**Walking Piano Handoff Documentation**

***GitHub code:*** [***https://github.com/Soundbendor/walking-piano***](https://github.com/Soundbendor/walking-piano)

***Assignments related to the project:*** [***Google drive***](https://drive.google.com/drive/folders/18Nqw7qWSTIXxFWF9BOZJtMES-dsnImXg?usp=sharing)

***Technical Documentation:*** [***GitHub Pages***](https://thricegreatest.github.io/WalkingPianoDocs/)

*Some screenshots of the game:*

A screenshot of a computer

Description automatically generated A piano keyboard with colorful squares and letters

Description automatically generated

Explanation of codebase:

For future development an understanding of the codebase is essential.

We can begin understanding the codebase by considering the program in two distinct pieces:

1. The menu
2. The piano game itself.

# The Menu

### Start.py

The menu is pretty straightforward and is handled exclusively by start.py. We utilize Pyglet for graphics, and I recommend reading some of the [Pyglet documentation](https://drive.google.com/drive/folders/18Nqw7qWSTIXxFWF9BOZJtMES-dsnImXg?usp=sharing) for help. Understanding the Pyglet library can help you implement things efficiently and effectively, and it is crucial to understand the tools at your disposal thanks to this powerful library. Alternatively, you could consult [Google's Gemini](https://gemini.google.com/app) for a simple understanding of how to implement certain things within Pyglet without needing to read extensive documentation. For example, you could gain a basic understanding by asking Gemini:

**"How do I create and display a simple rectangle shape and the word 'Hello' in Pyglet?"**

With a decent understanding of Pyglet, the menu should become quite simple. Although there is a lot of code within the start.py file, I believe everything can be understood easily. Nearly everything is managed by a single class, WalkingPianoGame. This class utilizes Pyglet functions for [creating shapes](https://pyglet.readthedocs.io/en/latest/modules/shapes.html) and [rendering text](https://pyglet.readthedocs.io/en/latest/modules/text/index.html). Additionally, we modified the on\_draw() function within start.py to handle the logic for determining what needs to be rendered. This built-in Pyglet function, [on\_draw()](https://pyglet.readthedocs.io/en/latest/modules/window.html#pyglet.window.Window.on_draw), is called every 1/60th of a second to update the display. Lastly, we use game states within the WalkingPianoGame class to dynamically update what needs to be rendered depending on what is clicked from the menu.

# The Piano Game

The piano game is more complex and is managed through two primary files:

1. midi\_processor.py
2. piano\_game.py.

It is essential to have a basic grasp of midi\_processor.py first because it handles all the logic that is critical to piano\_game.py.

### midi\_processor.py

Within midi\_processor.py, we handle reading MIDI files through a class which is then utilized by piano\_game.py to get the necessary information about any MIDI file that will be played within our game. The midi\_processor.py file includes the main() function that can be used by running midi\_processor.py to test MIDI files and listen to the song as our program interprets it. ***It is incredibly important to note that this playback does not 100% reflect how the song will sound in-game. Although the song should sound somewhat correct through midi\_processor.py, songs may sound slightly desynced. This is due to the playback implementation using Python's time.sleep(), but since it is just for testing, it is not a major issue.*** If you want to hear exactly how a MIDI file sounds, we have another file, test\_midi\_file.py, which plays the file exactly as it should be using built-in functionality from MIDO. You can then compare how that sounds versus running the song in game with autoplay enabled. Any songs can easily be trialed in-game by adding into the song database within start.py.

The most critical function of the midi\_processor.py is extract\_track\_messages(int). This function returns a list of tuples for a single track of the MIDI file containing the note\_on and note\_off messages as well as the delay between the notes in seconds, in the form (message, delay). This data is then utilized by piano\_game.py and works well for our game, allowing one-player mode to play just one track using extract\_track\_messages. For two-player mode, we can use extract\_track\_messages twice for two separate tracks (typically track 0 and track 1) to easily facilitate two players with two distinct groups of notes.

### piano\_game.py

The logic for piano\_game.py overall is complex, but here’s a breakdown to help you understand it better.

Initialization: The PianoGameUI class starts with the initialization process, where a large number of variables are set up for use within the program, and the parameters given to the class via the start menu can specify certain settings to run the game with. For example, the dimensions of the piano keys are determined based on the controller size, either 49-key or full-sized. Note management is another crucial part of initialization, where arrays and dictionaries for managing notes and their states, along with various visual elements like visibility lines and active note lines, are created.

Creating the Piano: Functions create\_white\_key and create\_black\_key are used to create individual piano keys. The create\_piano function uses these to create and position all the keys on the screen, with special labels for keys like Middle C. The piano is dynamically positioned based on the controller size to ensure proper alignment. Drawing the piano is managed by rendering both the white and black keys on the screen, including the note labels for each key. However, note labels are only drawn if the game is not in Jukebox mode to keep the user interface clean and focused on the gameplay.

Drawing the Piano**:** The draw\_piano method handles rendering the white and black keys on the screen, including note labels. This method is called by Pyglet’s on\_draw() function, which updates the display every frame. Game logic, particularly the scheduling and updating of rectangles, is handled through note scheduling, where notes are scheduled to fall at specific times based on the MIDI track messages. This uses Pyglet’s scheduling functions. Rectangle updates occur each frame, managing the position of falling rectangles and handling note timing and scoring. This is where most of the game logic is executed, ensuring smooth and accurate note-falling mechanics.

Handling MIDI Input: The play\_piano\_user method starts a separate thread to handle MIDI input in real-time without blocking the main game loop. This method highlights keys when notes are played, providing visual feedback to the player.

Loading MIDI Files: The load\_midi\_file method utilizes the MIDIProcessor class to extract track messages from a MIDI file, preparing the data needed for gameplay. This function supports both single and two-player modes by managing multiple tracks.

Scheduling and Updating Notes: The start\_rectangle\_game and update\_rectangles methods handle the scheduling of notes to fall at specific times based on MIDI track messages. start\_rectangle\_game handles creating the falling rectangles that represent the notes, and update\_rectangles handles updating their positions each frame. Within initialization we set update\_rectangles to be called every 1/60th of a second so the game logic is constantly moving everything downwards towards the piano automatically. Both these functions are the most ‘logic heavy’ parts of the whole of the program, and for adjusting any of the logic surrounding the game, most changes will be made in ‘update\_rectangles. Anything about timing, how fast notes fall, and the different behavior for game modes is handled here.

Scoring and Game States: The update\_score method updates the score based on the accuracy of note hits, with points awarded for hitting notes correctly. The end\_of\_song method displays a game-over message when the song ends, signaling the end of the gameplay session.

Practice Mode and Clock Management**:** The ClockPauseManager class manages the scheduling and pausing of game events. This is crucial for practice mode; it manages scheduled functions, allowing them to be paused and resumed. It tracks the start time of the game, the pause time, and the unpause time, and then functions can be scheduled with a specific delay, paused, and resumed with adjusted intervals to maintain correct timing. The implementation of practice mode works, but it is very inefficient. In short whenever a pause happens, the manager loops through every single remaining note and unscheduled them, before rescheduling them at a new time based on the moment the game is un-paused. For songs with a lot of notes, I believe there are issues because some time is lost looping through incredibly large number of notes. This makes practice mode only compatible with basic songs with not many notes, and even then sometimes there are glitches. This is one major area of improvement for future development if the practice mode is viewed as a game mode, and you want to make it more robust. Currently the practice mode is very finicky and works for basic songs *usually* but is by far the most buggy part of the program.

With this foundational understanding, you should be able to delve into the Walking Piano project and contribute effectively to its development.

# Recommendations for Future Development

For future development, here are some ideas and improvements that could take the Walking Piano project to the next level. After the #1 recommendation about fixing practice mode, these are presented in no particular order.

1. **Fixing or Reworking Practice Mode and Clock Management:**
   * The most critical area needing improvement is the practice mode, which relies on the ClockPauseManager class. This class manages the scheduling and pausing of game events, tracking the start time, pause time, and unpause time. While this implementation works, it is highly inefficient. When a pause occurs, the manager loops through every single remaining note, unschedules them, and then reschedules them at a new time based on when the game is unpaused. For songs with many notes, this process is slow and causes issues because significant time is lost looping through a large number of notes. This makes practice mode compatible only with basic songs that have fewer notes, and even then, it can be glitchy. Improving this system is crucial if practice mode is to be viewed as a robust game mode. Given the limitations of Pyglet's clock, which cannot truly pause, it might be worth considering a complete rework or even archiving this feature altogether. It turned out to be a much more challenging feature to implement than anticipated due to these limitations.
2. **Integration with a Physical Walking Piano Controller:**
   * It would be fantastic to link the program with a physical walking piano controller as intended in the original vision of the project. Either a 49-key or 88-key size would work easily with the current program, but it could be adapted to fit other size controllers too. This would make the game more interactive and engaging and complete the original idea of our “Walking Piano”.
3. **Automatic MIDI Device Detection:**
   * Implement automatic detection of the MIDI device size so it could work with more than just 49-key and 88-key controllers. An override in the settings page would also be useful if, for example, a 49-key piano wanted to still play with the visuals for 88-key.
4. **Audio Through Web Server or Bluetooth:**
   * Assuming there is a physical walking piano controller, another idea is to implement a way for users to hear the audio through a web server or their own Bluetooth device. This would be useful in environments where noise might be a concern, allowing only the player to hear the piano.
5. **Song Compatibility for Different Key Sizes:**
   * Specify which songs work within 49-key mode versus 88-key mode. Currently, all songs are available in any mode, but sometimes you can't hit the necessary note within 49-key mode. This 49-key mode was a last-minute addition that improved visibility for users at the engineering expo, so making it more robust and intuitive would be great. I don’t think it makes sense for a user to choose a song and then not be able to play notes within the song.
6. **Improvisation Mode:**
   * A new mode similar to Freeplay could help users improvise. Leveraging music theory knowledge, (possibly with input from music majors or others knowledgeable in music), the mode could highlight only the keys that sound good together. For example, all notes in the C pentatonic scale or a chord like Cmajor7. Users could pick from a list of various chords or scales to experiment with improvisation, allowing creativity without needing extensive music theory knowledge.
7. **Freeplay Mode Integration:**
   * Transforming Freeplay mode to be part of the main menu could be an interesting change. The piano could always be playable and visible at the bottom of the main menu. This would require some big changes, as it would make the menu and the game no longer separate standalone entities.
8. **Speed Slider:**
   * Adding a speed slider before playing a song would be useful for some players who might want to play songs slower. Implementing this feature should not be too hard, though some logic changes would be needed in midi\_processor.py.
9. **Improving Visuals:**
   * Improving the visual aspects of the game could make it more appealing. Updating graphics, adding animations, and refining the overall aesthetic would enhance the user experience.
10. **Enhanced Autoplay Feature:**
    * Improve the autoplay feature so it can automatically play one track, allowing the user to play along with the other track. This would help users enjoy complex songs without needing another skilled player.
11. **Transforming Hard Songs (potentially with AI):**
    * Finding a way to transform insanely hard songs into something playable on a walking piano or with relative ease on a real piano could be a significant improvement. Machine learning could be used to isolate or predict a melody from the track messages, allowing users to play the melody while the rest of the notes are played automatically in a way similar to the autoplay feature. This would make complex songs more accessible, allow way more songs to be in the game overall, and likely make the user experience much more enjoyable. I think there is much greater satisfaction if you are contributing to some amazing music being played compared to a super easy song.
12. **Dynamic Resizing:**
    * Implement dynamic resizing to adapt the game interface to different screen resolutions and aspect ratios more fluidly.
13. **Search Menu for Jukebox Mode:**
    * Add a search menu to the jukebox mode
14. **Adding pagination for song select in normal game modes.**
    * Add multiple pages for songs in the normal game modes. As more songs are added, this feature will become necessary to manage the growing library.
15. **Adding More Songs:**
    * Simply adding more songs would keep the game fresh and engaging for users.
16. **Comprehensive Error Handling for MIDI Files:**
    * Adding comprehensive error handling for MIDI file loading and playback will ensure a smoother user experience, especially if individuals try to add their own MIDI files that are not necessarily supported. Although most MIDI files work, they are highly dependent on the specific format of MIDI files set up for piano with separate tracks for each hand. Most of our files came from musescore.com, where most piano songs can be downloaded as MIDI in the correct format.

By focusing on any or all of these changes, the Walking Piano project has the potential to become something truly amazing. Although many of these changes may seem ambitious, I am confident that with effort and collaboration from staff and peers alike, many of these dreams can become a reality. As the primary creator of the Walking Piano software, I personally would love for the project continue to grow and I would be especially thrilled to see collaboration between musicians/music majors to make the improvisation mode a reality. Furthermore, any work towards the AI melody isolation feature would be super cool and very interesting for a research type project.

***Take it slow and progress is inevitable – slow and steady wins the race🐢.***

*- Devin Martin.*

***Contact me at martindevin2001@gmail.com***

***or 971-506-4216 if assistance is needed!***