6. Scale

<input type="checkbox"/>	C10	<input checked="" type="checkbox"/>	C#10	<input type="checkbox"/>	D10	<input type="checkbox"/>	D#10	<input type="checkbox"/>	E10	<input type="checkbox"/>	F10	<input checked="" type="checkbox"/>	F#10	<input type="checkbox"/>	G10								
<input type="checkbox"/>	C9	<input checked="" type="checkbox"/>	C#9	<input type="checkbox"/>	D9	<input type="checkbox"/>	D#9	<input type="checkbox"/>	E9	<input type="checkbox"/>	F9	<input checked="" type="checkbox"/>	F#9	<input type="checkbox"/>	G9	<input checked="" type="checkbox"/>	G#9	<input type="checkbox"/>	A9	<input type="checkbox"/>	A#9	<input checked="" type="checkbox"/>	B9
<input type="checkbox"/>	C8	<input checked="" type="checkbox"/>	C#8	<input type="checkbox"/>	D8	<input type="checkbox"/>	D#8	<input type="checkbox"/>	E8	<input type="checkbox"/>	F8	<input checked="" type="checkbox"/>	F#8	<input type="checkbox"/>	G8	<input checked="" type="checkbox"/>	G#8	<input type="checkbox"/>	A8	<input type="checkbox"/>	A#8	<input checked="" type="checkbox"/>	B8
<input type="checkbox"/>	C7	<input checked="" type="checkbox"/>	C#7	<input type="checkbox"/>	D7	<input type="checkbox"/>	D#7	<input type="checkbox"/>	E7	<input type="checkbox"/>	F7	<input checked="" type="checkbox"/>	F#7	<input type="checkbox"/>	G7	<input checked="" type="checkbox"/>	G#7	<input type="checkbox"/>	A7	<input type="checkbox"/>	A#7	<input checked="" type="checkbox"/>	B7
<input type="checkbox"/>	C6	<input checked="" type="checkbox"/>	C#6	<input type="checkbox"/>	D6	<input type="checkbox"/>	D#6	<input type="checkbox"/>	E6	<input type="checkbox"/>	F6	<input checked="" type="checkbox"/>	F#6	<input type="checkbox"/>	G6	<input checked="" type="checkbox"/>	G#6	<input type="checkbox"/>	A6	<input type="checkbox"/>	A#6	<input checked="" type="checkbox"/>	B6
<input type="checkbox"/>	C5	<input checked="" type="checkbox"/>	C#5	<input type="checkbox"/>	D5	<input type="checkbox"/>	D#5	<input type="checkbox"/>	E5	<input type="checkbox"/>	F5	<input checked="" type="checkbox"/>	F#5	<input type="checkbox"/>	G5	<input checked="" type="checkbox"/>	G#5	<input type="checkbox"/>	A5	<input type="checkbox"/>	A#5	<input checked="" type="checkbox"/>	B5
<input type="checkbox"/>	C4	<input checked="" type="checkbox"/>	C#4	<input type="checkbox"/>	D4	<input type="checkbox"/>	D#4	<input type="checkbox"/>	E4	<input type="checkbox"/>	F4	<input checked="" type="checkbox"/>	F#4	<input type="checkbox"/>	G4	<input checked="" type="checkbox"/>	G#4	<input type="checkbox"/>	A4	<input type="checkbox"/>	A#4	<input checked="" type="checkbox"/>	B4
<input type="checkbox"/>	C3	<input checked="" type="checkbox"/>	C#3	<input type="checkbox"/>	D3	<input type="checkbox"/>	D#3	<input type="checkbox"/>	E3	<input type="checkbox"/>	F3	<input checked="" type="checkbox"/>	F#3	<input type="checkbox"/>	G3	<input checked="" type="checkbox"/>	G#3	<input type="checkbox"/>	A3	<input type="checkbox"/>	A#3	<input checked="" type="checkbox"/>	B3
<input type="checkbox"/>	C2	<input checked="" type="checkbox"/>	C#2	<input type="checkbox"/>	D2	<input type="checkbox"/>	D#2	<input type="checkbox"/>	E2	<input type="checkbox"/>	F2	<input checked="" type="checkbox"/>	F#2	<input type="checkbox"/>	G2	<input checked="" type="checkbox"/>	G#2	<input type="checkbox"/>	A2	<input type="checkbox"/>	A#2	<input checked="" type="checkbox"/>	B2
<input type="checkbox"/>	C1	<input checked="" type="checkbox"/>	C#1	<input type="checkbox"/>	D1	<input type="checkbox"/>	D#1	<input type="checkbox"/>	E1	<input type="checkbox"/>	F1	<input checked="" type="checkbox"/>	F#1	<input type="checkbox"/>	G1	<input checked="" type="checkbox"/>	G#1	<input type="checkbox"/>	A1	<input type="checkbox"/>	A#1	<input checked="" type="checkbox"/>	B1
<input type="checkbox"/>	C0	<input type="checkbox"/>	C#0	<input type="checkbox"/>	D0	<input type="checkbox"/>	D#0	<input type="checkbox"/>	E0	<input type="checkbox"/>	F0	<input checked="" type="checkbox"/>	F#0	<input type="checkbox"/>	G0	<input checked="" type="checkbox"/>	G#0	<input type="checkbox"/>	A0	<input type="checkbox"/>	A#0	<input checked="" type="checkbox"/>	B0

Description

The scale object holds a selection of notes that are allowed to be played by the modules that use a scale. The octave range from which notes in the scale are taken is chosen based on what is defined in the configuration of the respective module. Scales can be set by manually choosing which notes are active, or by generating them from a set of available scales.

4. Technical implementation of experiment

To perform and evaluate the experiment, which is part of the thesis, a custom solution for time measurement as well as survey provision and collection was implemented. In the following, the individual components of the experiment solution are presented.

4.1. Experiment Starter

Description

Responsible for initiating the experiment. For the thesis, three different experiment groups were implemented. When a participant starts the experiment, they are randomly assigned to one of the three groups.

The probability to be assigned to a specific experiment group can be adjusted. This is useful when participants are not equally distributed among the given groups. However, to change the probabilities in the experiment application, it must be updated. In the future, participants shall be automatically assigned to the group that has the smallest number of participants by checking previous submissions.

4.2. Experiment Conductor

Description

Holds a series of actions that define the experimental process, such as resetting the player position, controlling music playback (e.g., turning the music off when text is displayed, or turning it on during the rounds), displaying the survey, enabling/disabling controls, etc.

4.3. Game Action

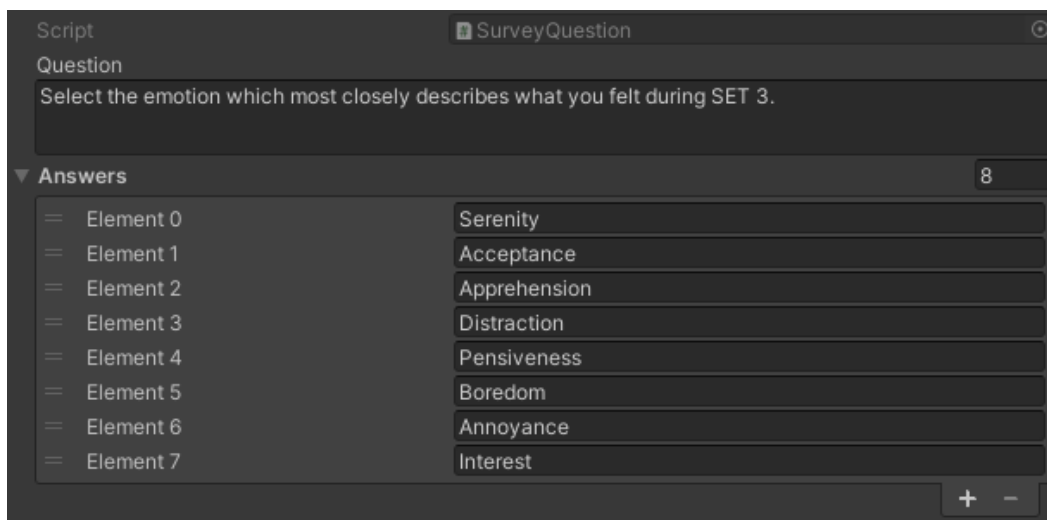
Description

The game consists of the participant having to find a star-shaped object. The Game Action component measures the time the participant takes to find the object for each round. It is further responsible to abort the round if it takes more than 180 seconds. In that case 180 is recorded as the time taken, however, this is merely a feature to avoid long experiment sessions and has in fact not been triggered by any participant.

4.4. Survey

Description

The survey component is responsible for displaying the survey and recording the submitted answers. Any number of survey questions can be added. Questions can be customized, allowing to prepare for other experiment types than were performed in the course of the thesis. Below is an example for how a survey question is set up in the engine. The question text can be customized, as well as the number and content of answers. Pictured below is an example for a survey question configuration:



4.5. Azure

Description

Interface component that is used to update the experiment results to a Microsoft Azure storage.

4.6. Data collection and processing

Description

At the end of the experiment, the experiment data is automatically uploaded to a Microsoft Azure server.

Declaration of Authorship

This is to certify that the content of this project, documentation and thesis is my own work. It has not been submitted for any other degree or other purposes. I certify that the intellectual content of my submission is the product of my own work and that all the assistance received in preparing it as well as all sources used have been properly acknowledged.

Cologne, September 17, 2021

Vadim Nickel