# 3 – Methodology

The application was developed in the game engine Unity and written in C# using Visual Studio 2017.

it is a proven technology used to create a wide range of commercial titles

These were chosen for the project as they were the most accessible for the developer and as the development of a simple game was required this was the most time efficient platform to use.

The main class for the project is the MusicController, this contains the rules and information for the music theory. It also keeps track of the current scale that the music is to be generated in.

## 3.1 - Reading in MIDI files

The MIDI file format was chosen to store the musical data as it has a low file size and the appropriate data can be easily stored and accessed, with minimal impact on the memory allocation of a game. MIDI stores musical data as a series of events, such as: when a note is played (note on), when a note is stopped (note off), and if the music’s tempo changes, etc. While not always the case, it is convention to have each channel only contain the information for one melody. As there is no simple way to differentiate between multiple melodies sharing a channel and one melody that consists multiple notes playing at the same time, one melody per channel was assumed to be the case for this project.

To read in the MIDI files the NAudio library was used to access the event data. To do this the application retrieves all the events from a specified file and channel, and then loops through these looking for the ‘note on’ events. The ‘note on’ event was the only event that was considered as all the relevant musical data can be inferred from them. The other events consist of musical structure which will be controlled by the game events (see below) and so would not be used in the analysis. The relevant musical data extracted from these events was the time the note starts, its length, and its pitch. The note’s pitch is represented as an integer, i.e. C4 (261.63 Hz) is given the value of 60 and C#4 (277.18 Hz) is 61. When ‘note on’ is found its information is extracted and added to a MidiHolder class. Unfortunately, the NAudio library does not retain the length of each note in an easily accessed form, however it does retain it at the end of ‘note on’ events name, this string is then parsed and the length is extracted and then converted to a float. To make changing the musical mode easier (see section X below) each note’s pitch gets reduced down to the key of C (if original pitch is A, each note’s pitch is subtracted by 9, the semitone difference between A and C). This would also allow multiple songs to be combined at the read in stage.

As the MIDI files can be created from live performances, notes that would be equal in standard music notation can have discrepancies in their length and so would be counted as different notes when the frequency distribution is calculated (see below). To remedy this, each note’s length is rounded to specified minimum number, this is defaulted to 0.25 and then the note is added to a list. As MIDI files have no need to record the musical information for rests, as they just use a lack of notes playing for this, these need to be added so they can be included in the analysis process (see below). This was done by looping through the list of extracted MidiHolders and checking if the time a note stopped was the same as when the next note started, if this was not the case a rest note was created at this point (shown below in Figure X), to specify that this note was a rest its pitch was set to -1.

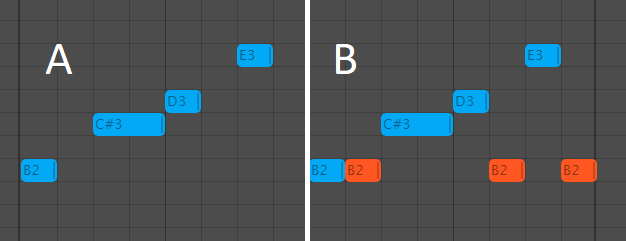


Figure 1: The MIDI sequence before adding rests (A), and the sequence after rests have been added into the spaces between notes (B). Blue bars represents note, orange bars represent rests.

The application then checks the length of the created rest, if this is greater than a specified amount (default value is 0.25) the rest is added into the list of MidiHolders at the correct position, if it is not greater than this, the rests length gets added onto the previous notes length. This is to combat MIDI files that have not been created properly, as shown below in Figure X the majority of the notes are 0.25 in length, but spaced at 0.5 and so a 0.25 length rest is created between most notes. When this file is used for a Markov chain, each note (state) would then always go to a rest, and that rest could then go to essentially any note creating a much more disjointed piece of music.

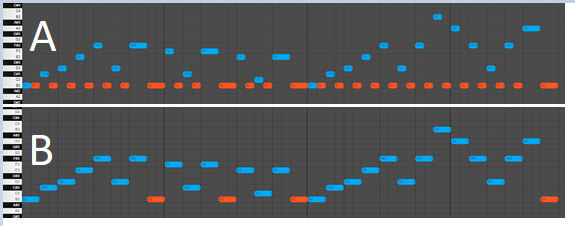


Figure 2: Adding rests with no minimum length (A), and where the minimum length is set to 0.25 (B) note the vastly reduced amount of rests added (from 26 to 3)

Once all the required rests have been added to the list of MidiHolder, the sequence is then looped through again to check for chords, that is when multiple notes start at the same time, if this is the case they are moved from the sequence to the first note’s list of pitches.

## 3.2 - Markov Chains

### 3.2.1 - Frequency Distributions

The next step in the process is to calculate the frequency distribution of the note pairs from the MidiHolder list. Two classes were created to retain the data for this, NextNote and DependHolder. NextNote contains a MidiHolder which saves the data from the second half of the note pair and the frequency that the pairing has occurred. Each DependHolder also contains an instance of a MidiHolder and a list of NextNotes. The frequency distribution is created with a list of DependHolders. The application loops through the sequence of notes and checks if each note has occurred in the MidiHolder of each DependHolder. If this is not the case, there is a new note pair and so a new DependHolder is created, and its MidiHolder is then set to the current note. The next note in the sequence is then added to the new DependHolder’s NextNote list, with a frequency of 1. Conversely if the current note was found in the DependHolders it then checks if the next note in the sequence has occurred in the NextNote list of that DependHolder, if it has it increments the frequency of that NextNote by 1, if not it adds the note to the NextNote list. This creates a data structure, an example of which is shown in Table X

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **MidiHolder** | **NextNotes** | | | |
| **DependHolder (16)** | D | (1) REST | (4) B | (2) C# | (9) A |
| **DependHolder (8)** | C# | (3) A | (1) F | (4) REST |  |
| **DependHolder (7)** | REST | (4) B | (3) G |  |  |

Table 1: A Representation of a frequency distribution. Each row represents a DependHolder with its MidiHolder in the left column and the rest of the cells being the NextNote list. The number in brackets before each NextNote is the frequency of the note pair ending in that note. The number after each DependHolder is the totalled frequencies of its NextNotes. (Note: this does not show values for note length, that is a NextNote list can contain multiple C#, but each has a different length)

For each DependHolder it then sums the frequencies of each of its NextNotes and then sorts them by length into numerical order, this is so the note selection process can be waited towards shorter notes (see section X).

### 3.2.2 - Choosing Notes

The function to generate a new note takes the frequency distribution and a MidiHolder as inputs. Although, it first gets the current intensity of the game from the MusicController and the intensity weighting, this is the amount the intensity will affect the note choosing process (discussed below). It then loops through the frequency distribution and checks to see if the inputted MidiHolder has occurred, if it has it then goes onto to choosing the new note to generate. It first generates a random number between 0 and the summed frequency count (from the DependHolder) and checks to see if the frequency of the first note in the NextNote is greater/equal than this, if not it adds the next NextNote’s frequency to this and checks again. This continues until the running frequency count is indeed greater than the random number. This is shown below in Figure X, where ‘A’ is chosen as the new note to be generated.

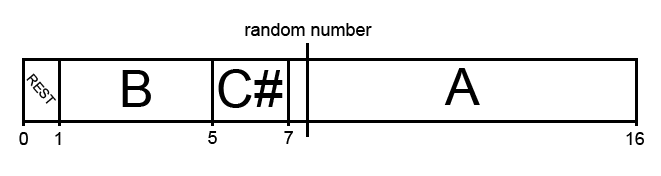


Table 2: The note generation process, in this case A is chosen.

To allow the intensity of the music to influence this process, a weighting is calculated using the following formula:

random\_weighting = -1 \* intensity \* intensity\_weighting \* max\_frequency

Figure 3: The calculation for the random number weighting, using the intensity, the intensity\_weighting (how much the output of this calculation effect the random number generation, and the max\_frequency.

The lower limit of the random number generation is then set to this random weighting, while the upper limit is set to maximum frequency plus this. Both limits are then clamped between 0 and the max frequency. This causes the random number generation to be weighted towards low numbers when the intensity is low (and so negative) and high numbers when its positive, this is shown in Figure X. Coupled with the fact that the NextNote are ordered by their length. This causes the note generated to more likely be longer when the intensity is low and shorter when the intensity is high.

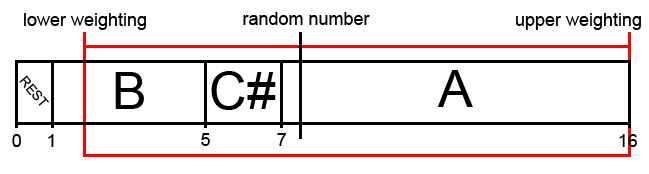


Table 3: The note generation process, the upper and lower limits are shown in red. Note that the rest cannot be generated in this scenario.

If this process goes through the entire frequency distribution and does not find a match, it tries this previously discussed process again, but ignoring the length of notes when comparing them, this causes the process to have a higher likelihood of finding a match, at the cost of musical integrity. If this also fails to find a match, a random note is chosen from the frequency distribution.

### 3.2.3 - Creating Structured Melodies

As both Period and Sentence structures (discussed above, see section XX) are made up of four short phrases, two of which are the same, three phrases are generated. This process uses the Markov chains process to generate notes (discussed above, see section [3.2.2 - Choosing Notes](https://docs.google.com/document/d/1-eXPu0vgbwyuGxYmOtao0lSkEjurGrp4qaqC3nrLUo4/edit#heading=h.2et92p0)), while it is doing this it keeps track of the current temporal length of the phrase, once this length exceeds a specified number, the generation stops and the last note’s length is reduced so the total length of the phrase complies with the specified maximum length. The three phrases are then added to a list of MidiHolders, the order this happens in depends on the structure used. For example, if the three phrases are named A, B, and C, Period structured would be ABAC and Sentence would be AABC.

### 3.2.4 - Playing the Music

The main concern of the application in terms of playing notes is keeping them in the correct timing. To keep track of the time for this, the application updates a variable using the MusicControllers ‘time\_step’, this is the length in seconds that the shortest note length (defaulted to 0.25 beats) takes at the current bpm, the calculation is as follows:

time\_step = 60 / (bpm / shortest\_note\_length)

Figure 4: How the MusicController calculates the time\_step

By comparing this against time from Unity’s inbuilt AudioSettings, the application determines if it needs to play a note, this is done by checking a counter against the current note’s length, this counter is also increased by the ‘shortest\_note\_length’ and reset to zero when a note is played. To stop the music going out of time if the bpm changes while a note is playing, each note is passed to a coroutine to be played.

This starts by checking if the note is a rest (if it is the pitch will be -1), if this is not the case it will continue. It then retrieves the necessary information from the note to be played (pitch and length). The pitch is then corrected, based on the current mode (discussed below, see XX). This is done by first getting the ‘base version’ of the note, which consists of subtracting 12 from the note’s pitch until it is below 12, for example if this process is done to C4, its note number of 60, the resulting ‘base’ note is 0, for D7 with a note number of 98 would be reduced to 2. The number of times 12 has to be subtracted is also saved as the amount the note’s pitch has to be reduced by. As the musical scales are in the range of 0 to 11, this base note is used to find the closest note in the scale, numbers that are in the scale are used immediately while numbers that are not are increased by 1. For example, if D is used (base note of 2) and the current scale is Locrian, D (i.e. 2) is not present in this mode so it is increased to D# (i.e. 3).

Once the base has been converted into the correct mode, it is then raised back into the correct pitch by adding 12 multiplied by the pitch shift. This new note is then passed to the synthesiser to be played. The note’s length will then be converted into a time in seconds using the following formula;

time = note\_length \* (60 / bpm)

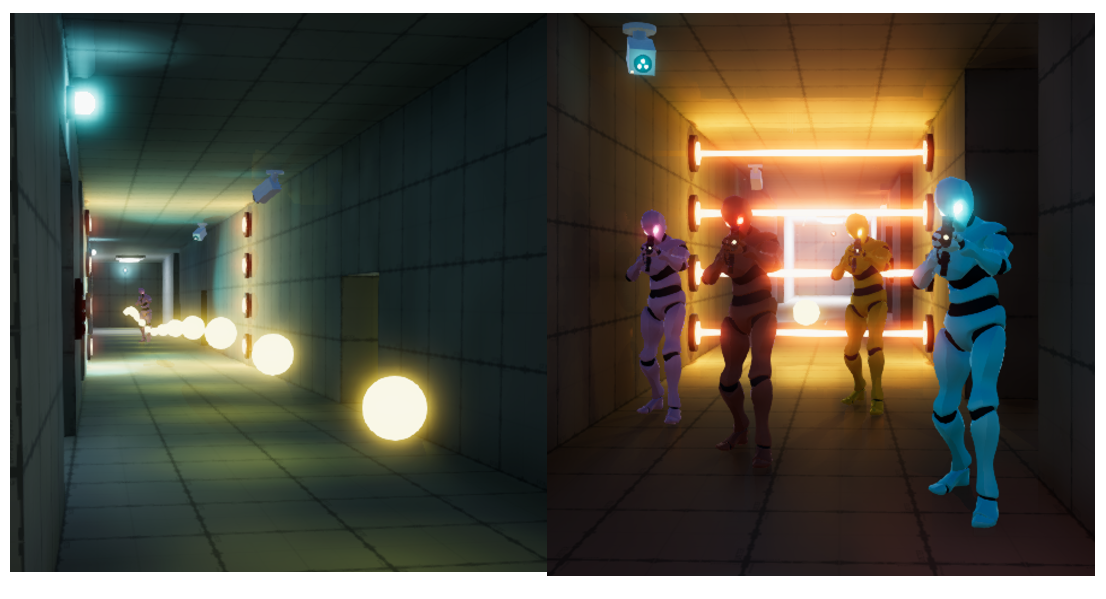
Figure 5: The calculation to find a note's length in seconds at the current bpm

The coroutine is then paused for the calculated length of time. Once this time has passed the synthesiser is then told to stop playing the note. The next note is then chosen, if the structured music is currently being played the next note is taken from the list of progenerated notes, if not then the current note is passed to the ‘choosing notes’ process (see above in section [3.2.2 - Choosing Notes](#_3.2.2_-_Choosing))

## 3.X - The Game

### 3.X.1 - Basic Idea

The first objective for designing the game was to determine the most appropriate genre which would provide enough variability in the gameplay to give the music generation application sufficient space to showcase how it can modify the produced music. The game that was chosen was a first-person stealth game, inspired by Pacman (XXXXXXXXXXXXX, 19XX), the player is tasked with collecting 200 orbs in a sci-fi style facility, while trying to avoid robotic guards, shown below in Figure X



A B

Figure 6: The orbs the player must collect (A) and the robotic guards they try to avoid (B)

### 3.X.2 - Game Intensity

The first way the game varies the intensity is the player’s movement speed, the player has three options for this; standing still, walking, and running, each having a larger effect on the intensity than the one before.

The player’s interactions with the guards also influences the intensity, the closer they are to the guards the higher the intensity is. The guards also have four states they can be in (ordered in decreasing intensity);

* Hunting - The player’s position is always known to the guards and they actively track them down.
* Chasing - When they see the player the guards will chase after them, until they can no longer see them.
* Searching - When they lose sight of the player they will go to the player’s last known location, when they reach this position they will turn in a circle to see if they can locate the player again.
* Patrolling - This is the default state where they follow pre-set paths around the facility.

These states interact with each other according to the following state diagram (Figure X).

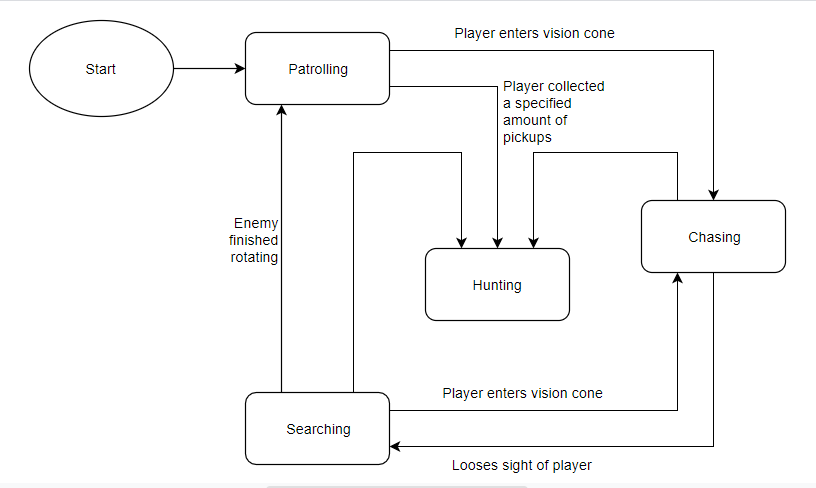


Figure 7: State diagram for the robotic guards

The amount of pickups the player has collected also has a positive effect on the intensity, the more they collect the higher the intensity. Other factors affect the intensity of the game, although they do not directly affect the music they make the game harder to complete and so the player will get more stressed. The first of which is that the player must also look out for security cameras (top left in Figure X - B), when they see the player they activate a laser grid which will block the player’s path (again shown in Figure X - B). The second is when half of the orbs are collected the facility goes into ‘lockdown’ mode, which causes the laser grids to be activated for the rest of the game, this change is indicated by a dialogue line and warning lights (Top right if Figure X - A) which change from their default blue to orange. When 80% of the orbs have been collected the facility goes into ‘hunting’ mode, indicated by another in-game dialogue announcement, the warning lights turning red. The robot guards at this point actively hunt down the player by permanently being in their ‘Chasing’ state.

### 3.X.3 - Game Valence

The first variable that affects the music’s valence is the number of orbs collected as this is the main way the player will track their progression, the higher this is the higher the valence is. The second is the number of lives the player has, starting with three and each time they collide with a guard they lose a life, and the valance is decreased. The state the guards are in affects the valence in a similar way that it affects the intensity, although for valence it has a negative effect.

### 3.X.4 - Intensity/Valence Calculations

To allow the various events in the game (described above) to affect the music, their value must first be calculated. This is done by scaling each value to be between 0 and 1. The exception to this, is the speed factor which is ranged from -1 to 1. These values are calculated as follows:

* Progression - The ratio of collected orbs to the total number that available.
* Guard Proximity - The distance between the player and the closest guard, this value is then scaled down by a minimum and maximum distance, clamped between 0 and 1, and then subtracted from 1.
* Guard State - The guard’s state is stored as an enum and so the integer value is used (patrolling = 0, hunting = 3), the highest of which is used, by dividing it by 3.
* Lives Left - The player’s current lives are divided by the starting number of lives.
* Speed - If the player is still the speed variable gets set to -1, walking sets it to 0, and running to 1. This value is then interpolated between these values so the change in the speeds affect on the intensity is smooth.

The values are then each multiplied by a specific scaling factor, so they have the desired effect on the intensity and valence. The default scaling values are shown below in Figure X. However, these values can be change from the Unity inspector to better suit what a developer wants to form the system.

|  |  |  |
| --- | --- | --- |
| **Variable** | **Intensity** | **Valence** |
| Progression (P) | 1 | 1 |
| Guard Proximity (Gp) | 1 | 0 |
| Guard State (Gs) | 1 | -1 |
| Lives Left (L) | 0 | -1 |
| Speed (S) | 1 | 0 |

Figure 8: The default scaling values for the various event values.

The intensity and valence targets are then calculated using the following equation:

I = (a1S + a2Gp + a3Gs + a4P) / 4

V = (a5P + a6L+ a3Gs) / 3

Figure 9: Intensity and Valence equations, a is each value’s scaling factor, as shown above in Figure 5

The final intensity and valence are then modified with delta time so the transition between states is not too jarring (the speed at which this happens can also be set in the inspector). These values are then clamped between 1 and -1.

## 3.X - Questionnaire

The main aim of the questionnaire was to determine the effectiveness of the application. For ease of access the questionnaire was created using a Google form, so it could be sent to participants (the complete form can be found below in Appendix XX). The first section is on the previous experiences of the participants, this asks how much time they spend in a week playing video games, as video games are a common source of procedural/reactive music. It was hypothesised that people with this previous experience would be able to hear the effects with greater ease. For this same reason participants were also asked if they played a musical instrument, as they would be more familiar with changes in music. They were also asked if they thought reactive music was important.

They are then asked to download and play the game (a Windows and Mac build is supplied for ease). After they have done this, they are asked to describe the music at four points in the game: at the start. when the warning lights turn orange, when the lights then turn red, and when the guards are chasing them.

They are given seven options taken from the circumplex model (see section XX, Figure X), if they choose other they are given the option to write how they would describe the music in their own words, shown below in Figure X.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Exciting | Stressful | Happy | Sad | Relaxed | Boring | Other |

(Optional) If other, how would you describe it?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Figure 10: The options given to the questionnaire participants when asked to describe the music

To ascertain if the participants could tell which events were causing the music to change they were shown nine distinct events and asked if they affected the music. These included the five events which did affect the music (discussed above, see section XX) and the remaining four had no direct effect on the music, these four were as follows;

* The player is close to a laser grid
* If a security camera has seen the player
* The player is inside a secret passage
* If the player has used a teleporter

To make sure the participants understood which event they were being asked about, a picture of the object in question was supplied. This was not done for the questions on the number of lives, speed of the player, and the score as these events contained several temporal elements and so a picture of it could not be easily obtained.

The participants were then asked to give more general feedback on the music and the game itself, to show the main way the application could be improved if worked on further.