

Kinecting With Music: A Human Interface for a Digital Orchestra

Rachel Adams, Joe Brown, Calder Coalson, Morgan Marks, Frederick Stein, Grace Whitmore
Carleton College

Northfield, MN, 55057

{adamsr, brownj, coalsonc, marksm, steinf, whitmorg}@carleton.edu

Abstract

We have developed Virtual Orchestra, a Microsoft Kinect game written in C# that creates an interactive virtual experience analogous to that of conducting an orchestra[6]. It allows an untrained user to virtually manipulate components of the playback of a MIDI song file including tempo, volume, pausing, and starting using motion detection with gesture recognition. A graphical user interface (GUI) reflects the changes a player has created in the MIDI song by displaying instruments currently playing, volume, and the next two measures of notes. This project required the communication between three main disparate components: gesture recognition, a MIDI controller, and an interactive GUI, all of which communicate via a central dispatch.

1 Introduction

Since the middle ages, professional conductors have been guiding orchestras using nuanced gestures to indicate timing and dynamics[6]. In creating Virtual Orchestra, we faced the unique problem of translating those interactions between a conductor and an orchestra into the interactions of an untrained player and a computer. We strived to create an analogous conducting experience that allows a user to manipulate song elements like tempo, volume, starts, and stops in an intuitive way, following these established conducting gestures, yet being flexible enough to accomodate users with no

previous conducting experience.

Research in the past few decades has explored aspects of virtual conducting, including developing computer vision algorithms and creating an intuitive visual interface[5][4][2][1][3]. As we use the Kinect to map the movement of the conductors hand, we are most interested in research that focuses on conducting gestures. Krom [2] created a system which recognizes a comprehensive list of gestures, including gestures that dictate dynamics, tempo, note articulation, entrance cues, and fermata. However, as many of the recognition techniques are somewhat intuitive (using vertical acceleration of the right hand, standard deviation and mean over a sample, etc.), we independently developed recognition algorithms using the x and y coordinates provided by the Kinect.

Virtual Orchestra, as we have created it, is a composite of motion detection, gesture recognition, MIDI song file reading, preprocessing, and playback, and GUI components, all of which communicate via a central dispatcher. We use the Microsoft Kinect to detect a user's movements, which are interpreted by gesture recognition algorithms. These recognized gestures then affect the playback of a MIDI song file and those changes are visualized in the GUI.

2 Architecture

We structured the application as a series of communicating components. These components send mes-

sages to and subscribe to messages from a global dispatch. This approach was selected because it has served the authors well in the past.

The Microsoft Kinect generates 30 skeletons per second and sends these to the dispatch, where various gesture detection algorithms listen for skeleton updates, process them, and send volume and tempo updates back into the dispatch. The MIDI player and GUI listen for these messages and update the playback and graphics.

This architecture allowed us to easily test components in isolation. We could record raw Kinect skeletons in CSV files, edit them, and play them back through different gestures to observe their responses, and test song playback using fake tempo and gesture signals.

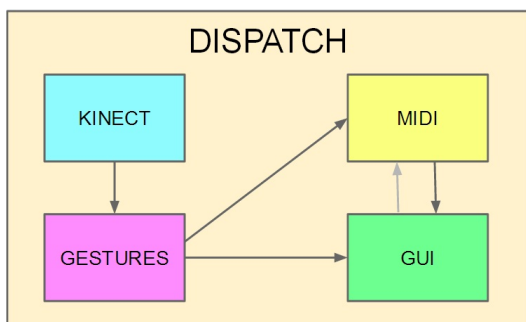


Figure 1: High level architectural overview

3 Gesture Recognition

Coding recognizable Kinect gestures was an algorithmically simple task, but each feature required a trade-off between realism and detectability. The biggest challenge was trying to balance coding realistic and recognizable conducting gestures with coding gestures that were reliably recognizable by the Kinect and flexible enough to be used by different users. They also needed to be intuitive and easy to use for first-time players. We settled on four main gestures, that we believe satisfy all of these requirements.

Tempo Gesture

Our tempo gesture is relatively simple but ef-

fective. Whenever the right hand hits a minimum, a beat is triggered. This means that a player can conduct using any gesture desired, as long as their hand is at its lowest point when a beat should be triggered. It allows a high level of flexibility on the part of the user to be able to conduct however they would like, while still accomodating traditional conducting gestures. However, the issue with our simple conducting gesture is that it cannot influence the music until a beat is made, so if there is a large change in tempo over one beat, the music will keep playing at the previous speed until the next beat is registered.

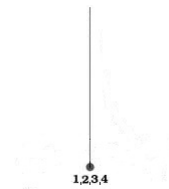


Figure 2: A basic gesture pattern that allows a player to trigger tempo beats.

Tempo Volume Gesture

Our tempo volume gesture uses the same basic code as the tempo gesture, but it also keeps track of the size of the user's gesture. They must be conducting with a 4/4 gesture, but from there, the larger the gesture is, the louder the music is. This is an option that can be chosen, instead of the default left-hand volume gesture.



Figure 3: The standard 4/4 gesture pattern that must be used in the tempo volume option.

Volume Gesture

The trickiest part of our volume gesture was teaching the computer when to start and stop recognizing the gesture. When the hand is above the hip, an acceptable box of (x, y) values is created based on the positioning of the hand. While inside the box, the user simply has to raise his left hand to raise the volume, and lower his hand to lower the volume. If the hand moves outside of this box, the player's movements will no longer be recognized.

Stop Gesture

Our stop gesture began with an attempt to get the Kinect to recognize a circle. When it was discovered that due to inherent resolution issues with the Kinect, it would not be possible, we considered other gestures that would be as natural for a user to use and easier to recognize. We settled on a gesture where both hands must be at the same height, and move down and away from each other at roughly the same velocity. When the user's hands pause at the bottom of this trajectory, the song playback stops.

4 MIDI Manipulation

MIDI (Musical Instrument Digital Interface) is a communication system for controlling the playback of pre-recorded sound files. In our setup, thousands of samples of single notes played by individual instruments are compiled and stored in a SoundFont file. Virtual Orchestra is designed to read the MIDI file, analyze and manipulate the MIDI commands, then send that data to both the SoundFont for audio, and the GUI for visualization.

4.1 Reading the MIDI file

The MIDI portion of the code begins by reading a .mid file. The MIDI format was released in 1983, and has become a ubiquitous standard. MIDI does not communicate sound information. Instead, it sends data to a SoundFont, which generates the

sound. MIDI is essentially sheet music for a computer. In practice, it is an efficient binary encoding, based on chunks. Chunks begin with a two-byte ID, and are followed by a number of data bytes. Most of the IDs also contain channel information. A channel is similar to a musical "part", like second trombone, or first clarinet. There are sixteen available channels to be populated by instrument parts. If a musical piece has more than sixteen parts, there are commands to switch out which instruments are playing on which channels. For our purposes, there are only three important ID's.

0x9* Command to turn a note on. The **9** signifies a Note-on command, the the second byte signifies the channel that receives the command.

0x8* Command to turn a note off. The **8** signifies a Note-off command. The second byte signifies the channel that receives the command.

0xC* Applies an instrument to a specific channel. The **C** signifies an instrument change, and the second byte signifies the channel to set the data to.

0xB* + 07 Change the volume on the specified channel. There are over 50 unique B data commands, but volume is the most important to us.

0xFF Command signifying a meta message. Meta messages can change how the song as a whole is played.

0xFF 51 signals that the following data contains information about a tempo change, specifically, the number of microseconds per tick. The tick is the smallest timing interval recognized by MIDI. The rate of change of ticks is proportional to the tempo dictated by the user's gesture.

0xFF 59 indicates time signature data, including ticks per quarter note.

4.2 Manipulating the data

We use a software package called **Sanford.MIDI** for C# to read in and write out MIDI data[7].

However, rather than write out what is read in, we need to alter the volume and the tempo of the song based on the conductor’s gestures, and the GUI needs comprehensible input. We make two major changes to the raw MIDI data:

Meta and Volume Messages need to be suppressed. In order to give the user full control of volume and tempo, we modified Sanford.MIDI so that it would not send any of these commands.

The GUI needs input. Note-on commands followed by note-off commands only work for audio; the player only needs to know which notes to play as they play them. Our GUI, specifically the piano roll, depends on knowing a bit about the future. We decided to create a new data structure that contains note-on commands paired with durations. This new structure information is stored in a dictionary and passed to the GUI. As the audio plays, temporal information is sent to keep the audio and GUI synchronized.

4.3 Transformation into Sound and GUI

Sanford.MIDI keeps the MIDI data in what it calls a sequencer. Once we have stripped out all of the meta and volume commands, the sequencer is ready for playing. When the user issues the start gesture, the sequencer plays. The sequencer begins at tick zero and proceeds through the song at the user-defined tempo. When the current tick matches any number of MIDI events (note-on, note-off, instrument change, etc.), they are loaded into a MIDI buffer and sent out to the soundfont to be rendered into actual sound.

Meanwhile, the user interface is being updated to synchronize with the music.

5 User Interface

Our user interface consists of a song selection menu (Figure 4), and a gameplay window (Figure 5).

The song selection window shows all available songs. When a song is clicked, it displays the album art and plays a short clip of the song until the user clicks “Confirm,” starting the gameplay window.

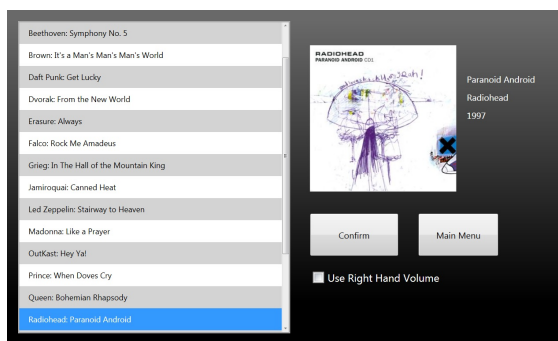


Figure 4: The song selection menu displays song options, displays album art, and plays a short clip of the song.

The gameplay window features images of each instrument in the song that become opaque when the instrument is playing and transparent when it is not, a piano roll note visualizer with note color corresponding the instrument background color so the user may easily identify which instruments are playing which notes, and a volume level indicator.

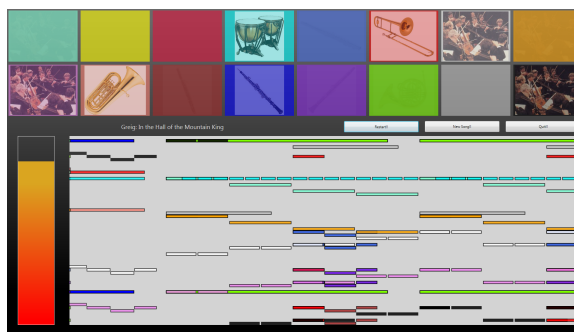


Figure 5: The gameplay window features a piano roll note visualizer, images of the instruments currently playing, and a volume indicator.

Once a song is selected, the GUI receives a pre-processed dictionary of events that happen at given ticks from MIDI, via the dispatcher.

MIDI also constantly sends the GUI the current tick, which enables the GUI backend to search for that tick in the dictionary to determine whether an event is occurring. This allows us to determine instrument image placement and background color, and piano roll note color based on the channel in which an instrument plays. An instrument's image opacity is proportional to the velocity of the note playing at the current tick, which allows the image to dim and brighten according to the intensity of the note.

The piano roll visualizer takes in a two-measure buffer of notes to display, and the speed of the scrolling is determined by the user's conducting tempo, reflected by the speed at which the current tick is changing. When a user alters the volume level, that gesture is interpreted by MIDI which in turns sends the message to the GUI, where the volume indicator's height and opacity are then set relative to the volume level.

6 Conclusions

With the addition of an interactive tutorial that provides indications of where the user's hands should be to perform each of the gestures, untrained users were quickly able to learn the basics of conducting and play the game. Our product could potentially be enhanced by the use of more sophisticated machine learning techniques and curve fitting techniques like Fast Fourier Transforms.

7 Acknowledgements

We would like to thank our research advisor and Carleton professor, Dave Musicant, for guidance and unending patience in this project. We would also like to thank Mike Tie, the Carleton College computer science department, and everyone who

put up with our arm-waving and MIDI tunes while trying to work.

References

- [1] Tommi Ilmonen, *Tracking conductor of an orchestra using artificial neural networks*. 1998.
- [2] Matthew Wayne Krom, *Machine Perception of Natural Musical Conducting Gestures*. MIT, 1993.
- [3] Michael A. Lee, Guy Garnet, David Wessel, *An Adaptive Conductor Follower*. International Computer Music Conference, 1992.
- [4] Declan Murphy, Tue Haste Andersen, and Kristoffer Jensen, *Conducting Audio Files via Computer Vision*. Proceedings of the Gesture Workshop, 2003.
- [5] Andrea Salgian, Micheal Pfirrmann, and Teresa M. Nakra, *Follow the Beat? Understanding Conducting Gestures From Video*. The College of New Jersey, 2007.
- [6] H. C. Schonberg, *The Great Conductors*. Simon and Schuster: New York, 1967.
- [7] Leslie Sanford, *C# MIDI Toolkit*. MIT, 2004.