# 3 – Methodology

The application was developed in the game engine Unity and written in C# using Visual Studio 2017. 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 they have 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 songs 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 that is 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 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 note 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.

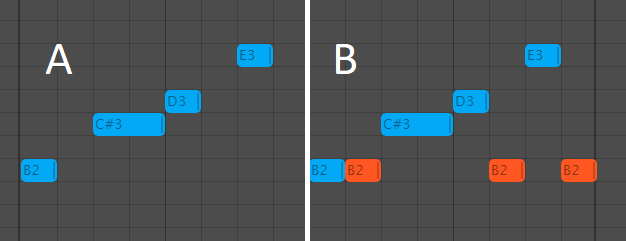


Figure 1: The MIDI sequence before adding rests (A), and the sequence after rests have been added into the spaces between notes (B). Blue bars represents note, orange bars represent rests.

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 file 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 note. 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 disorganised music

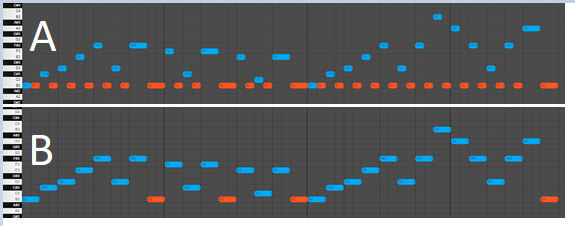


Figure 2: Adding rests with no minimum length (A), and where the minimum length is set to 0.25 (B) note the vastly reduced amount of rests added (from 26 to 3)

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 creates a new DependHolder with its MidiHolder set to the current note and then it adds the next note to its NextNote list and sets its frequency to 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 occurs 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.

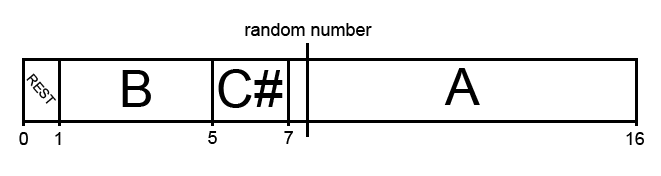


Table 2: The note generation process, in this case A is chosen.

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

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.

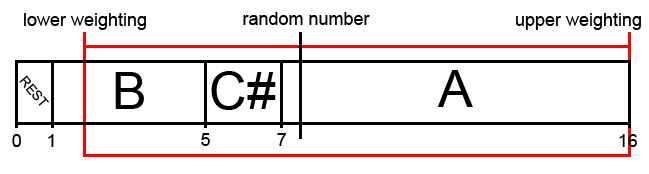


Table 3: The note generation process, the upper and lower limits are shown in red. Note that the rest cannot be generated in this scenario.

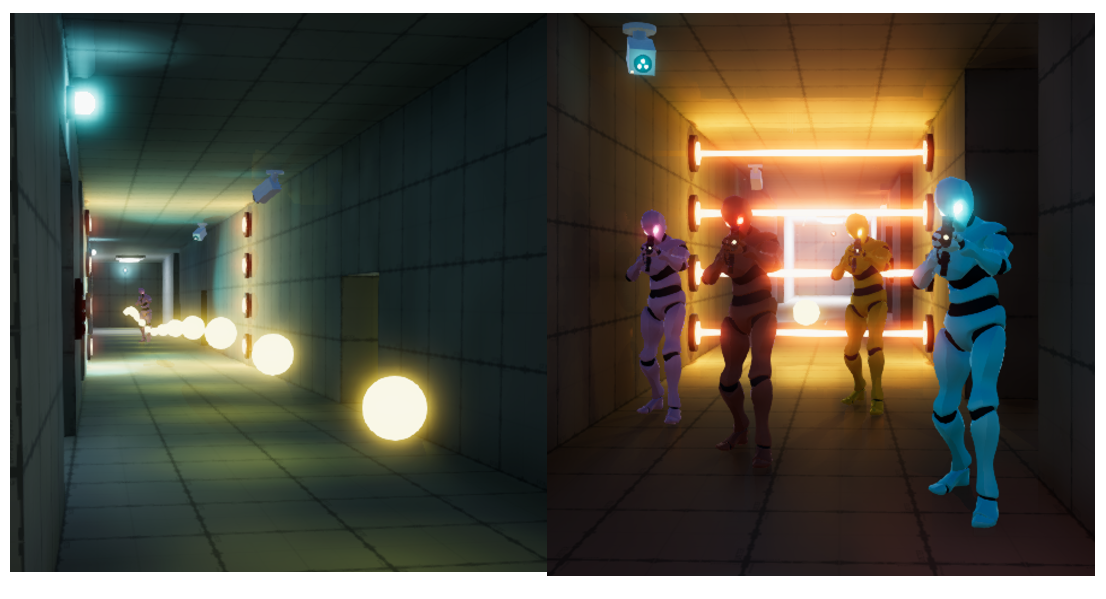
If this process goes through the entire frequency distribution and does not find a match, it tries this previously discussed process again, but ignoring the length of notes when comparing them, as this would have a higher likelihood of finding a match, at the cost of musical integrity. If at this point a match has still been found, a random note is chosen from the frequency distribution.

## 3.X - Questionnaire

## 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 variability 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 avoid the robot guards, shown below in Figure X



A B

Figure 3: The orbs the player has to collect (A) and the robots guards they have 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 guards also has an effect on the intensity, the closer they are to them the higher the intensity is. The guards also have three states they can be in (in order of decreasing intensity);

* Chasing - When they see the player the guards will chase after then, until they can no longer see them.
* Searching - When they lose sight of the player they will go to the players 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.

The player has to also look out for security cameras (top left of in Figure X - B), when they see the player they activate a laser grid which blocks the player’s path (again shown in Figure X - B). To increase the intensity of the game, when half of the orbs are collected the facility goes into ‘lockdown’ mode, which causes the laser grids to be on for the rest of the game, this is change is indicated by a dialogue line and the warning lights (Top right if Figure X - A) change 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, and an alarm playing. The robot guards at this point actively hunt down the player, by permanently being in the ‘Chasing’ state. These two changes actively increase the intensity of the game and thus increase the intensity of the music.

### 3.X.3 - Game Valence

The first variable that effects 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, they start with three and each time they come into contact with a guard they lose a life, and the valance is decreased. The state the guards are in effects the valance in a similar way that it effects the intensity, although for valence it has a negative effect.