Open-Source Gesture Recognition for Virtual Orchestra Conducting Using the Kinect

Edward Ly
Earlham College
801 National Road West
Richmond, Indiana 47374
esly14@earlham.edu

ABSTRACT

Developing a virtual orchestra conducting application has been a difficult problem in computer music research in part due to the difficulties faced in gesture recognition research. Choosing the right hardware for the job is no easy task, and the amount of open-source resources available can wildly vary between each device as well. The Microsoft Kinect is one such recent gaming device that has a great amount of potential to make live virtual conducting possible. However, the scarcity of open-source libraries for the Kinect, especially gesture recognition libraries, makes open-source applications for the Kinect more difficult to develop than it should be. Therefore, we developed "Kinect Konductor," a virtual conductor application for the Kinect for Windows v1 that is dependent on nothing but open-source, cross-platform libraries. The popular freenect library extracts depth images from the Kinect, while a library called XKin extracts the hand image and classifies the gesture being performed. We then use the hand's velocity and acceleration to detect beats, and then trigger musical notes via the MIDI interface using the FluidSynth library. With these algorithms in our application, we were able to detect beats with 97.8% accuracy and play music with minimal latency.

Keywords

beat pattern gestures, beat tracking, conducting gestures, gesture recognition, music conducting

1. INTRODUCTION

The idea that one could conduct a virtual orchestra is nothing new. Indeed, as early as the debut of the Buchla Lightning in 1991, the possibility has been ever present, but the limits of existing technology make such a possibility impractical in a mass setting. Nevertheless, the problem of virtual conducting in computer music research still contains plenty of room for further development. While there have been a number of efforts made in this realm using a wide variety of available hardware, there is yet to be a virtual conductor that is powerful enough to be suitable for live performance, whether it be in a concert hall or in a recording studio. The Microsoft Kinect for Xbox 360, which debuted in 2010, is one recent piece of hardware that presented itself with some potential for finally addressing this problem, with the OpenKinect community in particular making open-source development for the Kinect possible. The motion tracking capabilities of the Kinect in particular provide an straightforward way to extract the necessary data for virtual conducting applications to function. Moreover, with a later re-release for Windows computers called the Kinect for Windows in 2012 (hereafter referred to as "Kinect v1"), we believe this device paves the way forward for further development into this research problem.

However, this does not negate the fact that the device can present itself with issues for open-source software developers. For one, the freenect library developed by OpenKinect community,[1] along with similar libraries such as OpenNI, could work only with the Kinect v1. As it turns out, Microsoft released a newer model of the Kinect called the Kinect for Xbox One in 2013, which would later be released for Windows computers as the Kinect for Windows v2 in 2014 (hereafter referred to as "Kinect v2"). Even though the Kinect v2 provides, among other things, higher video resolution, a wider field of view, and better motion tracking compared to that of the Kinect v1, features that would be attractive to researchers and developers, the data that Kinect v2 transmits is no longer in a format that freenect recognizes. Along with the fact that the Kinect v1 was discontinued earlier this year, it is no wonder that open-source development around the Kinect has become stagnant as well.

Moreover, open-source libraries that are able to perform the gesture recognition needed to detect beats are few and far between. The OpenNI framework and the NITE middleware provided one possible solution for this task until around 2013, when Apple bought PrimeSense, the company developing the OpenNI framework. In the following year, the official OpenNI website[2] was shut down, and many other associated libraries, including NITE, are no longer available for download. While the OpenNI 2 binaries and source code are still currently available,[3] NITE is not. For this reason, an open-source alternative for OpenNI and NITE must be established.

In the next section, we will highlight and compare sev-

[1]`https://openkinect.org/wiki/Main_Page`
[2]`http://www.openni.org/`
[3]`https://github.com/occipital/openni2` for the source and `http://structure.io/openni` for the binaries

ACM ISBN 978-1-4503-2138-9.
DOI: 10.475/123_4

eral papers detailing the different implementations that have been attempted while listing some of the benefits and drawbacks of each system. Afterwards, we will propose our own system that addresses some of the concerns that have been raised by using one possible alternative for OpenNI and NITE, and then test the performance of this system by measuring its accuracy and latency.

2. PREVIOUS RESEARCH

2.1 Earlier Hardware

In 2006, Lee et al. used modified Buchla Lightning II batons to create a system that controls the tempo, volume, and instrumental emphasis of an orchestral audio recording [4]. Additional control of the playback speed of the accompanying orchestral video recording has been implemented as well. The gesture recognition itself uses a framework called Conducting Gesture Analysis (CONGA) to detect and track beats, while a variation of a phase vocoder algorithm with multi-resolution peak-picking is used to render real-time audio playback. While video control will likely be outside the scope of our research, volume control and instrumental emphasis can provide an added sense of realism on top of what the Kinect already provides. In the ten years since the paper was published, however, the Buchla Lightning has been discontinued, making the CONGA framework obsolete as well. In addition, the framework itself, while touting a recognition rate of close to 100 percent, has a latency described as "acceptable for non-professionals, [but] professionals will find the latency much more disturbing" [3]. Indeed, the system has made only one public appearance in a children's museum in 2006.

In 2012, Han et al. developed their own virtual conductor system that relied on ultrasound to gather 3D positional data [2]. Hidden Markov models were implemented to process the data and recognize the gestures that would then control tempo, volume, as well as instrumental emphasis. Compared to the Kinect, their model is more simplistic in that the computer only has to recognize the position at one point on a baton rather than at multiple points on the body. We hypothesize that this may reduce CPU load significantly, allowing the system to be accessible to more consumer hardware. However, there is no mention of the amount of latency that is involved at any stage in the system when the gesture recognition is touted to be reliable with about 95 percent accuracy. Further research will need to determine the amount of latency of this system and whether or not this system remains viable for use in a live performance.

In 2014, Pellegrini et al. proposed yet another gesture recognition system using RGB/depth cameras, but they use this system for the specific purpose of soundpainting, composing music through gestures in either live or studio environments [6]. Hidden Markov models are also used here to detect a custom set of gestures and to trigger a variety of different musical events. Gesture recognition for the purpose of creating custom virtual instruments is a problem in computer music research that bears a large resemblance to the problem of conducting a virtual orchestra, as both attempt to manipulate sound in some shape or form. However, when a song is already predetermined, such as in a live orchestral performance, being able to detect beats is crucial for a system to be successful.

2.2 Microsoft Kinect

One of the earliest known uses of the Kinect for the purpose of virtual conducting was in 2014, when Sarasúa and Guaus developed and tested a computer's beat-detection capabilities using the Kinect [7]. The application was built with the ofxOpenNI module,[4] a wrapper around OpenNI as well as NITE and the SensorKinect module. Human participants were also involved to contribute to the computer's learning capabilities, even though human error and time deviations had to be taken into consideration. This approach is especially useful as a starting point given that the beat-detection algorithm is about as simple as calculating the current amount of vertical acceleration and finding local minima or maxima. Their application, however, only serves to test the effectiveness of the algorithm and not yet have the music react to the change in tempo given by the live placement of beats. Our project aims at the very least to include the ability for the music to react to said beats.

The following year, Graham-Knight and Tzanetakis use the Kinect for yet another purpose, namely for creating touchless musical instruments for people with disabilities [1]. Here, the positional data from the Kinect is sent to the visual programming language Max/MSP through the Open Sound Control (OSC) Protocol for analysis and playback. While the system does react to the gestures being made, the application usually requires more than one attempt for the gesture to be recognized. Moreover, a bigger limiting factor of this system is the 857 ms average latency, which is too large to be practical for live performances. It is unclear which part of the system contributes the most to latency, whether it be the Max/MSP language or the Kinect itself. Nevertheless, our project aims to develop an application with a latency small enough that both the performers and the audience would not notice.

3. PROGRAM DESIGN

We developed "Kinect Konductor," a virtual conductor application for the Kinect v1 that is dependent on nothing but open-source, cross-platform libraries. The overall framework of the flow of data in our application is shown in Figure 1. As of the writing of this paper, the source code repository for our application is hosted both on GitHub[5] and on GitLab.[6]

3.1 Reading Data from the Kinect

We first use the libfreenect library to extract the depth images from the Kinect. We then use the XKin library,[7] which was developed by Pedersoli et al. [5], to track the location of the hand and classify the current gesture being performed. Just as the libfreenect library serves as a replacement to OpenNI, the XKin library serves as a replacement to NITE.

To visually see the hand position tracking in action as well as provide a simple GUI for the user, we took advantage of the high-level GUI interface and image processing capabilities of OpenCV.[8] For testing purposes, the XKin li-

[4]`https://github.com/gameoverhack/ofxOpenNI`

[5]`https://github.com/edward-ly/kinect-konductor`

[6]`https://gitlab.cluster.earlham.edu/seniors/edward1617`

[7]`https://github.com/fpeder/XKin`

[8]`http://opencv.org/`

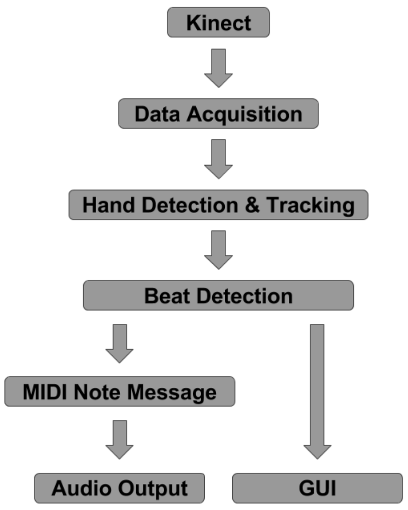

Figure 1: Overview of the flow of data in our application.

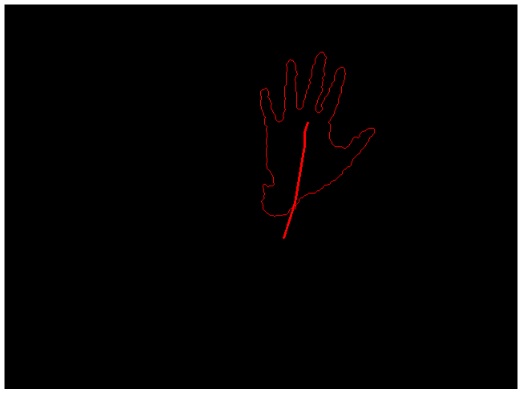

Figure 2: A screenshot of the GUI of the application.

brary provides tools for testing the detection of the hand by drawing the contour of the hand onto the screen. In our application, however, we simply extract the single point position of the hand and then draw the motion of the hand using the last few positions read. We implement a modified queue to store said points, as well as the time at which those points were recorded. Once the queue is filled to capacity, the oldest point is automatically popped before the next point is added to the queue. Figure 2 shows a screenshot of the GUI with the contour and motion of the hand inside the window.

This stream of positional data is also what allows us to detect beats. However, we opted not to use the functions provided by XKin that allowed us to track and classify gestures. A major drawback of this feature is the fact that the library needs to know when a gesture starts and ends. In this case, they are indicated by the closing and opening of the hand, respectively. This is truly inconvenient for orchestra conductors, as opening and closing the hand every beat is not only tiring for the conductor, but also not a typical part of orchestra conducting in general. Instead, we simply used XKin's ability to track the position of the hand and then detect beats based solely on the velocity and acceleration of the hand. Since this method does not track any specific motion in particular, both left-handed and right-handed conductors can freely use this application.

3.2 Producing Sound

After a beat has been detected, there are several directions one could take in terms of audio output. We first directly generated and played a simple sawtooth wave via the PortAudio library.[9] While this provides a simple solution for producing monophonic music, it introduces two major problems. The first is that the application goes to sleep in order to play any sound at all, halting any further input from the Kinect and "freezing" the GUI for the duration of the sound. The second is that because the application goes to sleep, finding the difference in time between two consecutive beats, and hence the tempo, does not take into account the duration of the sound while it is playing, so the tempo reading would be inaccurate as a result.

Due of the above issues, we then replaced PortAudio with FluidSynth,[10] which allowed us to send MIDI note messages to an internal synthesizer and sequencer for playback. This solution not only alleviates both problems found from using the PortAudio library, but provides a framework for playing any sound or combination of sounds imaginable and at any point in time. Due to this, our application now allows for the ability of the user to conduct music of any genre as well.

Each time the music reaches a certain beat, we send some set of MIDI messages to the sequencer. This sequencer maintains its own clock by counting "ticks," and increments the count by one after some time has passed. By default, each tick is one millisecond apart. It can then schedule these MIDI messages to trigger at a certain time after some number of ticks have passed. Some messages may occur precisely on the beat, so those messages are immediately sent to the synthesizer for processing and playback. Some messages may want to occur, say, an eighth note after the current beat, or about halfway in between the current beat and the next beat. In that case, the sequencer keeps the messages for some specified duration, usually before the next beat has arrived, before releasing them to the synthesizer for playback. However, the duration between any two consecutive beats may vary wildly, whether it be due to flawed beat detection or to artistic creativity. For safety and simplicity, we then take the average time over the past five beats in order to make the tempo more stable.

As for specifying the messages themselves, and hence the music, such information is usually stored in a separate file, typically a MIDI file. A MIDI file, among other things, consists of a series of "note on" and "note off" messages. Each of these messages is also associated with a number indicating when it is triggered, in milliseconds after the start of the music. A MIDI "note on" message consists of a channel number, a note number, and a velocity value, telling us what instrument to play, which note to play, and at what volume, respectively. A "note off" message is the same as a "note on" message, except that the velocity value is always set to 0. For our application, the two variables that remain constant are the channel and note numbers. The timing and velocity of each note, however, are determined in real-time by each

[9] `http://portaudio.com/`
[10] `http://www.fluidsynth.org/`

gesture read by the Kinect. A MIDI file alone, however, may not be able to tell us on which beats a certain note starts and stops. Therefore, we used our own CSV files in a custom format instead of MIDI files, replacing millisecond timing with beat numbers and removing the velocity values altogether.

MIDI note messages alone, however, still do not tell us which instruments map, or connect, to which channels, as one could change this mapping at any time via program changes to load the desired instruments to play. Furthermore, these instruments are not embedded into any of the messages themselves, but rather loaded into the application from SoundFont files, which includes the information needed for a synthesizer to produce various sounds for the various instruments they contain. Therefore, when running the application, one must also specify the SoundFont file to use as well as the mapping of each channel to the desired instrument. SoundFont files that conform to the General MIDI standard, such as the Fluid (R3) General MIDI SoundFont in Linux, are generally enough to produce music of almost any genre, including orchestral music.

3.3 Playing Music

A CSV file that our application can read must have a specific set of numbers separated by whitespace in order to play any music at all. These numbers store all of the program changes and all of the note messages that the MIDI protocol needs in order to play music. The first line of the file consists of three numbers, which tell the application how many program changes and how many note messages there are in the music, as well as the PPQN (pulses per quarter note) value of the music, in that order. The PPQN value subdivides each beat into smaller units of time, usually called pulses or ticks, so that we can schedule MIDI notes to start and stop at a certain tick rather than just at a certain beat.

Next, the program changes are listed as pairs of numbers, specifying the MIDI channel number to change followed by the program number of the instrument specified by the SoundFont file. As MIDI hardware supports up to 16 channels, the acceptable values for the channel number range from 0 to 15. Similarly, the General MIDI specifications define 128 possible instruments, so the program number can range from 0 to 127. For an example, in our `ensemble.csv` file included with the application, we program 13 orchestral instruments into the first 14 channels, keeping them in the order that they are listed in a typical orchestra score. A summary of the program changes and instruments used is listed in Table 1. Note that channel 9 is reserved for percussion instruments, so a manual program change in that channel is not needed. In addition, not all program changes are required to be used in the music. A piece for woodwind quintet, for example, only uses the first five channels while the remaining channels can be left alone without harm, which can be seen on our woodwind quintet rendition of Beethoven's "Ode to Joy" in the `ensemble.csv` music file.

Finally, the note messages are listed as sets of five numbers containing the following information in order.

- beat number
- number of ticks after the current beat
- channel number
- key/note number
- note on/off indicator (1 for "on" or 0 for "off")

Channel No.	Program No.	Instrument
0	73	Flute
1	68	Oboe
2	71	Clarinet
3	70	Bassoon
4	60	French Horn
5	56	Trumpet
6	57	Trombone
7	58	Tuba
8	47	Timpani
9	-	Percussion
10	40	Violin
11	41	Viola
12	42	Cello
13	43	Double Bass
14	-	(other instrument)
15	-	(other instrument)

Table 1: Summary of Program Changes and Instruments for Orchestra.

It is required that each message be sorted by increasing beat number (and then preferably by increasing tick count too) as each MIDI message will be read and scheduled in the same order as in the file. One must also be careful that the actual number of note messages in the file does not exceed the message count indicated at the start of the file either. Once our application has finished reading all of the notes in the music, we simply reset the music by turning off any remaining notes left on and send our beat counter back to the start of the music to be played again.

4. RESULTS

We base the performance of our virtual conductor system on two metrics: accuracy and latency. In our case, we define "accuracy" to mean how well the beats detected by the application match up with the conductor's intended beats. Quantitatively, we can express accuracy as a percentage using the following expression:

$$\frac{b-t}{b+f} \times 100\%,$$

where b expresses the number of beats elapsed in the music, while t and f express the number of true negatives and false positives, respectively, over the elapsed period. Here, a true negative is when a beat gesture is performed, but the application does not recognize it as a beat, while a false positive is when the application detects a beat that is not actually performed.

In our testing, we played our version of "Ode to Joy", which is 64 beats long, 32 times for a total duration of 2,048 beats. Half of the time was given to conducting with the left hand and half with the right hand, and we varied the amount of movement in every single gesture. The t and f values for every single performance, as well as the total and average values, are listed in Table 2. Since we encountered 7 true negatives and 39 false positives in that entire period, this gives our application an average accuracy of approximately 97.8 percent. While this percentage may seem satisfactory at first, in actuality, this means that on average, there will

#	t	f	#	t	f
L1	**0**	**0**	R1	0	3
L2	0	2	R2	0	2
L3	0	1	R3	0	1
L4	**0**	**0**	R4	1	2
L5	0	1	R5	0	1
L6	0	2	R6	0	2
L7	0	1	R7	**0**	**0**
L8	1	1	R8	1	0
L9	0	1	R9	0	1
L10	0	1	R10	0	2
L11	**0**	**0**	R11	1	2
L12	1	1	R12	0	1
L13	1	1	R13	0	3
L14	0	2	R14	0	1
L15	**0**	**0**	R15	0	1
L16	0	2	R16	1	1
Total	3	16	Total	4	23
Avg.	0.19	1.00	Avg.	0.25	1.44

Table 2: t and f Values for Each Performance of "Ode to Joy".

be at least one false positive or true negative every 48 beats, or 12 measures in common time. Considering the fact that the length of most music around the world exceeds well over 48 beats, making even just one mistake over the course of a song would prove to be disastrous for a live conductor. Even in our testing, we were only able to record five mistake-free performances (both t and f equal 0) out of 32 tries, or 15.6 percent of the time, which are not great chances to take into a live performance.

While we should strive for 100 percent accuracy to maintain control of the music, minimizing latency is still just as important. Here, we define "latency" to mean the time it takes to play a note from the moment the corresponding gesture is performed. Admittedly, it is difficult to measure this latency with some level of precision, but over the course of development and testing, we observed that the latency was never large enough to be noticeable. Due to this characteristic, our application has better latency than most virtual conductor systems before it.

5. CONCLUSION AND FUTURE WORK

While our application performs well in terms of latency, a number of improvements can still be made to better the accuracy of our application as well as the algorithms implemented to process data at each step. As previously mentioned, the XKin library has plenty of room for improvement in two major ways. The first is in the hand detection algorithm, as there are times when other nearby objects or even the walls of a small room are mistaken to be part of the hand if they are in the same depth plane as the hand. Even with a removal of objects and a bigger room, however, other parts of our body, most commonly the arm, can still be mistaken as part of the hand as well. The second is that the provided method for learning the start and end of a gesture is not suitable for all applications, and does not take into account gestures based on timing. We believe that further research and development for this library should focus on both of these areas, and that improving on these areas would surely lead to better accuracy as a result.

Aside from the XKin library, another improvement that can be made in our application is in our beat detection algorithm, as the current method for calculating velocity and acceleration is simplistic but crude. While calculating only two velocity values and one acceleration value each frame could still suffice, a large difference in volume between two consecutive beats can be problematic if the acceleration value is not precise. A more robust algorithm would take the entire path of the hand into consideration while also self-correcting any stray points should the hand contour at any given frame ever produce unexpected results. In addition, our application has only been compiled and tested under Ubuntu Linux 14.04 LTS, but in the long run, the cross-platform nature of all libraries used should allow for future work to add compatibility for Windows and Mac computers as well.

Another issue of our application is not from within the application itself, but rather from the CSV files that the application reads. When composing a new song or adapting an existing song, keeping track of every single MIDI note message in the music can be a daunting task, especially as the count gets larger with each new instrument and each new beat of the music. Furthermore, even though our CSV files have a similar structure to that of actual MIDI files, there is not yet any way to adapt existing music from MIDI files into CSV files. For this reason, a tool that can convert between these two formats can help reduce much of the time spent copying music for conducting.

In the end, we developed an open-source application that allows the user to conduct not just orchestral music, but any genre of music one desires. As long as the music can conform to whatever sound font one chooses to use, one can compose and playback music using our software. Although the Kinect for Windows v1 may no longer be in production, the research using this device for a variety of applications has not. If a similar device with better hardware were to be released in the future, that would only make the idea of a mass virtual concert one step closer to reality.

6. ACKNOWLEDGMENTS

I am grateful to my co-advisors, Forrest Tobey of the Music Department and Charlie Peck of the Computer Science Department, for their support for this project as well as their advice that helped shape the scope and direction of this research project. In addition, Xunfei Jiang of the Computer Science Department and Sierra Newby-Smith of the Earlham College Writing Center graciously provided feedback and help during the writing process of this paper. I am also grateful to Fabrizio Pedersoli [5] for developing the XKin library and providing the necessary tools to make our application a reality.

7. REFERENCES

[1] K. Graham-Knight and G. Tzanetakis. Adaptive music technology using the kinect. In *Proceedings of the 8th ACM International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '15, pages 32:1–32:4, New York, NY, USA, 2015. ACM.

[2] S. Han, J.-B. Kim, and J. D. Kim. Follow-me!: Conducting a virtual concert. In *Adjunct Proceedings of the 25th Annual ACM Symposium on User Interface*

Software and Technology, UIST Adjunct Proceedings '12, pages 65–66, New York, NY, USA, 2012. ACM.

[3] E. Lee, I. Grüll, H. Kiel, and J. Borchers. Conga: A framework for adaptive conducting gesture analysis. In *Proceedings of the 2006 Conference on New Interfaces for Musical Expression*, NIME '06, pages 260–265, Paris, France, France, 2006. IRCAM &#8212; Centre Pompidou.

[4] E. Lee, H. Kiel, S. Dedenbach, I. Grüll, T. Karrer, M. Wolf, and J. Borchers. isymphony: An adaptive interactive orchestral conducting system for digital audio and video streams. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '06, pages 259–262, New York, NY, USA, 2006. ACM.

[5] F. Pedersoli, N. Adami, S. Benini, and R. Leonardi. Xkin - extendable hand pose and gesture recognition library for kinect. In *Proceedings of the 20th ACM International Conference on Multimedia*, MM '12, pages 1465–1468, New York, NY, USA, 2012. ACM.

[6] T. Pellegrini, P. Guyot, B. Angles, C. Mollaret, and C. Mangou. Towards soundpainting gesture recognition. In *Proceedings of the 9th Audio Mostly: A Conference on Interaction With Sound*, AM '14, pages 18:1–18:6, New York, NY, USA, 2014. ACM.

[7] A. Sarasúa and E. Guaus. Beat tracking from conducting gestural data: A multi-subject study. In *Proceedings of the 2014 International Workshop on Movement and Computing*, MOCO '14, pages 118:118–118:123, New York, NY, USA, 2014. ACM.