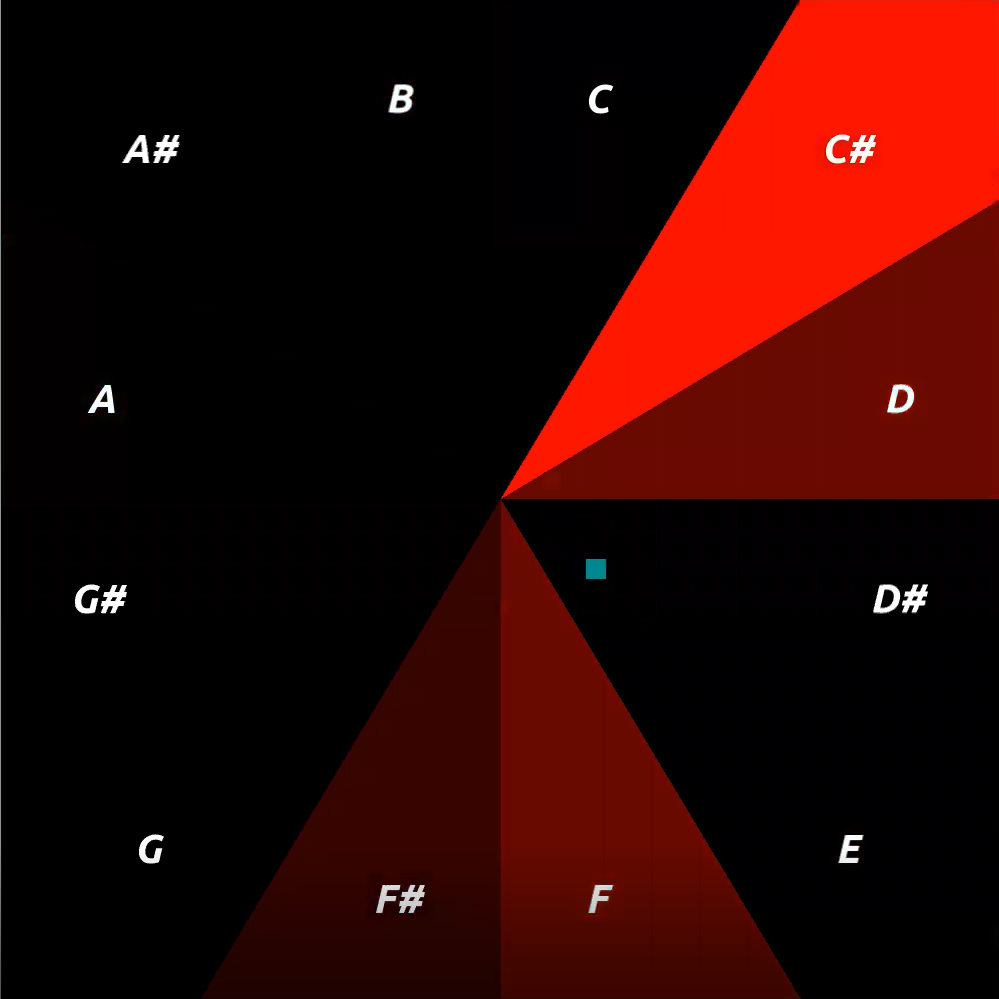
**Lightshow**

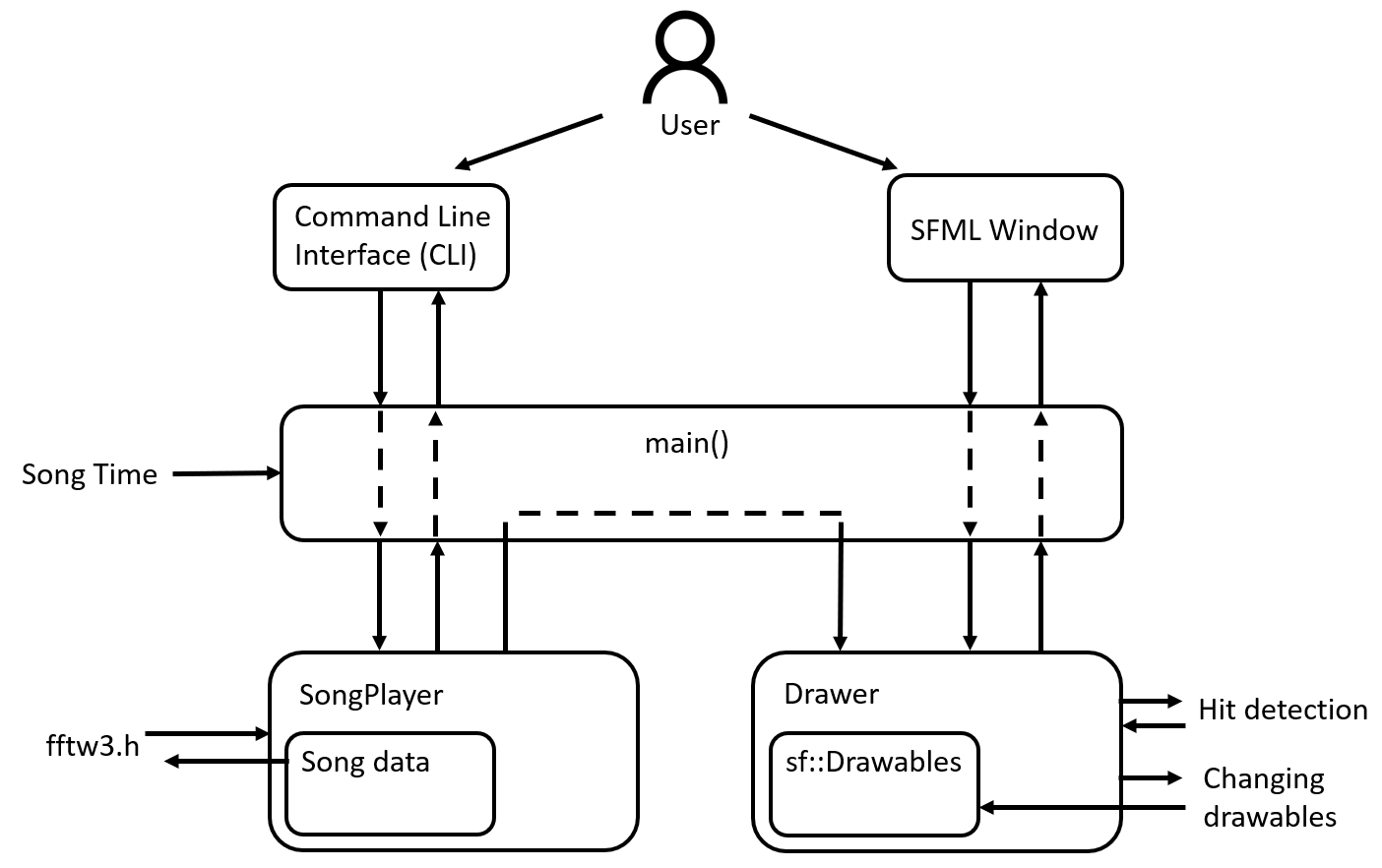
**Quan Pham – U42516796**

**GitHub:** <https://github.com/QuanMPhm/LightShow>

**YouTube Demo:** <https://youtu.be/vVqVnEHk5Yw>



**Software Architecture**



Lightshow contains two main classes called SongPlayer and Drawer which are responsible for parsing the input WAV file and drawing the SFML render window, respectively.

The user interacts with the program through the command line interface (CLI) and the SFML window. With the CLI, the user can pass in the name of a WAV file, set the tempo, and provide other inputs. Most of these inputs are passed directly from main() to the SongPlayer for parsing the WAV file and initializing internal fields.

When the player starts the game, a render window is displayed. The objects displayed on the window are determined by Drawer, which takes in inputs such as the frequency spectrum at a given beat and the user’s keyboard presses to color and move different SFML objects. The frequency spectrum is obtained from SongPlayer using the fftw3 library. All drawable objects are stored in Drawer.

Neither SongPlayer or Drawer is aware of the current song time. Only main() knows the song time by polling it every frame within the SFML window. By knowing the song time, main() can determine if a song beat has occurred, or if the song has ended. This information allows main() to call the appropriate functions in SongPlayer and Drawer, allowing the game to synchronize with the music.

**Application Overview**

Lightshow is a music-based 2-D reaction-time game. The player can upload any WAV song file they want, set a certain tempo, then take control of a square which must survive the entire duration of the song to win.

At each beat of the song, Lightshow attempts to determine the loudest 2 notes being played, and briefly lights up the tiles on the screen corresponding to those notes. If the player-controlled square is in a tile when it is lit, they lose a life. The player has 3 lives in total. To give players a sporting chance, the tiles are dimly lit prematurely to give players time to react. The difficulty of the game can be adjusted by playing songs with slower tempos.

Lightshow is purely an application for entertainment. The game is targeted towards people of all ages who are interested in listening to their music while also playing a somewhat involving game.

Lightshow is primarily inspired by the game “Just Shapes and Beats” by Berzerk Studio, following a similar minimal design and gameplay concept. While there exist many music-driven games which involves synchronization of the music to events happening in the game, as far as this student is aware of, there are few games in which the frequency of the notes plays a significant role during gameplay. Lightshow also allows any song to generate its own unique level, effectively making Lightshow a procedurally generated game. This is in contrast to games in which the choices of music are limited, and every level must be “hardcoded”.

**Component Description**

**SongPlayer**

To make Lightshow music-driven and discern the amplitudes and frequencies in any given song, the application must be capable of parsing sound files, perform signal analysis, and configurate the game based on the user-provided tempo. These functionalities are bundled in the SongPlayer object.

Given a song file, SongPlayer parses the file’s metadata to determine its size, encoding format, and other relevant information. Due to time constraints, SongPlayer currently only supports WAVE encoding at 16 bits per sample.

To determine the frequency and volume of the song at a given beat, SongPlayer inserts 0.05s worth of samples into an internal buffer, then performs the Discrete Fourier Transformation (DFT) provided by the fftw3 library to obtain a frequency spectrum.

**Drawer**

After obtaining the frequency spectrum from SongPlayer, the application must be able to lit the corresponding tiles in the SFML render window. This is in addition to drawing other objects such as the player’s square and the win/lose/pause texts, as well as determining if the player was hit when a tile lit up. These functionalities are bundled in the Drawer object.

To associate a tile with a frequency and note, Drawer contains a vector of Shard objects, which encapsulates this information. Each shard contains a note, its scientific frequency (“A” to 440Hz), and a SFML ConevexShape object. Given the frequency spectrum of a certain beat, Drawer determines the loudest frequencies and whether they pass a threshold volume before lighting the tiles (or Shards) associated with those frequencies.

In order to reposition the player’s square after a movement key press, or to display text when the game is paused, the main() function contains SFML event handlers which call the corresponding functions in Drawer whenever a key is pressed.

To determine whether the player’s square was within a tile when it is lit (collision detection), Drawer applies the ray casting algorithm.

Drawer itself cannot draw SFML objects in the render window because the window is within the scope of the main() function. Therefore, Drawer contains several “get” functions which main() frequently calls to get the SFML drawable objects.

**Main function**

As mentioned in the Software Architecture section, SongPlayer and Drawer has no knowledge of the current time of the song. This is because the SFML object responsible for playing the user-chosen song is in the scope of the main() function, similar to the render window. Therefore, in order for SongPlayer to obtain the DFT spectrum then pass it to Drawer, and for Drawer to know when to light the tiles and perform collision detection, main() must poll the song’s current time every frame, determine whether this was the start of a new beat, then appropriately call functions in SongPlayer and Drawer which perform the aforementioned actions.

Apart from timekeeping, main() allows user inputs on the CLI and keyboard to be passed to SongPlayer and Drawer.

**Team Assessment**

After much deliberations and internal reviews, it was agreed that Quan Pham had worked on 100% of the Lightshow project but only spent 85% effort due to his failure at meeting the expectations of a Specification Lead.

Within the first two weeks of the project, much of the time was devoted to proving the feasibility of the DFT transformation and music synchronized graphics. This meant very little thought was given when designing the software architecture in a way that could smoothly satisfy the game’s other major specifications, namely its timing system. This is most evident in the fact that the role of playing the song and managing its current time was not given to an object literally called the SongPlayer. Additionally, other oversight from both the Specification Lead and Technical Lead led to the problem of memory management becoming unmanageable and ultimately ignored towards the end of the project.

**Project Timeline**

Week 1 – 06/14: A program to play a WAVE file and parse its metadata and song data, was written and debugged. This was soon followed by experimentation with the fftw3 library to confirm the accuracy of the DFT in identifying volumes and frequencies in a song. WAVE files containing simple scales were used, and the frequency spectrum from the DFT were plotted to confirm their relative accuracy. Further experimentation and calibrations were made to prove the feasibility of obtaining DFTs while simultaneously playing the analyzed music.

Accuracy and memory problems persisted with use of the fftw library due to a lack of technical understanding of the Fourier Transformation as well as memory management. These problems would continue to plague the project throughout its development.

Week 2 – 06/ 21: After demonstrating music-DFT synchronization, an initial software architecture was drafted which contained a SongPlayer and Drawer object for handling WAVE file parsing and object drawing respectively. A simple command line interface (CLI) was made to load a song, set a tempo and starting time, all done manually.

More testing was made to prove that the CLI and SFML render window could perform actions that were synchronized with the beat of the song. These tests included logging to the command line and lighting rectangles on the render window whenever a beat is played. Frequent testing demanded a reorganization of the timing system, and due to technical ineptitude, it was decided that song timing was handled by the main() function. This decision, along with others, consequently forced main() to be exposed to some of SongPlayer’s and Drawer’s features, causing the source code to be de-modularized.

After finishing the DFT component, a rough sketch of the UI was made and quickly implemented. The ray casting algorithm for hit detection was implemented using online resources.

A single Boolean toggle was later added in the source code, allowing the render window to either display the DFT spectrum or normal UI for quick testing.

Attempts were made to modularize the code but were ineffective. After play testing the game, the decision to add multi-shard and delayed lighting was made to tune difficulty. Implementing the delayed lighting proved surprisingly difficult due the entangled timing system. To somewhat solve the problem, the Shard object was made to encapsulate the SFML rectangles and their corresponding frequencies.

Week 3 – 06/28: The UI was polished. With the game effectively done, final attempts were made to debug memory errors that emerged when testing the application. The memory bugs proved difficult to fix, but fortunately did not made the game unplayable.