RECREATING FINGER MOTION FROM AUDIO DATA FOR LIVE PERFORMANCE IN VIRTUAL SPACE

By

Pranabesh Sinha

A project proposal submitted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Supervised by

Dr. Joe Geigel
Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York
January 2011

Approved By:

Dr. Joe Geigel
Associate Professor, Department of Computer Science
Primary Advisor

Dr. Reynold Bailey
Assistant Professor, Department of Computer Science
Reader

Dr. Hans Peter Bischoff
Professor, Department of Computer Science
Graduate Program Coordinator

ACKNOWLEDGEMENTS

I would like to thank Dr. Joe Geigel for the wonderful graphics course structure that he has set up and for always greeting me with a smile whenever I needed his advice and making me feel that I was on to something good. I thank Dr. Reynold Bailey for nudging me in the right direction in the CG 2 class, which although doesn't have a direct connection with this project, helped me to get more confident in programming in computer graphics. I also thank Dr Warren Carithers for making the world of shaders especially the GLSL world more interesting.

I owe this project to the countless Sunday morning spent watching Disney cartoons and to Pixar. Without them I'd have never thought of doing an MS in Computer Graphics.

Lastly I owe a big thanks to my parents and to Troy for keeping faith in me while I hemmed and hawed and stayed away from finishing the project.

ABSTRACT

Although motion capture allows us to animate human motion, the data first needs to be processed before it can be applied to models. Hence if this data is used directly in real time, the resulting animation will have artifacts. If there are multiple joints in a small area, such as in fingers the amount of noise in the data will be even higher.

The purpose of this project is to create an alternative technique by which finger movement while playing a musical instrument such as the piano can be animated in real time by analyzing the music that is being played.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Animation is regarded as more of an art than a science. Traditional cell animation was perfected by the likes of Walt Disney, Chuck Jones and other animators. The process required an adept skill at drawing. When Mickey played Mozart on the piano, an enormous amount of drawing was needed, just for the fingers alone.

Computers brought several algorithms which could simplify the process to a certain extent. The two most common techniques used today are keyframing and motion capture. While the former technique still needs a certain artistic skill in being able to decide the right poses which are to be keyframed the latter one can be completely automated and is widely used in games and movies to get believable human motion.

However, as with all things in the world, motion capture also has its side effects. The technology relies on transmitting position data of joints in the human body to a receiver and this data is often noisy. So a data cleaning phase is required before the data can be applied to a 3D character.

This poses a problem in systems like Virtual Theater which apply motion capture to characters in real time. Especially problematic is when such data is used for joints which are too numerous and too close together such as fingers.

This project presents an implementation of an idea that allows the Virtual Theater system to show finger motions of a character playing a musical instrument such as a piano in real time.

1.1 Background

The Virtual Theatre (VT) project is an endeavor started by Dr. Joe Geigel and Dr. Marla Schweppe at Rochester Institute of Technology. Their intention was to combine their love for theatre and technology to create a virtual stage where live actors perform via animated avatars in real time [1].

At its final stage VT will allow actors, audiences and the crew members to be in different parts of the world and yet be able to share the experience of having produced and enjoyed a show.

This is made possible with the use of motion capture. The actors are either made to wear the by now traditional motion capture suits or use the new suit-less technology which requires them to simply walk into a small stage where their motion get recorded by multiple cameras.

Ideally the motion capture data should be cleaned to remove noise and then applied to a character, but since VT needs to work in real time this stage has to be jettisoned. The data is mapped on to the virtual character that the actor is playing by using Autodesk's Motion Builder software. This serves two purposes:

1. Every motion capture system uses their own software to read the motion captured data from the suits or cameras. But they have plugins which allow the data to be streamed into Autodesk's Motion Builder. This in turn allows the VT system to stream the data from motion builder by writing a plugin for it.

2. Motion Builder maps the incoming streaming data into Autodesk's Maya characters which have a specific number of bones. This sometimes reduces the incoming motion capture data.

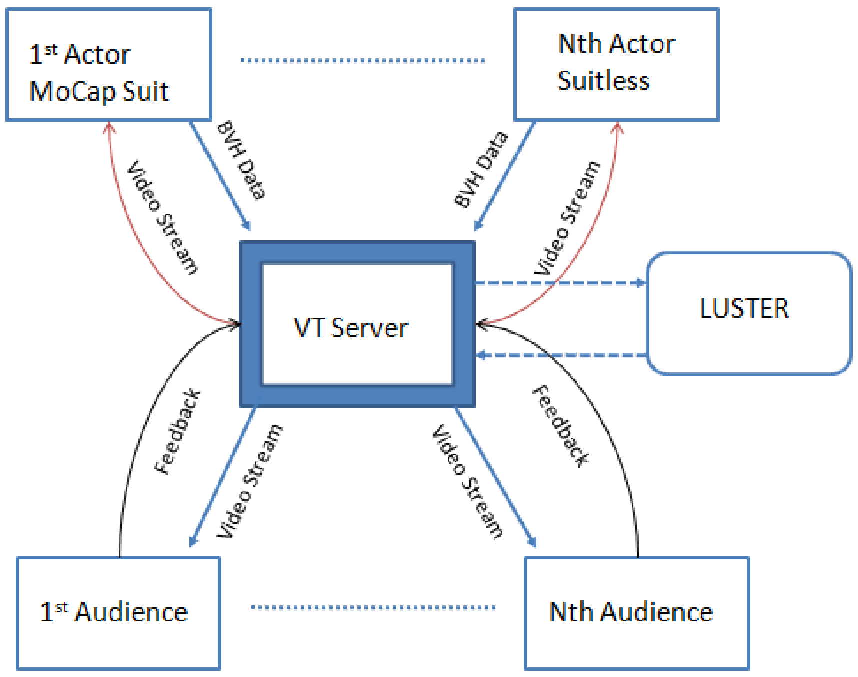


Figure 1.1 Overview of the VT system

At the heart of VT is a game engine called Luster [2]. Some of its chief functionalities are:

- It renders the stage, lights, characters and the props.
- It provides a live video feedback of the stage that it renders to the audience and to the actors. This helps the actors localize themselves on the screen and interact with other actors.
- It gives the audience controls to express their feelings with cheers or boos.
- It triggers the audio files that are used in the play.
- It provides various other controls that the director of the play would need.

Luster reads the BVH data that is streamed from the Motion Builder systems and maps it onto the specific characters.

1.2 Problem Introduction

VT has always had problems with animating the hands of the characters. This currently makes it impossible to have scripts for the plays in which the character uses their hands to perform a complex operation such as playing a musical instrument.

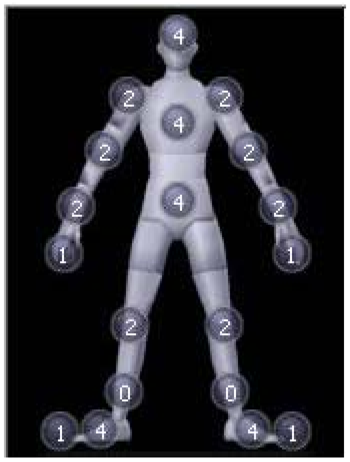


Figure 1.2 Figure courtesy of [3]

Motion Builder has some very specific points to which it allows data to be mapped as shown in the figure 1.2. The points marked at the end of each hand maps to the wrist. There are no points that map to the bones of the finger. So there is no way to capture finger movement data.

There are two ways to solve the problem. The first involves using a system such as Shape Hand which is shown in the figure below:

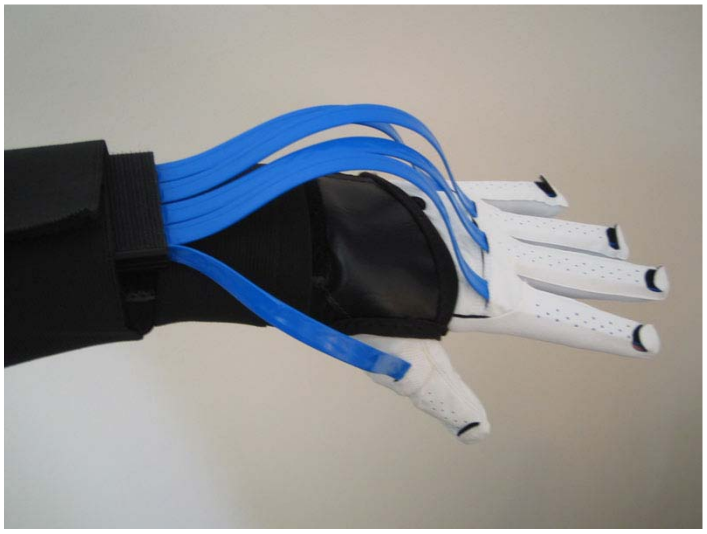


Figure 1.3 The shape-hand system courtesy of [4]

The shape-hand is a special motion capture glove designed specifically for capturing finger movement. Although such a device makes it easy to capture finger movement it makes holding and playing a musical instrument cumbersome. Also, each actor who plays an instrument in the script will have to be provided with a shape-hand system which will increase the cost of production.

The second option involves using the information in the music that is being played to estimate the finger motion and is the topic of this project.

1.3 Previous Work

The work done by Jun Yin, Ankur Dhanik, David Hsu and Ye Wang [5] serves as a direct inspiration for this project.

Their approach uses pre recorded wave files which are transcribed via a transcriber module as shown in figure 1.4 to create a note table which is then passed on to the Animator which is an Inverse Kinematics (IK) engine. The IK engine produces a video output of an animated character playing the violin.

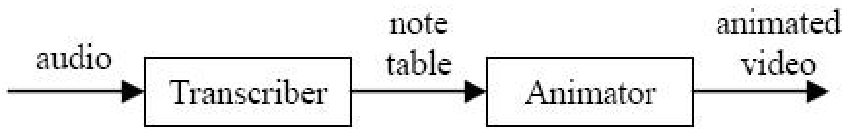
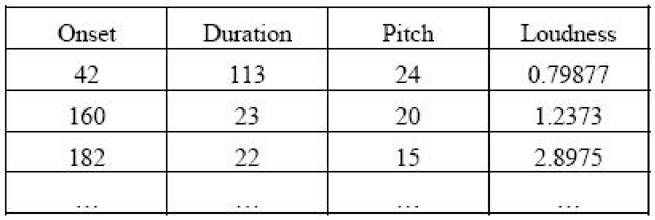


Figure 1.4 Figure courtesy of [5]

To create the note table as shown in figure 1.5, their approach was to create an energy spectrum of the music. The spectrum was then used to estimate the pitch and loudness in the music from which the onsets of the music were determined. This allowed them to create a note table which contains the following information.



Onset	Duration	Pitch	Loudness
42	113	24	0.79877
160	23	20	1.2373
182	22	15	2.8975
…	…	…	…

Figure 1.5 Figure courtesy of [5]

For the purpose of this project it was decided to use MIDI signals. MIDI (Musical Instrument Digital Interface) was developed in the early eighties as a protocol for enabling communication between musical instruments and computers. This protocol does not directly concern itself with audio signals; rather, it stores information such as the pitch, intensity, volume and velocity which can later be used to recreate the music. This is the essentially the same information that is produced by the transcription process in their paper.

The second stage of their paper uses an IK engine to animate the hand. IK is a very widely used technique in robotics and several papers exist which address the problem of simulating hand movement using IK.

One of the most useful papers was by George ElKoura and Karan Singh [6] on the Handrix. This is a system made for playing the guitar using the IK engine.

Their solution was to use a greedy algorithm which minimized the cost of moving the fingers. One of the advantages that their system had was the ability to look ahead at the notes being played and plan the finger movement accordingly. But in a real time system such an assumption cannot be made.

However their paper provides some very useful measurements about the human hand which was used in the project to tune the IK engine. These will be outlined in section 2.3 which deals with the IK module.

1.4 Project Goals

With the problem defined in the previous section, the goal of this project is to create a framework that takes in MIDI input signals through a USB keyboard controller and provides an animated view of the fingers that are likely to be used in playing the notes and also to stream out this data in the form of BVH data which can be received by VT and applied to the hands of their characters.

Since the project will synthesize the data in real time it will be possible to verify the accuracy of the process by looking at the finger movement of the person playing the piano and comparing it with the animated view that the project provides.

II. ARCHITECTURE OF THE SYSTEM

The following diagram shows the components of the system:

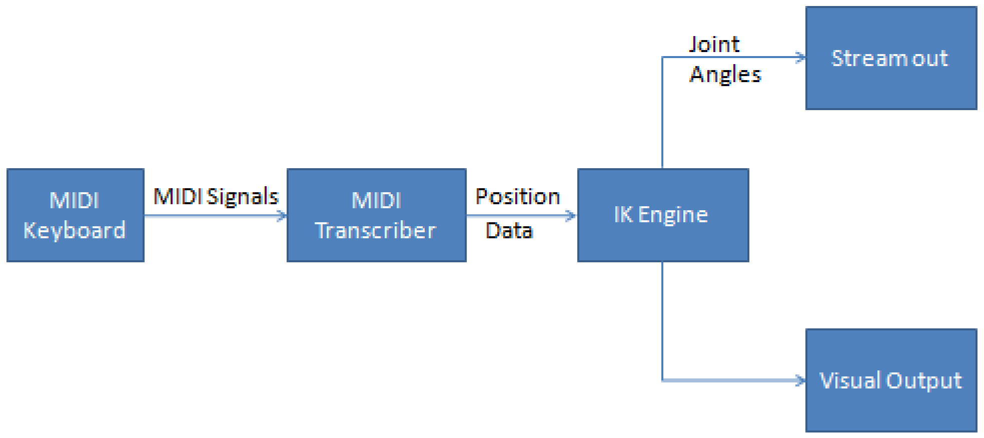


Figure 2.1 Architecture of the system.

2.1 MIDI Keyboard

A simple MIDI USB controller can be used for this purpose. Several libraries exist which allow interfacing between the USB controllers and the OS.

For the purpose of this project the RtMidi [7] library was chosen. It is written in C++ and follows an object oriented approach. It allows both MIDI input and output functionality through the use of two classes called RtMidiIn and RtMidiOut respectively. Only the former is used in this project.

A MIDI signal is 3 bytes long. The first byte contains status information which includes

- Note turned on/off data.
- The channel data.

The second two bytes contain the data of the signal which includes

- The value of the note in the second byte
- The velocity with which the note was hit in the third byte.

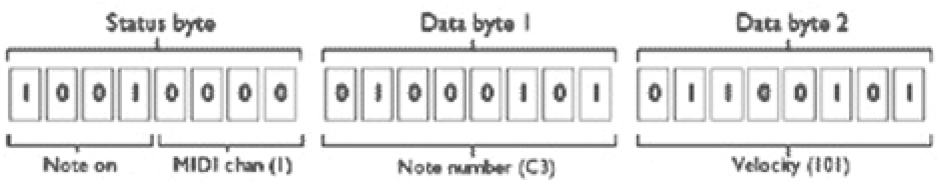


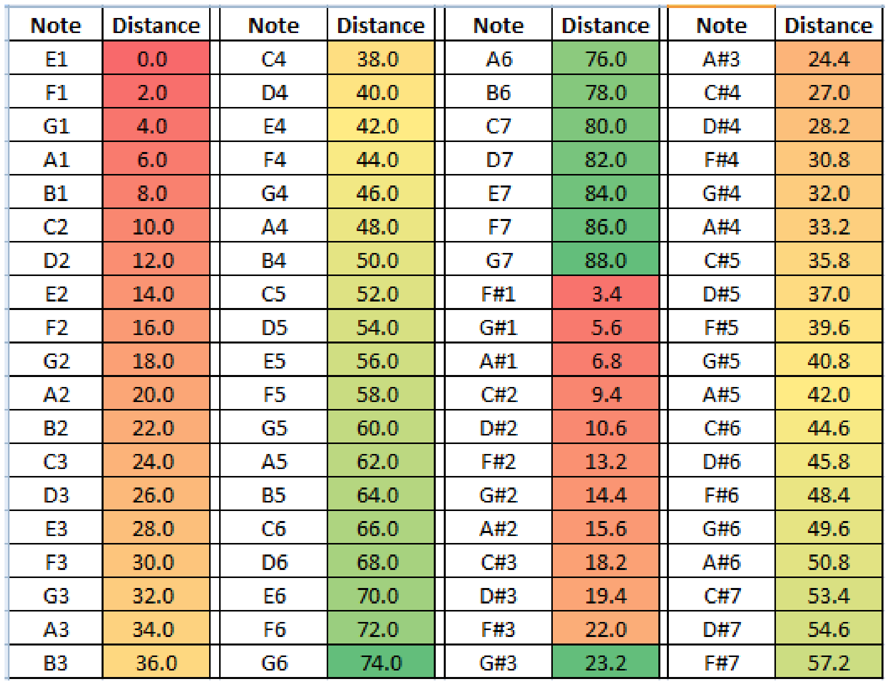
Figure 2.2 The MIDI bytes courtesy of [8]

For the purpose of this project the channel information is unnecessary since only one instrument is used. The velocity of the note is not used as the IK engine used does not allow speeding or slowing down motion.

2.2 MIDI Transcriber

The purpose of this module is to take MIDI signals as input and convert it into position data for the IK engine and forms the bulk of the work done in this project

It consists of two units. The first unit converts the MIDI signal into distance offsets on the keyboard. Figure 2.3 shows a note table that was used for this purpose.



Note	Distance	Note	Distance	Note	Distance	Note	Distance
E1	0.0	C4	38.0	A6	76.0	A#3	24.4
F1	2.0	D4	40.0	B6	78.0	C#4	27.0
G1	4.0	E4	42.0	C7	80.0	D#4	28.2
A1	6.0	F4	44.0	D7	82.0	F#4	30.8
B1	8.0	G4	46.0	E7	84.0	G#4	32.0
C2	10.0	A4	48.0	F7	86.0	A#4	33.2
D2	12.0	B4	50.0	G7	88.0	C#5	35.8
E2	14.0	C5	52.0	F#1	3.4	D#5	37.0
F2	16.0	D5	54.0	G#1	5.6	F#5	39.6
G2	18.0	E5	56.0	A#1	6.8	G#5	40.8
A2	20.0	F5	58.0	C#2	9.4	A#5	42.0
B2	22.0	G5	60.0	D#2	10.6	C#6	44.6
C3	24.0	A5	62.0	F#2	13.2	D#6	45.8
D3	26.0	B5	64.0	G#2	14.4	F#6	48.4
E3	28.0	C6	66.0	A#2	15.6	G#6	49.6
F3	30.0	D6	68.0	C#3	18.2	A#6	50.8
G3	32.0	E6	70.0	D#3	19.4	C#7	53.4
A3	34.0	F6	72.0	F#3	22.0	D#7	54.6
B3	36.0	G6	74.0	G#3	23.2	F#7	57.2

Figure 2.3 The note table used in the project.

The distances were measured on a standard CASIO keyboard taking the left end as the origin. This data is stored in a map. When the musical note information is obtained from the keyboard the corresponding distance is looked up from the map. On a piano the sharp notes denoted by the # symbol correspond to the black keys.

The second unit takes the distance offset and decides, based on the current state of the ten fingers which finger is most suited to play the note and sends this information to the IK engine.

Figure 2.4 shows a class diagram which provides a holistic view of the relations between the main classes.

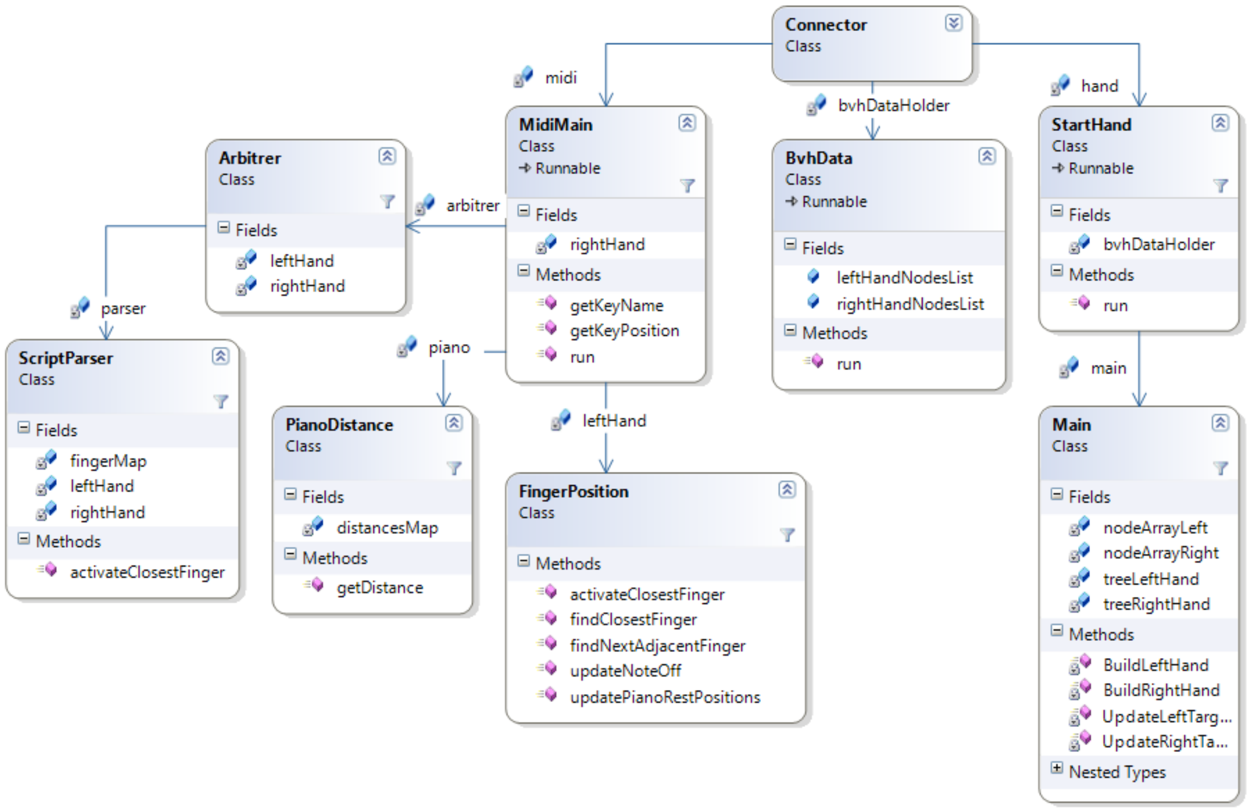


Figure 2.4. Holistic view of the classes

The **Connector** class is the entry point which contains references to the **MidiMain**, **BvhData** and **StartHand** classes. These three classes inherit from the **Poco Runnable** class. The **Connector** class starts these three classes and waits for them to join.

The **BvhData** class contains two lists of **Node** objects (which is data required by the IK engine) for each hand. It is instantiated and passed into the **StartHand** class so that when the two hands are created during the startup of the IK engine the **Node** objects that get created can be added to the lists. From then on the **BvhData** class reads the angle values from the lists at 24 fps as the IK engine moves the hands. A client can be created that would send this string data to the VT server.

The **StartHand** class exists so as to be able to tie the IK engine into the multithreaded structure of this project. It also passes the reference of the **BvhData** class to the IK engine so that when the IK engine creates the two hands it adds each **Node** object into the list contained in the **BvhData** class. The **StartHand** class starts up the main method in the **Main** class which is the starting point of the IK engine. This will be described in the next section.

The **MidiMain** class encapsulates the RtMidi classes. Once it is instantiated it listens for incoming MIDI signals on the USB port. It contains a **PianoDistance** class which contains the note and distance information shown in the note table in figure 2.3. The **MidiMain** class instantiates two objects of the **FingerPosition** class (one for each hand) and passes them to the **Arbitrer** and **ScriptParser** classes.

The purpose of the **Arbitrer** class is to arbitrate which hand should be used to play the note. For this purpose it invokes methods in the **FingerPosition** class.

There are two objects instantiated of the **FingerPosition** class, one for each hand. The class diagram in figure 2.5 shows its main variables and methods. For simplicity only the variables corresponding to the index finger is shown in the figure.

There are two sets of **Point** variables that store the co-ordinates of the hand. One set, called the currIndex variables store the co-ordinates of the points where the hand actually is located and is rendered by the IK engine. The second set, called the pianoRestIndex variables store the coordinates of the hands as they would be poised on the center of the piano. It was decided to use two different sets of variables so as to facilitate the transformation of the co-ordinate space in which the IK engine works and the co-ordinate space in which the piano exists.

The calculation of which finger is nearest to the note is done on the pianoRestIndex variables and the result is applied to the currIndex variables.

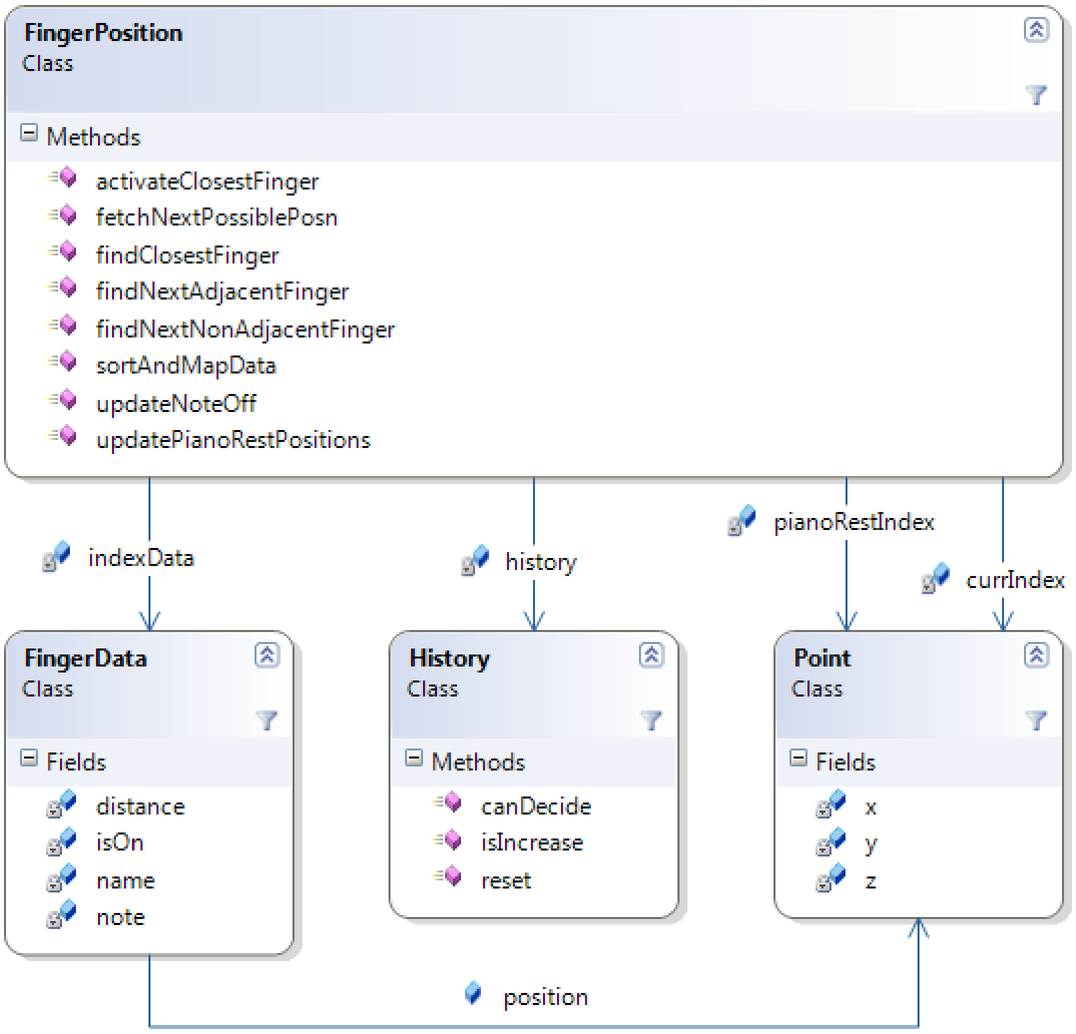


Figure 2.5 The FingerPosition class

The currIndex and the four other **Point** variables named currMiddle, currRing, currPinky and currThumb store the co-ordinate data of the end effectors of each finger in each hand. It is the actual co-ordinates that the IK engine uses to render the hands. It is by changing the x, y and z vales in these variables that the finger motion is attained. To obtain the effect of playing a key the z value of these points is decreased. This simulates the effect of pressing on a key. When the z value is restored to its previous value, it simulates the effect of lifting the finger off from the key. For the sharp notes, the y and z values are changed simultaneously. Currently the x values are not changed, but they can give a more realistic effect of playing the piano.

The pianoRestIndex and the four other **Point** variables named pianoRestMiddle, pianoRestRing, pianoRestPinky and pianoRestThumb store co-ordinates that correspond to the imaginary position of the two hands on the piano.

The **Point** information stored in the pianoRestIndex variables are used to create the indexData variables which are objects of type **FingerData**. Each indexData object encapsulates the instantaneous information of each finger such as the name of the finger, whether it is on a note, the distance that it is currently at, the note it is on and the **Point** position. The indexData variables are stored in a list.

The **History** class works as a memory for each hand. It stores the distance of the note that was last pressed by the finger and uses that to determine if the next note corresponds to an increase or a decrease and accordingly selects the finger.

As MIDI signals are read by **MidiMain,** the distance information is read from **PianoDistance** and converted into a **Point** data which is sent to the **Arbitrer** class. This class invokes the findClosestFinger method for the left and the right hand respectively. A **FingerData** object corresponding to the finger in each hand which is close enough to be able to play the note is returned. The **Arbitrer** then invokes the activateClosestFinger method on whichever hand is closest to the position of the note.

The findClosestFinger method at its simplest structure takes the **Point** data which corresponds to the location of the key which is to be pressed and calculates the distance of each finger in the specified hand from that point. It then sorts this distance and returns the finger which is at the least distance from the point and is currently not being used to press a note. However the sorting is applied only if the **History** class is unable to decide if the next note is a higher note than the previous or a lower note.

A cross over is necessary when the hand is progressively playing higher or lower notes. For example if the left hand keeps playing higher and higher notes it must then move from the left side of the keyboard to the right side. Once the hand reaches the thumb finger, to play the next higher note the middle finger must cross on top of the thumb and occupy the note.

Similarly a cross under occurs when the hand keeps playing lower and lower notes. Once the little finger of the hand reaches a note, to play the next lower note the thumb cross under the little finger.

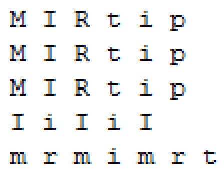
Cross over and cross under styles are commonly seen in the C major scale. For the system to understand if this style is to be played the **History** object of each hand must be able to decide if the new note is a higher note than the previous or a lower note. Once the **History** class makes its decision, the findNextAdjacentFinger and findNextNonAdjacentFinger methods are invoked to ascertain which finger to use.

The structure described works provided the first finger that gets pressed when no fingers are on the piano can be determined with accuracy. But for practical purposes the first finger that is pressed is more a matter of personal preference and style and therefore impossible to judge. Certain heuristics can be applied such as

- Giving the index or middle finger higher preference if no other finger in the hand is on a key or if a sharp note is pressed.
- Using the little finger if the note pressed is close to the left or right extreme of the keyboard.
- Ensuring that if the note is on the left side of the piano then the left hand is used to play it provided the left hand is free to play the note.

Despite these heuristics, if a note is located in the center of the piano, it becomes impossible to tell which finger to use first. For this purpose another experiment was conducted to test how well the system works if the note information and the fingers used to play it are known and provided to it as a script. This is similar to lip syncing for actors.

In figure 2.4 it can be seen that the **Arbitrer** class contains a **ScriptParser** object. This class contains a text file which is a script which contains information about which finger to press and in which hand. Currently the script used in the system is shown in figure 2.6.



```
M I R t i p
M I R t i p
M I R t i p
I i I i I
m r m i m r t
```

Figure 2.6 Sample of the script

The letters stand for the first letter of the finger (I for Index etc) and the case is used to specify a particular hand.

Each time a note is pressed on the piano the **Arbitrer** transfers control to the **ScriptParser** class if there is a script and if there is data left to be read from it. The **ScriptParser** class then reads the next letter from the script and using the **FingerPosition** objects that were passed into it when it was created invokes the activateClosestFinger method.

This technique works very well, especially when several keys on the piano are pressed simultaneously, since each key press is a signal to read one more letter from the script.

2.3 IK Engine

For the purpose of this project, it was decided to use an existing IK engine rather than make a new one. After going through a few libraries it was decided to use the IK engine created by Samuel R. Buss and Jin-Su Kim [9] for their paper on the technique of Selectively Damped Least Squares for IK. Since their IK engine was written in C++, it became the de facto language in which the project was written.

The following image shows the kind of tree structure that the engine solved initially.

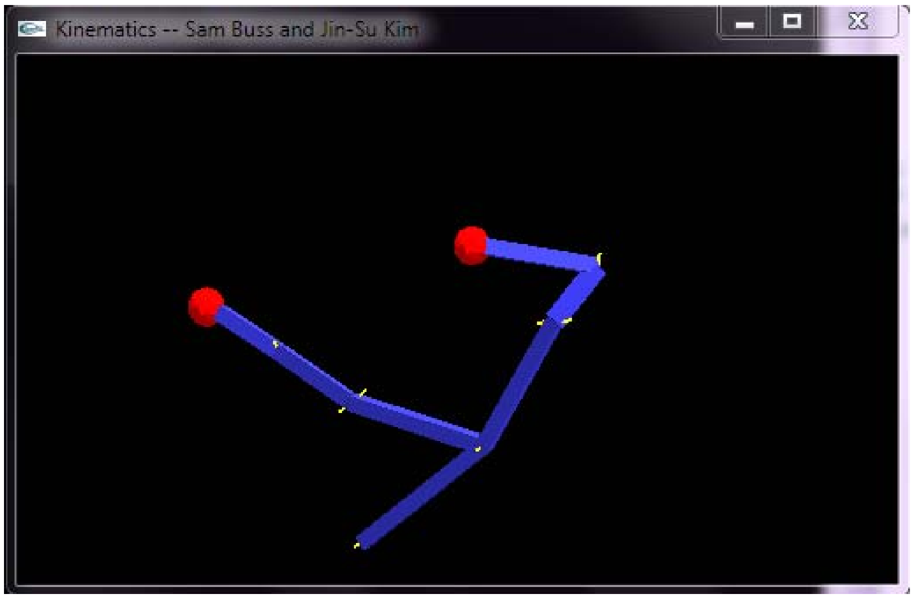


Figure 2.7 Initial tree structure solved in [9]

The yellow lines represent the rotation axis about which rotation occurs in the joint. The red spheres represent the targets which the end effectors of the tree try to reach. This provided an extremely simple interface that could be manipulated to create the necessary movement in the fingers.

In order to create a chain which can mimic the functionalities and restraints of the hand the measurements required were obtained from the paper on the Handrix [6]. The hand has three kinds of joints:

- Distal Interphalange joint (DIP)
- Proximal Interphalange joint (PIP)
- Metacarpophalangeal joint (MCP)

Rotation about the PIP and DIP joints occur in only one axis. The MCP can rotate about two mutually perpendicular axes but with restrictions in the angles. Figure 2.8 below shows the position of the joints.

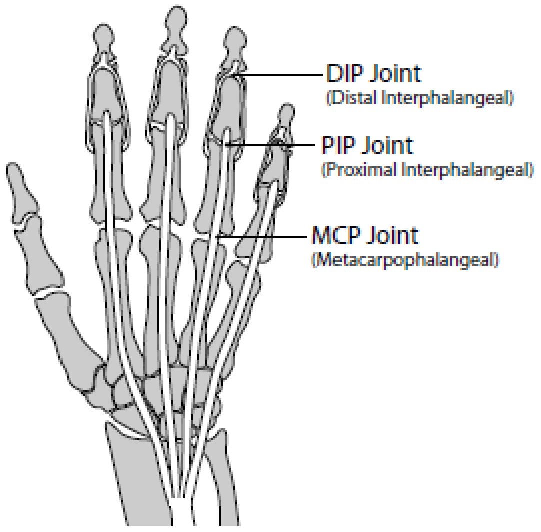


Figure 2.8. The types of joints

In Samuel Buss and Jin-Su Kim's IK engine, they used the concept of **Node** objects to create the chains. Each **Node** object has a vector representing the length of the bone, the rotation axis about which it must rotate and the minimum and maximum angles of rotation that it is allowed to rotate through. These nodes are ultimately added to a Tree object which gives rise to the IK chain which is then recursively solved.

For the purpose of this project the recursive nature of the tree was replaced by a list of **Node** objects called the metaCarpalsList which denotes the metacarpal bones. Each hand is a **Tree** object and has such a list. Each bone was given a string name such as 'index', 'middle' etc. **Nodes** were created for each joint in each finger separately with specific distance vectors and rotation constraints. The root of each such chain was then added to the list under the appropriate name.

Figure 2.9 shows a class diagram of the classes showing only the parts that were modified for this project.

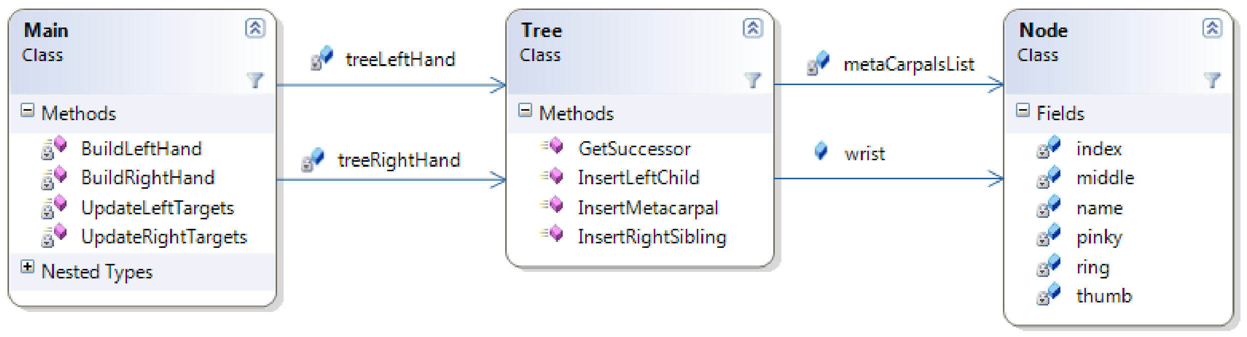


Figure 2.9. Simplified class diagram of the IK engine

The **Main** class is a continuation of the class shown in brief in figure 2.4. It is started by the **StartHand** class. It invokes the BuildLeftHand and BuildRightHand methods which create the two **Tree** objects for each hand

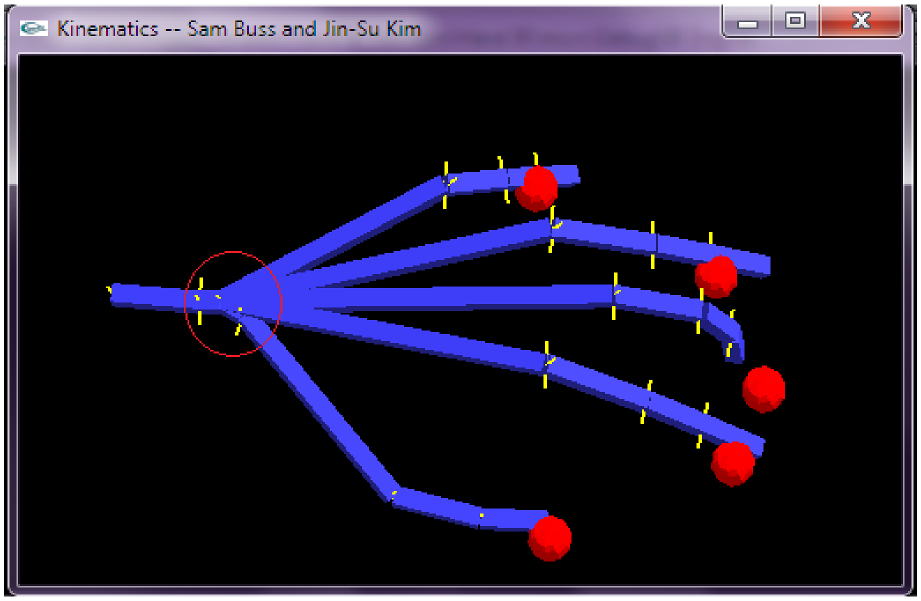
When the IK engine traverses the tree, instead of doing a recursion like in the initial library, it first iterates over the list to get to each metacarpal bone and then recurses over the node chain in that bone.

For the MCP joints which have more than one axis of rotation an initial **Node** object is added to the list with one of the axes and then to this **Node** another **Node** is added with the exact same distance vector but different axes of rotations and corresponding restriction angles.

The two images in figure 2.10 show the final result from two different perspectives with the yellow lines showing the rotation angles.

The **Tree** starts as a simple **Node** chain as understood by the IK engine. The wrist (shown with a red circle) has two mutually perpendicular axes of rotation. While rendering the wrist the IK engine has to iterate through the metaCarpalsList, where each item is another chain leading to each finger.

For each finger a red sphere has been added. Once the system determines which finger to use to play the note then the UpdateLeftTargets and UpdateRightTargets in the **Main** class methods are invoked. These methods take the **Point** position of the end effectors of each finger as input which they use to relocate the red spheres. The IK engine then moves the finger so as to reach the position of the spheres.



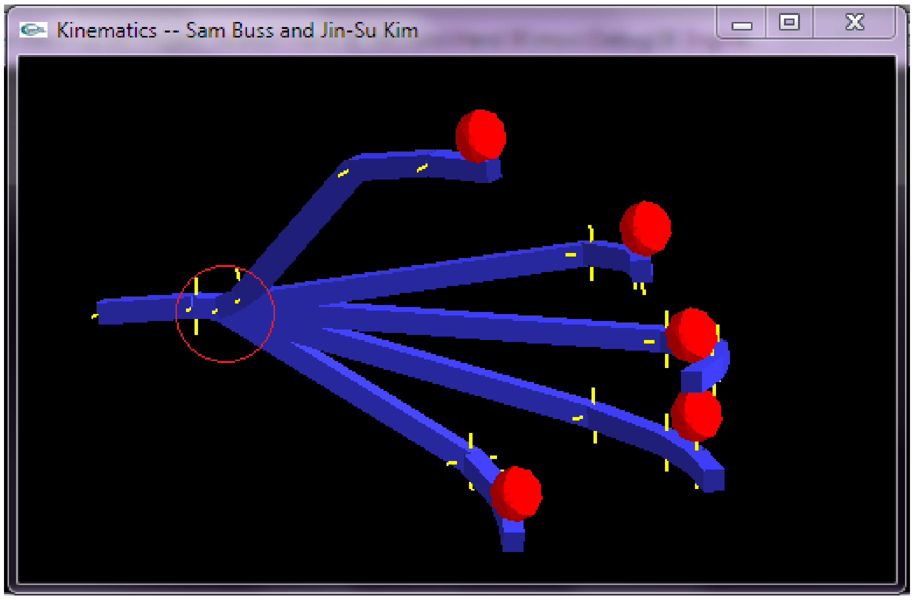


Figure 2.10. The final form of the hand

Once the left hand was created, a mirror image of it was created to obtain the right hand.

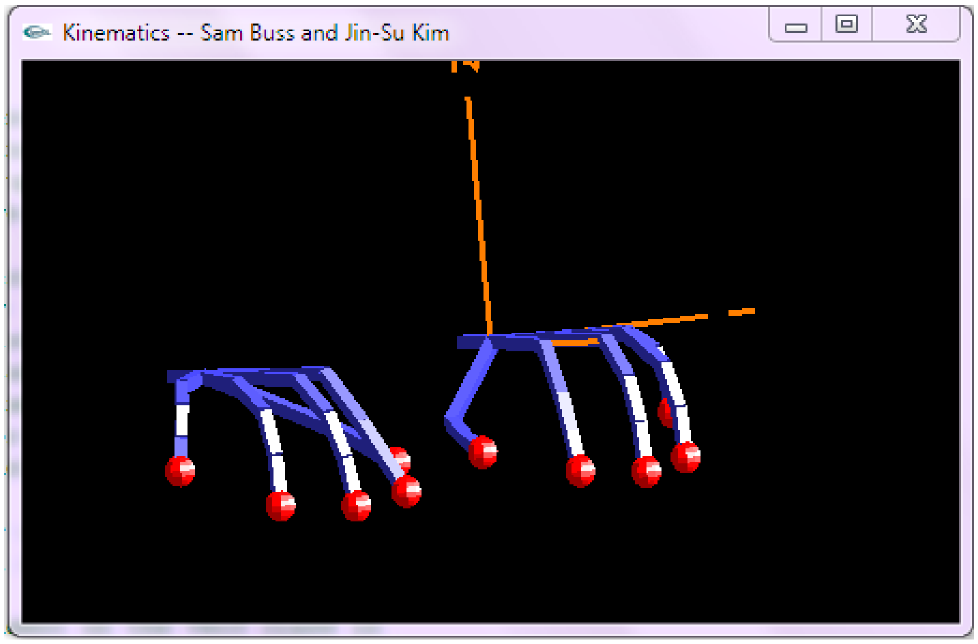


Figure 2.11 The two hands

III RESULTS

There are two modes by which the system can be used as mentioned in Section 2 viz.

- In real time but with a script which has finger movement mapped in it.
- In real time with no previous knowledge of the music being played.

3.1 Running with a script

Figure 3.1 shows the results of pressing keys one finger at a time and one hand at a time. As can be seen there is a perfect match.

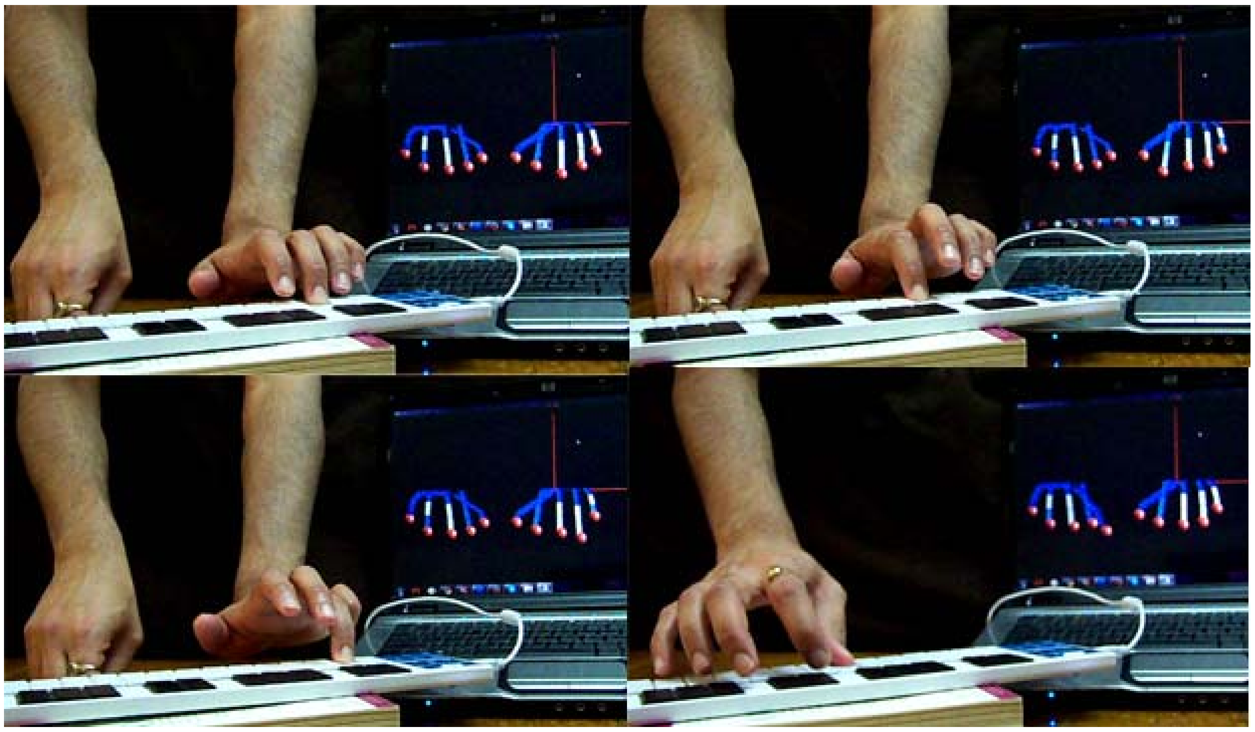


Figure 3.1 Individual fingers using one hand at a time with script

Figure 3.2 shows the results of hitting three keys simultaneously from the same hand. As before there is a perfect correspondence.

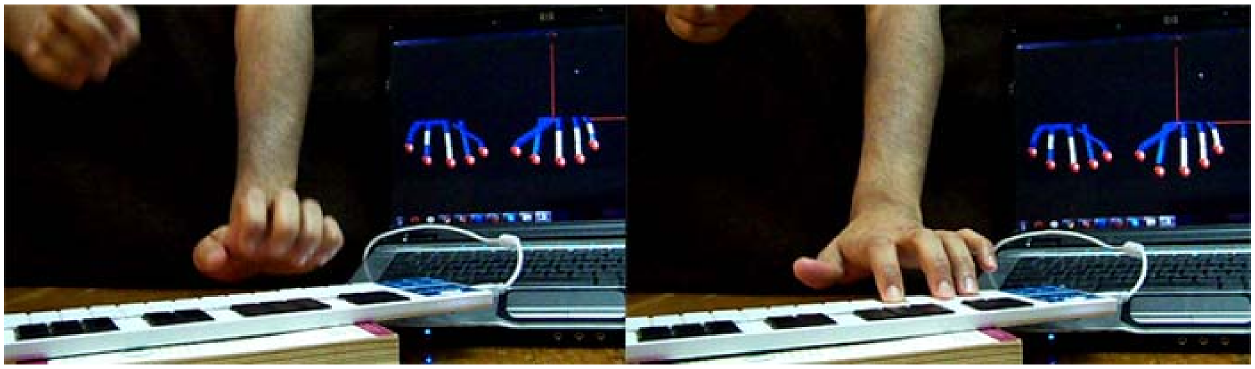


Figure 3.2 Three fingers in left hand with script

Figure 3.3 shows the result of pressing three fingers in each of the hands simultaneously. Once again there is no error.

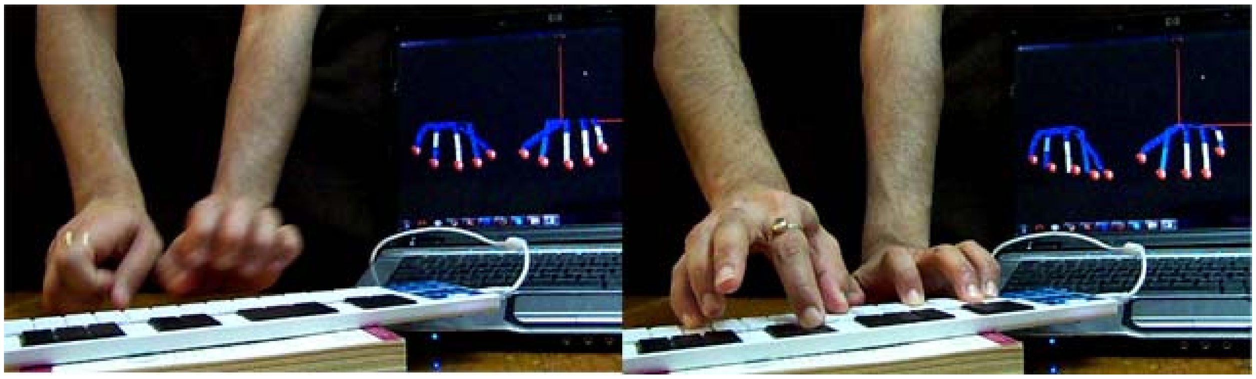


Figure 3.3 Three fingers in both hands with script

Figure 3.4 shows the effect of pressing keys individually in one hand but with each note progressively going for a higher note or a lower note.

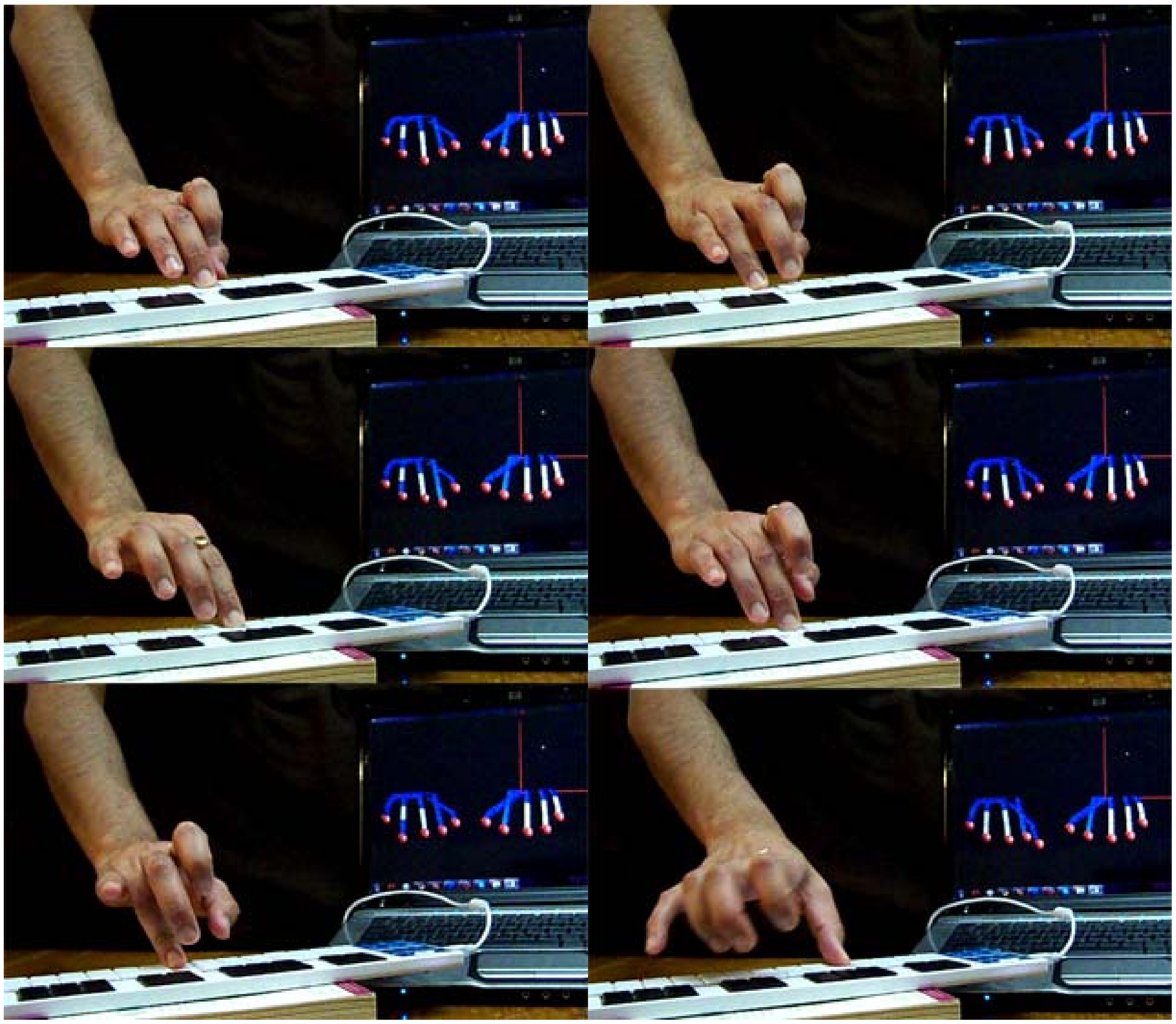


Figure 3.4 Individual fingers in a sequence with script

3.2 Running without a script

Next the results of pressing keys without using a script will be shown. The algorithm works properly when the first key that is pressed is at the left end or the right end of the keyboard as figures 3.5 and 3.6 will show.

In figure 3.5 the first key that is pressed is the little finger in the left hand at the left end of the keyboard. With this order even crossovers can be simulated. Once again there is a perfect correspondence. Figure 3.6 is a mirror image of the settings of figure 3.5.

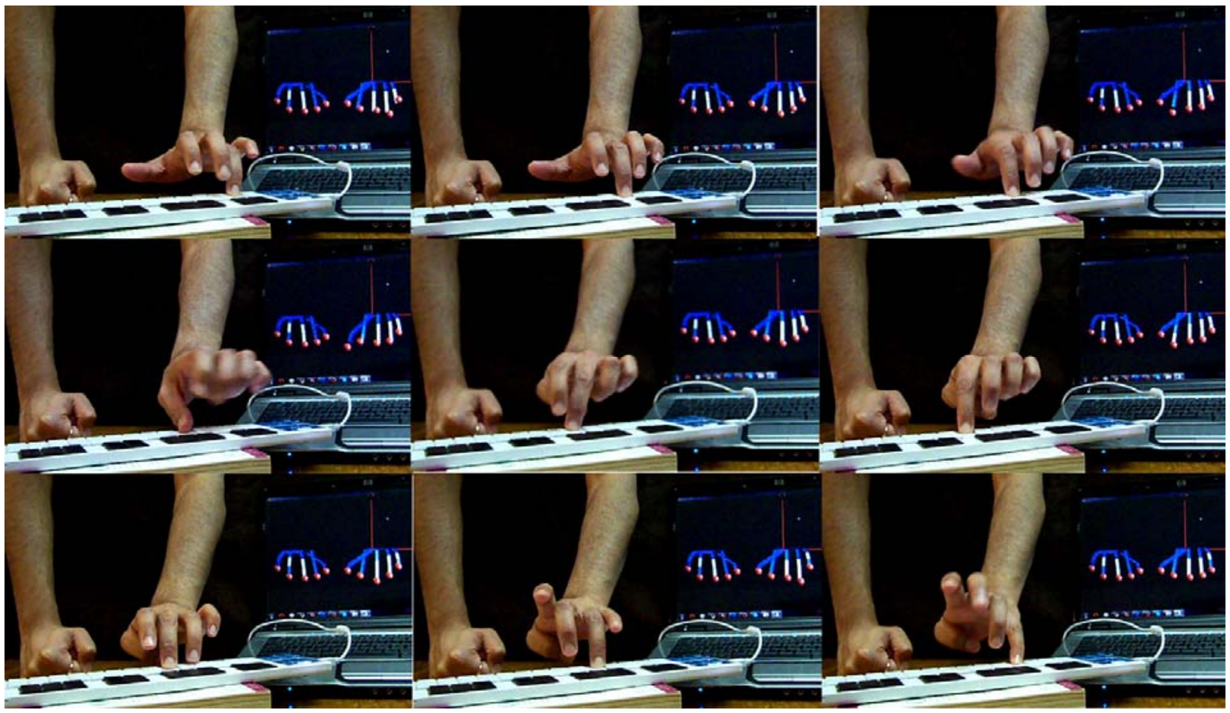


Figure 3.5 Starting from the left end with the left hand without script

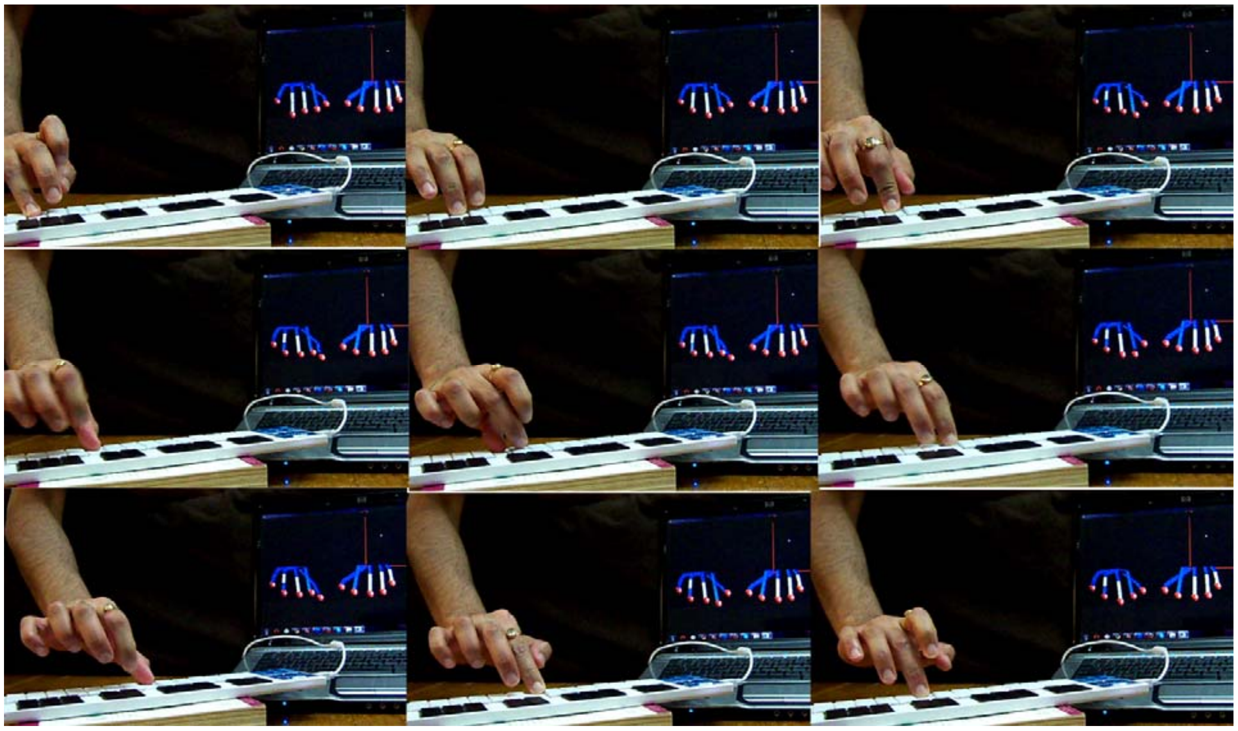


Figure 3.6 Starting from the right end with the right hand without script

Figure 3.7 shows the effect of pressing three fingers simultaneously. In each case the result does not match the finger movement but is still physically possible. In the second and third image even though the same three fingers are being pressed, yet because of the notes that they affect, the results are completely different, the system replicates the motion by moving fingers in both the hands.

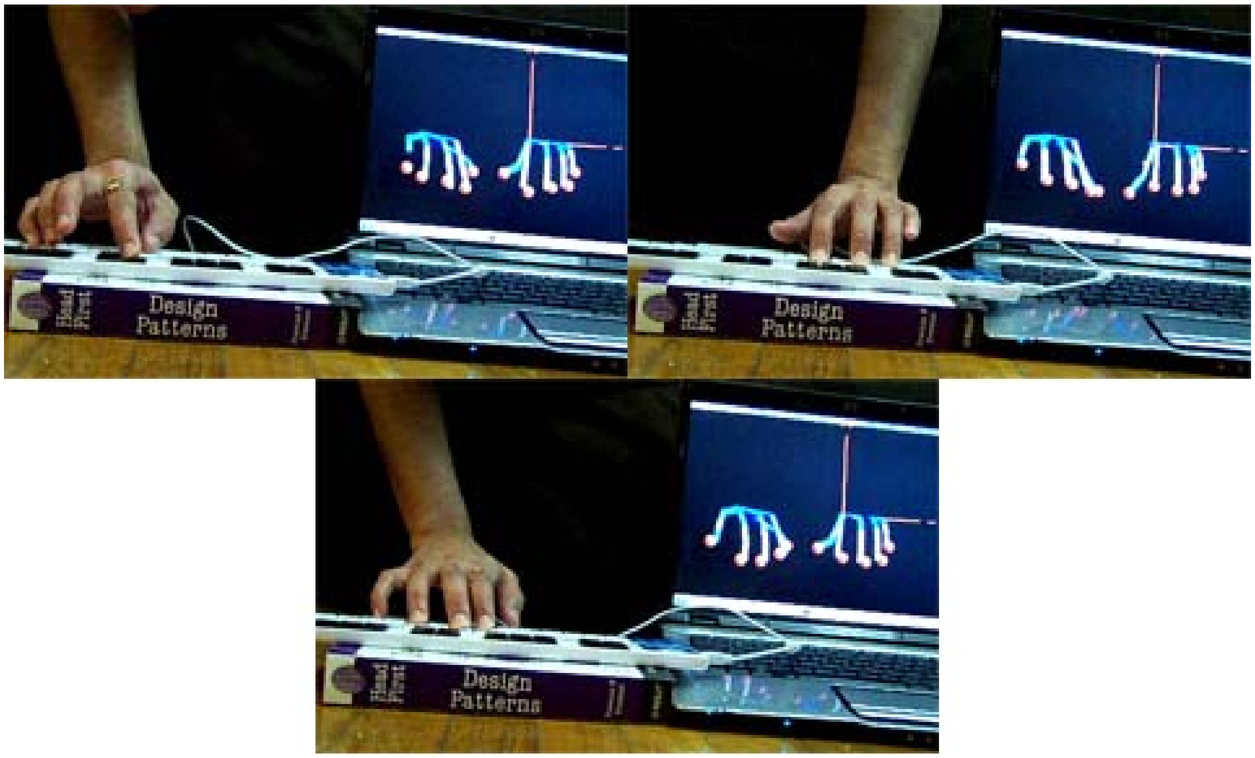


Figure 3.7 Pressing three fingers without a script

Figure 3.8 shoes another example of the effect demonstrated in figure 3.7 with the difference that in this case each finger is pressed one after the other. The middle finger of the right hand is replicated by the index finger of the right hand, the index finger by the thumb and the thumb of the right hand by the thumb of the left hand. However, this is still a physically possible movement.

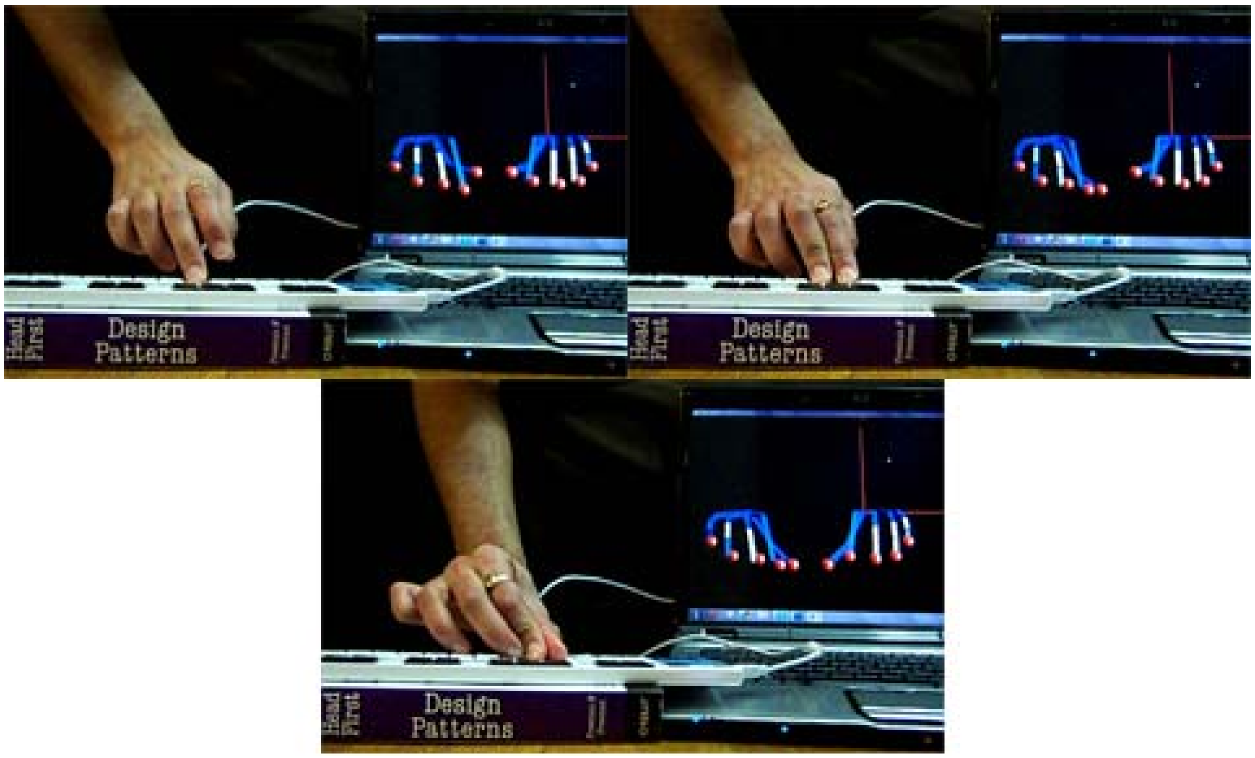


Figure 3.8 Pressing three fingers one after the other without a script

Figure 3.9 shows the same index finger of the left hand moving on two adjacent notes, yet in one case the motion is replicated by moving the index finger and in the other case by the thumb.

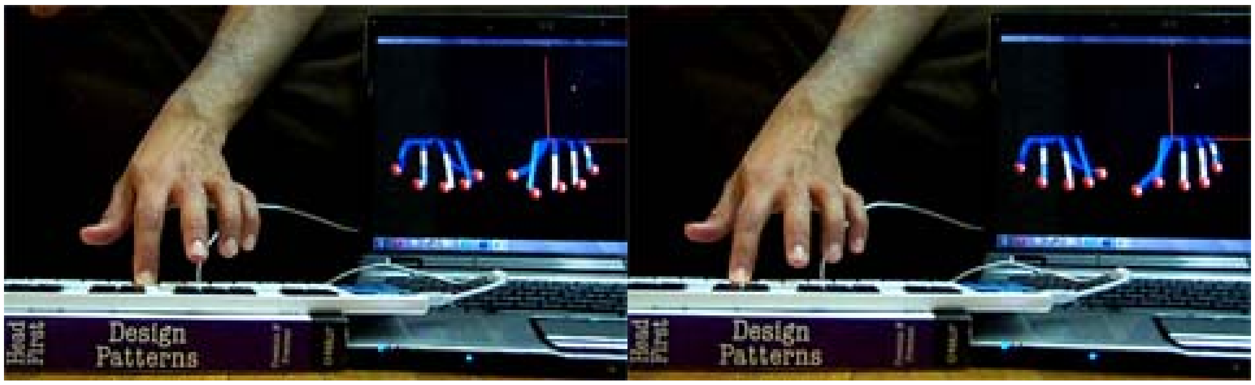


Figure 3.9 Moving the index finger on adjacent notes without a script

IV CONCLUSION

As was seen from the results of using the system without a script, creating a perfect correspondence between the actual finger movement and the ones replicated becomes increasingly complex as the left hand moves over to the right side of the keyboard and vice versa.

There are essentially two parameters that can be used to evaluate the effectiveness of the project.

- Strongly defined parameters such as response time and speed: This directly depends on the refresh rate of the IK engine and so they can be sped up or slowed down as needed.
- Weakly defined parameters such as accuracy: This is weakly defined because there can be several styles of playing the piano. What is significant in this part is that even though the system is unable to match the finger movement, yet, it produces results which are not physically impossible. For example if the middle finger of the left hand is currently on a note and the next note that is being played is a higher note, then, at that instant, there are three fingers that can be used to reproduce this correctly

    1. The index finger on the left hand.
    2. The thumb on the left hand.
    3. Any finger on the right hand provided the right hand is free to move.

  The pianist will, in most cases either choose option 1 or 2, but the system may choose any one of these options and it is not an incorrect move.

  In the example given above a wrong move will be if the system uses the ring finger or the little finger of the left hand to reproduce the motion.

V FUTURE WORK

The question then remains, whether it is possible to replicate finger motion with a more higher accuracy. One of the things that was planned during the development of this project but ultimately not carried out was to create an AI layer that would trigger the finger movement.

It is possible to create a mesh of neural networks, one for each finger using the current algorithm to represent the naive state of the application. While the idea seems to work as a thought experiment, collecting and then applying the training data is a rather complex process, which is why it wasn't tried in this project.

VI REFERENCES

[1] Geigel, J. and Schweppe, M., "What's the buzz?: A theatrical performance in virtual space," *Advancing Computing and Information Sciences, Reznik, L., ed., Cary Graphics Arts Press, Rochester, NY*, 2005, pp. 109-116.

[2] World in 3D is vision, venture of RIT graduates: http://www.democratandchronicle.com/article/20100124/BUSINESS/1240336/World-in-3D-is-vision-venture-of-RIT-graduates

[3] MotionBuilder Actor: http://atec.utdallas.edu/midori/Handouts/motionBuilder_actor.htm

[4] Shapehand :http://www.pierzak.com/motioncapture/index.htm

[5] J. Yin, A. Dhanik, D. Hsu and Y. Wange, "The creation of a music-driven digital violinist", *Proceedings of the 12th annual ACM international conference on Multimedia*, October 2004.

[6] G. ElKoura and K. Singh, "Handrix: Animating the human hand", *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, September 2003.

[7] The RtMidi Tutorial: http://music.mcgill.ca/~gary/rtmidi/

[8] http://www.planetoftunes.com/sequence/messages.html

[9] S. R. Buss and J. Kim, "Selectively damped least squares for inverse kinematics", *J. Graphic Tools*, vol. 10, pp. 37-49, 2004.