The sEV3n Nation Army

Group 8 - 29

Mahir Mahota - 20991134
Ryan Scomazzon - 21003008
Iram Anwar - 21016841
Noah Yacowar - 21030438

MTE 100/121
2022/12/06

Summary

The main goal for the project is to create an EV3 robot that reads a given song through a colour sensor, sets time intervals and frets based on the colour inputs, and plays the song on one string. It will do this by using both a mechanical strumming system to pluck the string and a mechanical fretting system to rotate a camshaft and press a specified fret. The C++ functions will convert MIDI files to pairs of coloured blocks with the first block being the time and the second block being the fret being pressed. The RobotC functions will start the tasks and follow by reading the coloured blocks that were printed on paper and convert the information into two arrays: the first array stores time and the second one stores the fret. Then, the code will cycle through both parallel songs to play the song.

By the end of the demo, the robot successfully played two different songs: Seven Nation Army and Eye of the Tiger. The colour reading system read each colour correctly and each array had the correct data. The plucking system successfully followed the timer, and it did not go in the wrong direction during the demo. The fretting system pressed down the correct frets and it passed the auditory test.

Acknowledgements

The group acknowledges that the robot is built on the traditional territory of the Neutral, Anishinaabeg and Haudenosaunee peoples. It is situated on the Haldimand Tract, the land granted to the Six Nations that includes six miles on each side of the Grand River [1].

The C++ music conversion system is based on sample code from David Barr, Copyright 2018-2020, OneLoneCoder.com [2]. Details regarding which parts are original are in code comments located in Appendix A.

Image file writing was created with a stb image implementation library created by Sean Barrett, Copyright (c) 2017-MIT open-source licence [3].

Introduction

The idea for this robot was conceptualised from a simple thought. Playing the guitar is cool! However, there are a couple problems. Firstly, the team, being engineering students, has no time to play. Secondly, none of the team members have the expertise to shred up the guitar like bands such as The White Stripes. While the average engineer may not be great at the guitar, they sure can build a robot that is. That is why the sEV3n Nation Army was created.

Scope

At its simplest, the robot plays one string on a guitar. An extra functionality that was decided on was to add a colour reading mechanism that could read notes into the program, somewhat like a coloured barcode that defined the notes and timing for a particular song.

The robot has three major systems that work together to fulfil its aim. Essentially, playing a guitar consists of pressing down on a fret to adjust the length of a string and then plucking the string to produce a sound. Thus, two of the systems are fretting and strumming which are dedicated to playing the appropriate sound. The third system is dedicated to the colour reading tasks.

The project takes in input from multiple sources. Two types of sensors are used, one colour sensor for reading the coloured strip and an ultrasonic sensor to detect when the paper has ended. Apart from this, the enter button is added as a pause condition before crucial steps in the program so that the user has more control of when the robot starts its next task. It also uses motor encoders to measure the angle the motors have travelled. Each fret is located at a certain angle offset from the one next to it and this angle is defined as a motor encoder value in the software.

Three motors are used throughout the project. The medium motor is used in the strumming mechanism to strike the string. The bigger ones are used to rotate the camshaft in the fretting mechanism and to drive the paper strip in colour reading.

One change that was made in the scope of the project is the range of notes that the robot could play. The initial goal was for the robot to be able to read in and play the first 11 notes on the guitar but due to colour sensor limitations, this was reduced to seven notes.

Constraints

One main constraint is that the frets must be pressed accurately and firmly. Without fulfilling this, the robot would sound bad and not be able to perform to the required level. To meet this, the fretting system had to be mounted correctly and alignment of the mounting was a significant factor to consider. The distance the string is pressed down from the fret affects the quality of sound of the note played. To test if the frets are fulfilling the criteria, each double cross block piece must be pushed down onto the string so that the string is firmly pressed against the neck of the guitar. Additionally, if the string is plucked while a fret is engaged and the sound is obviously incorrect, then it is evident that the constraint is not being met. This is the primary method used to calibrate the fretting system, and when the sound for each fret was adequate, the constraint for the system was met.

The robot must play at least one string. It was decided that only one string would be used to simplify the fretting mechanism. This is mainly due to the limited motors and LEGO parts available as well as due to the relatively short time span for the project. This was a consideration that only affected our overall project design but did not limit the robot very much as there is a large collection of songs that can be played on one string successfully.

The robot must also be able to distinguish notes by colour. This is one of the distinguishing factors of the robot. Instead of a song being coded into the program, it is read in using the hardware and this way multiple songs can be played depending on what strip of paper is inserted. To be successful, the colour sensor must accurately recognise the colours and the speed of the paper strip had to be adjusted to ensure this. In the final demo, two different strips of paper were inputted with two distinct songs and each colour was demonstrated to have been read correctly.

Finally, the music that the robot reads must be automatically generated from a music file. Specifically, it must read a MIDI file which is a common format in the music industry that contains data about each note of a song.

Criteria

The strumming must be timed correctly and should be clear. The strumming system had to be synced to the fretting in the program. It should hit the string as soon as the fret is set to the correct position, and it should not play while the frets are being changed. It should also strike the string with enough force to produce a clear sound without damaging the string. This can be accomplished by testing various systems to see which one produces the strongest sound with the same motor power. The robot will function without this working at its best, but this is a good area to focus on.

Moreover, overcoming the EV3 motor speed and accuracy to fulfil all the song speed requirements fully would be a valuable addition. This was a major factor that would affect the design of the project as the optimal speed had to be found through testing. Moreover, this restricted the project from playing overly fast songs that the motors would not be able to handle. The songs can be played at a slower pace without any problems but speeding them up sounds smoother.

The robot should be able to play more than one song with the frets available and the frets should be chosen in a way that multiple recognizable songs are playable on them without rearranging or moving the fretting mechanism.

Mechanical Design and Implementation

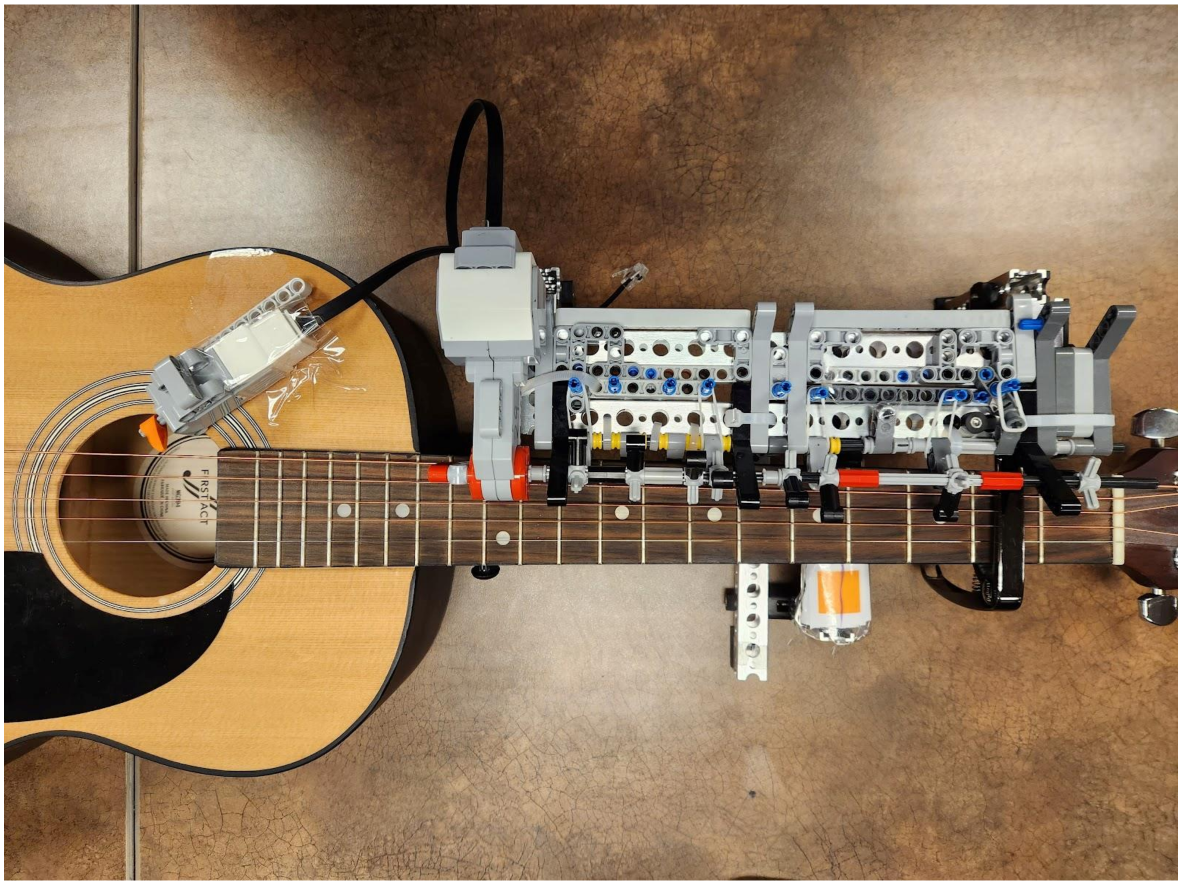


Figure 1: Strumming and fretting system mounted onto guitar

Fretting

The fretting system is based on the concept of a camshaft where rotational motion is converted to linear motion. This type of shaft converts the power of a singular large motor into something that can press down onto the string for multiple frets. An axle with 3-spoke angular blocks is connected to the motor such that the 3-spokes put force on another axle (free-rotating) which has multiple double cross block pieces connected to it. These pieces are shown in Figure 2 for clarification.

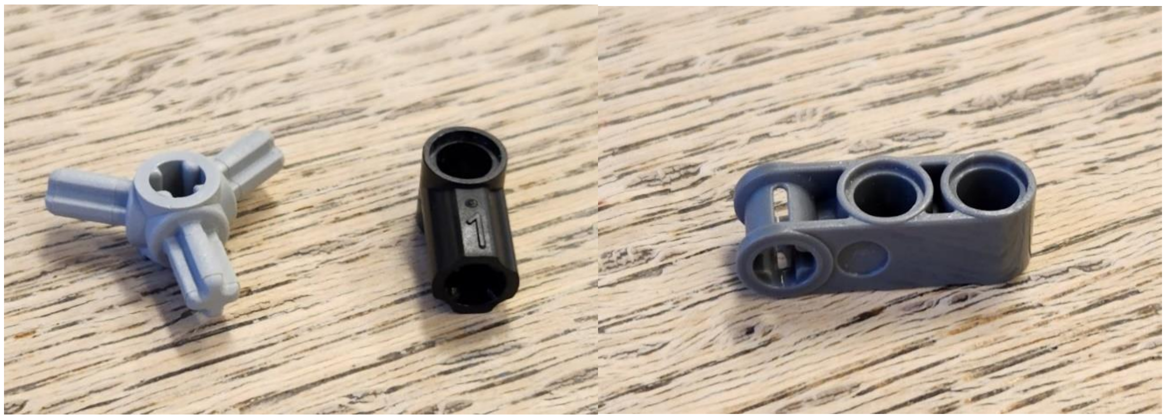


Figure 2: From left to right, 3-spoke angular block, camshaft head, double cross block piece

The string is pressed down at a particular fret when the camshaft is rotated to the right-angle using motor encoders. With the LEGO pieces provided, a total of twelve frets can be pressed down one after the other in this way. Five of the twelve were utilised in this project. One of these is shown in Figure 3.

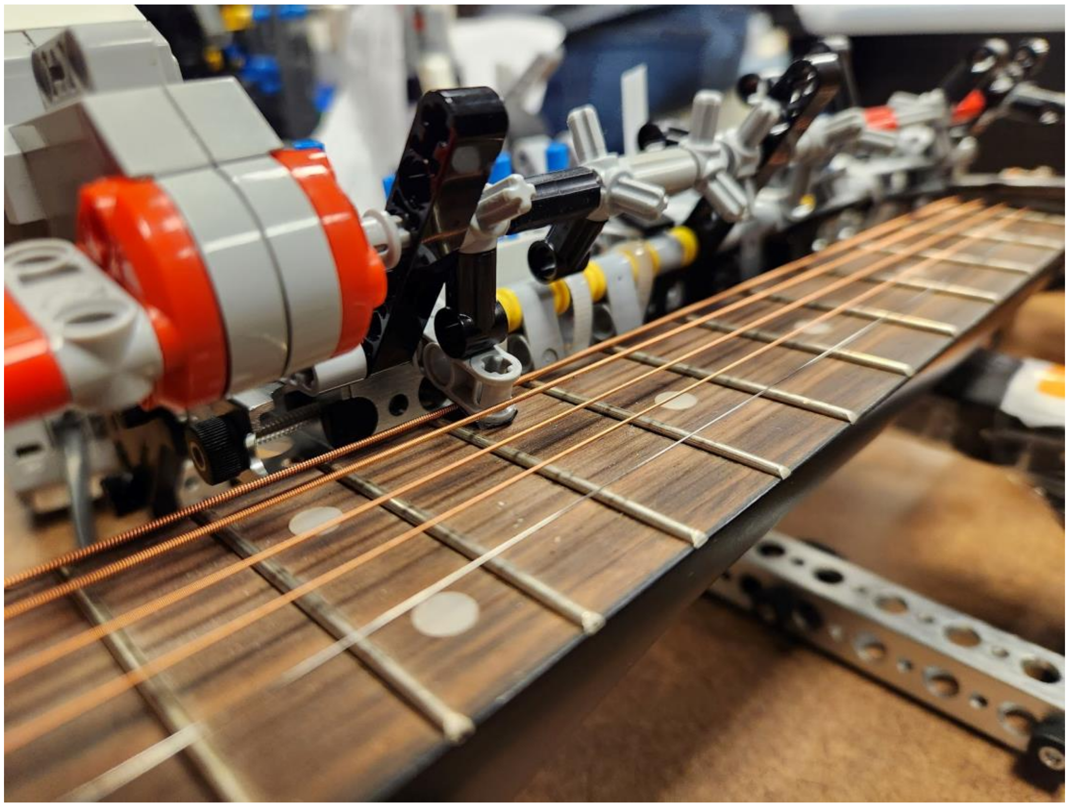


Figure 3: String pressed down on tenth fret

The mounting for fretting was a challenge. When a string isn’t pressed down with enough force, it buzzes against the metal of the fret making the wrong sound. Getting the right alignment and force was an issue. There was very little room for error as the axle cannot handle too much force. However, this was a limitation of the LEGO pieces provided. Therefore, metal beams were used from a Tetrix kit to support the axles. The axle was fastened to a beam to stop it from bending during the song.

Moreover, the height of the camshaft relative to the guitar had to be finetuned as well. If it was too high, not enough force was produced on the string and if it was too low, the camshaft head would get stuck against the neck of the guitar restricting its range of motion. In order to reach the right height with the limited Tetrix beam lengths, padding was added underneath the neck of the guitar to adjust its height relative to the fretting mechanism.

Challenges with Fretting

One of the problems for fretting is that the group did not have an axis long enough for the length of 10 frets. Therefore, it was done by combining multiple axles together with connectors. This meant the alignment of the cams is not perfectly in-tune with the centre of the pieces and the sound of the string as it was plucked was not ideal. Therefore, the system had to be broken and rebuilt multiple times to fix the alignment. Through experimentation, there were 12 different angles that the 3-spoke angular block could have so it was relatively simple to re-align the camshaft after understanding how the angles worked.

Another problem with the fretting system was the fact that the double cross block pieces would be resting on the strings when they were not being pressed on. This is a problem because guitars make sound based on the vibrations the strings produce. Therefore, it was important to ensure that there was nothing stopping the vibrations of the string. To do this, there are rubber bands for each double cross block piece such that they are retracted above the string unless they are pressed down. This ensures that only one fret is pressed at a time in addition to ensuring that a maximum of one fret is touching the string at once, essentially solving two problems with one solution.

Also, the fretting system required a significant amount of minute adjustments to achieve acceptable sounds from each fret. Through trial and error, a singular location was found where each double cross block piece had just enough pressure on the string without getting stuck on the string or striking the neck of the guitar. To mark this piece (which was named the datum), a parallel line was drawn on the Tetrix base to allow consistent testing.

Strumming

The strumming system is relatively simple but efficient. Since the project is constrained to playing just one string, the motor only needs to pluck one string. Therefore, it is as simple as attaching a LEGO pointer part to an axle and having it rotate to pluck the strings. The range of rotation is restricted to 45 degrees to allow the plucking to respond quickly while still having a range of error if it is not initialised in the correct start position. Another design consideration is the angle of the motor.

Challenges with Strumming

One big challenge with this mechanism is that the pointer part needed to be precisely in the right spot. If it was too close, it would pull the string with it or jam up. If it was too far, it would miss the string. I had to be within millimetres of the correct spot. After some iteration, the team found that the motor must be placed at an angle to the string so that the curved side of the LEGO pointer hits the string instead of the flat end. This is more forgiving in terms of the positioning of the motor as the string does not get stuck on the pointer itself and can slide off the curve if necessary. This angle and the strumming setup can be seen below in Figure 4.

However, it is also important to note that the LEGO pointer part was not in the original design of the robot. Initially, the intention was to create a CAD model of a pick holder that could bend as it moved. However, after trialling the pick with a makeshift holder, it was apparent that the sound produced was worse than the pointer. This was because picks were designed to glide over strings horizontally at an angle whereas our motor was designed to move something perpendicular to a single string. This meant that the pluck slapped the string without creating any sounds. Therefore, the pointer was preferred even though it also negatively affected how the guitar sounded.

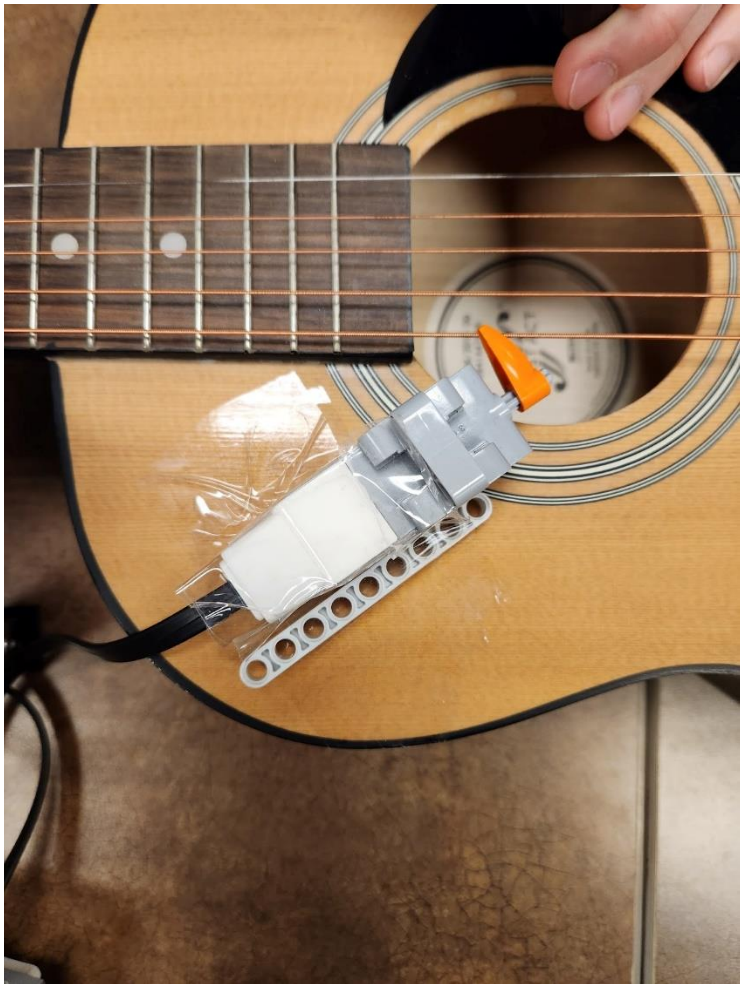


Figure 4: Strumming motor

Colour Reading

This mechanism has two wheels run by one motor that roll the paper strip through the system. The coloured strip passes under the colour sensor that reads the values and sends them back to the EV3. There is an ultrasonic sensor also placed above the paper right before the colour sensor. This detects when the paper has run out so that the robot knows to stop reading and it does not read anything below as a note. It also indicates that the robot can move on to play the song itself. Finally, pegs and axles are used as braces to keep the paper a constant distance from the colour sensor and to keep it continuously aligned. Figure 5 shows a coloured strip being fed through the system.

Challenges with Colour Reading

Initially, the plan was to spool the strip of paper around a circular paper holder, but it became apparent that this would create more problems than solutions. The paper would get crinkled or the system itself would jam because the paper would fold. This meant that the easiest way to feed the paper was to have it go straight out onto a flat surface so the paper would remain flat and so the risk of jamming is minimal.

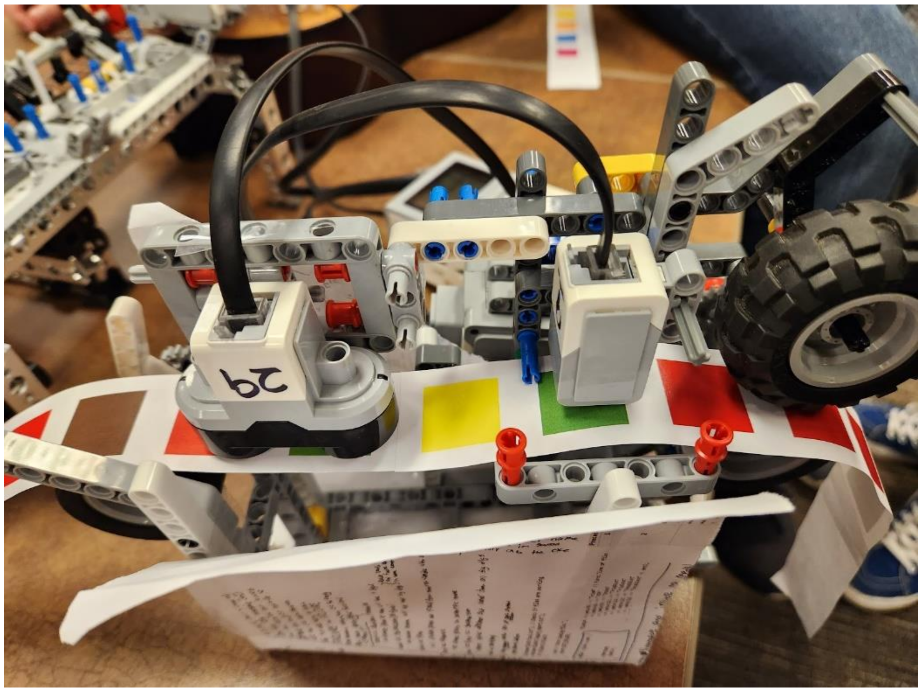


Figure 5: Colour reading system showing ultrasonic sensor and colour sensor

Software Design and Implementation

The robot contains many different systems, each of which is its own challenge. To keep the project organised, the software for each of these systems is created in separate files with separate functions. To keep the main program simplified, most systems are designed to have only one or two primary functions that are used by main while the rest of the functions are kept isolated from the rest of the program and only used in their respective systems primary functions. Finally, the detail in main such as start-up and shutdown were also abstracted.

The software is split between the various mechanical systems as all of these contain complex logic to function and can tend to get complicated if not dealt with separately. However, they can all be simplified into one function that abstracts this detail away, so they do not get out of hand when combined. Figure 6 and Figure 10 show how the program executes its tasks while treating each system in separate functions.

Data types

Two integer arrays are passed to various functions and are used to store the important data that the robot needs to function properly. Namely, these are the fret numbers and the timings between the notes. Both sets of numbers are stored in parallel arrays. For example, the fifth timing value is the time to wait before playing the fifth fret in the song. It can also be noted that the index of these arrays represents the current note. In the C++ portion, there is also a parallel boolean array that stores if a note has been read into the specific index. This just makes calculations easier when iterating through many times to create an image.

Significant Issues

The team faced two major unexpected issues relating to software. The first was with processing colour data. The colour sensor is very inaccurate, and this causes the reading to give incorrect readings when switching notes. This made it nearly impossible to get an accurate reading. To solve this, the software must be advanced enough to process out the incorrect data. This means that a task as simple as detecting when the colour changes become an extremely complex problem to solve that takes days of work. After testing different failed methods such as using delays to skip this period of awkward change, the final solution repeatedly checks the sensor over an interval until it detects the same colour many times. This means it only returns a colour when it is confident that this colour is not just a temporary fluctuation. This is elaborated on more in the function description for getRepeatRead().

The second issue was with the cam shaft overshooting its target. Mechanical constraints would jam the robot, so this must be fixed with a software solution. Slowing down the motion would fix this, but it would also increase the time it takes for the robot to play notes. Using the motor encoders to judge the distance would still cause it to overshoot as the momentum continues the motion slightly. As more frets are played, this small offset compounda and causes larger errors. To fix this, the motor encoder is only reset on start-up and the constantly increasing encoder value is used for all calculations. For the calculations to work, math must be done to convert this ever-growing encoder value into a more readable, 0-360 value. This means that the camshaft can overshoot slightly, and it will not affect future motions as the encoder values stay relative to the start.

Functions

RobotC

Main()

Written by: Mahir Mahota

The main function mostly consists of calling each function and is kept as simple as possible. Most of the robot's functionality is handled in separate functions. When the program is first started, the startAllTasks() function is called to configure everything before anything else runs. Next, when the enter button is pressed, the readSheet() function is called. The colour reading motor starts running and the colour sensor starts reading values into the data arrays. After the ultrasonic detects no paper, text is output onto the EV3 display. The data arrays are passed into the playNotes() function which handles the operation and synchronisation of both the fretting and strumming systems. After all notes finish playing, the endAllTasks() function runs and the program finishes after all mechanisms reset to their default positions.

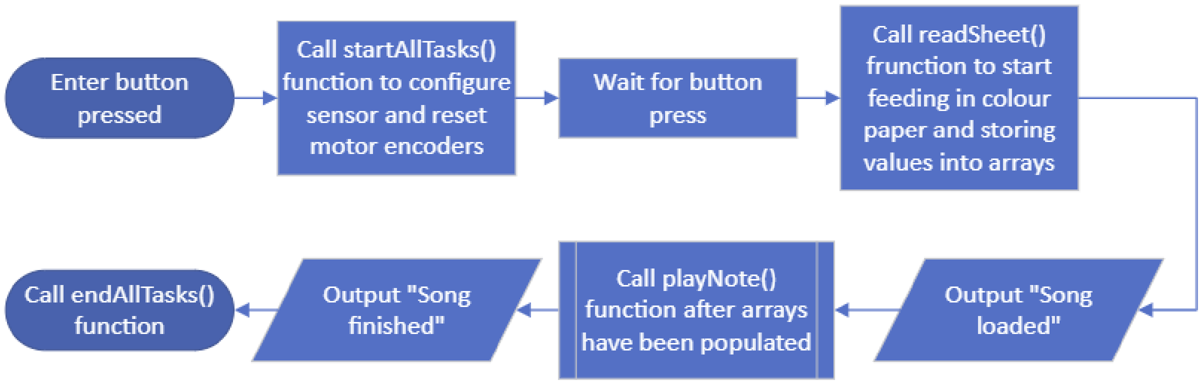

Enter button pressed
Call startAllTasks() function to configure sensor and reset motor encoders
Wait for button press
Call readSheet() frunction to start feeding in colour paper and storing values into arrays
Output "Song loaded"
Call playNote() function after arrays have been populated
Output "Song finished"
Call endAllTasks() function

Figure 6: Flowchart for main()

startAllTasks()

Written by: Noah Yacowar

Return type: Bool to signify successful start-up

Parameters: int for ultrasonic sensor port, int for colour sensor port

The start-up function configures all the required sensors and then resets motor encoders so that encoder values can be used reliably later on in the program. It then returns a boolean as a signal to proceed to the next step. The full flow of the function is shown in Figure 7.

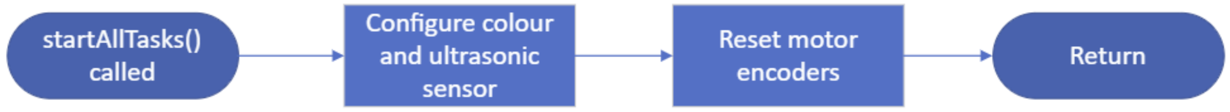


Figure 7: Flowchart for startAllTasks()

setAndPlayFret()

Written by: Iram Anwar

Return type: Void

Parameters: int for fret number, int for timing between frets, int for direction of strumming motor and int for the total number of rotations of the fretting motor

This is the function that is called inside playNotes() for each individual note. It is used to set each fret and it also deals with the timing of the song. The function takes in an integer for the fret number and then multiplies that by the motor encoder value between each fret. It then starts a timer and makes the fret mechanism turn to that fret. The function waits for the timer to reach the desired time between strums and then calls the pluck function. It has another wait to sustain the note for a period and then resets the fretting mechanism to its original position, ready to play the next fret. The flow of the function is depicted in Figure 8.

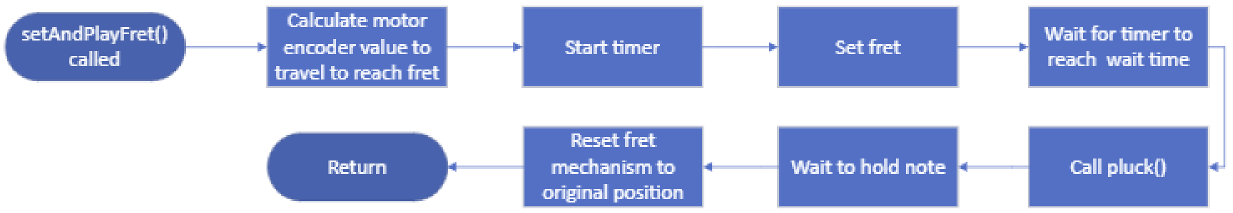

setAndPlayFret() called
Calculate motor encoder value to travel to reach fret
Start timer
Set fret
Wait for timer to reach wait time
Call pluck()
Wait to hold note
Reset fret mechanism to original position
Return

Figure 8: Flowchart for setAndPlayFret()

pluck()

Written by: Iram Anwar

Return type: Void

Parameters: int for previous direction of strumming motor

This function is used to operate the strumming motor so it hits the string at the right time. It takes in the direction, which is either 1 or -1. It resets the strumming motor encoder back to zero and multiplies the strumming motor power (global constant) by the direction. It then waits for the motor encoder to reach the pluck angle defined as a global constant before returning.

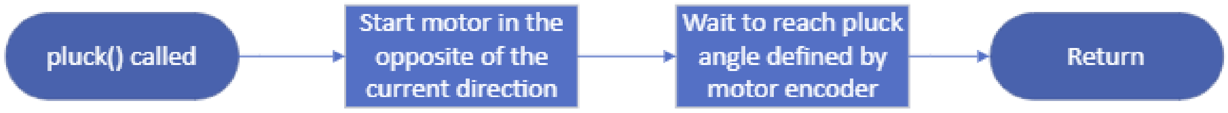

pluck() called
Start motor in the opposite of the current direction
Wait to reach pluck angle defined by motor encoder
Return

Figure 9: Flowchart for pluck()

playNotes()

Written by: Noah Yacowar

Return type: int for direction of strumming motor at the end of the song, to pass into the endAllTasks() function

Parameters: array with fret numbers, array with timings between frets, int for length of the song to play in terms of total notes

Here is where most of the song playing itself is operated. This function cycles through the notes in the fret and timing arrays and passes the values into the setAndPlayFret() function for each. It also keeps track of the current number of rotations of the fretting mechanism and the direction of the strumming motor. It keeps on running the loop while the current index is lower than the total note count passed as a parameter. It returns after all the notes have been played. This process is clear in Figure 10.

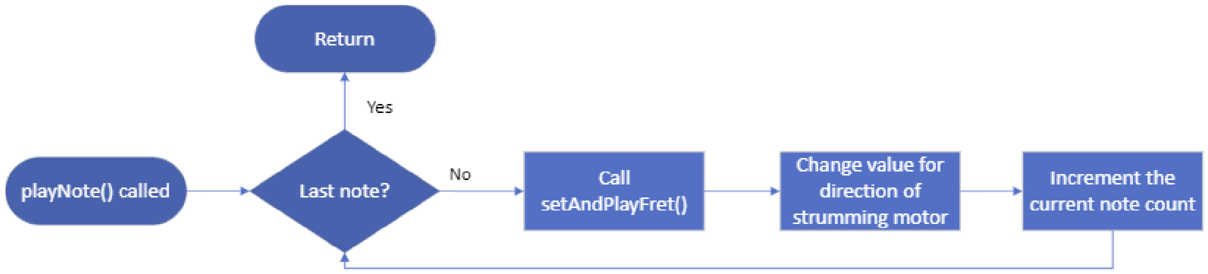

Return
Yes
playNote() called
Last note?
No
Call setAndPlayFret()
Change value for direction of strumming motor
Increment the current note count

Figure 10: Flowchart for playNotes()

readSheet()

Written by: Ryan Scomazzon

Return type: int for total notes read in

Parameters: array reference to be populated with fret numbers, array reference to be populated with the timings between frets

readSheet() is the primary function for the colour reading system and it is the only function that needs to be called from main to have all the paper data be read. Upon being called, it starts the motor, reads in every coloured square on the strip, populates the data arrays, detects the end of the paper, and then returns the number of notes. It yields until this process is finished so main does not have to worry about any logic to detect when notes are loaded. When the function finishes, the rest of the program can continue and easily reference the data from the arrays.

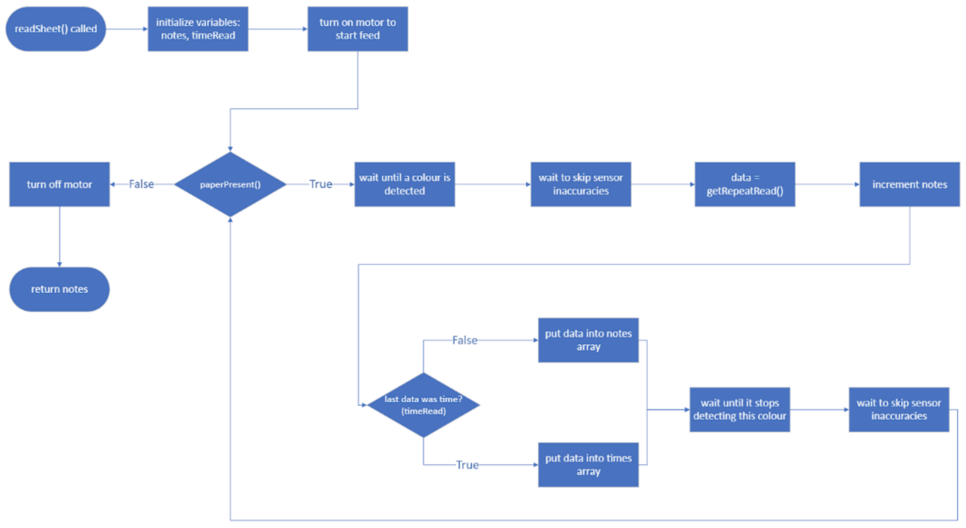

readSheet() called
initialize variables: notes, timeRead
turn on motor to start feed
turn off motor
False
paperPresent()
True
wait until a colour is detected
wait to skip sensor inaccuracies
data = getRepeatRead()
increment notes
return notes
False
put data into notes array
last data was time? (timeRead)
wait until it stops detecting this colour
wait to skip sensor inaccuracies
True
put data into times array

Figure 11: Flowchart for readSheet()

paperPresent()

Written by: Ryan Scomazzon

Return type: bool

Parameters: N/A

paperPresent() is not part of the initial planning but is included to simplify code. It is a trivial function that does the necessary checks to tell if paper is still in the feed and returns a boolean that represents if there is paper. It is used as a loop condition multiple times in readSheet() to tell if the colour reading should continue. Specifically, it does two checks. First, it checks to make sure the distance that the ultrasonic sensor detects is less than it would be if there is no paper. The other check is for the case that the paper is too close to the ultrasonic sensor for it to read. In this case, the sensor defaults to a value of 255, which is larger than the distance that the paper is at. This check prevents this from triggering the first condition.

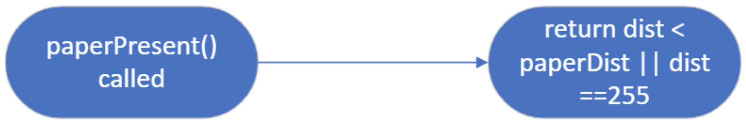


Figure 12: Flowchart for paperPresent()

getRepeatRead()

Written by: Ryan Scomazzon

Return type: int

Parameters: N/A

This function returns the first colour that is read by the colour sensor repeatedly over an interval. It is not one of the initially planned functions but is necessary because the sensor is much less accurate than expected. The sensor would consistently pick up seemingly random colours and flicker between them every time the colour beneath it changed. By comparing multiple checks and doing them over an interval of time this function gives much more accurate results than raw sensor data. It can be fine-tuned with constants that specify the time between checks and how many checks must be the same before concluding that it is the correct colour.

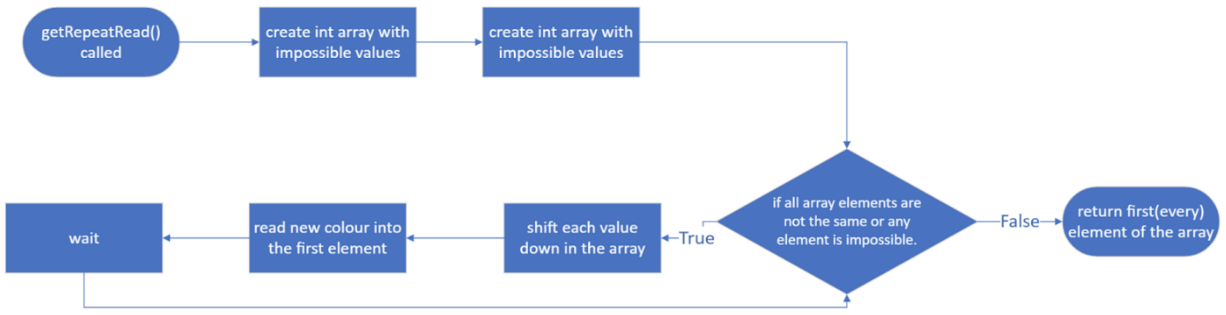

getRepeatRead() called
create int array with impossible values
create int array with impossible values
if all array elements are not the same or any element is impossible.
False
return first(every) element of the array
True
shift each value down in the array
read new colour into the first element
wait

Figure 13: Flowchart for getRepeatRead()

endAllTasks()

Written by: Mahir Mahota

Return type: Bool to signify successful shutdown

Parameters: int for direction of strumming motor after the song finishes

This function resets both the fretting and strumming mechanisms to their original positions so that the next song can play without the need for a person to manually reset each system. It returns a boolean to signify that all systems can reset successfully.

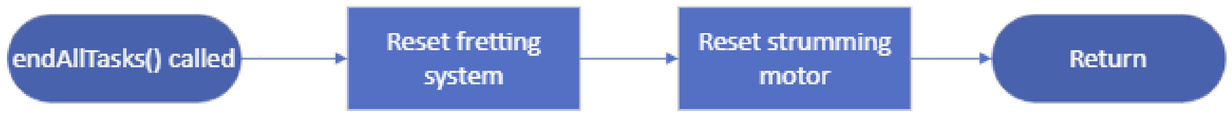


Figure 14: Flowchart for endAllTasks()

C++

readData()

Written by: Ryan Scomazzon

Return type: null

Parameters: MidiFile* data from file loaded, int note array to be populated reference, int time array to be populated reference, bool array reference for if a note exists.

*The MidiFile class is created in the portion of code from David Barr [2].

readData() iterates through the data that is loaded into a MidiFile object and populates the Notes and Times arrays with the data. In order to do this, it must find which part of the file contains the notes. It does this by looping through the different tracks of the MidiFile object until it finds notes. Once it finds them, it loops through the data for each note and calculates two things. First, it converts the musical note from the MidiFile to the fret number on the guitar using a constant offset. It then adds this number to the next position in the Notes array.

The second thing that it calculates is the timing for each note. The time is stored in the arrays as integer multiples of sixteenth notes in music notation. This means that the robot, when reading the paper, can play at any tempo and still understand the music. These standard intervals also mean that less colour sensor values are needed to communicate a range of timing while little capacity is lost as timing less than a sixteenth note is very rare in music notation. The timing is also stored as time between notes rather than absolute time, so the robot does not have to calculate as it goes from note to note. This also means the timing number will be smaller and easier to communicate using the limited colour sensor values. The MIDI data however, returns the time in milliseconds from the beginning of the song, so readData() calculates this desired format by saving and subtracting the time of the previous note, then dividing the entire thing by a fixed constant to get the sixteenth note intervals. This value is then pushed to the next spot in the Times array that corresponds to this note. Finally, it sets the corresponding index in the bool array to true.

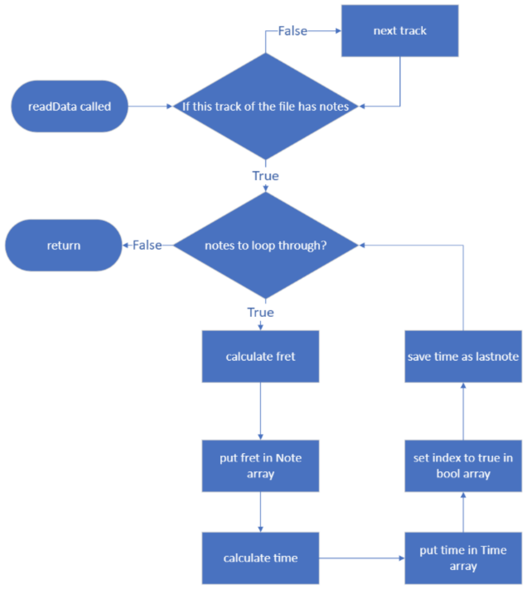


Figure 15: Flowchart for readData()

createImage()

Written by: Ryan Scomazzon

Return type: null

Parameters: int array reference for notes, int array reference for times, bool array reference for if a note exists at an index.

createImage() uses the note data from the three parallel arrays passed through to generate an image that contains the entire data of the song out of coloured squares. This image is formatted to be cut out and fed into the robot during the demo so the data can be read by the robot and played as a song. It uses the stb image library to generate the image [3]. The function iterates through every pixel in the image row by row from the top-left to bottom-right. For each pixel, it iterates through each coloured square and uses constants for square size and spacing to determine where in the image this square would be located and if this pixel is within these bounds. If the pixel overlaps with the square, it sets the pixel colour to the colour of the square. The colour is determined by feeding the stored integer value into a constant array which contains the RGB values of each colour, where the index represents the corresponding integer.

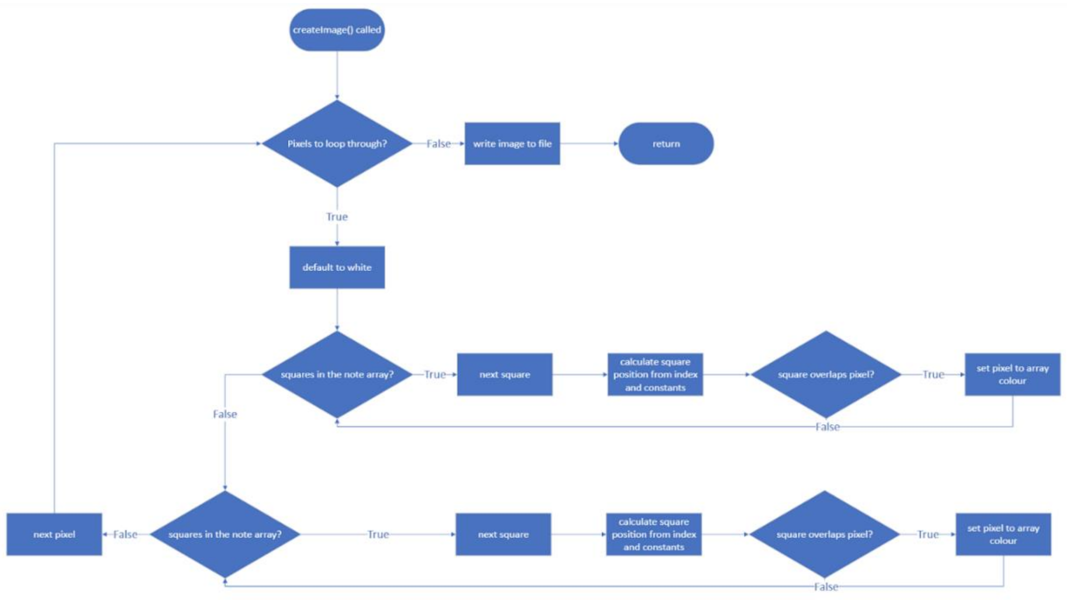


Figure 16: Flowchart for readData()

main()

Written by: Ryan Scomazzon

Return type: int

Parameters: N/A

This is the main function for the song-generating C++ code. It starts by initialising the parallel arrays for storing note data and proceeds to open and read the MIDI file. It then calls readData() to populate the arrays with the MIDI data and createImage() to output the image. Finally, it logs debug data to a file and then exits the function.

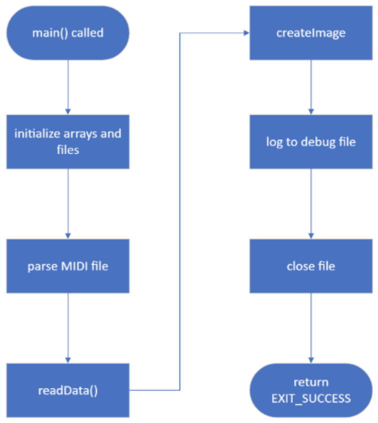


Figure 17: Flowchart for readData()

Task List

Start-Up

- Configure all sensors
- Start-up confirmation is displayed
- Paper is inserted into the colour reading mechanism
- Ultrasonic sensor must detect the piece of paper
- Enter button to start reading sheet

Operation

- Coloured strip is fed through colour reading system
- Colours are read and printed to screen
- Ultrasonic sensor detects the end of the paper and stops reading.
- Camshaft rotates to each fret
- Pluck hits the string.
- Waits correct delay between each note.
- All notes are played

Unexpected:

- Paper falls out of feed: No colour data or ultrasonic data detected, treat as song end
- String break / major mechanical failure: Emergency stop button on EV3 brick

Shutdown

- Final note is played.
- Camshaft rotation is reset to start.
- Pluck angle is reset to start.

Throughout the project, there has only been one change to the task list. In early stages, the option to add percussion that taps the guitar was considered, but this was discarded due to it being unnecessary and taking away time that could be spent polishing the more crucial components. Even when the range of the notes that could be played was reduced from eleven to seven, the range was chosen in a way where the songs could still be played.

Verification

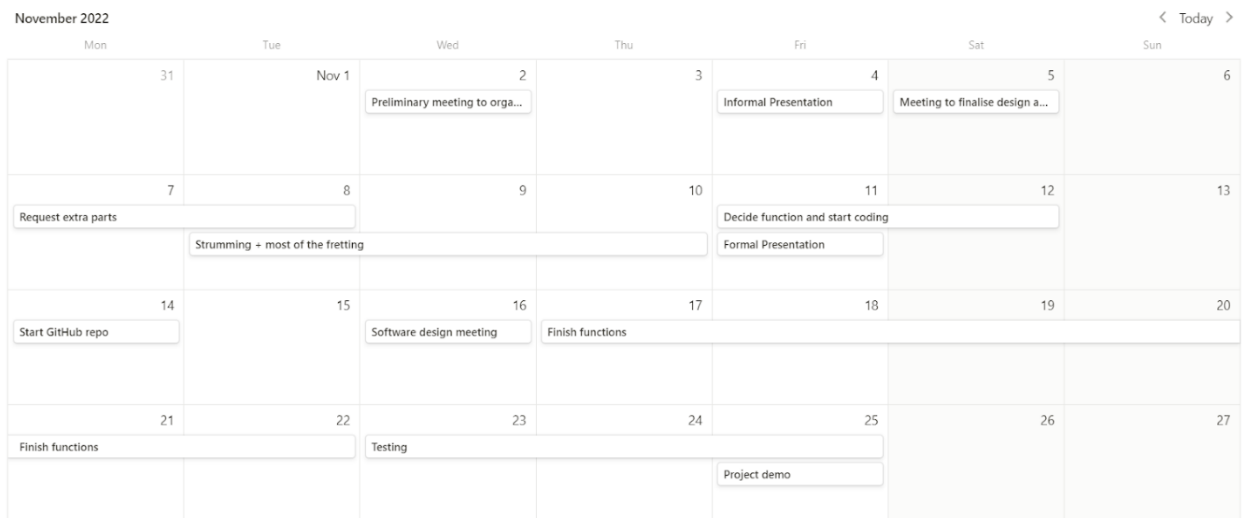
To ensure that the frets are pressed down clearly, testing, and optimising the design is crucial. The overall construction of the fretting mechanism is adjusted over time to straighten the axle and to ensure pressure is applied optimally. Mounting the fretting mechanism involves accounting for forces exerted onto the assembly from pressing. This is especially difficult to do with LEGO pieces because they have low durability. The entire assembly must maintain pressure on the fret without it lifting, otherwise sound is produced properly. The final assembly is tightly fastened to the guitar, and weak points in the assembly were reinforced to avoid lifting.

The robot successfully plays the string that was planned even though it is the toughest string to strum due to its thickness and weight. Moreover, the structure of the LEGO parts makes it difficult to align each of the double cross pieces right behind a fret for the optimal sound, so some trade-offs must be made for ease of mounting. With this, all the constraints planned have been met successfully and the project has also managed to meet all the criteria.

To meet the colour reading constraint, revisions had to be made to the initial plan. The sensor is not accurate enough to read the initially planned eleven colours by hue detection, so this constraint is forced to be at seven colours. The number of frets that the robot could play is therefore smaller and the song possibilities are lower. The six total frets are chosen strategically to ensure that our planned notes could be played. This way, the robot can still play the songs initially planned without any issues but it’s full potential had to be limited.

To ensure that the music sheet is generated correctly, the data printed to the paper is output to a file by the program in a legible form for humans to read. This is used to confirm that the data on the sheet is as intended and corresponds to the correct notes.

Project Plan



November 2022 ‹ Today ›

Mon	Tue	Wed	Thu	Fri	Sat	Sun
31	Nov 1	2 Preliminary meeting to orga...	3	4 Informal Presentation	5 Meeting to finalise design a...	6
7 Request extra parts	8 Strumming + most of the fretting	9	10	11 Decide function and start coding Formal Presentation	12	13
14 Start GitHub repo	15	16 Software design meeting	17 Finish functions	18	19	20
21 Finish functions	22	23 Testing	24	25 Project demo	26	27

Figure 18: Project schedule

Iram Anwar

Mechanical Tasks:

- Building the Tetrix mount for the fretting mechanism (Deadline: November 22)
- Attaching the double cross block pieces with the elastic bands (Deadline: November 22)
- Finish fretting mechanism and meet the minimum criteria (Deadline: November 23)
- Assisting with the strumming function (Deadline: November 12)
- Fixing the datum so the testing is consistent (Deadline: November 23)

Software Tasks:

- Wrote the pluck() and setAndPlayFret() functions (Deadline: November 20)

Noah Yacowar

Mechanical Tasks:

- Assist in designing and building the fretting mechanism where needed (Deadline: November 22)
- Assist with the mounting of the fretting mechanism (Deadline: November 23)
- Adjust the plucking mechanism with fine-tined adjustments (Deadline: November 12)

Software Tasks:

- Write the startAllTasks() function (Deadline: November 20)

- Write the playNotes() function (Deadline: November 20)
- Help on optimising various other functions where required (Deadline: November 23)
- Assist in the general plan for approaching software problems (ie. method to store sheet music within code) (Deadline: November 20)

Mahir Mahota

Mechanical Tasks:

- Assemble the prototype of the Colour Reading Mechanism with a working colour sensor and paper feeding method (Deadline: November 11)
- Finish the strumming mechanism (Deadline: November 11)
- Optimize fretting system so the right amount of padding is used on the double cross block pieces (Deadline: November 22)
- Stabilize Tetrix and LEGO pieces by using a combination of zip-ties, force fits, and tape (Deadline: November 23)

Software Tasks:

- Write the endAllTasks() (Deadline: November 20)
- Write the main() function (Deadline: November 24)
- Help optimize playNotes() function (Deadline: November 23)

Ryan Scomazzon

Mechanical Tasks:

- Create the first working camshaft fretting mechanism (Deadline: November 14)
- Modify colour reading mechanism for more accuracy (Deadline: November: 22)

Software Tasks:

- Create C++ paper image generation code so testing of the other features can begin. (Deadline: November 20)
- Create the colour reading code. (Deadline: November 22)

Deadline changes:

Fretting mechanism prototype deadline extended from November 10 to 14. The group realized that the current fretting design was not optimal and could not be ready for the formal presentation. The TA requested to see it the following week.

Conclusions

The SEV3N Nation Army Robot, in general, is successful and it meets all our criteria with the constraints given. Initially, it required considerable time management to ensure that the project was ambitious enough to test the skills of the team without going overboard. However, as the robot slowly took shape, it is evident that although our criteria and constraints are realistic, the sound produced by the robot is not the same as if a person played it. Essentially, the robot can play one string while changing the sound with the help of the frets which is what the essence of our criteria is. Additionally, it can also read songs that use just one string and the ten frets defined in our colour reading program. However, ten became seven after the team realised that the colour sensor only read 7 colours. Problems such as this arose throughout the design of the robot, but the basic criteria were always met.

After the tedious trial and error process, the robot's mechanical system satisfies the requirements we instilled. First, the colour sensor reads the strip of paper as it is fed through the system. The colours are in pairs of 2, with the first colour being the time interval between each pluck and the second colour being the fret being pressed down at that time. After the song is stored in the arrays, the fretting mechanism moves to the first fret in the fret array at intervals of 30° to press down on the specified fret. Then, after the specified time in the time array, the strumming mechanical system completes its first 45° rotation to pluck the string. This process repeats until the entire song is played and the "Song Complete" string is displayed on the EV3 brick.

Recommendations

Mechanical

The mechanical design for this project involves many iterations through the design and prototype process, so as a result, there are numerous changes which can be implemented if this project is redone.

1. Firstly, the mounting system relies on the assumption that 3D-printing was easily accessible and quick. Unfortunately, the Rapid Prototyping Lab's printers are always being used so there are little to no opportunities to print parts. Additionally, WATiMake requires training to 3D-print so it is also not a viable option for a time-constrained project.

2. If 3D-printing is not used, though, the Tetrix base could still be changed drastically. As of now, the base is a freestanding structure which does not connect to the guitar. However, this is not ideal because if the guitar or the base is moved then the alignment is ruined, and the fretting mechanism will not work. To fix this problem, the base could be extended to have four standing legs like a table instead of being one-side heavy. This would improve the stability of the fretting mechanism while also maintaining a high level of precision. In addition to this, there are multiple ways to attach the Tetrix base to the robot without ruining the guitar. For example, a structure like the white 3D-printed model in Figure 19 but with Tetrix pieces perpendicular to the neck of the guitar would be sufficient to lock the mechanism in place. This is because if you rotate the mount in any direction, it will automatically lock its location unless you rotate it in the other direction. However, since the structure would be on the ground, the friction will ensure that it will not move without considerable force.

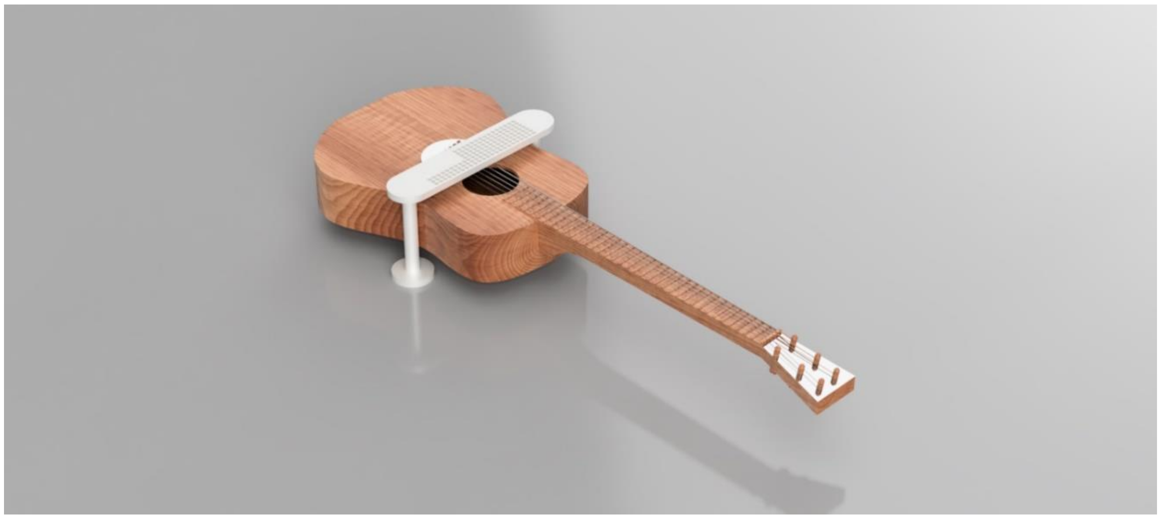


Figure 19: Strumming mount

3. The strumming mechanism is something that is incredibly simple but also difficult to get right. If it had to be done again, the first change would be to use a different piece to pluck the string. Ideally, something flexible such as TPU (Thermoplastic Polyurethane, a flexible 3D-printer filament) will work because it can bend while still being stiff enough to create a satisfactory sound.

4. The fretting mechanism is something that took as much time to plan as it did to build. As a result, it works well considering how complicated it is relative to every other mechanical part. However, this also means that there are multiple ways to do fretting which can produce better results. For example, it is possible to create a simpler solution by simply relying on rotational motion to press down on the frets. This makes aligning the frets an effortless task, but the camshaft idea avoids the other problems that arise due to the simplicity of this alternate solution.

5. Moreover, if custom 3D-printed parts are used to press on the strings instead of the double cross block pieces, the sound created by the guitar will be significantly better. Slots to account for the use of the elastic bands and for the cylindricity of the strings will also fix many of our alignment problems, but the testing phase for the implementation of this addition requires more time commitment towards something that doesn’t have a role in the main constraints of the project.

Software

1. The software driving the fretting mechanism is simple, and robust. The code can be made more complicated to account for smaller errors. This is especially important in longer songs, where the opportunity for error grows, and small errors accumulate. Firstly, the point at which the shaft applies pressure onto the fret can be made more accurate, and sequentially, can be made faster. To do this, errors in the fretting code can be accounted for, coupled with a PID controller, to smoothly alleviate accumulated errors.

2. Through testing, the turning of the camshaft both clockwise, and counterclockwise, causes problems. One of the flaws of the fretting system is that the elastics keeping the double cross blocks off the strings act like a ratchet that prevents the cam shaft from rotating in the reverse direction. Instead, the robot only moves the camshaft in one direction. This is problematic with faster paced songs, where the camshaft is not necessarily moving the shortest distance to press the appropriate fret. A full redesign is likely not required to solve this problem, but it will require some considerable modifications. With these changes, the largest possible angle of travel will go from 330 degrees clockwise, to 180 degrees in either direction.

3. Instead of resetting the fretting system after every pluck, the axle should only turn enough to reach the next fret. This will increase the speed with which consecutive frets are reached and will allow the robot to play songs at a faster pace than is currently possible. It is not possible to do this within the time frame without the motor encoder causing issues, but it is an upgrade to be considered if the project were to be done again.

References

[1] University of Waterloo, “Territorial Acknowledgement,” University of Waterloo, [Online]. Available: https://uwaterloo.ca/indigenous/engagement-knowledge-building/territorial-acknowledgement. [Accessed 05 December 2022].

[2] D. Barr, “GitHub,” 2018. [Online]. Available: https://github.com/OneLoneCoder/Javidx9/blob/master/PixelGameEngine/SmallerProjects/OneLoneCoder_PGE_MIDI.cpp. [Accessed 5 November 2022].

[3] S. Barrett, “GitHub,” 2017. [Online]. Available: https://github.com/nothings/stb . [Accessed 10 November 2022].

Appendix A

```
const int STRUMMING_POWER = 100;
const int FRET_SPEED = -100;
const int TIME_UNIT = 300;
const int COLOUR_PORT = S1;
const int ULTRASONIC_PORT = S2;
const int ARR_LENGTH = 100;
const int FULL_ROTATION = 360;
const int FRET_CLICKS = 30;
const int FRET_TIMING = 100;
const int PLUCK_ANGLE = 45;
const int SHUTDOWN_WAIT = 2000;
const int COLOURPORT = S1;
const int MOTORPORT = motorC;
const int USPORT = S2;
const int WAITCOLOUR = 500;
const int WAITWHITE = 200;
const int MOTORSPEED = -5;
const int SENSORDIST = 11;
const int REPEATTIME = 5;
const int REPEATS = 10;
const int colourValues[8] =
      {
            0,////empty 0
            0,//black 1
            2,//blue 2
            3,//green 3
            5,//yellow 4
            7,//red 5
            0,//white 6
            10//brown 7
      };
const int Remap[11] = {0,1,2,3,4,6,6,8,8,9,10}; //reverse of c++ code
//Declaring variables
```

```
bool startAllTasks(int ultrasonic, int colour)
{
      SensorType[ultrasonic] = sensorEV3_Ultrasonic;
      wait1Msec(50);
      SensorType[colour] = sensorEV3_Color;
      wait1Msec(50);
      SensorMode[colour] = modeEV3Color_Color;
      wait1Msec(100);
      //Configure sensors

      nMotorEncoder[motorA] = nMotorEncoder[motorD] = 0;
      //Reset encoders

      return true;
}

void pluck(int direction)
{
      motor[motorA] = direction * STRUMMING_POWER;
      //Set power according to motor direction

      nMotorEncoder[motorA] = 0;

      while(abs(nMotorEncoder[motorA]) < PLUCK_ANGLE)
      {}

      motor[motorA] = 0;
}

void setAndPlayFret(int fret, int time, int direction, int count)
{
      int clicksForFret = FRET_CLICKS * (fret - 1);
      int clicksToRotate = FULL_ROTATION - clicksForFret;
      // Calculate clicks to travel to fret

      time1[T1] = 0;

      motor[motorD] = FRET_SPEED;
      while(abs(nMotorEncoder[motorD]) % 360 < clicksToRotate)
      {}
      motor[motorD] = 0;

      while(time1[T1] < time * TIME_UNIT - FRET_TIMING)
      {}
      //Wait for fret time

      pluck(direction);

      wait1Msec(FRET_TIMING);
```

```
	motor[motorD] = FRET_SPEED;
	while(abs(nMotorEncoder[motorD]) < 360*count)
	{}
	motor[motorD] = 0;
	//Reset fretting system
}
int playNotes(int *fret, int *hold, int length)
{
	int direction = 1;
	int count = 1;

	for(int note = 0; note < length; note++)
	{
		setAndPlayFret(fret[note], hold[note], direction, count);

		direction *= -1;
		count++;
	}

	return direction;
}
bool paperPresent() //Checks if there still is paper to read
{
	return SensorValue[USPORT] < SENSORDIST || SensorValue[USPORT] == 255;
}

//returns the first colour that is read repeatedly over a set interval rather
//than that is read right away. This is to account for the inaccurate sensor.
int getRepeatRead()
{
	//initialize array with placeholder values
	int values[REPEATS]; //Stores consecutive loops
	for (int i = 0; i < REPEATS; i++)
	{
		values[i] = -1;
	}

	bool Loop = true;
	//Loop until definite colour is found
	while (Loop)
	{
		Loop = false;
		//Check if enough of a colour was read
		for (int i = 0; i < REPEATS; i++)
		{
			if (i!=-1 && i > 0 && values[i] != values[i-1])
			{
				Loop = true;
```

```
                        }
                }
        //Shift values down array and read new colour
                for (int i = 0; i < REPEATS; i++)
                {
                        if(i < REPEATS - 1)
                        {
                                values[i] = values[i+1];
                        }



                        else
                        {
                                values[i] = SensorValue[COLOURPORT];
                        }
                }

                //If we didnt read a full array yet
                if (values[0] == -1)
                {
                        Loop = true;
                }

                wait1Msec(REPEATTIME);
        }

        //Once array is full of one colour return that colour
        return values[0];
}

int readSheet(int * Times, int * Frets)
{
        // start to read in.

        int Notes = 0;
        bool timeRead = false;

        motor[MOTORPORT] = MOTORSPEED;

        //Keep reading if there is still paper to read
        while(paperPresent())
        {
                //Wait until there is a colour
                while (getRepeatRead() == (int)colorWhite &&
                        paperPresent())
                {}

                displayString(5, "Not White");
```

```
            //Wait to skip inaccuracy of the sensor changes
            wait1Msec(WAITCOLOUR);

            //read
            int value = getRepeatRead();

            displayString(5, "%d         ", value);

            //Choose to read into time or notes array
            if(!timeRead)
            {
                  Times[Notes] = Remap[colourValues[value]];
                  timeRead = true;
            }

            else
            {
                  Frets[Notes] = colourValues[value];
                  timeRead = false;
                  Notes++;
            }

            Wait until this colour is done.
            while (getRepeatRead() != (int)colorWhite &&
                  paperPresent())
            {}

            displayString(5, "White     ");

            wait1Msec(WAITWHITE);
      }

      motor[MOTORPORT] = 0;

      return Notes;
}

bool endAllTasks(int fret_start, int strum_start, int direction)
{
      motor[motorA] = motor[motorD] = 0;

      const int CURRENT_CLICKS = nMotorEncoder[motorD];
      const int OFFSET = CURRENT_CLICKS % 360;
      const int ADJUSTMENT = 360 - OFFSET + fret_start;

      nMotorEncoder[motorD] = 0;

      motor[motorD] = FRET_SPEED;
      while(abs(nMotorEncoder[motorD]) < ADJUSTMENT)
```

```
	{}
	motor[motorD] = 0;

	motor[motorA] = -1 * direction * STRUMMING_POWER;
	while(abs(nMotorEncoder[motorA]) < strum_start)
	{}
	motor[motorA] = 0;

	return true;
}

task main()
{
	bool startup = startAllTasks(ULTRASONIC_PORT, COLOUR_PORT);

	if(startup)
	{
		displayBigTextLine(5, "Startup successful");
	}

	int fret[ARR_LENGTH];
	int timing[ARR_LENGTH];

	while (!getButtonPress(buttonEnter))
	{}

	int const NOTE_COUNT = readSheet(timing, fret);
	displayBigTextLine(5, "Song loaded");

	int direction = playNotes(fret, timing, NOTE_COUNT);

	displayBigTextLine(5, "Song finished!");

	wait1Msec(SHUTDOWN_WAIT);

	bool shutdown = endAllTasks(0,0, direction);

	if(shutdown)
	{
		displayBigTextLine(5, "Shutdown successful");
	}
}
```

//**C++ code –----------**

```
//The following code is based on work from several sources
```

```
//MIDI reading basis code from David Barr, Copyright 2018-2020
//OneLoneCoder.com
//https://github.com/OneLoneCoder/Javidx9/blob/master/PixelGameEngine/Smaller
//Projects/OneLoneCoder_PGE_MIDI.cpp

//stb image library from Sean Barrett, Copyright (c) 2017
//- MIT open source license
//https://github.com/nothings/stb


//This next portion of the code sorts through the midi file, skips irrelevant
//data, and stores it as a more readable format
//This is NOT original code.
//It was written by David Barr, Copyright 2018-2020 OneLoneCoder.com



//-----------------------------------------------------------------------
#define OLC_PGE_APPLICATION

#include <fstream>
#include <array>
#include <vector>
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

struct MidiEvent
{
      enum class Type
      {
            NoteOff,
            NoteOn,
            Other
      } event;

      uint8_t nKey = 0;
      uint8_t nVelocity = 0;
      uint32_t nDeltaTick = 0;
};


struct MidiNote
{
      uint8_t nKey = 0;
      uint8_t nVelocity = 0;
```

```
	uint32_t nStartTime = 0;
	uint32_t nDuration = 0;
};

struct MidiTrack
{
	std::string sName;
	std::string sInstrument;
	std::vector<MidiEvent> vecEvents;
	std::vector<MidiNote> vecNotes;
	uint8_t nMaxNote = 64;
	uint8_t nMinNote = 64;
};
class MidiFile
{
public:
	enum EventName : uint8_t
	{
		VoiceNoteOff = 0x80,
		VoiceNoteOn = 0x90,
		VoiceAftertouch = 0xA0,
		VoiceControlChange = 0xB0,
		VoiceProgramChange = 0xC0,
		VoiceChannelPressure = 0xD0,
		VoicePitchBend = 0xE0,
		SystemExclusive = 0xF0,
	};

	enum MetaEventName : uint8_t
	{
		MetaSequence = 0x00,
		MetaText = 0x01,
		MetaCopyright = 0x02,
		MetaTrackName = 0x03,
		MetaInstrumentName = 0x04,
		MetaLyrics = 0x05,
		MetaMarker = 0x06,
		MetaCuePoint = 0x07,
		MetaChannelPrefix = 0x20,
		MetaEndOfTrack = 0x2F,
		MetaSetTempo = 0x51,
		MetaSMPTEOffset = 0x54,
		MetaTimeSignature = 0x58,
		MetaKeySignature = 0x59,
		MetaSequencerSpecific = 0x7F,
	};
public:
	MidiFile()
	{
	}
```

```
MidiFile(const std::string& sFileName)
{
      ParseFile(sFileName);
}

void Clear()
{

}

bool ParseFile(const std::string& sFileName)
{
      // Open the MIDI File as a stream
      std::ifstream ifs;
      ifs.open(sFileName, std::fstream::in | std::ios::binary);
      if (!ifs.is_open())
            return false;


      // Helper Utilities ====================

      // Swaps byte order of 32-bit integer
      auto Swap32 = [](uint32_t n)
      {
            return (((n >> 24) & 0xff) | ((n << 8) & 0xff0000) | ((n >>
                   8) & 0xff00) | ((n << 24) & 0xff000000));
      };

      // Swaps byte order of 16-bit integer
      auto Swap16 = [](uint16_t n)
      {
            return ((n >> 8) | (n << 8));
      };

      // Reads nLength bytes form file stream, and constructs a text
      // string
      auto ReadString = [&ifs](uint32_t nLength)
      {
            std::string s;
            for (uint32_t i = 0; i < nLength; i++) s += ifs.get();
            return s;
      };

      // Reads a compressed MIDI value. This can be up to 32 bits long.
      // Essentially if the first byte, first
      // bit is set to 1, that indicates that the next byte is
      required
      //to construct the full word. Only
```

```
// the bottom 7 bits of each byte are used to construct the final
// word value. Each successive byte
// that has MSB set, indicates a further byte needs to be read.
auto ReadValue = [&ifs]()
{
      uint32_t nValue = 0;
      uint8_t nByte = 0;

      // Read byte
      nValue = ifs.get();

      // Check MSB, if set, more bytes need reading
      if (nValue & 0x80)
      {
            // Extract bottom 7 bits of read byte
            nValue &= 0x7F;
            do
            {
                  // Read next byte
                  nByte = ifs.get();

                  // Construct value by setting bottom 7 bits,
                  // then shifting 7 bits
                  nValue = (nValue << 7) | (nByte & 0x7F);
            }
            while (nByte & 0x80); // Loop whilst read byte MSB
            is //1
      }

      // Return final construction (always 32-bit unsigned
      // integer internally)
      return nValue;
};

uint32_t n32 = 0;
uint16_t n16 = 0;

// Read MIDI Header (Fixed Size)
ifs.read((char*)&n32, sizeof(uint32_t));
uint32_t nFileID = Swap32(n32);
ifs.read((char*)&n32, sizeof(uint32_t));
uint32_t nHeaderLength = Swap32(n32);
ifs.read((char*)&n16, sizeof(uint16_t));
uint16_t nFormat = Swap16(n16);
ifs.read((char*)&n16, sizeof(uint16_t));
uint16_t nTrackChunks = Swap16(n16);
ifs.read((char*)&n16, sizeof(uint16_t));
uint16_t nDivision = Swap16(n16);
```

```
for (uint16_t nChunk = 0; nChunk < nTrackChunks; nChunk++)
{
      std::cout << "===== NEW TRACK" << std::endl;
      // Read Track Header
      ifs.read((char*)&n32, sizeof(uint32_t));
      uint32_t nTrackID = Swap32(n32);
      ifs.read((char*)&n32, sizeof(uint32_t));
      uint32_t nTrackLength = Swap32(n32);

      bool bEndOfTrack = false;

      vecTracks.push_back(MidiTrack());

      uint32_t nWallTime = 0;

      uint8_t nPreviousStatus = 0;

      while (!ifs.eof() && !bEndOfTrack)
      {
            // Fundamentally all MIDI Events contain a timecode,
            // and a status byte*
            uint32_t nStatusTimeDelta = 0;
            uint8_t nStatus = 0;


            // Read Timecode from MIDI stream. This could be
            // variable in length
            // and is the delta in "ticks" from the previous
            // event. Of course this value
            // could be 0 if two events happen simultaneously.
            nStatusTimeDelta = ReadValue();

            // Read first byte of message, this could be the
            // status byte, or it could not...
            nStatus = ifs.get();

            // All MIDI Status events have the MSB set. The data
            // within a standard MIDI event
            // does not. A crude yet utilised form of
            compression
            // is to omit sending status
            // bytes if the following sequence of events all
            // refer to the same MIDI Status.
            // This is called MIDI Running Status, and is
            // essential to succesful decoding of
            // MIDI streams and files.
            //
            // If the MSB of the read byte was not set, and on
            // the whole we were expecting a
            // status byte, then Running Status is in effect, so
            // we refer to the previous
            // confirmed status byte.
            if (nStatus < 0x80)
            {
                  // MIDI Running Status is happening, so refer
                  // to previous valid MIDI Status byte
                  nStatus = nPreviousStatus;

                  // We had to read the byte to assess if MIDI
                  // Running Status is in effect. But!
                  // that read removed the byte form the stream,
                  // and that will desync all of the
                  // following code because normally we would
                  // have read a status byte, but instead
                  // we have read the data contained within a
                  // MIDI message. The simple solution is
                  // to put the byte back :P
                  ifs.seekg(-1, std::ios_base::cur);
            }



            if ((nStatus & 0xF0) == EventName::VoiceNoteOff)
            {
                  nPreviousStatus = nStatus;
                  uint8_t nChannel = nStatus & 0x0F;
                  uint8_t nNoteID = ifs.get();
                  uint8_t nNoteVelocity = ifs.get();
                  vecTracks[nChunk].vecEvents.push_back({
                         MidiEvent::Type::NoteOff, nNoteID,
                         nNoteVelocity, nStatusTimeDelta });
            }

            else if ((nStatus & 0xF0) == EventName::VoiceNoteOn)
            {
                  nPreviousStatus = nStatus;
                  uint8_t nChannel = nStatus & 0x0F;
                  uint8_t nNoteID = ifs.get();
                  uint8_t nNoteVelocity = ifs.get();
                  if(nNoteVelocity == 0)
                        vecTracks[nChunk].vecEvents.push_back({
                              MidiEvent::Type::NoteOff, nNoteID,
                              nNoteVelocity, nStatusTimeDelta });
                  else
                        vecTracks[nChunk].vecEvents.push_back({
                              MidiEvent::Type::NoteOn, nNoteID,
                              nNoteVelocity, nStatusTimeDelta });
            }
```

```
else if ((nStatus & 0xF0) ==
      EventName::VoiceAftertouch)
{
      nPreviousStatus = nStatus;
      uint8_t nChannel = nStatus & 0x0F;
      uint8_t nNoteID = ifs.get();
      uint8_t nNoteVelocity = ifs.get();
      vecTracks[nChunk].vecEvents.push_back({
            MidiEvent::Type::Other });
}

else if ((nStatus & 0xF0) ==
      EventName::VoiceControlChange)
{
      nPreviousStatus = nStatus;
      uint8_t nChannel = nStatus & 0x0F;
      uint8_t nControlID = ifs.get();
      uint8_t nControlValue = ifs.get();
      vecTracks[nChunk].vecEvents.push_back({
            MidiEvent::Type::Other });
}

else if ((nStatus & 0xF0) ==
      EventName::VoiceProgramChange)
{
      nPreviousStatus = nStatus;
      uint8_t nChannel = nStatus & 0x0F;
      uint8_t nProgramID = ifs.get();

      vecTracks[nChunk].vecEvents.push_back({
            MidiEvent::Type::Other });
}

else if ((nStatus & 0xF0) ==
      EventName::VoiceChannelPressure)
{
      nPreviousStatus = nStatus;
      uint8_t nChannel = nStatus & 0x0F;
      uint8_t nChannelPressure = ifs.get();
      vecTracks[nChunk].vecEvents.push_back({
            MidiEvent::Type::Other });
}

else if ((nStatus & 0xF0) ==
      EventName::VoicePitchBend)
{
      nPreviousStatus = nStatus;
      uint8_t nChannel = nStatus & 0x0F;
```

```
			uint8_t nLS7B = ifs.get();
			uint8_t nMS7B = ifs.get();
			vecTracks[nChunk].vecEvents.push_back({
				MidiEvent::Type::Other });

		}

		else if ((nStatus & 0xF0) ==
			EventName::SystemExclusive)
		{
			nPreviousStatus = 0;

			if (nStatus == 0xFF)
			{
				// Meta Message
				uint8_t nType = ifs.get();
				uint8_t nLength = ReadValue();

				switch (nType)
				{
				case MetaSequence:
					std::cout << "Sequence Number: "
					<< ifs.get() << ifs.get()
					<< std::endl;
					break;
				case MetaText:
					std::cout << "Text: " <<
						ReadString(nLength) <<
						std::endl;
					break;
				case MetaCopyright:
					std::cout << "Copyright: " <<
						ReadString(nLength) <<
						std::endl;
					break;
				case MetaTrackName:
					vecTracks[nChunk].sName =
						ReadString(nLength);
					std::cout << "Track Name: " <<
						vecTracks[nChunk].sName <<
						std::endl;

					break;
				case MetaInstrumentName:
					vecTracks[nChunk].sInstrument =
						ReadString(nLength);
						std::cout << "Instrument
						Name:
            "<< vecTracks[nChunk].sInstru
            ment
            << std::endl;
      break;
case MetaLyrics:
      std::cout << "Lyrics: " <<
            ReadString(nLength) <<
            std::endl;
      break;
case MetaMarker:
      std::cout << "Marker: " <<
            ReadString(nLength) <<
            std::endl;
      break;
case MetaCuePoint:
      std::cout << "Cue: " <<
            ReadString(nLength) <<
            std::endl;
      break;
case MetaChannelPrefix:
      std::cout << "Prefix: " <<
            ifs.get() << std::endl;
      break;
case MetaEndOfTrack:
      bEndOfTrack = true;
      break;
case MetaSetTempo:
      // Tempo is in microseconds per
      //quarter note
      if (m_nTempo == 0)
      {
            (m_nTempo |= (ifs.get() <<
                  16));
            (m_nTempo |= (ifs.get() <<
                  8));
            (m_nTempo |= (ifs.get() <<
                  0));
            m_nBPM = (60000000 /
                  m_nTempo);
            std::cout << "Tempo: " <<
                  m_nTempo << " (" <<
                  m_nBPM << "bpm)" <<
            std::endl;
      }
      break;
case MetaSMPTEOffset:
      std::cout << "SMPTE: H:" <<
            ifs.get() << " M:" <<
            ifs.get() << " S:" <<
            ifs.get() << " FR:" <<
```

```
                  ifs.get() << " FF:" <<
                  ifs.get() << std::endl;
            break;
      case MetaTimeSignature:
            std::cout << "Time Signature: " <<
                  ifs.get() << "/" << (2 <<
                  ifs.get()) << std::endl;
            std::cout << "ClocksPerTick: " <<
                  ifs.get() << std::endl;

            // A MIDI "Beat" is 24 ticks, so
            //specify how many 32nd notes
            //constitute a beat
            std::cout << "32per24Clocks: " <<
                   ifs.get() << std::endl;
            break;
      case MetaKeySignature:
            std::cout << "Key Signature: " <<
                  ifs.get() << std::endl;
            std::cout << "Minor Key: " <<
                  ifs.get() << std::endl;
            break;
      case MetaSequencerSpecific:
                  std::cout << "Sequencer
                  Specific: "
                  << ReadString(nLength) <<
                  std::endl;
            break;
      default:
            std::cout << "Unrecognised
                  MetaEvent: " << nType <<
                  std::endl;
      }
}

if (nStatus == 0xF0)
{
      // System Exclusive Message Begin
      std::cout << "System Exclusive Begin: "
            << ReadString(ReadValue())  <<
            std::endl;
}

if (nStatus == 0xF7)
{
      // System Exclusive Message Begin
      std::cout << "System Exclusive End: " <<
            ReadString(ReadValue()) <<
            std::endl;
}
```

```
                }
                else
                {
                      std::cout << "Unrecognised Status Byte: " <<
                            nStatus << std::endl;
                }
          }
    }


    // Convert Time Events to Notes
    for (auto& track : vecTracks)
    {
          uint32_t nWallTime = 0;

          std::list<MidiNote> listNotesBeingProcessed;

          for (auto& event : track.vecEvents)
          {
                nWallTime += event.nDeltaTick;

                if (event.event == MidiEvent::Type::NoteOn)
                {
                      // New Note

                listNotesBeingProcessed.push_back({event.nKey,
                            event.nVelocity, nWallTime, 0 });
                }

                if (event.event == MidiEvent::Type::NoteOff)
                {
                      auto note =
                            std::find_if(listNotesBeingProcessed.begi
                            n(), listNotesBeingProcessed.end(),
                            [&](const MidiNote& n) { return n.nKey
                      ==
                            event.nKey; });
                      if (note != listNotesBeingProcessed.end())
                      {
                            note->nDuration = nWallTime -
                                  note->nStartTime;
                            track.vecNotes.push_back(*note);

                            track.nMinNote =
                      std::min(track.nMinNote,
                                  note->nKey);
                            track.nMaxNote =
                      std::max(track.nMaxNote,
                                  note->nKey);
```

```
                                    listNotesBeingProcessed.erase(note);
                              }
                        }
                  }
            }

            return true;
      }

public:
      std::vector<MidiTrack> vecTracks;
      uint32_t m_nTempo = 0;
      uint32_t m_nBPM = 0;

};
```

```
//------------------------------------------------------------------------


//The code below this point is all either original or very heavily modified


#include <sstream> //Included for use of sample code
#include <cstdlib>
```

```
//The following are imported for use with external library: stb image
//implementation
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"

#define STBI_MSC_SECURE_CRT
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"

//Setup constants
const int STROFFSET = 52; //String Offset - represents the open low E string
                              //in midi format
//Output setup
const int WIDTH = 400;
const int HEIGHT = 400;
#define CHANNEL_NUM 3
const int SQUARESIZE = 40;
const int SPACING = 20;
const int MAXNOTES = 100;
const int MAXFRET = 12;
const int MAXTIME = 10;
const int ROWSQUARES = (WIDTH - SQUARESIZE + SPACING)/(SQUARESIZE+SPACING);
const int TIMEFACTOR = 24;
const string fileName = "TestMidi5.mid";

//Color for each int 0-10
//Colours full range
const int FULLRANGE = false; //changes encoding to trade off range for
                              //accuracy

//Colours accuracy
const int Colours[11][3] =
{
      {0,0,0},//0 -----
      {255,255,255},//1
      {0,100,255},//2 // ----
      {0,200,0},//3 // ----
      {255,255,255},//4
      {255,255,0},//5 // ----
      {255,255,255},//6
      {255,0,0},//7 // -----
      {255,255,255},//8
      {255,255,255},//9
      {96,64,0}//10 // ----
};
const int Remap[] = {0,1,2,3,4,5,5,7,7,9,10}; //allows for the time and note
            //arrays to have the same colour correspond to different
            //integers.
```

```
void readData(MidiFile & midi, int squareColour[], int squareTime[], bool
      squareExists[])
{
      //Go through all notes
      //Print to debug file
      //Populate square arrays
      cout << endl << endl << "Custom Output:" << endl;
      int noteCount = 0;
      bool doneTrack = false;
      for (auto& track : midi.vecTracks)
            {
                  if (doneTrack)
                  {
                        break;
                  }
                  if (!track.vecNotes.empty())
                  {
                        cout << endl << track.sName << endl;
                        int lastStartTime = 0;
                        for (auto& note : track.vecNotes)
                        {

                              noteCount++;
                              char asciiKey = note.nKey;
                              //Convert ascii value to int
                              int key = int(asciiKey);

                              //Convert to location on string
                              key -= STROFFSET;

                              //Stop from progressing to future tracks
                              doneTrack = true;

                              //Populate square arrays
                              squareColour[noteCount-1] = key;
                              int timeValue = (note.nStartTime -
                                    lastStartTime)/TIMEFACTOR;
                              if (FULLRANGE == false)
                              {
                                    timeValue = Remap[timeValue];
                              }
                              squareTime[noteCount-1] = timeValue;
                              squareExists[noteCount-1] = true;
                              lastStartTime = note.nStartTime;

                        }
                  }
            }

}


void createFile(int squareColour[], int squareTime[], bool squareExists[])
{
      //Output squares
      uint8_t* pixels = new uint8_t[WIDTH * HEIGHT * CHANNEL_NUM];

       int index = 0;
       for (int y = 0; y < HEIGHT; y++)
       {
        for (int x = 0; x < WIDTH; ++x)
        {
         int r = 255;
         int g = 255;
         int b = 255;

            //Check if a timing square is here
            for (int square = 0; square < MAXNOTES; square++)
            {
                  int row = square * 2 / ROWSQUARES;
                  int rowPos = SPACING + (SQUARESIZE+SPACING)*row;

                  int pos = SPACING + 2*square*(SQUARESIZE+SPACING) - row *
                        WIDTH + 2*row * SPACING;
                  if(x >= pos && x < pos + SQUARESIZE
                        && y >= rowPos && y < rowPos + SQUARESIZE &&
                        squareExists[square])
                  {
                        r = Colours[squareTime[square]][0];
                        g = Colours[squareTime[square]][1];
                        b = Colours[squareTime[square]][2];
                  }
            }

            //Check if a note square is here
            for (int square = 0; square < MAXNOTES; square++)
            {
                  int row = (square * 2 + 1) / ROWSQUARES;
                  int rowPos = SPACING + (SQUARESIZE+SPACING)*row;

                  int pos = 2*SPACING + SQUARESIZE +
                        2*square*(SQUARESIZE+SPACING) - row * WIDTH + 2*row
                  *
                        SPACING;
                  if(x >= pos && x < pos + SQUARESIZE
                        && y >= rowPos && y < rowPos + SQUARESIZE &&
                        squareExists[square])
                  {
                        r = Colours[squareColour[square]][0];
                        g = Colours[squareColour[square]][1];
                        b = Colours[squareColour[square]][2];
                  }
            }

         pixels[index++] = r;
         pixels[index++] = g;
         pixels[index++] = b;
        }
       }

      //write image with stb image library
      // if CHANNEL_NUM is 4, you can use alpha channel in png
      stbi_write_png("ImageOutput.png", WIDTH, HEIGHT, CHANNEL_NUM, pixels,
            WIDTH * CHANNEL_NUM);

}


int main()
{
      //Setup square arrays
      int squareColour[MAXNOTES] = {};
      int squareTime[MAXNOTES] = {};
      bool squareExists[MAXNOTES] = {};



      MidiFile midi;
      bool success = false;

      //Read in midi file
      success = midi.ParseFile(fileName); //insert name of file here

      if (!success)
      {
            cout << "Failed to load file" << endl;
            return EXIT_FAILURE;
      }
      else
      {
            cout << "Loaded midi file" << endl;
      }

      //Open output file for logging debug values
```

```
	ofstream fileout("LogOutput.txt");

	if (!fileout)
	{
		cout << "Failed to load file" << endl;
		return EXIT_FAILURE;
	}
	else
	{
		cout << "Loaded output file" << endl;
	}

	//Read in data
	readData(midi, squareColour, squareTime, squareExists);

	//Output to file
	createFile(squareColour, squareTime, squareExists);

	//export values to a file for debugging.
	for (int i = 0; i < MAXNOTES; i++)
	{
		if (squareExists[i])
		{
			fileout << squareTime[i] << endl << squareColour[i] <<
				endl;
		 }

	 }

	fileout.close();

	cout << "Finished";

	return EXIT_SUCCESS;
}
```