Campanile-Carillon Model: Phase II

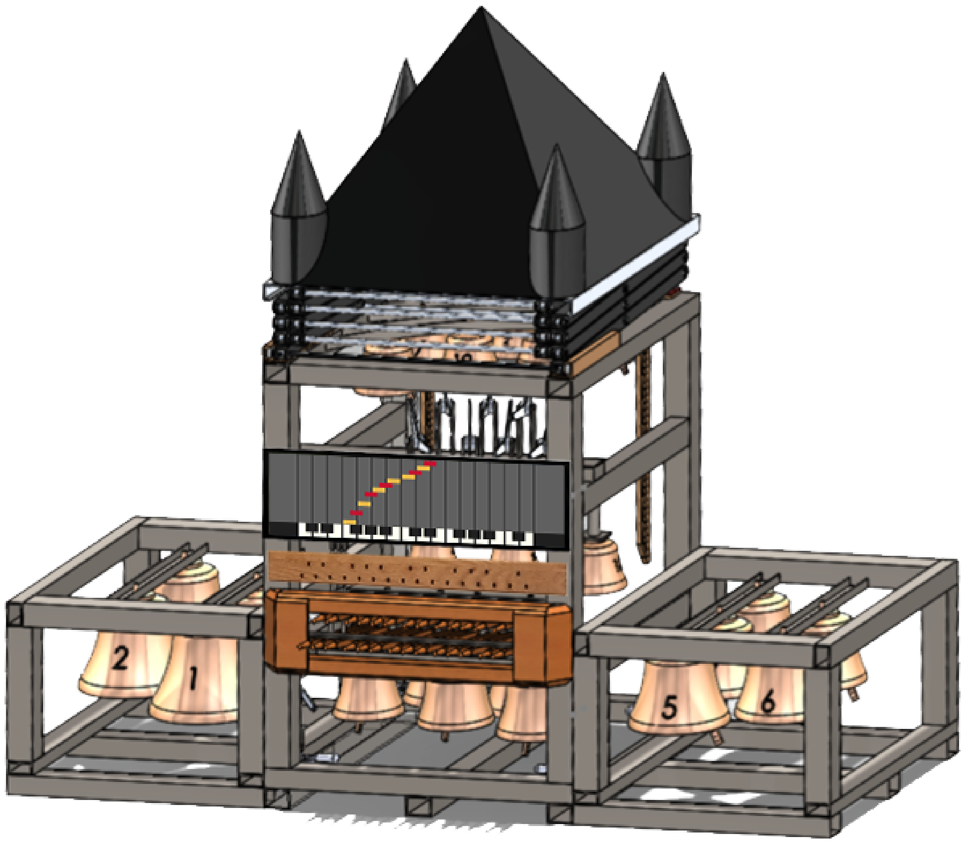
Final Report - 12/10/19

sddec19-12

Client: Dr. Tin-Shi Tam
Advisor: Gary Tuttle

Gabriel Stackhouse - Software Lead
Grant Mullen - Integration Manager
Kienan Otto - Report Manager
Ryan Roltgen - Meeting Scribe
Sam Habel - Meeting Facilitator
Yicheng Hao - Power Systems Lead

Email: sddec19-12@iastate.edu
Website: http://sddec19-12.sd.ece.iastate.edu/



List of Figures

Terms & Definitions

Arduino: A microcontroller that is widely used for interacting with digital devices

C++: A general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing facilities for low-level memory manipulation

Git: A version-control system for tracking changes in computer files and coordinating work on those files among multiple people. It is primarily used for source-code management

GitLab: A web-based Git-repository manager that is used at Iowa State

GPIO: **G**eneral-**P**urpose-**I**nput-**O**utput

I/O: **I**nput/**O**utput - Usually related to user input (keyboard and mouse) and graphical output (screen)

LED: **L**ight-**E**mitting-**D**iode - A light source

LED/Light Bar: The bar that is below the monitor which has LEDs that light up when notes reach the bottom of the screen

Linux: An open-source lightweight operating system

MIDI: **M**usical-**I**nstrument-**D**igital-**I**nterface - A file type that represents digital music

Power Inverter: A device that converts DC power into AC power necessary for electronic components. 12VDC to 120 VAC

PSU: Power-**S**upply-**U**nit - Entails all components necessary to supply power

Repository (Git): A repository is a collection of files that are managed by Git

SFML: **S**imple-and-**F**ast-**M**ultimedia-**L**ibrary - C++ library that *"provides a simple interface to the various components of your PC"*

USB: Universal-**S**erial-**B**us - An industry-standard cable connection type

Visual Studio: Software created by Microsoft that allows for the creation of programs in C++

Software Design

Overview

The software for this project was written entirely in C++. The language was chosen because the previous group laid the groundwork for the program in C++, making it easy for us to extend. C++ is great for a project like this because it's able to produce efficient code and interact with the hardware of the entire system much easier than other languages. The SFML library was also used to generate the graphics, which is much more intuitive and faster than if we tried to generate them manually.

Another C++ library, aptly named MidiFile, was used to convert MIDI files into a C++ data structure. This allows the program to read the music and know when and where to add the falling notes for each song.

The final piece was the computer the Arduino connection. C++ made it easier to communicate with the hardware by opening a serial port and talking directly to the Arduino, which in-turn communicates with the LED lights. Since the Arduino uses C natively and the serial connection is done entirely through C libraries, C++ made this all very intuitive to implement.

Libraries

Below are the requirements to build the project's source code from scratch. All libraries should be built and installed on the computer that is running the program. This step is about where to find and download the required files. Anything regarding installation will be covered in the next section.

SFML

The SFML library is used to draw the graphics to the screen. For our project we use SFML 2.5.1 for Windows. We have an installation script to help install the needed libraries to the proper location to the computer.

Midifile

MidiFile is a C++ library that handles reading MIDI files, a file format that represents music digitally. The library converts MIDI files provided by the end-user to a data structure readable by C++ code. MidiFile is maintained by Craig Sapp in his GitHub repository.

FastLED

FastLED is a C library for the Arduino Uno that allows for easily controlling LEDs on various chipsets. In our case, this was used to emit the LED lights in the light bar at the correct times and colors to correspond with the falling notes on the screen above. FastLED is publicly maintained on the Github repository.

GameStates

The game is split up into different states represented by a class named GameState. Each GameState can handle a specific set of actions. There can only be one active GameState at a time, but the game can switch between GameStates based on the actions that can be taken. The game will run indefinitely, so long as the active GameState tells the game which GameState to go to next.

MenuState, SongSelectionState, and DifficultyState

There are three menus that are in the game currently: MenuState, SongSelectionState, and DifficultyState. Each one gives a list of options that the user can select and when selecting an option it will perform an action.

For example, if on the main menu, one selects the “Exit” option, the game will exit. Instead, pressing the “Play” option the MenuState will tell the program to make the SongSelectionState active. On the song selection menu, a song can be selected which will make the DifficultyState active. The difficulty state lets users select how fast they’d want the song to play, and the song starts immediately upon selecting this.

Navigation for these menus works with the arrow keys and the Enter button. The Escape button can be used to go back a menu if you selected something wrong. Controls function similarly with an external remote.

PlayState

PlayState is where the musical notes are shown on screen. The track is played back to the user while the notes “fall” towards the bottom of the screen. When the notes touch the bottom of the screen, two things happen:

- That note is marked as active
- A pulse is sent out via serial connection to alert the connected Arduino that a note change has occurred

When PlayState is first initialized:

- PlayState’s internal Timer is reset. The timer necessary to keep track of how much time has passed while the song is playing, which is needed to update the note’s graphical position
- The active song’s MIDI file is read into memory
- Every note is iterated over, and a note is generated at the specific time offset to represent that MIDI note in the song
- Every note is then set as inactive

While PlayState is running, each frame calculates how much time has passed since the previous frame. The difference in time is what is used to update the notes' positions. This "delta-time" ensures that the framerate of the computer won't alter the positions of the notes.

The way active notes are determined is based off of the note's position. For each note, if the note is within the bounds of the bottom of the screen then the note is considered active. If a note is active then that note will be lit up on the LED bar. This happens for all 27 notes in the song.

Loading External Files

Since the project needed to support the ability to play any music, the program automatically checks a directory for the existence of MIDI files. Those MIDI files are listed in the SongSelectionState and when selected the MidiFile library reads the data for the program to use.

All file I/O is handled by the ResourceManager class. Its main function is to reread those directories and repopulate the list of available MIDI files to play.

Source File Class Definitions

GameState

For all GameState-related information, refer to the previous section.

KeySprite

KeySprite inherits from two SFML base classes - sf::Drawable and sf::Transformable. It is essentially a sprite that generates a rectangular entity representing a key that appears on the screen. The key will change color, location, and size depending on these factors:

- Whether or not the key is a natural note or a sharp (white key or black key)
- The millisecond offset at which the note is first played
- The duration of the note

Note

A Note acts as an intermediary between MIDI file notes and KeySprite objects. When the MIDI file is parsed during the initialization of PlayState, a Note object is generated for each MIDI note, and then based on the MIDI data parsed into that Note, a KeySprite is generated. This also takes into account notes that fall outside the initial range of 27 keys that are available on the carillon.

VerticalBarSprite

The VerticalBarSprite is a background of vertical stripes that allow the user to see the spacing of the notes and line up the notes with the real-life instrument more accurately. On top of that, these stripes help differentiate the natural notes from sharps and flats because sharps/flats will be on the line while the natural notes will be between two lines. This provides more usability for people that are colorblind and can't easily tell the difference between yellow and red.

ResourceManager

As mentioned previously, ResourceManager's main purpose is to populate the list of available MIDI files in pre-configured directories. It will also handle the creation of directories for those files if they do not exist.

Serial

Serial is used for serial connection I/O. Its main purpose is to send data from the main program to the Arduino connected via USB serial cable. It contains mostly standard C code instead of C++, as low-level file descriptor interaction (which serial connection is) works best in C. Serial is used within PlayState to send active note updates to the Arduino which in turn updates the lights.

Timer

Timer is used within the PlayState class to keep track of how much time has passed during the execution of the song. The main benefit that the Timer class provides over SFML's built-in Clock class is the ability to pause time and start it again, which Clock doesn't allow. That being said, Timer is essentially a wrapper around Clock that extends its functionality to provide the pause/start effect.

Hardware Design

LED Bar

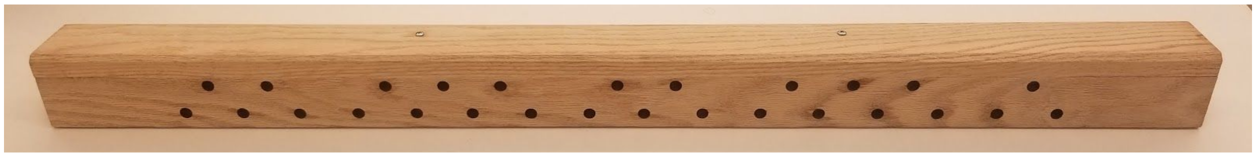


Figure 1: LED bar

The LED bar replaces a decorative crossbeam on the structure. The embedded LEDs line up with the falling notes on screen and will light up for each note's duration. The LEDs sit on one of three custom PCB designs with three, five, and seven LEDs each. There are five boards currently being used inside the beam to match the pattern of holes in the beam and batons on the carillon.

The boards are controlled by an Arduino Uno which receives commands over a serial connection with the main program. The power for the Arduino and the five LED boards also comes from this connection to the main system.

Battery

The best solution for a battery is to buy an off-the-shelf system. This allows for easy replacement in the event of failure or battery degradation. Additionally, this will provide for long-term support as opposed to a student-designed solution. There were multiple recommendations made for the battery.

For a single long-life battery system, the Sungzu Portable Power Station 1000W system is recommended. With the recommended PC, monitor, and peripherals there is an estimated battery life of six hours. For shorter battery life, the Sungzu 500W Portable Generator Power Station is recommended. This provides an estimated 3 hours of runtime. Ideally, multiple batteries at this smaller capacity would be purchased. This would cut down on the required space inside the structure for a battery. However, this would require the system to be powered down when swapping batteries. This multi-battery concept could be used in any combination to achieve the desired battery life.

Implementation Details

Software and Hardware Integration

The program is being run in Windows 10 and is developed in C++. The program uses multiple libraries to perform important data conversions, including creating usable data out of MIDI files. The information includes note and duration for each note in the song. When the note is at the bottom of the screen (indicating it should be played at this time), a signal is sent via serial connection to the Arduino Uno controlling the LED bar. The Uno then sends the activation signal to the specific LED. When the note leaves the screen (indicating it no longer needs to be played), the Uno receives a signal to deactivate the specific LED and does so immediately.

Port from Linux to Windows

The program was originally developed for a Raspberry Pi running Raspbian, a Linux distro. The program was functional in this state but the hardware was not adequate performance-wise. The team made the decision to port the program from Linux to Windows to make it easier for future users and owners to navigate and modify. The original version on Linux was developed in C++, making the conversion simple. The biggest changes were in the libraries being used and the IDE being used to develop the software. This also enables easier system replacement in the future, as the installation process on a brand new system can be scripted and run easily by anyone familiar with Windows OS.

Project Obstacles

Our original project plan made assumptions about the timeline of receiving necessary components that were not correct. The plan was to get the monitor at the end of March 2019 and continue developing the program in Linux until our deadline of October 2019. In April 2019, the monitor had not arrived and we were given a new task of porting the program from Linux to Windows. This shifted priorities away from the planned feature developments. After successfully porting the program, the monitor arrived October 2019. The testing period was cut short due to the late acquisition of the monitor. In addition, the previous group did not provide the source code for the Arduino that controlled the LED light bar. The team was forced to find the software library that was used with the LEDs and recreate the source code from scratch.

Testing Process & Results

Process

The LED bar and main program were tested together, as the LED bar's functionality is dependent on the on-screen notes. The tests occurred after each major iteration, structural change, or optimization made to the program. There were multiple tests run on the program:

1. Continuous operation for one hour
2. Long duration midi files
3. High note count midi files

The LED bar was tested by watching how closely it aligned with the displayed notes using the same tests as the program.

Results

The program successfully passes each of these tests in the current version.

1. The test can run any number of songs for any amount of time without having failures due to the program's runtime.
2. Longer songs do not lead to lowered performance or failures.
3. Notes are being only temporarily drawn and stored in memory, so a high number of notes does not lead to lowered performance or any related issues.

The LED bar passes the tests as well.

1. The Arduino and LED bar do not lose functionality after extended use. The test was mainly focused on discovering memory issues with the Arduino.
2. The Arduino and LED bar do not become overwhelmed after being used nonstop for individual long duration songs. This test also addressed memory issues.
3. The system continues functioning even when multiple notes are overlapping and changing quickly. This was to test the responsiveness to a change in active notes.

Related Products and Literature

Products

The styling of the interactive guide itself has been used in multiple games and programs already. The most popular of these games is Guitar Hero, but other common variations are found in Rock Band and Synthesia, a piano themed instructional guide. Our guide is most similar in functionality and design to Synthesia, but specialized for a 27-key carillon and extended to interact with our LED bar.

Literature

This project started about 3.5 years ago as a capstone project for mechanical engineering students. The goal to create a mobile carillon that can be easily accessible by people who don't have the ability to make it to the top of the campanile. The project is overseen by Dr. Tin-Shi Tam and the Student Carillonneur Leadership Council.

In Spring 2018 the first ECpE team was assigned to start work on a way for users to be able to play popular ISU songs despite how much musical knowledge they have. Over the year, they were able to get a small working prototype that we have refined and added to over our time on the project. They also helped construct the lightbar and make the code for the arduino which controls when the lights are lit up.

Appendix I - Operation Manual

Project Laptop Credentials

Username: user
Password: carillon

Initial Setup

1. Run the carillon installer, found on the Releases page of the project repository

Each Use Setup

1. Connect the HDMI cable between the monitor and the laptop
2. Double check that the MIDI files are in the proper folder (look at "How to add MIDI or MID files)
3. Check that the SFML-2.5.1 folder is in the C: drive

How to add MIDI or MID files

1. Go to **C:\Users\<username>\Documents\CampanileCarillon\midi** on the computer and drag any midi files you want into the folder
2. Start or restart the program to reload the song list

Running the Program

1. Double-click the executable. Program should load into the main menu

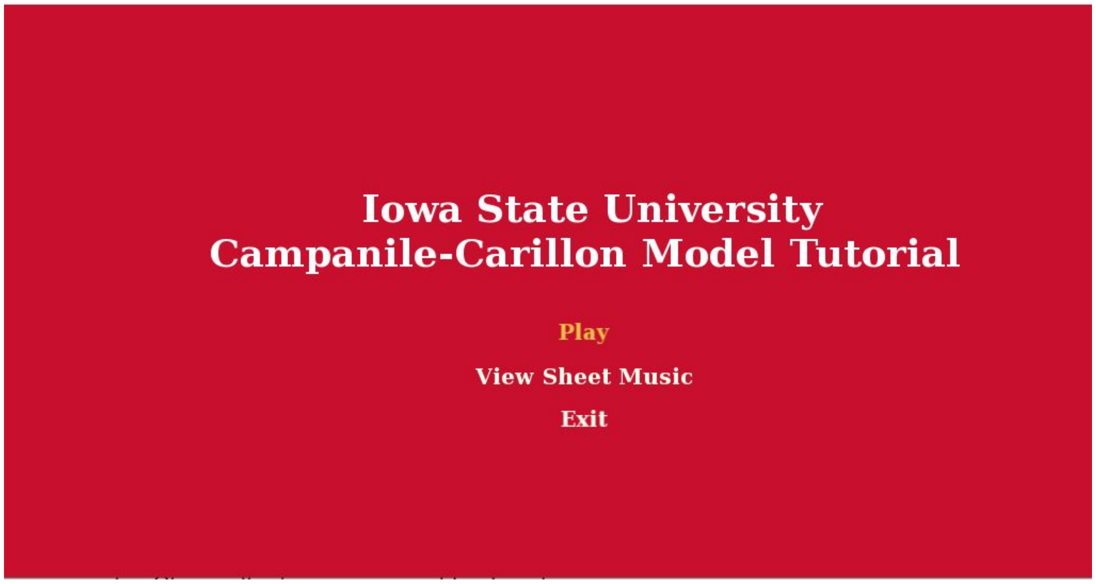


Figure 2: Main Menu / MenuStater

2. Use the up/down arrow keys for movement and enter to make the selection
    a. Selection is in yellow
3. Escape can be used to go back a menu and to leave a song that is currently playing
4. To play a song, select **Play** from the main menu
    a. Choose the song you want to play
    b. Choose the tempo you want to play at
        i. **Adagio** (Easy) = 50% speed
        ii. **Andante** (Medium) = 75% speed
        iii. **Allegro** (Hard) = 100% speed
        iv. **Presto** (Advanced) = 125% speed

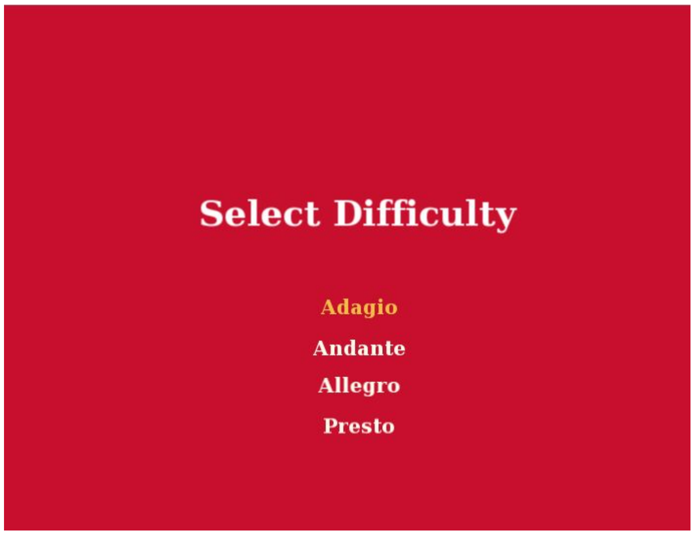


Figure 3: DifficultyState

   c. Play through the song, or you can leave anytime with the **Escape** key

5. To leave the program select **Exit** from the main menu

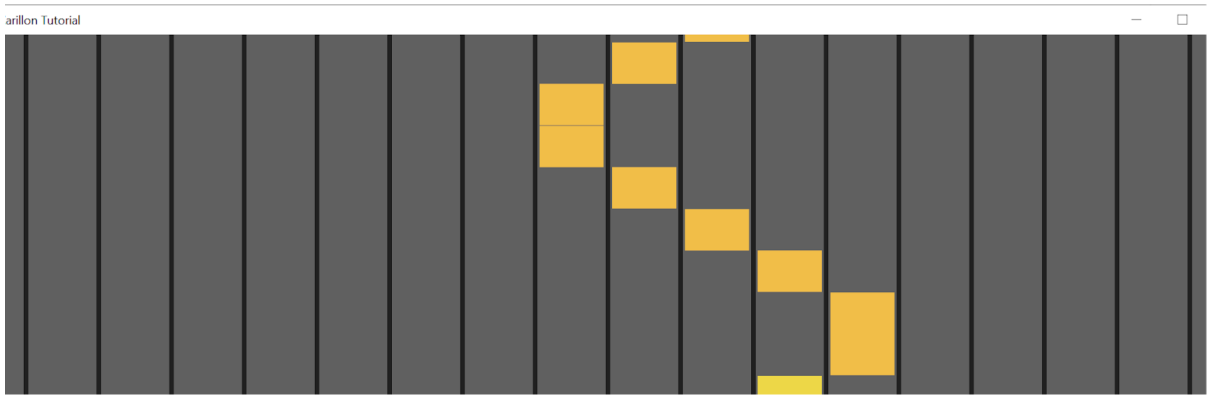


Figure 4: PlayState

Compiling from Source

1. Install Visual Studio 2017 or newer onto your computer
2. Clone the git repository for the project. Link is earlier in this report
3. Launch the .sln file for the program
4. Changes can be made and recompiled as needed

Appendix II - Alternative Versions

Linux Version

- When starting this project the program was already in Linux and we worked on the program for about 3 months, fixing bugs before our client showed interest for the program to run on MacOS or Windows.

Appendix III - Future Plans/Other Considerations

PDF Sheet Music Viewing

We have been working to view PDF sheet music within the program to allow musicians to use our program effectively instead of having to follow the screen. Most of our work for this feature has been working towards having a way to turn PDFs into a format that SFML can use and display. Due to not having the final system in our possession, the feature was not fully implemented

The conversion from PDF to bitmap to on-screen sprite requires the following programs and libraries to function:

1. Google
    a. Depot_tools: A set of extensions to allow automatic updating of Google's tools and development of Chromium-based software.
    b. PDFium: The program written in C++ used to display PDFs on Chrome and Chromium.
2. Apache
    a. Subversion: Similar to git, and used to track changes in code and keep the current codebase up-to-date.
3. Microsoft
    a. Debugging Tools for Windows: Feature from the Windows 10 Software Development Kit to aid in Windows development.
4. Standalone
    a. Ninja: Developed by Evan Martin, it is a software building tool.
    b. Python: Designed by Guido van Rossum, Python is a scripting language.
    c. Git: git is a version control system created by Linus Torvalds.

PDFium is the core of the feature, as it is responsible for rendering the PDF to a bitmap. After a PDF is processed, there will be one image per page of the PDF, and the name follows the scheme of the original PDF followed by a number indicating order. Our program would then be responsible for taking the bitmap and rendering it as a sprite, which is the format SFML uses for shapes and images. It could be implemented similarly to the normal SelectionState but the PlayState would require a new format.