

SVEUČILIŠTE U ZAGREBU  
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1372

**Modeling and simulation of a 3D  
musical instrument**

**Modeliranje i simulacija 3D  
glazbenog instrumenta**

Luka Kunić

Zagreb, lipanj 2017.

Zagreb, 3 March 2017

## MASTER THESIS ASSIGNMENT No. 1372

Student: **Luka Kunić (0036468102)**  
Study: Computing  
Profile: Computer Science

Title: **Modeling and Simulation of 3D Musical Instrument**

Description:

Investigate techniques for modelling of musical instruments, particularly for a guitar model. Peruse the standard MIDI file format. Realize a three-dimensional model of a musical instrument. For the MIDI file format design a parser which will enable acoustic and visual simulation of a soundtrack. Implement a framework for analysis and comparison of the investigated models. Show examples of results. Evaluate the results and the implemented algorithms.

Implement the appropriate software solution. Use the game engine Unreal Engine for simulation of audio and visual functionality of the musical instruments models. Make the results of the thesis available online. Supply algorithms, source codes and results with appropriate explanations and documentation. Reference the used literature and acknowledge received help.

Issue date: 10 March 2017  
Submission date: 29 June 2017

Mentor:



---

Full Professor Željka Mihajlović, PhD

Committee Secretary:



---

Assistant Professor Tomislav Hrkać, PhD

Committee Chair:



---

Full Professor Siniša Srblić, PhD

Zagreb, 3. ožujka 2017.

## DIPLOMSKI ZADATAK br. 1372

Pristupnik: **Luka Kunić (0036468102)**  
Studij: Računarstvo  
Profil: Računarska znanost


Zadatak: **Modeliranje i simulacija 3D glazbenog instrumenta**

### Opis zadatka:

Proučiti načine modeliranja glazbenih instrumenata, a posebice obratiti pažnju na model gitare. Proučiti standard zapisa datoteka MIDI. Realizirati trodimenzionalan model glazbenog instrumenta. Za zapis MIDI napraviti program za raščlambu (engl. parser) glazbenog zapisa MIDI na temelju kojega će biti ostvarena simulacija puštanja glazbenih uzoraka pripadnog notnog zapisa. Na različitim primjerima prikazati ostvarene rezultate. Načiniti ocjenu rezultata i implementiranih algoritama. Izraditi odgovarajući programski proizvod. Koristiti grafički pogon Unreal Engine za ostvarivanje funkcionalnosti simulacije sviranja izrađenog 3D modela. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Zadatak uručen pristupniku: 10. ožujka 2017.  
Rok za predaju rada: 29. lipnja 2017.

Mentor:

  
\_\_\_\_\_  
Prof. dr. sc. Željka Mihajlović

Djelovođa:

  
\_\_\_\_\_  
Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za  
diplomski rad profila:

  
\_\_\_\_\_  
Prof. dr. sc. Siniša Srbljić



# CONTENTS

<b>1. Introduction</b>	<b>1</b>
<b>2. Modeling the 3D bass guitar</b>	<b>3</b>
2.1. <i>Self-play</i> mechanisms . . . . .	4
2.1.1. Left-hand fretting mechanism . . . . .	4
2.1.2. Right-hand picks . . . . .	6
2.2. Strings . . . . .	8
2.3. Fretboard . . . . .	9
2.4. Exporting the model . . . . .	10
<b>3. Parsing MIDI files</b>	<b>11</b>
3.1. MIDI file structure . . . . .	11
3.2. Creating the MIDI files . . . . .	12
3.3. Extracting note data . . . . .	13
<b>4. Mapping the notes to fretboard positions</b>	<b>15</b>
4.1. Graph traversal algorithm . . . . .	16
4.1.1. Scoring . . . . .	17
4.1.2. Building the graph . . . . .	18
4.1.3. The algorithm . . . . .	18
4.1.4. Cleaning up the calculated sequence . . . . .	20
<b>5. Action timeline</b>	<b>21</b>
5.1. Actions for a bass guitar . . . . .	21
5.1.1. Pick actions . . . . .	22
5.1.2. Fretting finger actions . . . . .	24
5.1.3. String actions . . . . .	25
5.1.4. Audio actions . . . . .	26
5.2. Executing the actions . . . . .	27

<b>6. Results</b>	<b>28</b>
<b>7. Conclusion</b>	<b>31</b>
<b>Bibliography</b>	<b>32</b>

# 1. Introduction

Music has always been a major source of entertainment in our society. Throughout history, live concerts were used by musicians to display their skill to an audience. In the recent years, with the introduction of on-line media sharing services such as *YouTube*, music videos have become the go-to platform for promotion of new talent. This does not only apply to professional artists, but to amateur musicians as well.

The biggest challenge for aspiring musicians is learning the art of playing an instrument. This is a long and difficult process that can require years of hard work and dedication, which is why skilled musicians have always been highly valued members of the society. Another challenge is related to the acquisition of instruments, especially the more uncommon ones. Most instruments require complex designs and precise crafting in order to produce the correct notes and pleasant tones, which can make the crafting process quite expensive and time consuming. This thesis aims to tackle these two challenges, although not by inventing new ways of crafting and playing instruments in the physical world, but by moving the instruments into the virtual world.

A virtual environment provides two main benefits: programmability and design freedom. The programmability aspect allows the instrument to be interactive and self-playing, meaning that the animations and sounds are triggered automatically based on given input data. Design freedom comes from the fact that the instruments and all accompanying mechanisms will not need to actually work in the physical world. The main purpose would be visualization, so the design can instead be approached from a purely visual perspective, with focus on animations that can even be slightly overexaggerated to increase the interest.

The goal of this thesis is to develop an extensible framework for visualizing self-playing musical instruments. This task involves (1) designing an efficient process for modeling and animating the instruments, (2) creating a prototype model of an instrument that will be used for development and testing the framework, (3) importing the model into a suitable 3D environment that allows programming the required behavior, (4) finding a suitable data source which will provide input for the visualization, and

(5) developing a system that will guide the animations and sounds based on the given input data.

A bass guitar was chosen to be the instrument used in development. Firstly, the bass guitar is a string instrument so it shares similarities with many musical instruments of the same family, which means that the implementation can easily be adapted to other similar instruments. Another important consideration is the playstyle, which for the bass guitar is mostly based on playing single notes and arpeggios<sup>1</sup>, unlike the regular six-string guitar which is most often played by strumming chords. This simplifies the development of certain mechanisms involved in playing the instrument. Chapter 2 contains the description of how the bass and the accompanying mechanisms were modelled using Blender and then imported into Unreal Engine for further development.

MIDI files are used as input data for the system. Chapter 3 describes the process of parsing the MIDI data to extract the notes that will be played by the virtual instrument. Because of the way notes are distributed on the neck of a bass guitar, the given MIDI notes need to be mapped to the fretboard positions so that the fretting sequence is efficient and visually pleasing. This process is described in Chapter 4. Chapter 5 explains the timeline mechanism responsible for guiding the animations and sounds for the visualization. The final results of this work are presented in Chapter 6.

---

<sup>1</sup>Arpeggio is a form of a chord which is played as single notes in an ascending or descending order.



## 2. Modeling the 3D bass guitar

To create a self-playing virtual instrument, the 3D model of the instrument must first be created and animated. In the case of a bass guitar, this includes modeling the body of the instrument, creating and animating the strings, and adding fretting and picking mechanisms which will be used to press the strings onto the fretboard and pluck the strings as notes are played.

The body of the bass guitar, along with all the static accessories such as pickups, tuning heads and volume knobs, are all created manually and do not require a detailed description of the modeling process. The more interesting elements of the model are described in the following sections.



**Figure 2.1:** The bass guitar model with the picking and fretting mechanisms imported into Unreal Engine.

## 2.1. *Self-play mechanisms*

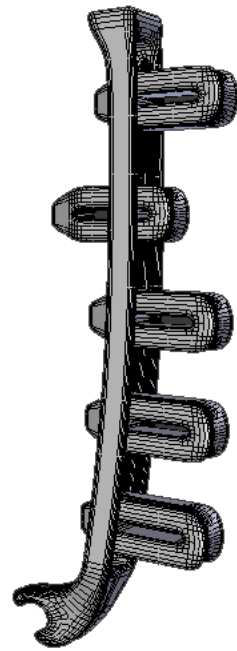
Although a musical instrument would generally be played by a musician, many mechanical engineers throughout history have attempted to build real robotic self-playing pianos, guitars, and other types of instruments [4, 9, 10]. These devices use complex mechanisms to produce sound, and designing such mechanisms poses a significant challenge when building robotic instruments. However, putting the instrument into a virtual environment allows a much easier implementation of such mechanisms, since there are no physical constraints limiting the design.

A self-playing bass guitar requires two such mechanisms. One mimics the left hand of a guitarist and is used to press strings onto the fretboard, while the other is used to pluck the strings, simulating movements of the right hand.

### 2.1.1. **Left-hand fretting mechanism**

To play notes on a guitar, the strings need to be pressed down onto corresponding frets. A mechanism for pressing strings must be robust enough to allow playing various note configurations efficiently, while maintaining simplicity and remaining visually interesting.

When designing such a mechanism, one can look at how real guitarists use their left-hand fingertips to press strings down on frets. The obvious idea would be to recreate a human hand and animate real finger behavior, which would give a realistic interpretation of the fretting hand. However, in order to properly simulate finger movement, this animation system would have to be very complex, yet at the same time highly constrained because of the physical limits of a human hand. The problem becomes apparent when looking at chords. The same chord can be part of multiple note sequences, and each sequence may require the chord to be played using slightly different finger positions, depending on the surrounding notes in the sequence. One must also consider the physical constraints of the fingers, such as the maximum fret span between the first and last finger, and the undesirable overlapping of fingers. Another problem rises from the fact that string spacing decreases from the body towards the head of the guitar, while the fret spacing grows larger towards the lower frets. This means that playing the same chord shape on various positions along the fretboard results in different finger spacing or positioning. While it may be plausible to develop a sufficiently robust and flexible system for a realistic fretting hand, this thesis does not further explore such an approach, and instead investigates a simpler alternative.



**Figure 2.2:** One fretting finger. The bottom of the finger attaches to the guide rail, allowing it to slide along the length of the neck. The pins can be spaced vertically or pressed down using the corresponding shape keys.

The designed fretting mechanism consists of several mechanical fretting fingers that slide along the neck of the guitar using a guide rail. The rail is positioned below the neck, with attach points on the back side of the neck. Each finger consists of a frame and a set of five pins, one pin per string. Each pin is used to press its corresponding string down onto the fretboard. The frame holds the pins and enables them to slide vertically, so that they can always stay aligned with the strings as the finger moves along the neck.

Shape keys are used to control pin movement in order to avoid creating an armature for each finger. Shape keys are animation mechanisms that are used to deform the mesh without the use of an armature. They store displacement information for the vertices of the string mesh, allowing interpolation between the initial vertex positions and the deformation. As shown by Figure 2.2, one shape key controls the spacing between pins, while the remaining shape keys are used for pressing each pin down towards the fretboard. These shape keys can be blended, which allows multiple pins to be pressed down at the same time. Since the vertex positions can be interpolated

between the initial position and the fully displaced position stored in the shape key, the interpolation value can be gradually increased or decreased, resulting in a smooth transition over time without the need for a baked animation.

This implementation offers several benefits over a realistic human hand model. The constraints for this system are much more relaxed because the fingers can move independently. This is especially true for the finger span constraint, as this system even allows two fingers to simultaneously press strings on the opposite sides of the fretboard. Also, one finger can easily press multiple strings on the same fret at the same time, which is not always a simple task for a real human hand. Furthermore, the animation system is very robust and flexible because there are no baked animations involved. All animations are created at runtime using shape keys, which results in a lot of freedom when determining the animation speed and blending the pin movements into a single flowing transition.

### 2.1.2. Right-hand picks

To pluck guitar strings, a musician would generally use a plastic plectrum or their fingers. A plectrum would usually produce a brighter sound compared to fingers, so the usage depends on the genre of music being played. Different fingerstyle techniques could also be considered, as classical finger picking will produce a significantly different sound to slapping or tapping the strings. The tone of the notes also depends on

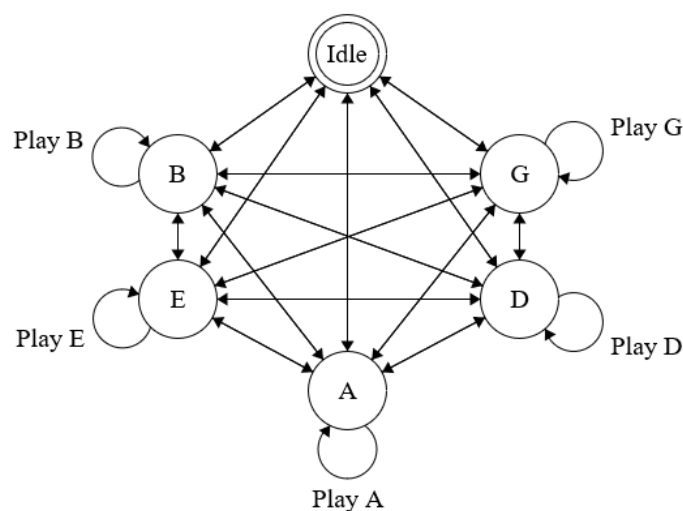


**Figure 2.3:** The five pick fingers used for plucking the strings. Each finger moves independently and can reach all five strings.

the location where the string is being plucked, because plucking a string in the middle would produce different amplitudes of certain harmonics than plucking the same string right next to the bridge. These considerations would be crucial when designing a physical picking mechanism, but are not as important in a virtual environment. The project described in this thesis does not aim to create a realistic simulation of a guitar, but to provide a visualization framework for musical instruments which only look realistic. The picks will not actually trigger string vibration, so they can be designed purely for the purpose of visual appeal.

Inspiration for the design was the way actual fingers move when plucking the strings using a classic two-finger technique. The mechanism consists of individual fingers attached to the guitar body just above the strings. Just like fingers on a human hand, each finger of the mechanism is built out of three joints that allow the finger to easily reach all five strings. To keep the animations reasonably paced, the finger count is expanded from two to five so that each finger may be used less frequently.

The animation system for a finger relies on a state machine with six states. The initial state corresponds to the idle pose of the finger, and the remaining five states correspond to the ready poses in front of each of the strings. In the idle pose, the finger is in a relaxed position above the top string. When a string needs to be plucked, the finger will transition from its current pose towards the ready pose in front of the string that needs to be plucked. After the finger is in the ready pose, the plucking animation for that string can be played. The plucking animation slides the tip of the finger behind the string, then quickly pulls the finger upwards before returning to the



**Figure 2.4:** The pick state machine. The states are labeled according to the root notes of the strings they represent – from top to bottom the root notes are B, E, A, D, and G.

ready pose. Since the final frame of the animation is the same as the first frame, the plucking animation can be seamlessly looped.

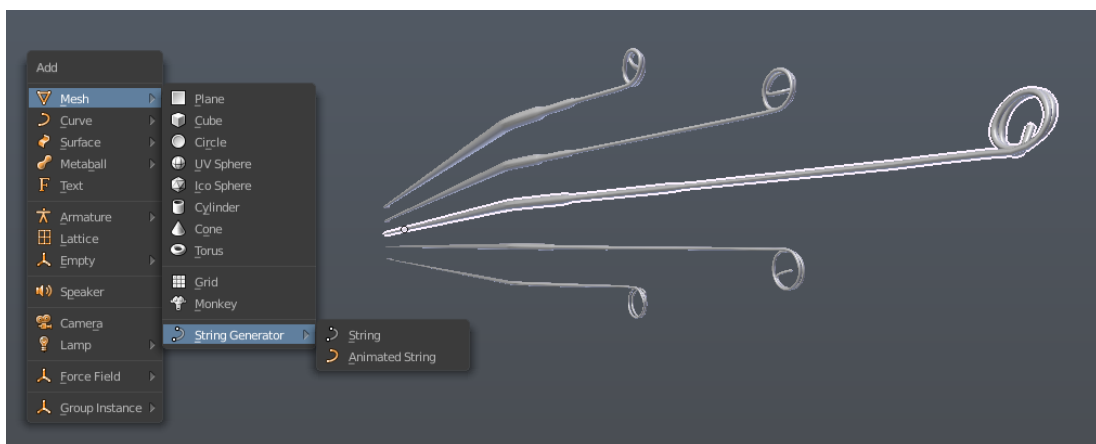
The states of the state machine are fully interconnected and every transition between any two states is created as a separate animation. All animations have been created manually. Since the animations are purely visualizations and the finger mesh does not actually interact with the string, the animations can contain slightly overexaggerated motions, which helps keep them visually interesting.

## 2.2. Strings

The strings of a guitar produce sound by vibrating with a certain frequency. The vibration is triggered by plucking the string using fingers, a plectrum, or in this case an animated picking mechanism. One way to create a visualization of a vibrating string would be to simulate the interaction between the pick and the string. This would make the animation seem more realistic, but simulating this type of behavior at runtime would be quite computationally demanding. To reduce the runtime complexity, the animations are precomputed and stored, which allows an animation to be simply played back as required while mostly preserving the realism of a simulation.

The task of modeling and animating guitar strings can be approached in several ways. One could attempt to do this manually, but it would require an immense amount of work to create realistic animations for the strings. Consequently, the only reasonable alternative would be to automate this process.

A detailed description on the entire string creation and animation process can be found in [6]. Blender and its powerful Python scripting API [1] are used to develop a



**Figure 2.5:** Five bass strings were created and animated using the string generator tool.

tool that automatically generates models of musical strings. The tool first generates a string mesh based on the given parameters such as string length, radius, and number of vertices. Then it creates and animates an armature which controls the vibration of the string. Finally, it adds shape keys that are used for the string bending. The bend positions are calculated using the equation (2.1) described in the next section.

For the bass guitar model, five strings were generated using the developed tool. The input parameter values for the strings were set to a range between .040 and .130 inches for the diameters, and to 34 inches for the length of the strings. For each of the generated string meshes, additional geometry was added to both ends of the mesh to form the static portions of the string which hold it in place – the start of the string at the saddle, and the ending of the string which wraps around a tuning post.

### 2.3. Fretboard

Fret positions on a guitar neck are mainly determined by the length of the string. When pressed down to a fret, the string is effectively shortened, so it produces a higher note than the root. Since the chromatic scale consists of 12 notes, a string pressed at the 12th fret should produce the note exactly an octave higher than the root, which means that the string should be shortened to half its original length. Similar logic applies for determining the positions of the other frets. The exact equation that determines the position of fret  $i$  for a string of length  $l$  is

$$d_i = l - \frac{l}{2^{i/12}} \quad (2.1)$$

where  $d_i$  is the distance from the nut on the neck of the guitar to the fret  $i$ . The finished fretboard model with the generated frets is shown in Figure 2.6.



**Figure 2.6:** The finished fretboard for the model. A script was used to calculate the fret positions and to scale the frets to match the width of the neck.

A script is used to generate individual frets for the fretboard. The script duplicates a template model of a fret and translates it to the correct position based on the result of (2.1) for each fret  $i$ . The duplicated fret is then scaled vertically so that it matches the width of the neck at its position.

## 2.4. Exporting the model

Models created in Blender can be exported in various formats. One of the most commonly used formats for this task is the FBX<sup>1</sup> file format. An FBX file stores information about the geometry, materials and animations of a 3D model, and is supported by Unreal Engine which will be used for the implementation. The created model of the bass guitar is split into separate elements in order to make exporting and managing animations easier:

- **Bass guitar body** – Together with the neck and head of the guitar, the body does not contain any animations. Only contains the mesh data with material channels.
- **Accessories** – Pickups, tuners, volume knobs, strap holders.
- **Fretboard** – Exported as a single file, but each individual fret is a separate mesh because their positions will be used for guiding the fret fingers along the neck.
- **One fretting finger** – Only one finger needs to be exported because it can be simply duplicated inside Unreal Engine. This avoids the need to import multiple identical animations and meshes, which would unnecessarily increase the size of the executable produced by the engine.
- **One pick finger** – Same logic applies as for the fretting finger.
- **Strings** – Each string is exported separately because they don't share mesh geometry nor animation data.

---

<sup>1</sup>FBX (Filmbox) is a digital content sharing file format owned by Autodesk.



## 3. Parsing MIDI files

A form of input is required in order to feed the note data into the visualization. The input data should be easy to parse, while providing all the required information about the notes: the pitch, timings and duration. Developing a custom file type to contain the data would offer some flexibility, as the data could be designed in a way that would facilitate parsing for this particular use case. However, a custom data type requires a custom way to create the data, and developing software specifically for this use would be a daunting task. A better alternative would be to use one of the standard file formats for storing music note data, such as the MIDI file format.

### 3.1. MIDI file structure

The MIDI file format (*Musical Instrument Digital Interface* [11]) is a binary format used to store MIDI stream data. The data is stored into several data blocks called chunks. Each chunk consists of a 4-character type descriptor and a 32-bit length value, followed by the chunk data. The type descriptor is used to differentiate between two types of chunks: header and track chunks. Each MIDI file contains exactly one header chunk that can be followed by one or more track chunks.

The header chunk provides metadata for the file. This metadata is used to specify the format of the file, the track count and the time division. The format determines how the file is organized – whether it contains a single track, multiple tracks played

Type	Length	Data
MThd	6	<format> <tracks> <division>
MTrk	32-bit value	<delta tick> <event> <delta tick> <event> ...

**Table 3.1:** The structure of the MIDI chunks. MThd indicates the header chunk, and MTrk is a track chunk.

simultaneously, or multiple independent tracks. The division is specified as the number of ticks per quarter note. Ticks function as timestamps that can be used to indicate the start time and duration of track events. Together with the information about the tempo of a song track, which equals the number of quarter notes per minute, one can easily determine the exact duration of each tick.

Track chunks contain the actual song data in the form of MIDI events: channel events, system events, or meta events. Channel events contain information about notes in the song. System events are used for controlling MIDI devices and modifying device settings. Meta events contain information about the track such as the tempo, time signature, and indicators for the start and end of the track. Each event is prefixed by a timestamp indicating the delta time since the previous event.

Meta event	Control code	Length	Data
sequence no.	FF 00	2	numeric index of the track sequence
track name	FF 03	var-length	string of ascii characters
tempo	FF 51	3	microseconds per quarter note

**Table 3.2:** The meta events useful for this implementation. All meta events start with the control code FF, followed by a specifier code, a length value, and the event content.

All length specifiers in the file are expressed as *variable-length* values. To prevent unnecessary allocation of memory, this system is used to express values up to a 4-byte integer using at least one, and at most four bytes. The lower 7 bits of each byte are used to store the value, while the most significant bit indicates whether the value is continued in the next byte. If the bit is set, the following byte contains the continuation of the value, and if it is unset, the byte is the last one in the sequence. As an example, the value  $0x817F$  would be used to store  $0xFF$ , and  $0x828000$  would result in  $0x8000$ .

## 3.2. Creating the MIDI files

FL Studio software [2] was used for generating the MIDI files used in this project. The software provides tools for recording and mixing audio input, as well as a MIDI sequencer that allows creating music using various sound samples.

An exported MIDI file contains a standard header, followed by an empty track chunk that only contains information about the tempo of the song. The subsequent

track chunks each specify a name that can be used to differentiate between the tracks, followed by the note data.

### 3.3. Extracting note data

A custom parser is used to read the note data from the MIDI file. The parser first reads the header to determine the track count and the number of ticks per quarter note. After that, each track is parsed and the notes are stored in separate containers.

The first track contains the tempo information. The tempo is stored as a 3-byte value inside a meta event, and is expressed in microseconds per quarter note. This value is used to calculate the number of quarter notes per minute and combined with the value for ticks per quarter note in order to determine the correct tick length.

The subsequent track chunks contain the note data in the form of MIDI events. Each note is represented by two 4-byte channel events, one for the start and another for the end of the note. The first byte indicates the number of ticks since the previous event. This means that the absolute tick value needs to be calculated for each event. The second byte is the control byte indicating whether the note is starting or ending. The note start event is indicated by the control code 9x, and the note end event uses the control code 8x. The x value indicates the MIDI channel used for playing the note.

Octave	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119

**Table 3.3:** The table of MIDI note pitch indices by octave.

The third and fourth bytes contain values for the pitch and velocity of the note.

Pitch is determined by the values in Table 3.3 which map 8-bit indicators to notes over eight octaves. The values start from the  $C_0$  note at the frequency of 16.35Hz and end with the  $B_8$  at 7902.13Hz. The commonly used reference note  $A_4$  with the frequency of 440Hz is represented by the value  $0x69$ . The note velocity indicates the audio volume of the note. The value can change between the start and end events, causing the note to fade in or out over time.

## 4. Mapping the notes to fretboard positions

One specific property of string instruments is that individual notes can be played at several points on the neck. For example, the same note can be played on up to five different positions on a five-string bass, as shown in Figure 4.1. This allows the musician to adjust the positions of their fingers depending on the surrounding notes that need to be played. Playing notes on different positions also slightly changes the tone of the notes – higher strings generally produce a brighter sound, while the lower strings are heavier and give softer tones.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
G	G#	A	A#	B	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#	D	D#	E	F	F#	G
D	D#	E	F	F#	G	G#	A	A#	B	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#	D
A	A#	B	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#	D	D#	E	F	F#	G	G#	A
E	F	F#	G	G#	A	A#	B	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#	D	D#	E
B	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#	D	D#	E	F	F#	G	G#	A	A#	B

**Figure 4.1:** The notes on the fretboard of a five-string bass guitar. The highlighted note  $A_4$  can be played once on each string of the bass.

This ambiguity creates a problem when finding the optimal fretboard positions for playing the notes in a given song. Parsing the MIDI file only gives information about the pitch of the notes, so the best position on the fretboard needs to be chosen for each note. The choice will mostly depend on the surrounding notes, with the goal of minimizing movement of the fretting fingers. Another problem rises from the fact that there are four fret fingers pressing the strings onto the fretboard. This complicates the matter even further because the finger indices also need to be considered when determining the optimal fretting positions.

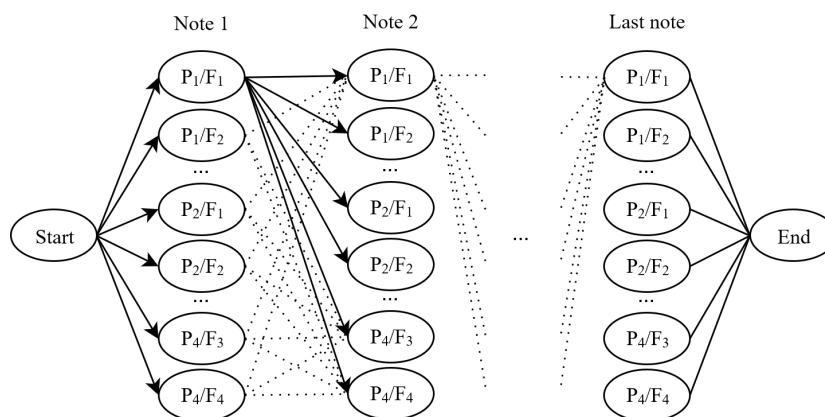
Finding the fretboard positions for a given set of notes can be approached as an optimization task. The goal of the optimization process is to produce a sequence of note positions that will allow the fingers to play the correct notes while abiding to a

set of rules and limitations posed by the design of the mechanism. The guitar fingering problem for a human hand has already been investigated in several papers which consider automatically generating tablatures [12], finding optimal chord sequences [5], and finding alternative viable fingering sequences for the same set of notes [8]. These works take various approaches to solving the problem using dynamic programming, genetic algorithms, and artificial neural networks.

The approach used here is similar to the one described by Radicioni et al. [7]. They use a form of dynamic programming to find the optimal sequence in a layered graph. The only major difference is the manner of calculating the cost because of the different fretting mechanisms used – Radicioni was simulating a real human hand, which has significantly more limitations when compared to the fingering mechanism implemented here.

## 4.1. Graph traversal algorithm

The optimal fingering is computed by building a graph of possible finger movements. Each graph node contains information about the note being played, a chosen fretboard position for the given note, the index of the finger that was assigned for pressing a string at that position, and a set of possible finger states. The finger states are based on the possible paths through the graph that have been taken up to that point. Each finger state tracks the positions of all fretting fingers and the states of their pins. For each finger pin, the finger state monitors whether the pin is pressed, and if so, the duration of the string press action.



**Figure 4.2:** The layout of the generated graph. The nodes correspond to position-finger combinations. The graph is built in layers, one layer per note in the song. The layers of the graph are fully connected.

The graph is built in layers by passing over all notes in the song and generating a layer of graph nodes for each note. The nodes of subsequent layers are fully connected. The transitions between them are scored to determine the cost, after which a backwards pass is performed to reconstruct the optimal fingering sequence.

#### 4.1.1. Scoring

To optimize the fingering and fretting positions, a measure of cost needs to be defined. The goal of the optimization process is to minimize the total cost of all movements of the fretting fingers. Sets of heuristics and rules are used to determine the cost value for a sequence of movements.

The heuristics guide the optimization process by discouraging undesired behavior. For example, playing empty strings is discouraged because it eliminates the need for using fretting fingers, making the visualization less active and therefore less appealing. Another example is determining which area of the fretboard contains most commonly played notes and favoring the nearby positions by inflicting a cost based on the distance from that area. The distance between the previous position of each finger and the new position can also be added to the final cost, as it will prevent fingers from unnecessarily moving too much. Having a large distance between the first and last finger is also discouraged to avoid playing all notes on the same string.

Criteria	Cost	Description
finger index	value of the index	fingers with lower indices are preferred
fret position	$6 \cdot  f_i - 6 $	positions around fret 6 are highly favoured
empty string	400	high cost in attempt to avoid empty strings
travel distance	$2 \cdot \sum_{i=1}^5 d_i$	sum of distances traveled by each finger
finger span	$2 \cdot  f_1 - f_5 $	distance between the first and last finger

**Table 4.1:** The main criteria for calculating the transition costs. The value  $f_i$  indicates the fret where the finger  $i$  is positioned.

Apart from those heuristics, a few strict limitations need to be set to avoid impossible finger movements. One major example are fingers that cross paths. This may never happen because the visualization would lose its realism. Also, if a finger is already being used to press a string on a certain fret, that finger cannot be moved until the note finishes playing because the string needs to remain pressed. This is closely related to

the finger crossing restriction because it can lead to a finger being stuck on a fret and blocking the path of other fingers, making it impossible for any finger to press the next note.

### **4.1.2. Building the graph**

The graph is built in layers, with each layer corresponding to one note in the given song. For each given note, a new layer is first added to the graph by calculating the possible fretboard positions for the note, and then combining each of those positions with each finger. For  $N$  fretboard positions for a note and  $F$  available fingers,  $N \cdot F$  new graph nodes would be initially created in each layer. One exception to this rule are graph nodes that correspond to empty strings being played, as they do not require any fingers to press down on the strings. The layers are initially fully connected, and the list of candidate finger states is empty. The correct values are calculated later by determining the best parent node and disconnecting all impossible connections between subsequent nodes.

### **4.1.3. The algorithm**

The forward pass is used to calculate the costs of the various paths through the graph. The algorithm uses a form of dynamic programming to gradually calculate the transition costs layer by layer.

As mentioned previously, every node has a collection of possible finger states. Those states are calculated from the finger states of the nodes in the previous layer. Initially, all fingers are resting at the base of the neck and are not pressing any strings. That makes every position-finger combination valid for the first note, since all fingers are available and no position restrictions are violated.

The process is less trivial for the subsequent layers, as some transitions between nodes may become impossible. The obvious first example can already manifest in the second layer. If the leftmost finger is already pressing a string on a fret and the next note needs to be played on a lower fret, no finger can reach that position because of the finger crossing restriction. The transition between the two nodes with that finger state is therefore impossible. However, it is worth noting that one parent node can have multiple finger states, and that some of those states may allow a valid transition, while the transitions for other states might be impossible. The nodes only become fully disconnected if no transitions are possible for any of the parent finger states.



After the impossible transitions have been eliminated, the remaining ones are scored. Every node in the parent layer stores a cumulative score for each of the finger states. The score of each state represents the cost of the optimal path through the graph which results in that finger state. The cost is calculated for every node in the current layer by combining the cost of the best finger state in a parent node with the transition cost between the two nodes. The best finger state is simply the one with the lowest cost, as long as it allows a transition to the current node. A new finger state is calculated by altering the best finger state with the position and finger data from the current node. A reference to the parent finger state is stored to enable reconstructing the path during the backward pass. The same process is repeated for each of the current nodes and all their connected parent nodes. As a result, the number of finger states for each of the current nodes equals the number of connected parent nodes from the previous layer.

---

**Algorithm 1** Forward pass used to calculate the transition costs

---

```

1: function CALCULATECOSTS
2:   for layer in graph do
3:     for node in prevLayer do
4:       simulate finger state changes for the current node tick;
5:       for node in currentLayer do
6:         calculate new transition finger states;
7:         filter the impossible finger states;
8:         calculate the transition costs for the remaining finger states;
9:         if newFingerStates.count is not 0 then
10:            add the finger state with the lowest cost to the current node;
11:            save the parent node and parent state;

```

---



---

**Algorithm 2** Backward pass for reconstructing the optimal path

---

```

1: function RECONSTRUCTOPTIMALPATH(node, state)
2:   if node is NULL then
3:     return;
4:   else
5:      $p \leftarrow node.parent$ ;
6:      $s \leftarrow p.states[state.parentIndex]$ ;
7:     RECONSTRUCTOPTIMALPATH( $p$ ,  $s$ );
8:     result.add(node);

```

---

When the costs in the final layer are calculated, a backward pass is performed to reconstruct the optimal sequence of fretting positions. The nodes in the final layer contain finger states with the accumulated costs of the best paths through the graph. The backward pass starts from the node in the last layer that contains the finger state with the lowest cost. Since the parent state for each node is stored during the forward pass, the path is easily reconstructed by simply backtracking over the parent references.

#### **4.1.4. Cleaning up the calculated sequence**

The calculated fretting sequence may contain redundant finger movements, so those can be cleaned up to reduce clutter in the animations. The redundant movements are mostly caused by the rule that encourages fingers to stay together, which causes fingers to unnecessarily move closer to each other under certain conditions. The cleanup function looks for cases where the same finger plays the same note twice in a row. The finger should not move in-between playing those two notes unless absolutely necessary. The only situation where the finger might have to move would be if it was blocking another finger from pressing its assigned note.

The cleanup is performed individually for each finger by recursively going through all the nodes and keeping track of the positions that the finger is pressing. If the same position is found twice consecutively, the recursion will pass through the nodes between the two consecutive finger presses and modify the finger states to make sure the finger stays at the same fret if possible. If the finger does have to move because it would block another finger, the finger states are not altered.

The final fretting sequence is then used to generate timeline actions for animating the strings, as well as the plucking and fretting mechanisms. The timeline is described in detail in the following chapter.

## 5. Action timeline

The action timeline is a system used for controlling all animations and sounds for the visualization. The timeline contains a list of actions that need to be executed, and each action is assigned a start time in ticks. Ticks are time units whose length is determined by the tempo of the parsed MIDI song.

The timeline is implemented as a list of abstract action objects, with a single function for getting all actions on a given tick. The timeline actions can perform various tasks, but they all provide a uniform interface for execution, so the main application controller can simply fetch all actions for a tick and then execute them all sequentially.

The timeline mechanism is designed to be generic, so the same system can be used for controlling various types of instruments at the same time. Each instrument has its own timeline and requires a set of actions to be calculated for a given set of notes. Furthermore, the system is not only limited to instrument actions, but can also be used to control any environment changes and camera movement with an additional timeline and a corresponding set of actions.

### 5.1. Actions for a bass guitar

Every instrument requires a custom set of actions that are specific to that instrument or family of instruments. For example, a string instrument requires actions that move fret and pick fingers, press fretting pins onto strings, bend the strings, play string vibration animations, play the note sounds, etc. Although string instruments come in various shapes and sizes, the actions mostly remain the same as long as they are played in the same manner (i.e. plucking the strings as opposed to using a bow). The specifics of each string instrument include the string count, fret count, and the tuning which determines the note positions on the fretboard.

In the following sections, the actions are described in terms of the five-string bass guitar. The operations initiated by the actions, such as playing the animations or sounds, are performed by a set of controllers. Each controller performs operations

that affect a specific area of the bass model, and the actions are split into categories based on the controller they use to initiate those operations: pick actions, fretting finger actions, string actions, and audio actions.

Category	Action	Description
Pick finger	prepare	moves the pick to the target string
	play	plucks the target string
	rest	returns the pick to the resting position
Fretting finger	prepare	prepares the finger for movement
	move	moves the finger to the target fret
	press	presses a pin down onto a string
	release	releases the pressed pin
	rest	returns the finger to the resting position
String	press	bends the string on the target fret
	release	releases the bent string
	play	starts the vibration animation
Audio	play	plays the sound sample for the given note
	stop	stops the sound playback

**Table 5.1:** The timeline actions required for the bass guitar.

### 5.1.1. Pick actions

Controlling the pick fingers is performed using three actions that trigger different types of animations. The first action prepares the finger for plucking the string by moving it from its current position towards the target string, the second action is used for plucking the target string, and the final action moves the finger from its current position back to the resting pose.

The actions are generated by simulating the song execution using temporary helper timelines which keep track of finger positions and availability as they pluck strings. Every finger has its own timeline, and initially the timelines are empty which indicates that the fingers are in the resting pose. As each note is processed, the content of the timelines is used to determine which finger is available for plucking the note.

For a finger to be available, it cannot already be playing a note. If the pick is on a different string than the one that needs to be plucked, it also needs to have enough

---

**Algorithm 3** Generating pick actions

---

```
1: function GENERATEPICKACTIONS(note, position)
2:   for finger in pickFingers do
3:     lastPlayTick  $\leftarrow$  finger.getLastPlayTick();
4:     if finger.canPlayNote() then
5:       if finger.canRestBeforePlaying() then
6:         generate rest action;
7:       if finger.currentString is not targetString then
8:         generate prepare action for the target string;
9:         generate the play action;
10:      return;

11:   // No finger was found, so the animation needs to be scaled
12:   finger  $\leftarrow$  the finger with the oldest lastPlayTick;
13:   calculate the required prepare animation scaling;
14:   generate prepare action with the scaled animation;
15:   generate play action;
```

---

time to perform the preparation animation before the string needs to be picked. The time required is determined by the length of the preparation animation, which depends on the current and target positions of the finger. Different transitions have different animation durations, e.g. transitioning to the neighboring string requires a shorter animation than transitioning from the first string to the last. The animation duration is given in seconds, so it must be transformed into ticks so that the start tick of the required preparation animation can be calculated.

In the situation that no pick has a large enough window for the preparation animation, the pick that was least recently used is chosen. The preparation animation is then scaled to fit the available window. The animation scaling is not very noticeable in practice, since those situations occur quite rarely and only in faster songs where notes are played more frequently.

Once a pick is found that can fit the entire animation, it can be assigned to playing the note. This is done by scheduling the pick play action that will be executed when the note is played. The play action starts the plucking animation of the pick finger. If the finger was not already in the correct position, the additional preparation action needs to be scheduled. After the actions are generated, the helper timeline for the used finger is updated to keep track of the current finger state.

The third possible action that can be scheduled for the pick fingers is the action that returns the finger to its resting position. This is performed when the finger is idle for a long time in-between two consecutive plucking actions. Also, all fingers are returned to the resting position after they play their last notes in the song.

### 5.1.2. Fretting finger actions

The fretting fingers are not controlled using animations, but by altering the shape key values and translating the fingers along the local Y-axis to slide them along the neck. Five actions are used for finger preparation, finger movement, pressing and releasing the pins, and returning the finger to the resting position.

The preparation and resting actions are only used to slide the fingers to the fretboard at the start of the song and return them to rest at the end, so only one of each is added for every fretting finger. The remaining actions are generated for each note in the song, based on the optimal fretting sequence calculated earlier.

For each node in the calculated sequence, the fingers need to be moved to their respective positions specified in the finger state of the node. Since the sequence nodes each represent a note that is played, the movements need to be performed earlier so that the fingers are in place when a string needs to be pressed by a pin.

The actions responsible for moving the fingers are first added for every finger. These actions merely translate the finger along the length of the neck to the position indicated by the target fret. Because the frets are imported as separate models, the finger positions in object space are determined by looking up the position of the fret model with the target index.

---

#### Algorithm 4 Generating fretting finger actions

---

```

1: function GENERATEFRETTFINGFINGERACTIONS(note, position, state)
2:   for finger in frettingFingers do
3:     targetFret ← state.fingerStates[finger].fret;
4:     if finger.isResting() then
5:       generate the prepare action;
6:     if finger.currentFret is not targetFret then
7:       generate move action to the target fret;
8:     if targetFret is position.fret then
9:       generate press action;
10:      generate release action;

```

---

Next, the pin actions are added for the finger that presses a string. The pin press and release actions are performed by modifying the shape key values for the affected pin. As described in section 2.1.1, the shape keys contain information about vertex displacements which press the pins towards the fretboard. The pin press action interpolates the shape key value from  $0.0$  to  $1.0$ , gradually pressing the target pin towards the string, and the release action performs the interpolation in the opposite direction. Since the string needs to be already pressed down before the note is played, the pin press action is scheduled to start slightly before the note start tick. On the other hand, the pin release action is scheduled to be executed at the note end tick.

### 5.1.3. String actions

The strings have animations for vibration and shape keys for bending, which makes them similar to both fretting fingers and picks. Just like the fretting finger actions, string actions are based on the calculated fretting sequence. The nodes in the sequence determine which string will be plucked for each note and to which fret the string needs to be bent. Three different actions are used for pressing strings, releasing them, and starting the vibration animation.

The first string action triggers the vibration animation. The animation should be properly aligned with the plucking animations of the finger picks, so that the string starts vibrating only after it is plucked. The animation lasts until the note finishes playing, and if the same string is plucked while it is still vibrating, the animation is simply restarted.

Pressing a string requires two steps, and each step modifies one shape key. The first shape key displaces the vertices along the length of the string so that the start of the vibrating section of the string aligns with the target fret. Then the second shape key can be used to gradually bend the string towards the fretboard. The two vertex displacements are blended, which results in the string being bent at the target fret. The release action does the opposite. The bending shape key value is gradually reduced to

---

**Algorithm 5** Generating string actions

---

```
1: function GENERATESTRINGACTIONS(note, position)
2:   string  $\leftarrow$  position.string
3:   generate press action;
4:   generate play action;
5:   generate release action;
```

---

simulate the fretting pin releasing the string. After the string is straight again, the fret shape key is deactivated, returning the string vertices back to their original positions.

Since the press and release animations are not instant, it is worth noting that a string can be pressed again on a different fret before it finishes fully releasing the fret that was played previously. These situations are expected and all problems are automatically avoided because of the way shape keys are controlled using floating point values. In this situation, the new fret shape key would be increasing at the same rate at which the old fret shape key is decreasing. That would make the string vertices just slide over from the old fret to the new one. This behavior opens the possibility for the song to contain sliding notes, which start at one fret and end at another.

#### 5.1.4. Audio actions

The audio actions are responsible for playing the sounds of the notes. Two actions are required for starting and stopping the audio. Each string has its own audio source, which allows multiple strings to be played at the same time, forming chords. Only one note can be played on a single string at any time.

The audio sources use sound samples to produce the notes that are played. Each string uses a sound sample which corresponds to the note played at the 12th fret of that string. The remaining notes on each string are produced by modulating the frequency of the sample. The correct frequency of each note is first calculated in relation to a pre-determined baseline note. The most commonly used baseline is  $A_4$  with the frequency of 440.0Hz. The frequency for note  $X$  is given by

$$f_X = 440.0 \times 2^{d/12} \quad (5.1)$$

where  $d$  indicates the note distance between the target note  $X$  and the baseline  $A_4$ . For example, the note  $C_5$  is three half-steps above  $A_4$ , so  $d = 3$  would give a resulting frequency of 523.3Hz. Likewise, the lower octave  $C_4$  is nine half-steps below  $A_4$  which gives  $d = -9$  and the frequency of 261.6Hz as a result.

The calculated value  $f_X$  is used to modulate the frequency of the sound samples for the strings. A pitch multiplier value is calculated as the ratio between the target frequency and the frequency of the sound sample. It is then given to the audio source while playing the sound sample to increase or decrease its frequency.

Since the sound sample needs to be long enough to play notes with high sustain, the stop action is required to terminate sound playback when a string is released. The side effects of manually stopping the playback are crackling noises because the audio



signal is abruptly cut. To fix this issue, the volume of the sound is gradually lowered just before the sound is stopped. That prevents the signal from cutting, removing the crackling noises.

## **5.2. Executing the actions**

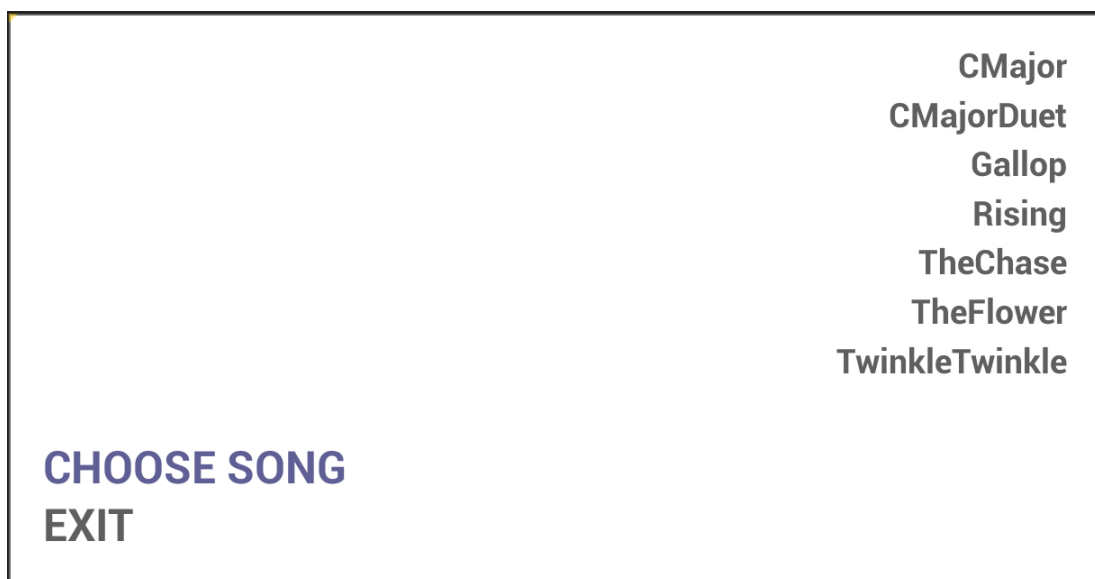
The actions are executed by the main application controller. The controller implements a timer system which periodically produces tick events. As mentioned previously, the ticks are time units whose length is determined by the tempo of the MIDI file. Each tick produces a callback to an event function that executes timeline actions.

Every time the tick function is called, it first polls all available timelines to find the actions that are scheduled to be executed at the current tick. Since the timeline actions are sorted, the polling is trivial to perform and executes in constant time because each timeline remembers the index of its last executed action. The actions are then sequentially executed, performing their scheduled operations.

## 6. Results

The application was implemented using Unreal Engine 4 [3], a 3D graphics engine that is primarily designed for game development, but can also be used for developing various interactive 2D or 3D visualizations.

When the application is started, it first displays a startup menu that provides the user with a list of songs available for playing, as shown in Figure 6.1. The song list is generated by detecting MIDI files located in the content folder of the application.



**Figure 6.1:** The startup menu of the application. Displays a list of songs that can be played.

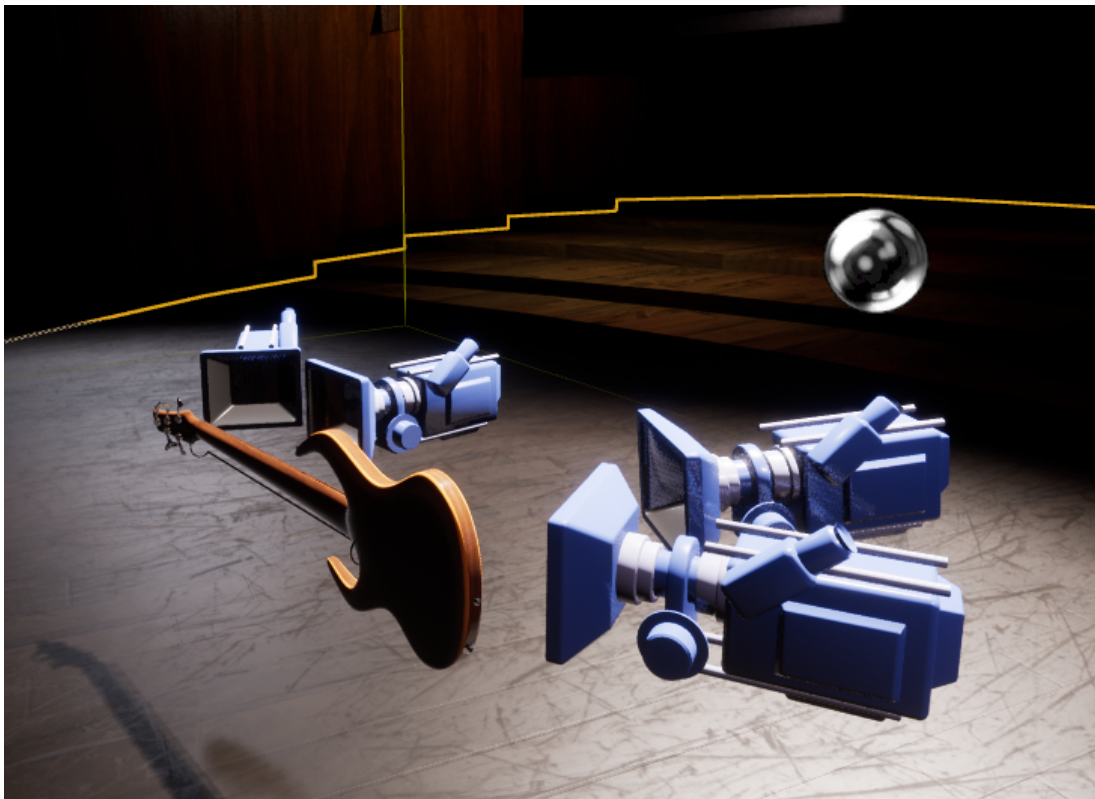
Once the user selects a song to be played, the MIDI parser loads the selected file and extracts the song track data. A sample of the console output while parsing the song is displayed in Figure 6.2. The notes for every track are forwarded to the corresponding systems for generating timeline actions based on the instrument information given by the MIDI track. The only supported instrument in the current implementation is the bass guitar, so tracks for all other instruments are ignored in this step. The bass guitar model is placed into the environment scene of a small concert hall where the visualization takes place. The environment is shown in Figure 6.3.

```

VirtusonicLog: Loading song: Rising.mid
VirtusonicLog: Track count: 6
VirtusonicLog: Ticks per quarter: 12
VirtusonicLog: Tempo: 130
VirtusonicLog: Added track Bass
VirtusonicLog: Added note to Bass: 48 63 62 100
VirtusonicLog: Added note to Bass: 60 66 69 100
VirtusonicLog: Added note to Bass: 69 75 69 100
VirtusonicLog: Added note to Bass: 66 81 62 100
VirtusonicLog: Added note to Bass: 72 84 74 100
VirtusonicLog: Added note to Bass: 78 93 69 100
VirtusonicLog: Added note to Bass: 84 105 62 100
VirtusonicLog: Added note to Bass: 90 108 76 100
VirtusonicLog: Added note to Bass: 102 123 69 100
VirtusonicLog: Added note to Bass: 114 126 73 100
VirtusonicLog: Added note to Bass: 108 141 62 100
VirtusonicLog: Added note to Bass: 126 141 69 100
VirtusonicLog: Added note to Bass: 132 144 74 100
VirtusonicLog: Added note to Bass: 144 159 62 100
...

```

**Figure 6.2:** The console output while parsing the MIDI file. The header is parsed first to get the values for the track count and time division. Then the track chunk for the bass guitar is parsed to get the tempo and all the notes. From left to right, the numbers for the notes indicate: (1) the start tick, (2) end tick, (3) note pitch identifier, and (4) note velocity.



**Figure 6.3:** The concert hall environment with the bass guitar.

During the song playback, the user has access to four different camera views to choose from – overviews of the entire model from both sides and two views that focus on the pick fingers and fretting fingers. After the visualization is completed, the scene fades out and the startup menu is displayed again.



**Figure 6.4:** The four available camera angles for displaying the song playback.

## 7. Conclusion

The implemented visualization presents a good starting point for a more feature-rich interactive application. The action timeline system and the MIDI parser, together with the powerful graphical capabilities of the Unreal Engine, form a robust and extensible framework for visualizations of self-playing musical instruments. With those core systems in place, the framework can be easily extended with new features.

One of the obvious upgrades would be the addition of new instrument models. These could be other string instruments, such as a guitar or an ukulele, as well as other types of instruments that are played in an entirely different manner, such as pianos or drums. Apart from creating the 3D models of those instruments, new action types would need to be added to the timeline to perform the required animations.

Another potential feature would be to add a graphical representation of the notes that are played, either in form of tablature or real musical sheets. This would be the first step towards educational use cases for the visualization, as it would allow the user to see exactly which notes are played in real time. The educational aspect would be even more reinforced by introducing realistic models for the picking and fretting hands, together with an improved fretting sequence optimization algorithm. Combining this with the graphical representation of the notes would produce a great tool for learning new techniques, practicing various fingering patterns and learning to play new songs.

# BIBLIOGRAPHY

- [1] Blender API documentation, Blender Foundation. <https://docs.blender.org/api/2.78c/>, 2017. [accessed June 10th 2017].
- [2] FL Studio, Image Line Software. <http://www.image-line.com/flstudio/>, 2017. [accessed June 10th 2017].
- [3] Unreal Engine 4 Documentation, Epic Games. <https://www.unrealengine.com/what-is-unreal-engine-4>, 2017. [accessed June 12th 2017].
- [4] R. B. Dannenberg, B. Brown, G. Zeglin, and R. Lupish. A robotic bagpipe player. *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2005.
- [5] N. Emura, M. Miura, N. Hama, and M. Yanagida. A system giving the optimal chord-form sequence for playing a guitar. *Acoustical Science and Technology*, 27(2), 2006.
- [6] L. Kunić and Ž. Mihajlović. Generating virtual guitar strings using scripts. *Proceedings of MIPRO, DC VIS*, 2017.
- [7] D. Radicioni and V. Lombardo. Guitar fingering for music performance. *International Computer Music Conference*, 2005.
- [8] A. Radisavljevic and P. Driessen. Path difference learning for guitar fingering problem. *International Computer Music Conference*, 2004.
- [9] F. A. Saunders. The mechanical action of violins. *Journal of Acoustic Society of America*, 1937.
- [10] E. Singer, K. Larke, and D. Bianciardi. Lemur guitarbot: Midi robotic string instrument. *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2003.

- [11] *Standard MIDI-File Format Spec 1.1*. The International MIDI Association, 1999.
- [12] D. R. Tuohy. Creating tablature and arranging music for guitar with genetic algorithms and artificial neural networks. *University of Georgia*, 2006.

## Modeliranje i simulacija 3D glazbenog instrumenta

### Sažetak

Rad opisuje aplikaciju za vizualizaciju realistične samosvirajuće 3D bas gitare. Gitara je izrađena i animirana upotrebom alata *Blender*, a zatim prenesena u *Unreal Engine* u kojemu je implementirana aplikacija. Ulazni podaci o notama za vizualizaciju dobiveni su iz MIDI datoteka.

Animacijama i zvukovima upravlja se putem vremenske linije s akcijama, pri čemu svaka akcija obavlja određenu operaciju u zadanom trenutku. Sustav je dizajniran tako da bude lako nadogradiv, kako bi omogućio dodavanje različitih vrsta akcija, od akcija koje upravljaju glazbenim instrumentima do akcija koje mijenjaju okruženje ili pomiču kameru. Akcije za bas gitaru služe za upravljanje mehanizmima lijeve i desne ruke, za animiranje žica i za puštanje uzoraka zvukova kod sviranja nota.

Cilj projekta bio je napraviti nadogradivi aplikacijski okvir za vizualizaciju samosvirajućih glazbenih instrumenata koji bi se kasnije mogao upotrijebiti u zabavne ili edukacijske svrhe.

**Ključne riječi:** Glazbeni instrumenti, 3D vizualizacija, MIDI, Blender, Unreal Engine.



## **Modeling and simulation of a 3D musical instrument**

### **Abstract**

This thesis describes an application that visualizes a realistic self-playing 3D bass guitar. The guitar was modeled and animated using *Blender*, and then exported into *Unreal Engine* where the application was implemented. The input for the visualization are MIDI files, which are parsed to extract the note data.

All animations and sounds are controlled using an action timeline system, where each action performs a specific operation at a given point in time. The system is designed to be extensible so that various types of actions can be supported, from various instrument actions to controlling the environment and moving the camera. The actions for the bass guitar include controlling the pick fingers, the fretting mechanism, the strings, and playing sound samples for the notes that are played.

The goal of the project is to provide an extensible framework for visualizing self-playing musical instruments, that could later be used for entertainment and educational purposes.

**Keywords:** Musical instruments, 3D visualization, MIDI, Blender, Unreal Engine.