Keyboard Music Player

Daniel Weber (dw475), David Chu (dc788), David Zhang (dyz3), Stephen Buttolph (sjb274)

System Description

We created a system that allows users to play complex songs and other pieces of music using nothing but their computer keyboard.

Key Features:

- The ability to trigger certain sounds with different keys.
- A GUI with a keyboard layout that give visual feedback on what key was pressed
- A file chooser to easily load a song configuration or midi file from a json file
- An midi player to play back a song at different speeds and scrub through a song
- Basic sound synthesis using a few different waveforms and effects.

How to install

We have supplied 2 install scripts in the `install` folder. The `vm_install.sh` script has been verified to successfully setup the environment on a fresh version of the course vm. It should also work for any linux machine that has the basic course ocaml installations. The `mac_install.sh` script should work on a mac, but we can't make any guarantees.

Then, in order to run the project, `cd` to the `src` folder and run `make main`. It should start up the GUI.

If the window is too big, you can set the window width and height from the command line. Running `./main.byte` will set the window size to be 1280 by 720. Running `./main.byte <width>` will calculate a reasonable height from the width. Running `./main.byte <width> <height>` will set the window width to be the given width and height.

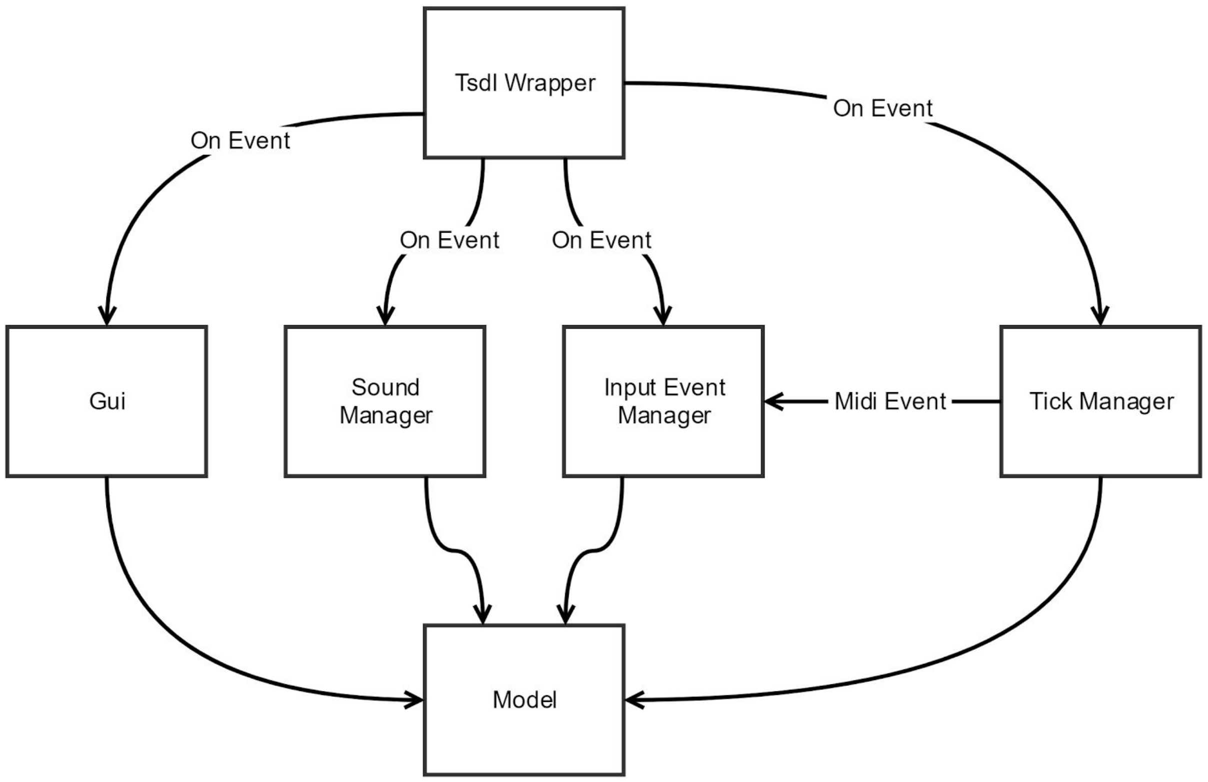
How to Use

- The initial mode of the system is Keyboard mode, which lets you play the default song file by pressing sequences of keys on your keyboard.
- Don't know how to play the song? Our midi player can help you out. Press the play button to start playing the default midi file, which will show you what keys to press and will also play the song for you.

- Is the midi player going to fast for you or did you miss something? No problem, use the scrub bar to scrub backwards or forwards in the song. Then, use the BPM slider to slow down the playback, so that you can figure out what keys are pressed. If you want to learn to play, start slow and then work your way up to full speed.
- Bored of the current song? Press the load button to select a different song file. This will automatically load in the first midi file for that song. Notice that there are multiple midi files for each song? That's because each song has multiple parts to it with different sounds mapped to different keys. We call each part a soundpack, and you can change the current soundpack by using the arrow keys. The midi file will change soundpacks for you automatically.
- Interested in some sound synthesis? Click the synth button to go to Synthesis mode. Each row of keys on your keyboard is now mapped to an octave, and each key in each row is mapped to a note in the octave. We have a few effects for you to play around with. You can select the waveform that you want and adjust the Attack-Decay-Sustain-Release (ADSR) envelope of the sound. You can also add a filter to the sound to create some interesting effects.
- Notice those bars in Keyboard mode when you play a song? Those are a visualization of the Fast Fourier Transform (FFT) of the audio buffer. What it does is it attempts to deconstruct the audio data into its fundamental frequencies and figures out the amplitude of each of those frequencies. So you'll notice that if you play a very low sound, the bars toward the middle will increase in amplitude. If you play a sound with a higher frequency, bars further away from the middle will increase their amplitudes.

System Design

Control Flow Diagram



Our system is mostly event driven, and all events originate from the Tsdl_wrapper module. In main.ml, we initialize the Tsdl_wrapper and set up all of the callbacks the the wrapper will call. When the wrapper receives a keyboard event, it calls the keyboard event callback with the event that was generated. When the wrapper receives a request to fill an audio buffer, it passes that request to the audio callback. The wrapper determines the refresh rate of the GUI, so when the GUI needs to be refreshed, it will call the draw callback. Additionally, about every millisecond it will send out a tick callback, which manages quick processes that must be called often, such as updating the metronome.

Initially, we had the data for our various systems stored inside of the corresponding system. For example, the sound manager used to contain the currently playing song. However, we soon started to run into abstraction and circular dependency issues when we had to access specific data from outside of its original module. So we redesigned our system around a model module, being loosely inspired by the Model-View-Controller architecture. The model module acts as a centralized repository for all of the data that our system uses. If the data is mutable, then the model has setters as well as getters for the data. The view was the gui module and its components, which simply query the model for the current state that it needs to draw and then draws that state.

The keyboard event controller is the input event manager, which works with the keyboard layout to generate key events and passes mouse events to their appropriate handlers.

When a key event comes in, the input event manager first passes the raw keycode through the keyboard layout which tells the input event manager what kind of event the keycode represents. Then it updates the keyboard state in the model and sends the event to the sound manager, which manages the currently playing sounds. For mouse events, we match on the state and then redirect to the appropriate button handlers.

The audio controller is the sound manager module, which has an internal state of currently playing sounds that get updated on key events from the input event manager. When the audio callback is called, the sound manager fills the supplied audio buffer base on the sounds currently being played. It also cleans up sounds that are done playing by flushing them from its list of currently playing sounds. When the system is in synthesizer mode, the sound manager

We also have a controller that handles the playback of a midi file. Due to circular dependencies, most of the logic for midi playback control was stored within the tick callback, which would decide what state the system is currently in (playback, pause, stop, load, or scrub) every frame and change the appropriate numbers in model, increment the metronome, and play the current midi file if necessary. GUI changes were based on the flags in the model set by the tick callback.

A complete control flow diagram is attached at the end of the document.

Data

Our system maintains two sources of data, the first is an internal source of data stored in records. The internal data primarily consists of an in memory representation of a song and the sounds that go with that song. There is also an in memory representation of a midi file, which is simply an ordered list of notes and the times at which to play those notes.

The external source of data that our system uses consists of a keyboard layout, song configuration files, and midi files for each song. The standard keyboard layout is stored as a json file made up of multiple elements. The first element is the layout of the keyboard that will be presented in the GUI. Since it will be presented in a 4x12 format visually, it is stored as a 4x12 2-d array. The order of the elements in the outer and inner arrays match the layout of a standard qwerty keyboard. The elements of the inner arrays will either be the ASCII decimal representation of the character at that specific location on the keyboard or another numerical representation for a special keyboard key that maps outside the ASCII range, e.g. the shift key. The other element of the json file stores the four arrow keys that will also be presented in the GUI. Similar to the keyboard element, the four arrow keys will be stored as a simple array of four numerical representations for the four arrow keys.

The second external source of data we store are song configuration files, each tailored towards a specific song. These files are in the json format and contain the name of the song stored as a string, the speed of the rhythm dictated as beats per minute stored as an integer, and an array of soundpacks. Each soundpack contains a different mapping of sounds to the keyboard keys with different settings for the sounds as well. In detail, each soundpack is made up of a 2-d array of pitches where each element is respectively mapped to the 2-d array that is the keyboard layout. Each element in the 2-d array may either be empty, meaning no there is no sound mapped to that keyboard key, or contain a series of settings that dictate what sound is to be mapped. The settings include a list of the .wav files that will be used to create the sound that is mapped, a 'hold_to_play' parameter that prescribes whether a key needs to be held down to keep a sound playing or not, a 'loop' parameter that loops the sound, and a list of group ids that the sound belongs to where in each group, there can be at most 1 sound played at any given time. This format for the song file allows for a large number of different songs to be played on our system.

The final external source of data are the midi files for each song. These midi files are json files that contain an ordered array of notes. Each note contains a key identifier which is a number between 0 and 50, a start time, and a length. Then start time and length are in units of beats, so if you adjust the bpm slider, the notes will be stretched out accordingly.

External Dependencies

We will be using the Tsdl library, as well as its sister library, Tsdl-ttf. We have made a VirtualBox install script that configures a fresh 3110 VM image to compile and run our project. Tsdl was chosen since it supported audio playback and graphics; tsdl-ttf is used to print words to the screen. Both are wrapper libraries for their C counterparts. OCaml graphics was another contender in early testing, but we soon determined that it could not capture key-up events, which are crucial for pausing audio playback. LibGTK was also tested and abandoned due to its convoluted objected-oriented structure, inability to draw words (due to a font error), and lack of support from Atom's Linter. Yojson will be used to parse files for that contain keyboard and song configurations, along with internal state.

For the song synthesis, we used and adapted some code from the ocaml-mm library. We originally wanted to use this library for sound synthesis, but we weren't able to compile it when trying to redirect audio to our speakers. So we adapted some of their code to work with the SDL audio buffers with some good results.

Testing Plan

The majority of testing was play testing. Some components like the parser were automatically tested to make sure that the file is being parsed correctly, however, because the application is mainly interactive, we had to do fair amount of manual testing. This also included making are revising design decisions about the UI. This was to make sure that the interface was as intuitive as possible to use. We also varied several internal parameters to make sure that the visual and audio portions of the interface have a low enough latency so that they don't look and sound too horrible.

Division of Labor

Who did what and how long did you spend on it. One paragraph per person

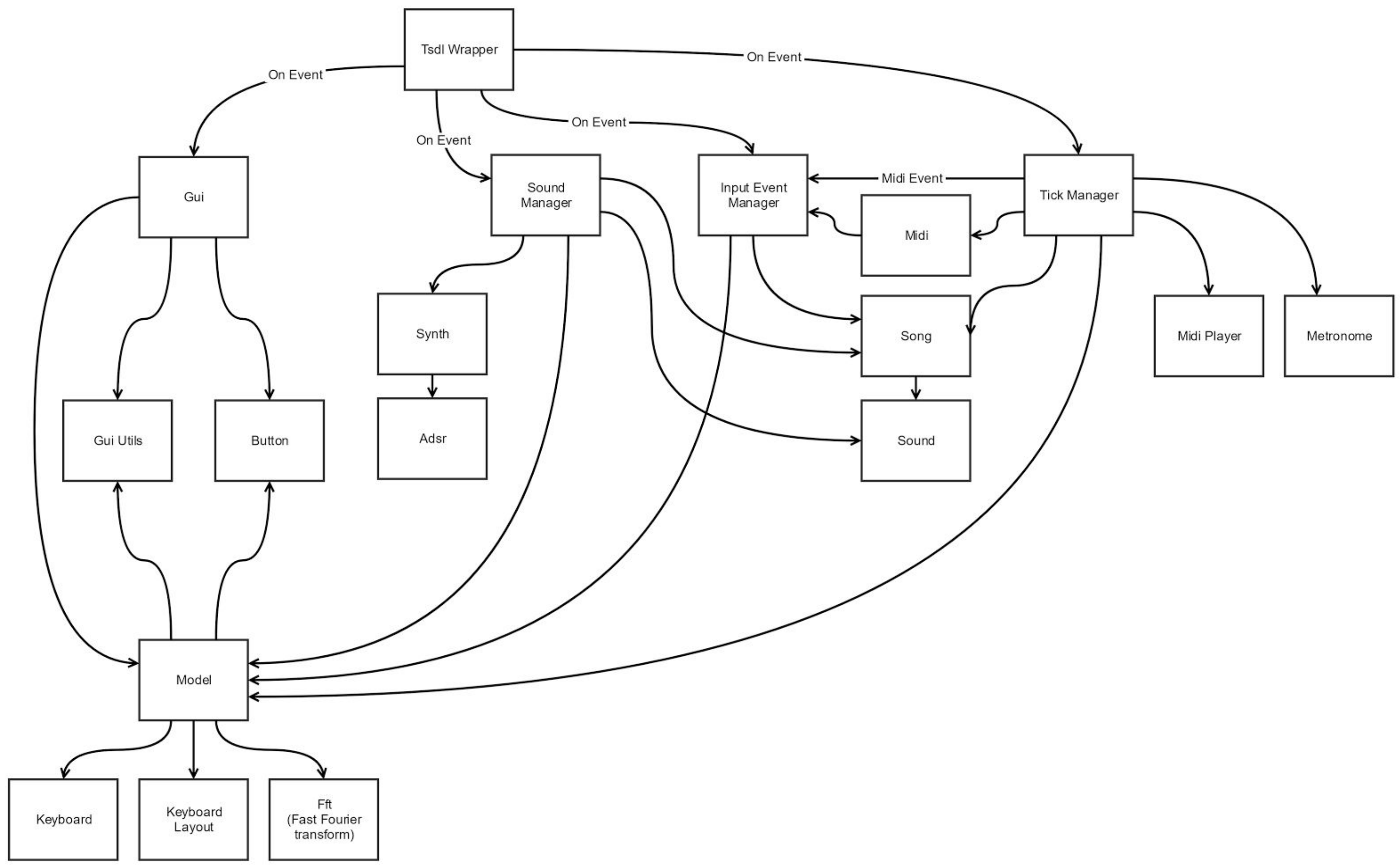
Daniel Weber worked on the tsdl_wrapper, sound_manager, and synth modules. SDL, and Tsdl by extension, does not expose a high level audio wrapper; it requires the user to fill a physical audio buffer with waveform data. This meant that Daniel had to figure out how to get the audio from the audio files that we had into memory and then into the audio buffer at the correct time. He also worked on the internal data representation for sounds and songs, which was informed by the knowledge gained from experimenting with Tsdl's audio buffer and audio callback system. Daniel also created the synth, adsr, and filter modules. He estimates he spent about 40 hours on this project, which includes the initial testing phase, the design phase, and the implementation phase.

David Chu worked on midi and metronome. He parsed the midi files into memory, used Daniel's sound_manager for playback and Stephen's keyboard to simulate key events, and handled user interaction with the play, pause, and stop buttons. He also created a scrub bar at the bottom of the keyboard to scroll to a specific section of the playing midi. He estimates that he spent about 20 hours on this project.

Stephen worked on the gui modules, keyboard modules, and event control flow. He made the gui keyboard visuals and visualized the current sounds playing in real time. He also made the gui for synthesising sounds. Stephen made a standard button module which was able to set event callbacks on a section of the gui canvas. He made the gui utils module such that buttons may use standard drawing methods. When pressed these buttons could now alter the model state. Stephen also made the amazing control flow diagram shown below. He estimates he spent about 30 hour on this project.

David Zhang worked on the file chooser and bpm manipulation. He linked up the Load button from David Chu's midi player to a new file chooser window that contained a list of all the .json midi and song files in the resources folder so that the user may toggle between songs. He also made the bpm of a track adjustable and added a visual bpm slider to allow for bpm manipulation. David also add other smaller features such as double clicking for choosing a file and using the spacebar as a hotkey for playing and pausing the midi. He estimates that he spent about 20 hours on this project.

Complete Control Flow Diagram



Tsdl Wrapper
On Event
On Event
On Event
On Event
Gui
Sound Manager
Input Event Manager
Midi Event
Tick Manager
Midi
Synth
Song
Midi Player
Metronome
Gui Utils
Button
Adsr
Sound
Model
Keyboard
Keyboard Layout
Fft (Fast Fourier transform)