3D LED Visualizer Group 2

John Burns, Henry Gillespie, Timothy Grant

May 27, 2021

Mentor: Brandilyn Coker

# Table Of Contents

## Section 1: Introduction

The purpose of the following document is to provide the project artifacts of the 3D LED Visualizer. The LED Visualizer consists of 3 main components. First, the physical hardware was created and designed for utilization of a PCB. The Logic Unit is the heart of the project, it directs the electricity flow in order to allow for targeted control of each LED as each row is cycled through fast enough to ensure the human eye is unable to detect any inconsistencies in the light display. Secondly, the software is the brain of the project. It utilizes the functionality and extensive libraries found within python to harness the utility of an Arduino Uno. The Arduino Uno may also receive commands from the user via Bluetooth, then send the correct logic to the Logic Unit to accomplish the desired animation that the user inputted.The third and final component of the project is the physical enclosure. The physical enclosure is constructed from a wooden base which contains all of the electrical components such as PCB, Arduino Uno, and connector wires. On top of the wooden base is a plexi-glass box. This plexi-glass is to protect the 3D LED matrix.

The project artifacts include: circuit schematic, code, mechanical drawings, block diagram, interface, PCB layers, materials used and time spent by each team member.

## Section 2: Documentation

The following documentation is sufficient for complete reconstruction of all sub-blocks of the project, and to then integrate all sub-blocks together into a fully functioning unit.

## Section 2.1: Schematic

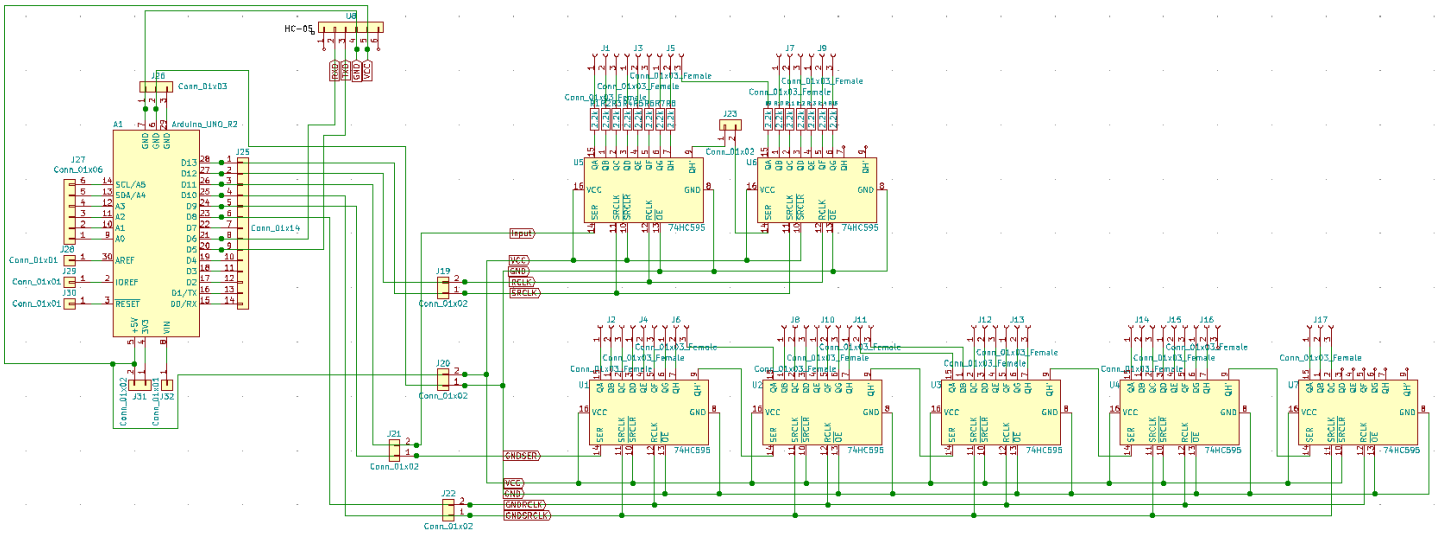


Figure 1: Full Schematic

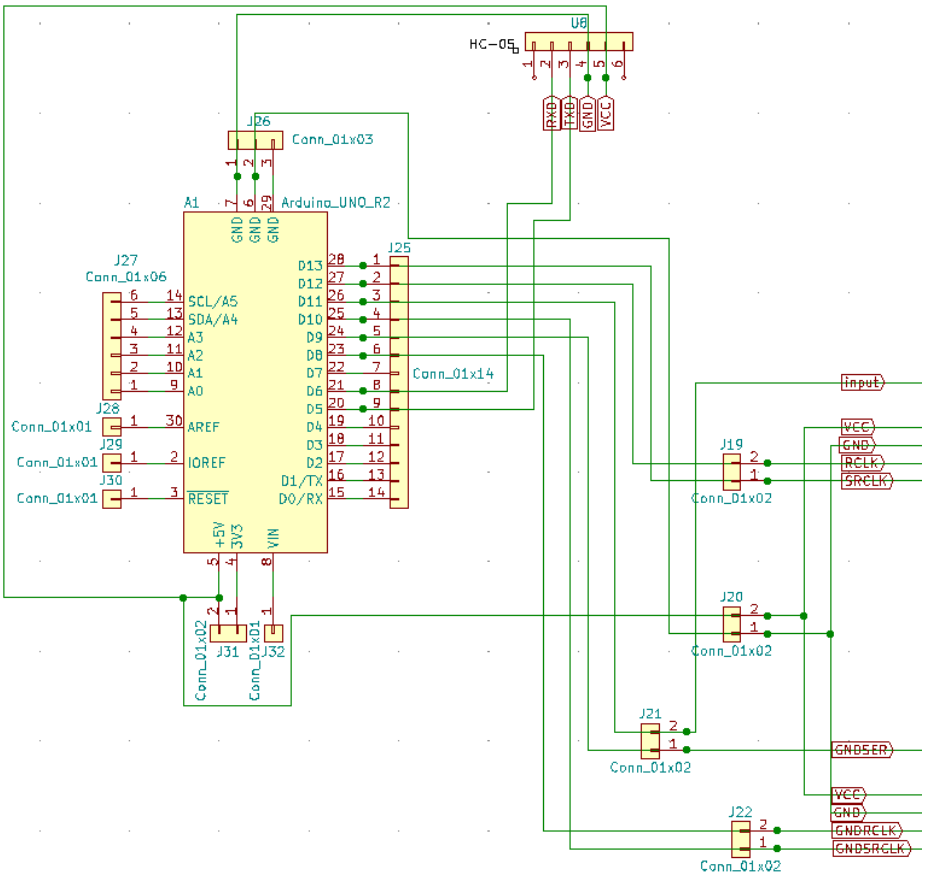


Figure 2: :Left Half of Schematic

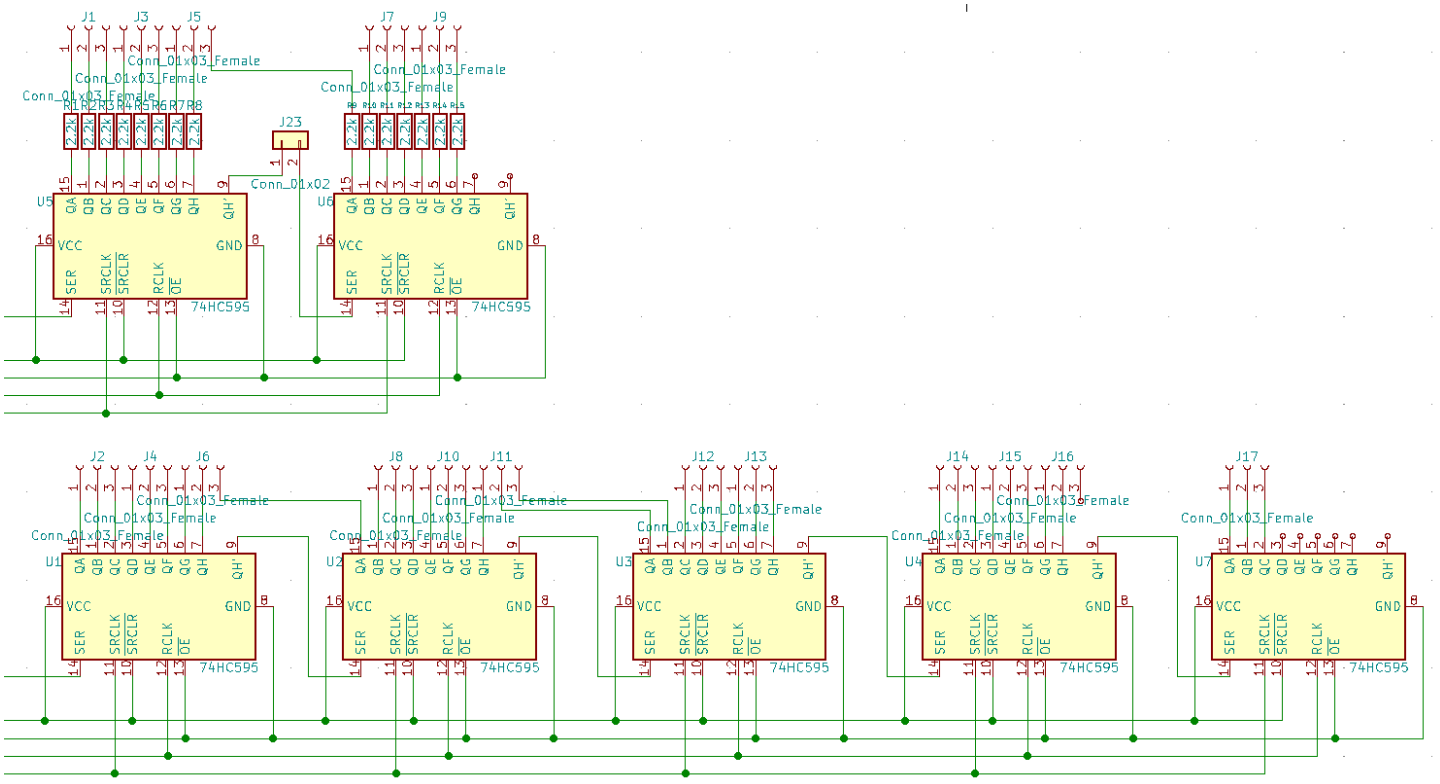


Figure 3: Right Half of Schematic

## Section 2.2: Code

There are three parts to the code of this project. It all starts at the user level. The user has an option. They may choose to select a programmed animation (via bluetooth or the graphical interface) or to create animations (exclusively on the graphical interface). If the user wants to make an animation, it must go through the User Interface. Once the desires of the user have been found, the information is sent to the Control System for processing.

The job of the Control System is to take the user's choice and provide it to the microcontroller in a way that it can understand. This did require some clever solutions. The biggest problem is, to make the program not blink, it has to be not only sent, but read in quickly. The solution that Henry found allows for nearly 4x the required 60 frames per second. This solution had us write the information into a header file. As header files use static memory, the arduino with its large memory space and low dynamic space was able to hold quite a bit of information. The three custom animations are contained in this header file, along with information about how fast they should play. This information is then used in the while loop in the customAnimation() function to slow the animations down by repeating the same frame until the time has passed. The pre-programmed animations do a similar thing, but use an extra for loop to repeat the frames. The main loop repetitively checks for new input from either the serial connection or the bluetooth connection, then goes into a case statement to run a particular animation. For this reason, the current animation will finish before the next one starts.

The Micro Controller takes the LED information and sends data to the Logic Unit in 1x1x5 row chunks. To reduce the amount of wires, Henry conceived the idea of flashing the LEDs twice. This gives the effect of a PWM and will allow for $3^3$ unique colors, much higher than the 10 color requirement.

Refer to appendix A.1-A.10 and B.1 for all code or retrieve the latest version (20) from GitHub.

## Section 2.3: Mechanical Drawings

Mechanical Drawings serve 2 purposes, showing what a project looks like, and showing how to reproduce it. As such we decided to include the renders as well as the drawings.

Figure 4 shows a rendering of the PCB that John Designed in KiCad. The custom PCB is 73 mm wide and 86 mm long. Some specific things of note are the 0 shaped formation of components, the holes, and the notes. The formation of the components was done this way to not only make the design more compact, but also keep its original clarity. The holes were added to allow for easy mounting. Finally, the notes are shown to help constructors understand what each section is. As a side note, J23 was included to make sure if something went wrong, there was a way to fix it without having to get a new board.

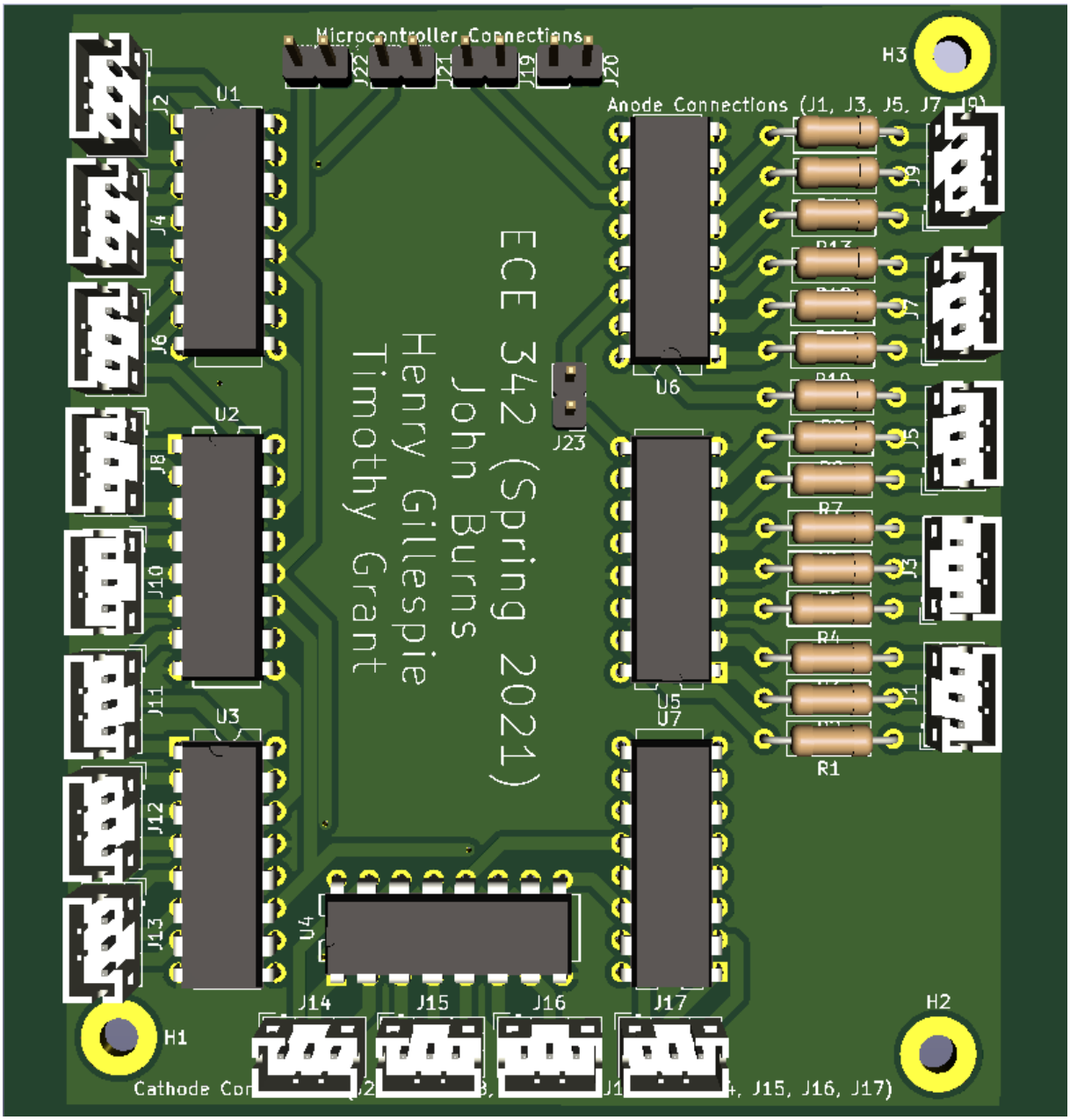


Figure 4: 3D View of PCB w/ Components

Figure 5 shows the CAD drawing of the bottom box. Originally the box was in the reverse orientation to the way it is now. Perhaps it shows the designs usability, the box will have the PCB, Bluetooth, and Arduino attached to the roof (upside down). The arduino's cords inputs are available through the hole provided on the back face. The connections to the ground wires are doing through the large hole in the back and the color wires that come through the smaller hole on the left side.

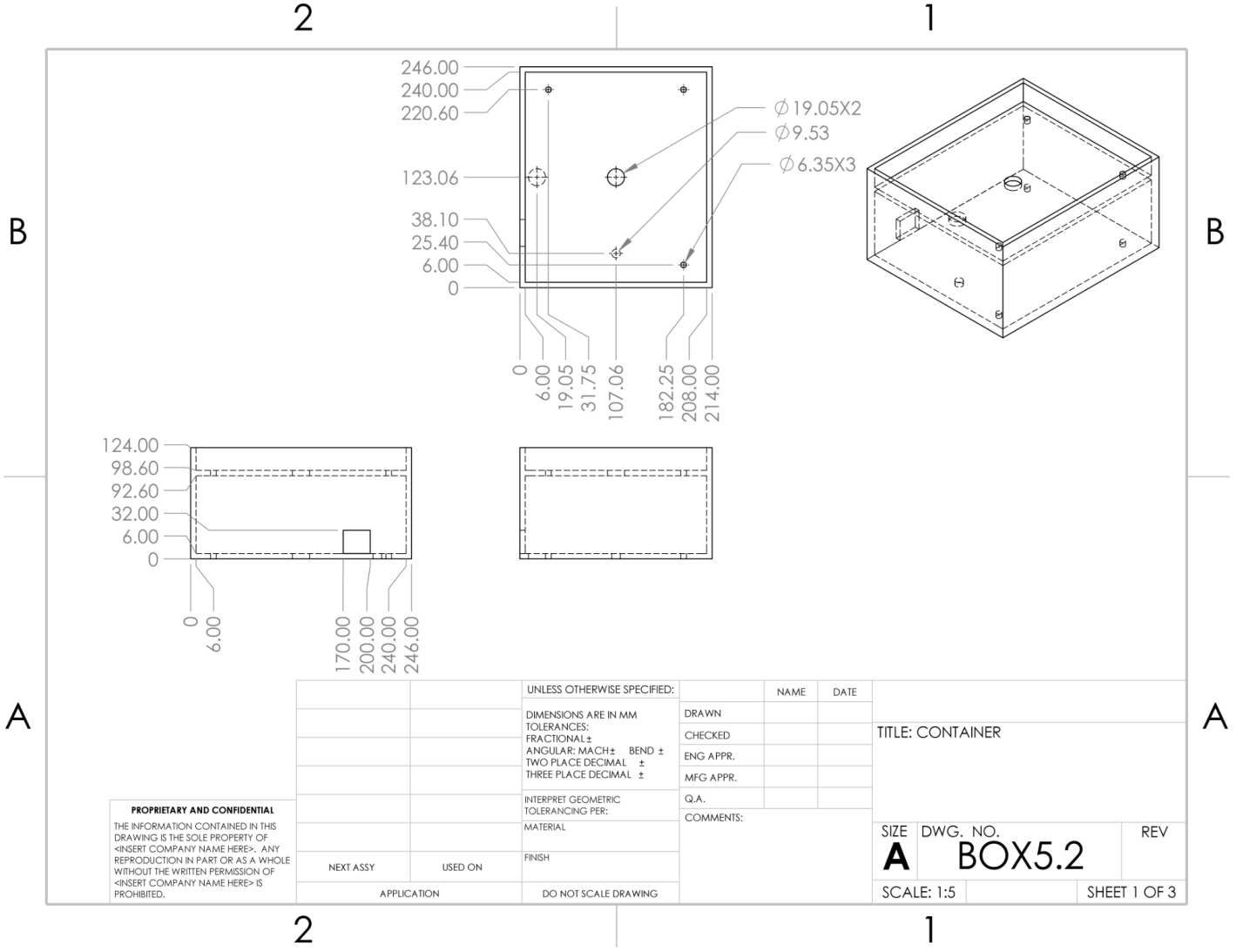


Figure 5: Drawing for Bottom of Enclosure

Figure 6 shows the plexiglass top to the box. The current design has it glues directly to the top of the box. It fits somewhat snuggly over the LED matrix. It is constructed from glueing 3 8x9 and 2 8x8 square inch plexiglass panes together at the edges.

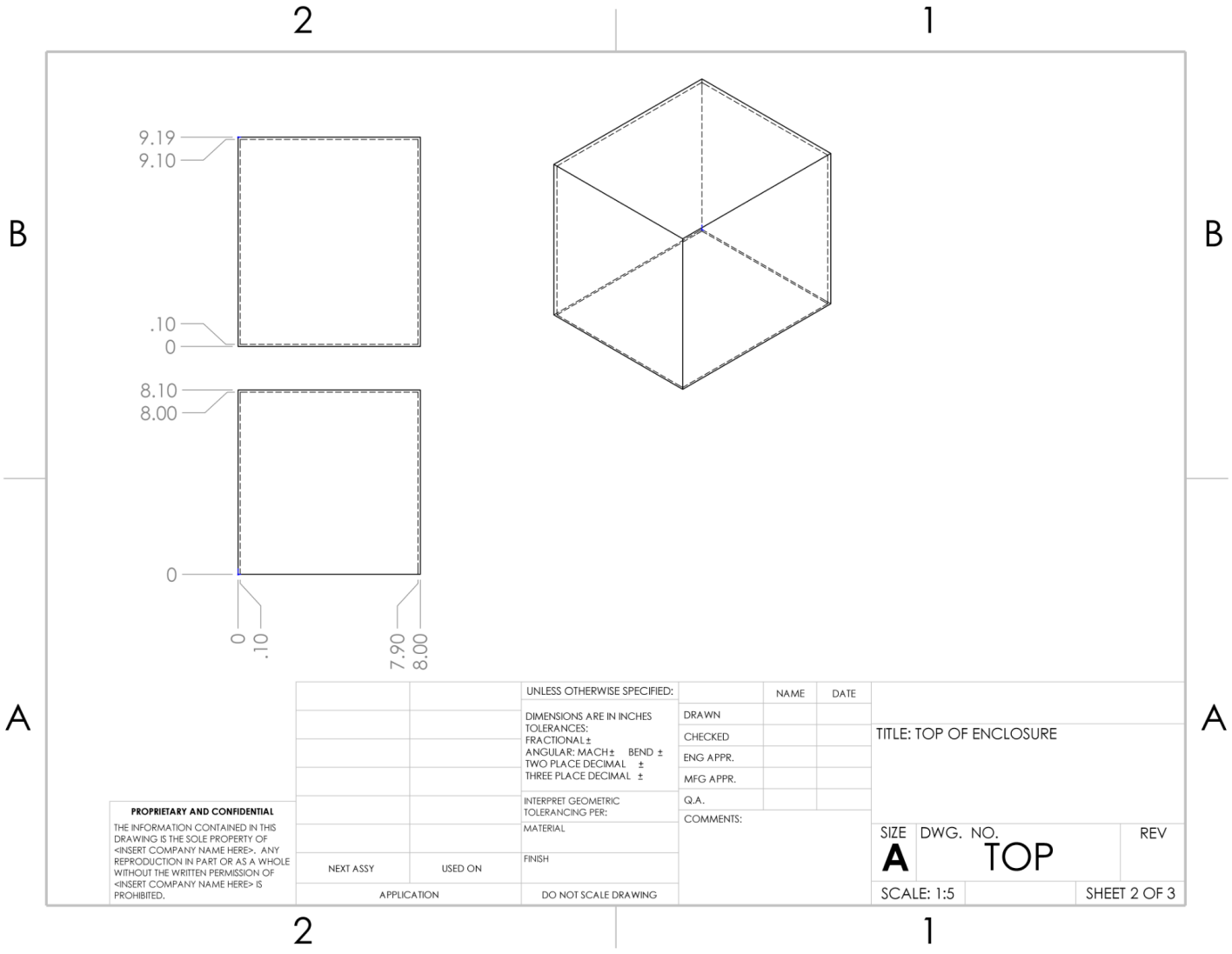


Figure 6: Drawing for Top of Enclosure

The full enclosure, shown below, is made in a way that provides an enclosure for all the components, but can be somewhat easily disassembled, but also rigid. The idea for disassembling is brought on by the possibility that something might eventually need to be replaced. The rigidity of the design is made to prevent users from easily breaking and having to replace parts of the design. Though it is not meant for extreme conditions, it can and has been tested sufficiently for vigorous shaking.

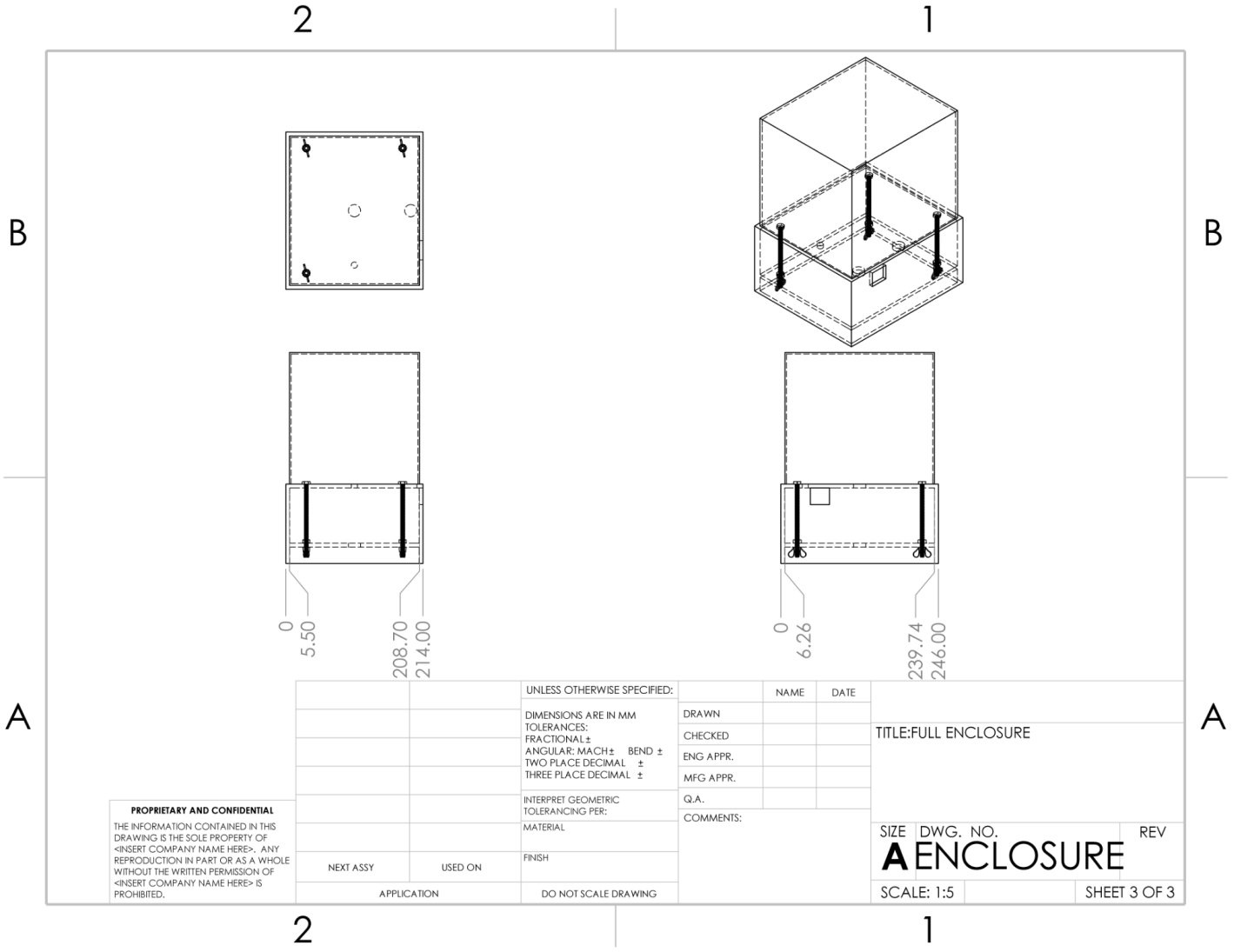


Figure 7: Drawing for Full Enclosure

The full enclosure is meant to look secure, but also maintain as much functionality as possible. For example, the components are protected by the bottom of the enclosure, but this also means that the LEDs cannot be viewed from as low. Additionally, the LEDs needed to be protected by some casing, so though it is not 100% see through, it protects the LEDs without obscuring the user's vision. As a side note, this is currently facing toward the computer. The hole in the top of the box is for the usb cord, the back hole is for the ground connections and the hole on the right is for the color connections.

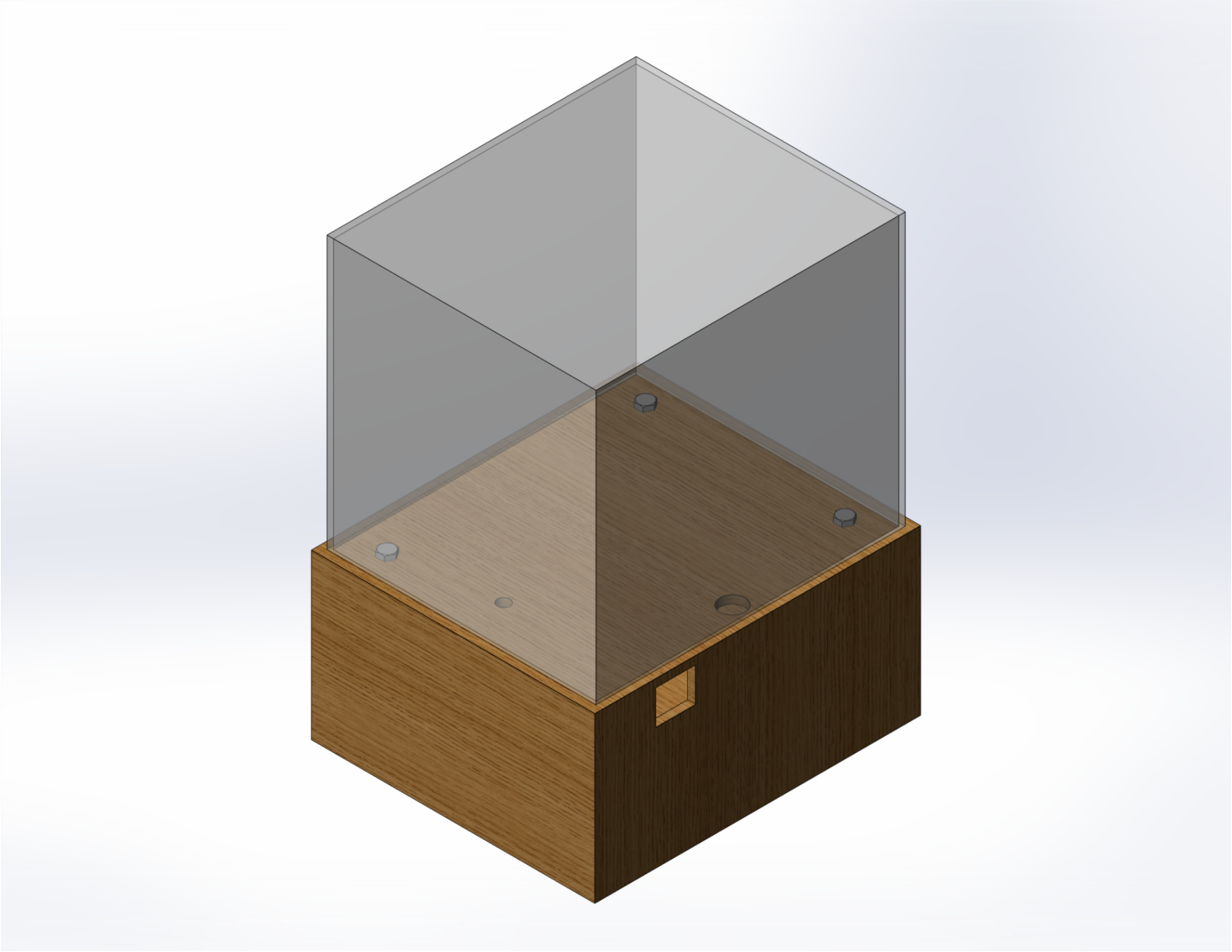

Figure 8: 3D View of Full Enclosure

## Section 2.4: Top-Level Block Diagram

In Figure 9, the block diagram can be seen. The block diagram allows the project to be viewed with all of the individual blocks put together to make up one cohesive unit. Present in the block diagram figure are the individual blocks along with the inputs and outputs of each of the blocks, and the pathways of those inputs/outputs. From the diagram it is possible to get an understanding of how each of the blocks interact with each other.

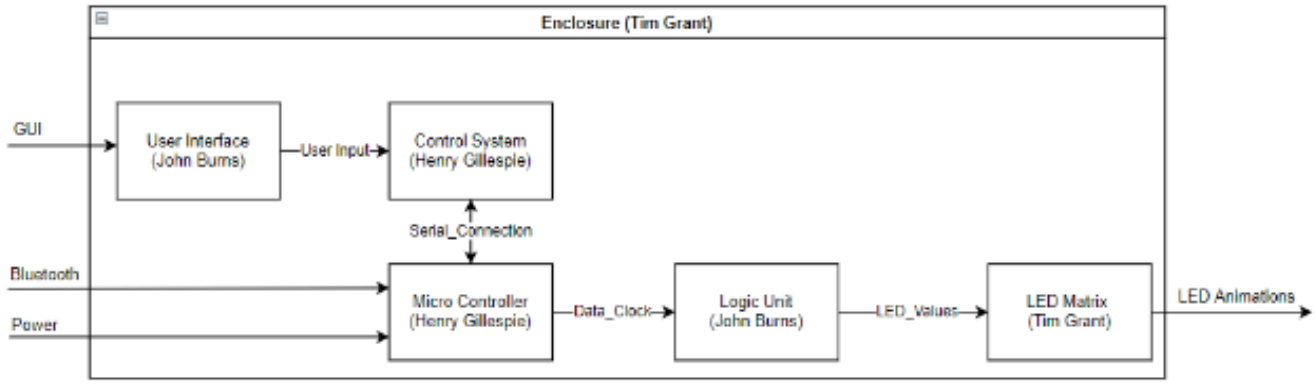


Figure 9: Top Level Diagram

## Section 2.5: Interfaces and Properties

The interfaces and properties section will allow the reader to understand the nature of the data that is being received and sent out from each of the interfaces, along with any of the specific requirements or attributes that the interface has. By utilizing this knowledge, the reader is able to ensure that all inputs of a block will accept data in the form given, and will output data in the form that is given. Without an agreement on the format that the data will be given and received the individual blocks will not be able to interact with each other. Therefore, it is vitally important that all of the data conforms to the agreed upon format as specified below.

Table 1. Interface Definition Table

| Interface | Interface Type | Specifics |
|---|---|---|
| Bluetooth_Input | Numeric | 2.45 GHz<br>Works from > 10 ft away |
| User_Input | Array of class objects & Numeric | Array of Animations:<br>- Each class animation object will have 30 frames, the speed of the animation, and the number of frames as attributes<br>- The frames are 5x5x7, and have color values specified by the Color dictionary<br>Numeric:<br>- 0-5, based on the selection of one of the 6 animations |
| Serial_Connection | String & File | String:<br>- 9600 Baud<br>- Data goes both directions<br>File:<br>- Uploads to Arduino through Windows OS<br>- COM3 port |
| Arduino_Power | DC Power | $V_{min}$ = 4.75 V<br>$V_{max}$ = 5.25 V<br>$I_{max}$ = 200 mA |
| Data_Clock | PWM Signal | $10_{min}$ kHz clock<br>1 bit shift register value<br>1 bit latch enable line |
| LED_Values | Analog Voltage | $Red_{min}$: DC 2.0-2.2V<br>$Blue_{min}$ & $Green_{min}$: DC 3.0-3.2V<br>$Power_{max}$: 0.06 Watts |
| LED_Animations | Light | Min switching time: 0.016 seconds (based on eyesight of 60fps)<br>Min 10 colors per LED |
| GUI | Events | Events are easily anticipated based on labels |

| | | |
|---|---|---|
| | | Speed and number of frames are adjustable<br>LEDs colors are intuitively changeable<br>Custom animations are not sent until the user clicks the submit button<br>All options for frames (0-6), animations (4-6), speeds (1-30), colors (27 colors between black and white), frame (0-29), and max frame counts (1-30) are available on the advanced settings window |

## Section 2.6: PCB Layers

This particular project required a custom PCB, however, we wanted to make the PCB as setup friendly as possible. One aspect of this is that the routes are easily followable. Though some vias were required to implement the logic correctly, they were used sparingly and it is blatantly obvious when they occur. Additionally John put a lot of effort into the silkscreen. The silkscreen is intended to make it easy to tell exactly what the intention of a component is.

The connections segment of the PCB is quite small, someone can see it with the naked eye, but it is borderline. Regardless, matching this up with the schematic in figure 1, someone setting this up can easily find the required connections. As a side note, the inputs are grouped by purpose. You can notice the clocks are put together, the input is put together, and the collectors are put together.

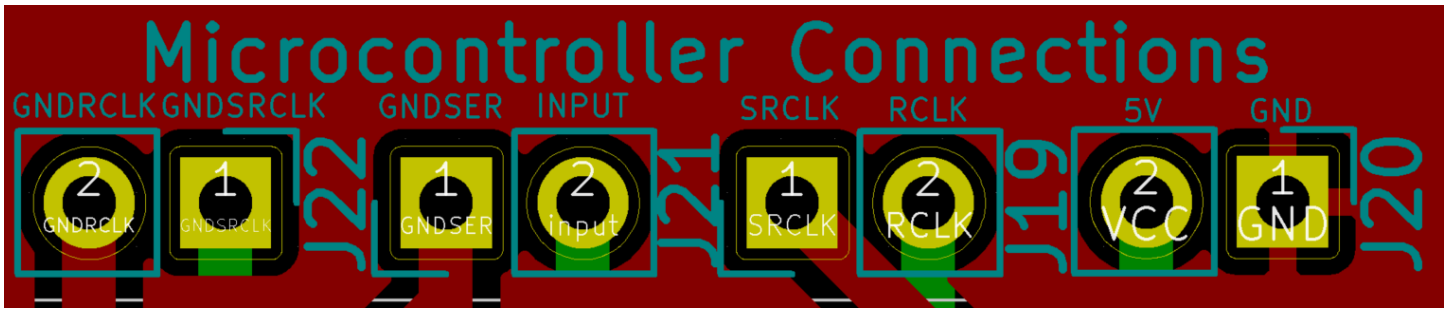


Figure 10: Input Interface for PCB

The top of the has most of the features. This was done in an attempt to prevent the pcb from having to be flipped back and forth to find locations for soldering. In keeping with this attempt, the silkscreen is entirely on the front layer of the PCB. Although, once the components are added, the visibility of the silkscreen is minimal.

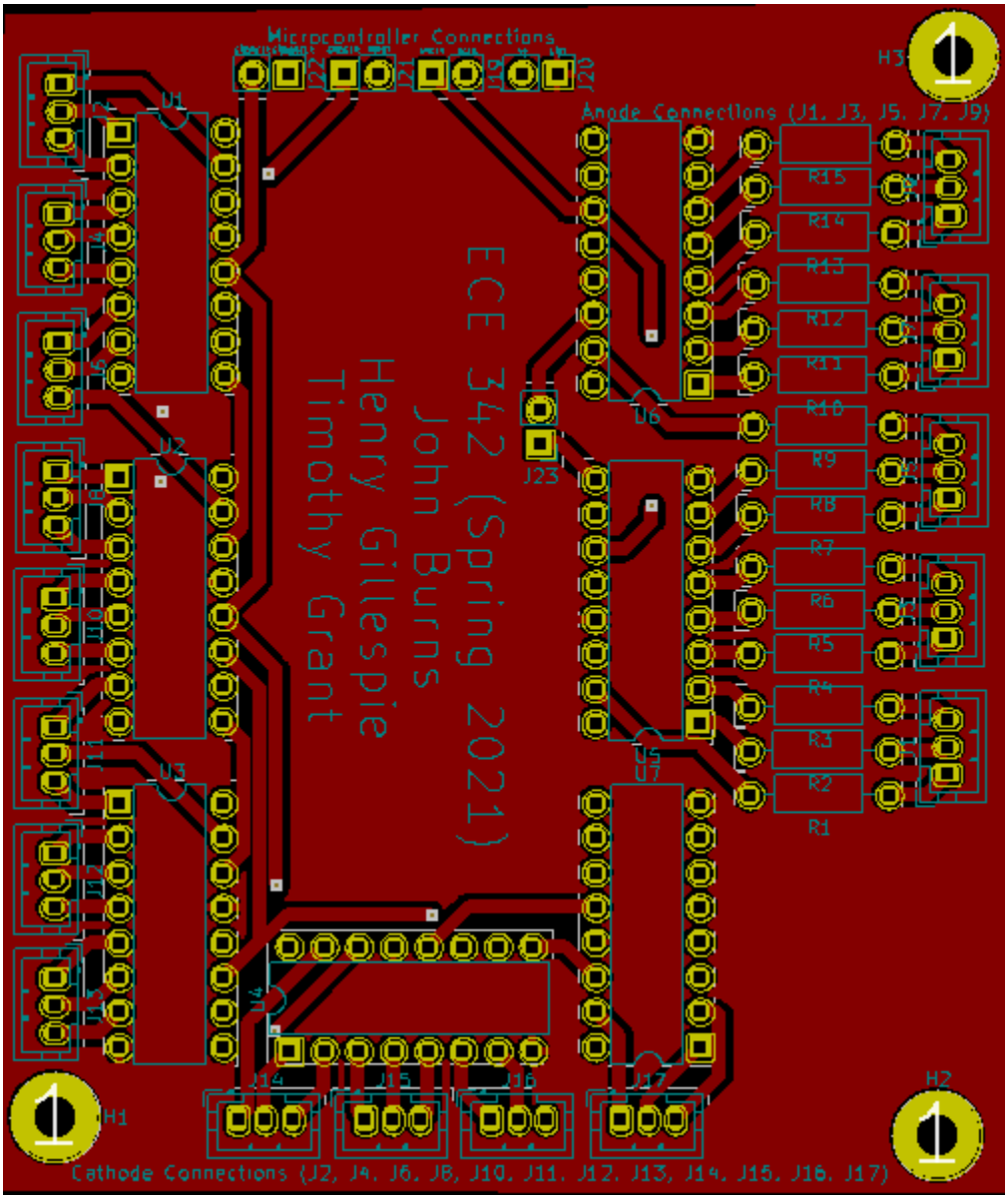


Figure 11: Top layer of PCB

The bottom layer was used when the top layer could not. Some of the traces for the input were done entirely through the bottom layer to increase clarity, but other points just show an unhindered view of the board. Once the PCB is fixed to the bottom of the container, the layer will not be visible.

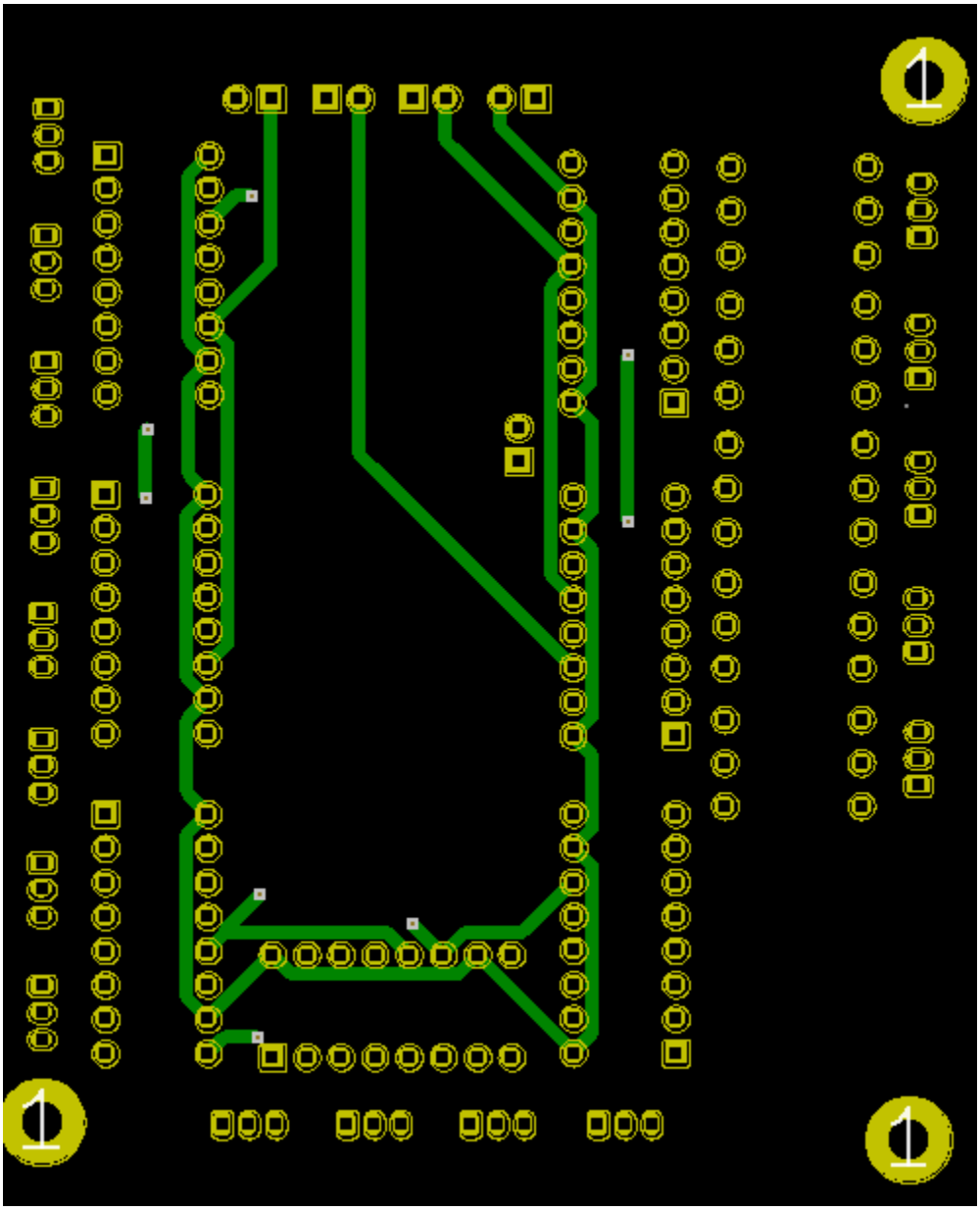

Figure 12: Bottom Layer of PCB

## Section 2.7: Bill of Materials

In order to be able to reconstruct the project from the basic materials that the original designers created the project from, it is necessary to consult the Bill Of Materials to ensure all parts used are the same as the materials used by the original creators. Below can be found all parts used in the project, along with their associated quantity, unit cost, manufacturer number, manufacturer, value, dimensions, and datasheet.

Table 2: Basic Information BOM

| Material | Designation | Name | Quantity | Unit Cost |
|---|---|---|---|---|
| 1 | N/A | Plywood | 1 | $4.83 |
| 2 | N/A | Bolt | 3 | $0.37 |
| 3 | N/A | ¼-20 Nut | 6 | $0.22 |
| 4 | N/A | Wing Nut | 3 | $0.30 |
| 5 | N/A | Machine Screw | 5 | $0.15 |
| 6 | N/A | #10-32 Nut | 5 | $0.05 |
| 7 | N/A | #10 Wood Screw | 3 | $0.10 |
| 8 | N/A | Corner Brace | 3 | $0.63 |
| 9 | N/A | Gorilla Glue | 0.01 | $5.00 |
| 10 | N/A | RGB Common Cathode LED | 175 | $0.09 |
| 11 | N/A | PCB | 1 | $3.00 |
| 12 | A1 | Arduino Uno w/ USB Connector | 1 | $17.60 |
| 13 | U8 | Wireless Bluetooth Module | 1 | $10.00 |
| 14 | R1-R15 | 2.2kΩ Resistor | 15 | $0.05 |
| 15 | U1-U7 | 8-bit Shift Register | 7 | $0.28 |
| 16 | J19-J23 | Pin Header | 10 | $0.01 |
| 17 | J1-J17 | Connector Pin | 17 | $0.06 |
| 18 | N/A | Jumper Wires | 62 | $0.06 |
| 19 | N/A | Roll of Silver Plated Copper Wire (26 gauge, 100ft) | 1 | $9.99 |
| 20 | N/A | Roll of Solder (Lead Free, 100ft) | 2 | $1.66 |
| | | | | Total: $78.61 |

Table 3: Manufacturing Information BOM

| Material | Manufacturer Number | Manufacturer | Value |
|---|---|---|---|
| 1 | NA | Home Depot | N/A |
| 2 | 800080 | Everbilt | N/A |
| 3 | 801736 | Everbilt | N/A |
| 4 | 802371 | Everbilt | N/A |
| 5 | 803331 | Everbilt | N/A |
| 6 | 800272 | Everbilt | N/A |
| 7 | 807491 | Everbilt | N/A |
| 8 | 13619 | Everbilt | N/A |
| 9 | NA | Home Depot | N/A |
| 10 | ED_YW05_RGB-4P-C_100Pcs | EDGELEC | N/A |
| 11 | NA | JLCPCB | N/A |
| 12 | A000052 | Arduino | N/A |
| 13 | B01MQKX7VP | NewZoll | N/A |
| 14 | 294-2.2K-RC | Xicon | 2.2kOhm |
| 15 | 74LV595N,112 | NXP USA Inc. | N/A |
| 16 | 826629-2 | TE Connectivity | N/A |
| 17 | B3B-EH-A(LF)(SN) | JST | N/A |
| 18 | 825 | ADAFRUIT | N/A |
| 19 | CWIR-S003 | KBeads | N/A |
| 20 | D96SCF192 | MULTICORE (SOLDER) | N/A |

Table 4: Material Specifications BOM

| Material | Physical Dimensions (L x W x H) | Datasheet |
|---|---|---|
| 1 | 8” x 48” | N/A |
| 2 | ¼”-20 (Thread) x 4.5” (L) | N/A |
| 3 | ¼”-20 (Thread) | N/A |
| 4 | ¼”-20 (Thread) | N/A |
| 5 | #10-32 x 1” | N/A |
| 6 | #10-32 | N/A |
| 7 | #10 x ¾” | N/A |
| 8 | 1” x 1” | N/A |
| 9 | NA | N/A |
| 10 | 5mm * 5mm * 29.5mm | https://www.sparkfun.com/datasheets/Components/YSL-R596CR3G4B5C-C10.pdf |
| 11 | 73mm * 85mm | N/A |
| 12 | 68.6mm * 53.4 mm | https://www.farnell.com/datasheets/1682209.pdf |
| 13 | 37.5mm * 16.5mm * 4mm | https://www.estudioelectronica.com/wp-content/uploads/2018/09/istd016A.pdf |
| 14 | 5.00mm x 12.00mm | https://www.mouser.com/datasheet/2/351/Royal_Electric_XC_600035-1893446.pdf |
| 15 | 20mm * 5mm | https://rocelec.widen.net/view/pdf/guynwn3fbt/PHGLS18880-1.pdf?t.download=true&u=5oefqw |
| 16 | 2.5mm * 2.5mm * 8mm | https://www.te.com/commerce/DocumentDelivery/DDEController?Action=srchrtrv&DocNm=826629&DocType=Customer+Drawing&DocLang=English&PartCntxt=826629-2&DocFormat=pdf |
| 17 | 7mm * 5mm | https://www.tlcelectronics.com/pub/media/wysiwyg/Datasheets/eXH.pdf |
| 18 | 205mm length | http://www.farnell.com/datasheets/2843167.pdf |
| 19 | 100ft * (0.157in diameter) | N/A |
| 20 | 2m length | http://www.farnell.com/datasheets/1724063.pdf |

## Section 2.8: Time Report

The time that was required by each of the 3 members of the group can be found below. It is important to remember that included in this time is the design process and the build process. For applications where the reader desires to create this project on their own, the time required will be significantly shortened due to the fact that this document encloses all of the required specifications, code, design features, and materials. This allows a reader to only be required to build the project from the already created design provided within this report.

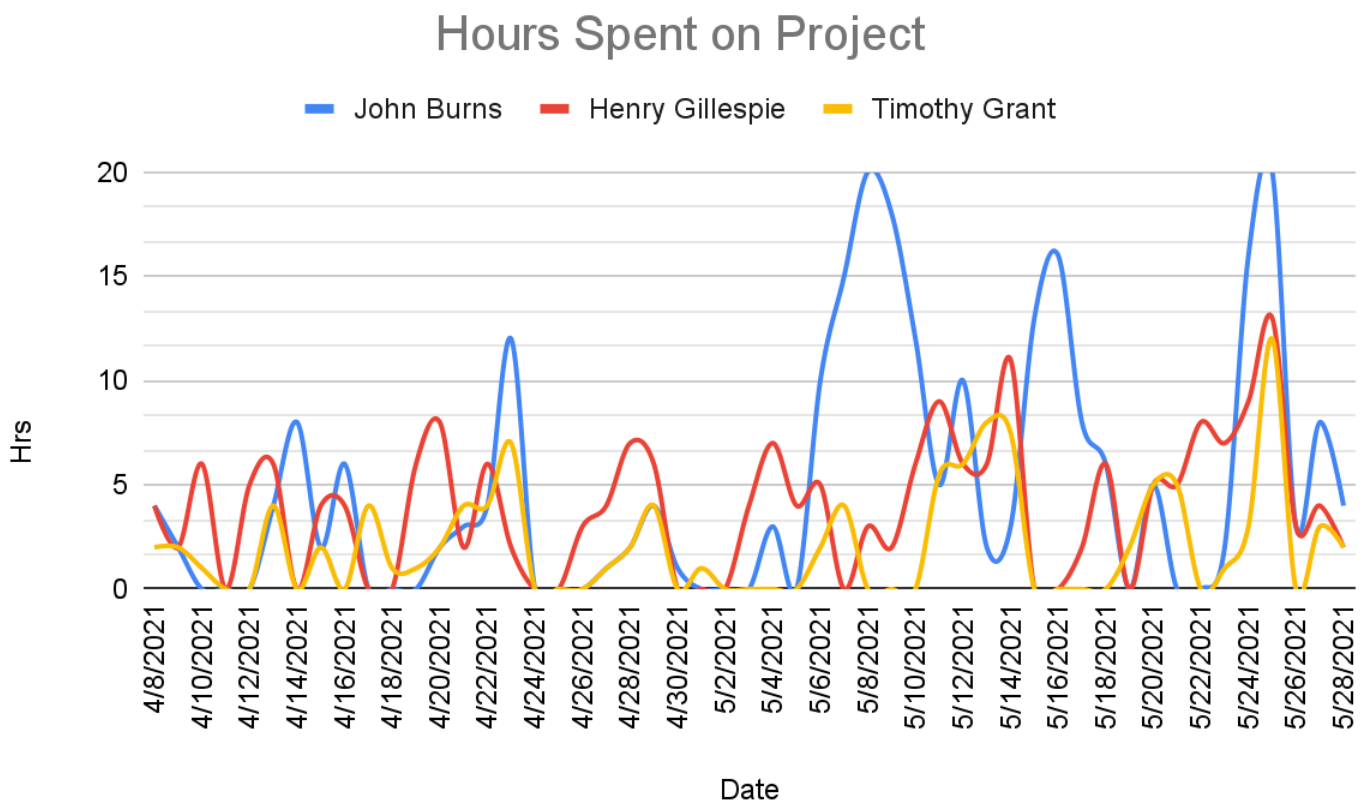


Figure 13: Time Report

## Section 3: Conclusion

The LED visualizer project was conceptualized by Brandilynn Coker, who acted as our mentor to ensure that we were on pace to complete the project within the space of the term. This project took a lot of time as shown in the time report, but it had a lot of high notes. We believe that, because this project was so time consuming, it gave an opportunity to be extremely through as well as add a lot of options. The other projects have not been demod yet, but it has been said that the LED visualizer projects vary vastly in their implementation.

## Appendix A.1: visualizer.py

```
#!/usr/bin/python
"""
Author: John Burns
Last Modified: 5/26/2021
OSU Email Address: burnsjo@oregonstate.edu
Course Number ECE 342
Project: LED Visualizer Group 2           Due Date: 5/28/2021
"""

from tkinter import *
from Animation import Animation
from ArduinoInterface import *
from Layer import LEDLayer
from CheckMarkList import CheckButtonBar
from Slider import SliderFrames
from Display import Display
from pythonCOMport import *
import sys
import numpy as np

#Constants that increase the readability of the slider array in one of the functions
shown below.
LAYER = 0
FRAME = 1
SPEED = 2
FRAMECNT = 3
EFFECT = 4

"""
A class that sets up the visualizer multiwindow.
The first window sends numeric data to sendValToArd(num) based on the chosen
animation.
The second window is built to customize animation 4, 5, and 6.
All three animations are saved during the session, but would have to be rebuilt if
the session was closed.
Clicking submit sends the animations to animationsForHeader(animations) based on the
selections made in the interface.
To change an animation, select the animation (default 4), choose a frame (default 0),
choose a layer (default 0),
   choose a color (default none or black), finally click an LED.
You will not only keep animations when going to other tabs, you will also keep the
place you were last working on that animation
   when you come back.
```

```
The speed of an animation is adjustable. Just adjust the slider.
The maximum frame count of an animation is adjustable. Just adjust the slider.
   Adjusting the maximum frame count will delete frames past that frame.
   You needn't worry about the current frame slider. It will be always be in a valid
state (will be reduced if max count is lower).
You may fill the animation with set commands. It is trivial to add more, just add
them in the Animation.py file.
You may demo the animations by clicking the play Animation button.

For the curious... non of the features of the default animation (1, 2, and 3) are
adjustable. Which is why they don't show up under
   advanced settings bar.

Verified os: Windows and Linux. Due to the pack and grid methods, the windows should
show up with the proper size on any system.
   The system as a whole will not be cross platform, but this gui is.

Parameters: master, the top frame (Frame).
"""
class MultiWindow:
   def __init__(self, master: Tk):
      #Setup windows.
      self.master = master
      self.secondaryWin = None
      self.playWindow = None

      #Prepare for button layer.
      self.currentBtnLst = [None]
      self.currentBtnLayer = None

      #Prepare for sliders.
      self.sliders = None

      #setup global variables that can be changed with widget actions.
      #These do not change between animations.
      self.colorChosen = IntVar(value = -1)
      self.commandChosen = IntVar(value = -1)
      self.animationChosen = IntVar()
      self.comChosen = IntVar(value = -1)

      #These do change between animations, so there is 1 for each animation.
      self.animations = []
      self.speeds = []
      self.frameCnts = []
```

```
      self.layers = []
      self.frames = []
      self.effects = []
      for i in range(3):
         self.animations.append(Animation())
         self.speeds.append(IntVar(value =
self.animations[self.animationChosen.get()].getSpeed()))
         self.frameCnts.append(IntVar(value =
self.animations[self.animationChosen.get()].getNumFrames()))
         self.layers.append(IntVar())
         self.frames.append(IntVar())
         self.effects.append(IntVar())

      #Make the com frame based on available coms.
      self.sendBtns = []
      self.comFrame = None
      self.submitBtn = None
      self.updateComFrame()

      #layout the buttons and their functionality.
      self.setupMainWindow()

      #Allow for creating another window with
      self.nextBtn = Button(self.master, text="Advanced Settings", width = 15,
command=self.advancedSettingOptions)
      self.nextBtn.grid(row = 2, column = 1)

      #Create an uodate button.
      self.updateBtn = Button(self.master, text="Update COMs", width = 15,
command=self.updateComFrame)
      self.updateBtn.grid(row = 2, column = 2)

   """
   update the sending buttons based on whether a com is chosen.
   """
   def updateComDependentBtns(self) -> None:
      #Activate for selected, Disabled if not.
      val = NORMAL if len(self.listOfComs) > 0 and self.comChosen.get() != -1 else
DISABLED

      #Update the buttons based on their functionality.
      for i in range(len(self.sendBtns)):
         self.sendBtns[i]['state'] = val
      if self.submitBtn:
```

```
            self.submitBtn['state'] = val

    """
    Update the com frame.
    """
    def updateComFrame(self) -> None:
        #Destroy the com frame if it exists.
        if self.comFrame:
            self.comFrame.destroy()

        #Find the descriptions of the available ports.
        self.listOfComs = getDescriptions()
        #If there aren't any coms, remove chosen com.
        if len(self.listOfComs) == 0:
            self.comChosen.set(-1)

        #Make a frame for the com. Put it at the far right. Shape it based number of
coms available.
        self.comFrame = Frame(self.master)
        self.comFrame.grid(row = 0, column=3, rowspan = len(self.listOfComs) if
len(self.listOfComs) > 3 else 3, sticky = N)

        #Update the buttons.
        self.updateComDependentBtns()

        #make the button list.
        self.comChecks = CheckButtonBar(self.comFrame, None, 'Select a COM',
self.comChosen, self.findPort(self.listOfComs), len(self.listOfComs),
self.listOfComs, -2)

    """
    update the port and get the ports. Verbose mode is activated if a comport was
chosen.
    Returns a function handle to be executed when check button is clicked. [funct1,
funct2].
    """
    def findPort(self, listOfComs: list):
        return lambda: [self.updateComFrame(),
getPort(listOfComs[self.comChosen.get()], True if self.comChosen.get() > -1 else
False)]

    """
    Sets up the main window. The main wondow contains 6 windows in a 2*3 grid of
buttons.
```

```
   The 6 buttons send a number to a function to be used by the Arduino Interface.
   """
   def setupMainWindow(self) -> None:
      #Keep track of the button number.
      counter = 0
      #Make 2 rows.
      for i in range(2):
         #Make 3 columns.
         for j in range(3):
            #Create uniform buttons with a function handle that allows for a number
to be passed every time it is called.
            self.sendBtns.append(Button(basic, text =f"Animation {counter+1}", width
= 15, height = 1, command = self.choiceLambda(counter)))
            self.sendBtns[counter].grid(row = i, column = j)
            if len(self.listOfComs) == 0 or self.comChosen.get() == -1:
               self.sendBtns[counter]['state'] = DISABLED
            counter += 1

   """
   function handle generator. This allows the buttons to use a command with a numeric
input.
   parameters: num, the numeric representation of the user's choice.
   returns: a lambda, a function handle that can be run any time one of the
determined buttons is clicked.
   """
   def choiceLambda(self, num: int):
      return lambda: self.displayChoice(num)

   """
   Let the user know what they did as well as let them know how to continue.
   parameters: num, the numeric representation of the user's choice.
   """
   def displayChoice(self, num: int) -> None:
      #Send the value over the to arduino once the user closes the messagebox.
      sendValToArd(num, self.listOfComs[self.comChosen.get()])

   """
   function handle generator. This allows one function to control multiple events via
a parameter.
   parameters: animation, the number attached to the animation button clicked.
   returns: a lambda, a function handle that can be run any time, with the preset
value.
   """
   def sendAnimationsLam(self, animations: list):
```

```
      return lambda: animationsForHeader(animations,
self.listOfComs[self.comChosen.get()])

   """
   Set up an advanced setting window temporarily closing the main window. This setup
process only needs
      to happen if this Secondary Window has not already been setup.
   As can be seen below the functionaliy of this is split into making a tab frame and
then an action frame.
      The tab frame allows one to select an animation to modify, go to the basic
settings window, or send the animation
      to the arduino (Submit)
   """
   def advancedSettingOptions(self) -> None:
      #If a window is not already setup. Make one.
      if not self.secondaryWin:
         #Make a top level window frame.
         self.secondaryWin = Toplevel()
         #If the user closes the GUI, exit the program.
         self.secondaryWin.protocol("WM_DELETE_WINDOW", sys.exit)

         #Setup the top button (tab) frame and action frame.
         #The action frame is the main place where settings are changed.
         tabFrame = Frame(self.secondaryWin)
         actionFrame = Frame(self.secondaryWin)
         tabFrame.pack(anchor = W)
         actionFrame.pack(anchor = W)

         #Allow easy access to the future btn pannel.
         self.btnFrame = None

         #Setup 3 animation tabs. They are Animations whose matrix will be edited by
changing the colors of the buttons.
         for i in range(3):
            Button(tabFrame, text =f"Animation {i+4}", width = 12, command =
self.selectActLam(i)).pack(side = LEFT)
         Button(tabFrame, text="Play Animation", width = 12, command =
self.playAnimationWindow).pack(side = LEFT)
         #Allow the user to return to the original window.
         Button(tabFrame, text="Basic Settings", width = 12,
command=self.basicSettingOptions).pack(side = LEFT)
         #Allow the user to submit the edited animations to the Arduino.
         self.submitBtn = Button(tabFrame, text="Submit", bg = 'green', width = 12,
command = self.sendAnimationsLam(self.animations))
```

```
            self.submitBtn.pack(side = LEFT)

            #Put the window in the top left so that no matter the os, it will show as
best as possible.
            self.secondaryWin.geometry("+0+0")
            #Hide the main window.
            self.master.withdraw()

            #Remove the animation demo window.
            if self.playWindow:
               self.playWindow.destroy()
               self.playWindow = None

            #Setup the action frame.
            self.setupActionFrame(actionFrame)
         else:
            #If the secondary window has already been setup, then just show the
secondary window and hide the main window.
            #withdraw() turns the window into a hidden icon. deiconify() undoes this
action.
            self.secondaryWin.deiconify()
            #if there is a play window, remove it.
            if self.playWindow:
               self.playWindow.destroy()
               self.playWindow = None
            self.master.withdraw()

         self.submitBtn['state'] = DISABLED if len(self.listOfComs) == 0 or
self.comChosen.get() == -1 else NORMAL

      """
      Method for togling the visible window.
         Hides the secondary window and shows the primary window.
      """
      def basicSettingOptions(self) -> None:
         self.secondaryWin.withdraw()
         self.master.deiconify()

      """
      Create a play Animation window.
      """
      def playAnimationWindow(self) -> None:
         #Create, title, place, and define closing behavior.
         self.playWindow = Toplevel()
```

```
      self.playWindow.title("Animation Demo")
      self.playWindow.geometry("+0+0")
      self.playWindow.protocol("WM_DELETE_WINDOW", sys.exit)

      #Hide the secondary window.
      self.secondaryWin.withdraw()

      #Show the display window, with ability to go back to secondary window.
      Display(self.playWindow, self.animations[self.animationChosen.get()],
["Advanced Settings", self.advancedSettingOptions])

   """
   Sets up the frame in the secondary window where the user makes all the edits.
   parameters: actionFrame, the frame to be used for the action frame.
   """
   def setupActionFrame(self, actionFrame: Frame) -> None:
      #Save the frame used for the buttons (changes every time one of the settings
changes).
      self.btnFrame = Frame(actionFrame)
      #Make a frame for the color selection frame.
      colorSelFrame = Frame(actionFrame)
      #make a command frame for the far right.
      commandSelFrame = Frame(actionFrame)

      #Put the frames on the window with some asthetic intent.
      colorSelFrame.pack(side = LEFT, fill = Y)
      self.btnFrame.pack(side = LEFT, fill = BOTH)
      commandSelFrame.pack(side = LEFT, fill = Y)

      #Make the slider frames and let them be accessessable elsewhere.
      self.sliders = SliderFrames(self.btnFrame, self.getInfo())

      #Setup the button layer functionality.
      self.LEDSelectionSetup()

      #Setup a color selection bar which enable/disable the buttons as well as allow
you to click the buttons until the user decides
      #to change color.
      CheckButtonBar(colorSelFrame, self.currentBtnLst, 'Possible Colors',
self.colorChosen)

      #Add a check list for the right side of the gui.
      listOfCommands = self.animations[self.animationChosen.get()].specialAnimations
```

```
      CheckButtonBar(commandSelFrame, None, 'Choose a Command', self.commandChosen,
self.doAction, len(listOfCommands), np.array(listOfCommands)[:, 0].tolist())

   """
   Fill the animation acording to the button checked. Update the layer buttons as
well.
   """
   def doAction(self) -> None:
      #Remove the current btn layer.
      self.currentBtnLayer.remove()

      #Execute the command on the current animation.
      if self.effects[self.animationChosen.get()].get() == 0:

self.animations[self.animationChosen.get()].specialAnimations[self.commandChosen.get(
)][1](self.frames[self.animationChosen.get()].get(),
self.layers[self.animationChosen.get()].get(), self.colorChosen.get())
      elif self.effects[self.animationChosen.get()].get() == 1:

self.animations[self.animationChosen.get()].specialAnimations[self.commandChosen.get(
)][1](self.frames[self.animationChosen.get()].get(), self.colorChosen.get())
      elif self.effects[self.animationChosen.get()].get() == 2:

self.animations[self.animationChosen.get()].specialAnimations[self.commandChosen.get(
)][1](self.colorChosen.get())

      #find the chosen animation based on the list of animations.
      chosenAnimation = self.animations[self.animationChosen.get()]
      #Setup the layer corresponding to the layer information chosen.
      self.setLayer(chosenAnimation, self.frames[self.animationChosen.get()].get())

      self.commandChosen.set(-1)



   """
   Retrieve the information required for the sliders.
      It is setup in such a way that more of either type of slider can be added
fairly simply.
      format of the parameters is: text (string), lo (int), hi (int), variable (needs
.get() method), command (function handle).
   returns: info, a 2 element array of slider parameters.
   """
   def getInfo(self) -> list:
      #Get the chosen matrix based on the chosen animation and chosen frame (matrix).
```

```
      matrix =
self.animations[self.animationChosen.get()].getFrame(self.frames[self.animationChosen
.get()].get())
      #Parameters for the info changing sliders (layer and frame).
      posInfo = [['Choose Layer', 0, matrix.getzlen()-1,
self.layers[self.animationChosen.get()], self.selAct], \
         ['Choose Frame', 0,
self.animations[self.animationChosen.get()].getNumFrames()-1,
self.frames[self.animationChosen.get()], self.selAct]]

      #Parameters for the feature changing sliders (speed and max frame count).
      featureInfo = [['Choose Speed (fps)', 1,
self.animations[self.animationChosen.get()].getSpeed(),
self.speeds[self.animationChosen.get()], self.selAct], \
         ['Max Frame Count', 1,
self.animations[self.animationChosen.get()].getNumFrames(),
self.frameCnts[self.animationChosen.get()], self.selAct], \
         ['Effect (layer, frame, whole)', 0, 2,
self.effects[self.animationChosen.get()], self.selAct]]

      return [posInfo, featureInfo]

   """
   Setup the selection of LEDs. A 5x5 (size of a matrix layer) button frame is
presented. Based on the layer, animation, frame,
      and color the values of the animations can be changed with a click. The change
in the animations is reflected by the
      color of these buttons.
   """
   def LEDSelectionSetup(self) -> None:
      #Get the chosen matrix based on the chosen animation and chosen frame (matrix).
      matrix =
self.animations[self.animationChosen.get()].getFrame(self.frames[self.animationChosen
.get()].get())
      #Make a new btn layer.
      self.currentBtnLayer = LEDLayer(self.btnFrame, matrix, self.colorChosen,
self.layers[self.animationChosen.get()])
      #Add the button layer to the button list. This makes it easier to destroy and
rebuild later.
      self.currentBtnLst[0] = self.currentBtnLayer.buttonLst

   """
   function handle, allows multiple buttons to have the same function with slightly
different (preset) values.
```

```
   parameters: animSelected, the animation that is being edited (int).
   returns: function handle, a way to change the features of the animation being
changed based on the variables set up
      and animation edited so far (lambda).
   """
   def selectActLam(self, animSelected: int):
      return lambda: self.selAnimAct(animSelected)

   """
   Definition of what happens when an animation is selected. It resets the sliders to
the values they were when editing that
      animation as well as changes the btn layer to the layer corresponding to the
sliders.
   parameters: animSelected, the animation being edited (int).
   """
   def selAnimAct(self, animSelected: int) -> None:
      #Remove the current btn layer.
      self.currentBtnLayer.remove()

      #update the chosen animation variable.
      self.animationChosen.set(animSelected)

      #get the scales from the information saved on the sliders (when the sliders
were created).
      scales = []
      for strip in range(len(self.sliders.listofscaleFrame_sels)):
         for scale in range(len(self.sliders.listofscaleFrame_sels[strip])):
            scales.append(self.sliders.listofscaleFrame_sels[strip][scale][1])

      #Update the scales to how they were with the current animation.
      scales[LAYER].set(self.layers[self.animationChosen.get()].get())
      scales[FRAME].set(self.frames[self.animationChosen.get()].get())
      scales[SPEED].set(self.speeds[self.animationChosen.get()].get())
      scales[FRAMECNT].set(self.frameCnts[self.animationChosen.get()].get())
      scales[EFFECT].set(self.effects[self.animationChosen.get()].get())

      #find the chosen animation based on the list of animations.
      chosenAnimation = self.animations[self.animationChosen.get()]
      #Setup the layer corresponding to the layer information chosen.
      self.setLayer(chosenAnimation, self.frames[self.animationChosen.get()].get())

   """
   Very similar to the above method however it had a fundemental difference. One is
that
```

```
      it is called by sliders (which pass changed parameters so animSelected would be
flagged
      inapropriately), but also that the variable are updated based on the sliders
rather than
      the slider on the variables. Since one slider is being used for all three
animations
      (akward things happen if you try to delete and resetup the sliders due to the
variables being attached)
      the best course of action is to update the variables based on the sliders and
then when the animation
      changes, change the slider values to those animation dependent variables.
   parameters: selected, the feature selected. Ghost parameter, not in use as it
would require a seperate function
      for each parameter. As seen in the description an adequate work arond was
found.
   """
   def selAct(self, selected: int) -> None:
      #Remove the current button layer.
      self.currentBtnLayer.remove()

      #get the scales from the information saved on the sliders (when the sliders
were created).
      scales = []
      for strip in range(len(self.sliders.listofscaleFrame_sels)):
         for scale in range(len(self.sliders.listofscaleFrame_sels[strip])):
            scales.append(self.sliders.listofscaleFrame_sels[strip][scale][1])

      #Update the variables for the animation depended variables based on the scale's
current value.
      #Update the layer variable based on the slider.
      self.layers[self.animationChosen.get()].set(scales[LAYER].get())
      #Update the maximum frame variable based on the slider.
      self.frameCnts[self.animationChosen.get()].set(scales[FRAMECNT].get())
      #Update the maximum effect variable based on the slider.
      self.effects[self.animationChosen.get()].set(scales[EFFECT].get())

      #Protect the user from moving the frame slider past the maximum value.
      if scales[FRAME].get() < scales[FRAMECNT].get():
         #Update the frame variable based on the slider.
         self.frames[self.animationChosen.get()].set(scales[FRAME].get())
      else:
         #Reset the slider to the maximum value if the user tried to take it past.
         scales[FRAME].set(scales[FRAMECNT].get()-1)
```

```
      #Update speed variable based on the slider.
      self.speeds[self.animationChosen.get()].set(scales[SPEED].get())

      #Make the chosen animation easily accessable.
      chosenAnimation = self.animations[self.animationChosen.get()]
      #Update the speed and maximum frame count of the chosen animation to the values
stored in the variables updated above.
      chosenAnimation.setSpeed(self.speeds[self.animationChosen.get()].get())

chosenAnimation.changeFrameCnt(self.frameCnts[self.animationChosen.get()].get())

      #If the user is moving the maximum frame count below the frame selected,
      #  move frame selected with it.
      frameNum = 0
      if chosenAnimation.getNumFrames() <=
self.frames[self.animationChosen.get()].get():
         #The frame chosen must be the maximum frame possible (0 indexed list)
         frameNum = chosenAnimation.getNumFrames() - 1
      else:
         #The frame chosen is just the value in the variable.
         frameNum = self.frames[self.animationChosen.get()].get()

      #Update the layer based on the animation and frame number.
      #Be warned that lowering the frame number will erase any custom frames in the
animation after that point.
      self.setLayer(chosenAnimation, frameNum)

   """
   Create a new layer based on the chosen values.
   parameters: chosenAnimation, the animation that has been chosen (IntVar).
               frameNum, the frame (matrix) to select (int).
   """
   def setLayer(self, chosenAnimation: IntVar, frameNum: int) -> None:
      #Make the chosen Matrix easy to access.
      chosenMatrix = chosenAnimation.getFrame(frameNum)
      #Create a new button layer.
      self.currentBtnLayer = LEDLayer(self.btnFrame, chosenMatrix, self.colorChosen,
self.layers[self.animationChosen.get()])
      self.currentBtnLst[0] = self.currentBtnLayer.buttonLst

#Initilizer a GUI window, with a title at the top left of the screen.
basic = Tk()
basic.title("LED Visualizer GUI")
basic.geometry("+0+0")
```

```
#Setup all frames and windows in the gui.
window = MultiWindow(basic)
#Allow the program to terminate if the main window is closed.
basic.protocol("WM_DELETE_WINDOW", basic.destroy)
#Listen for events in the setup gui until the window is destroyed.
basic.mainloop()
```

## Appendix A.2: Slider.py

```
"""
Author: John Burns
Last Modified: 5/15/2021
OSU Email Address: burnsjo@oregonstate.edu
Course Number ECE 342
Project: LED Visualizer Group 2           Due Date: 5/28/2021
"""

from tkinter import *

"""
Built to makes slider frames more intuitive for the gui.
Make a strips of frames and insert frames of sliders into those strips based on the
infos parameter.

Though this could be expandable with multiple
    types of sliders as well as sub sliders, that is untested and likely will look
awfull.
    parameters: frame, a frame to put the sliders on.
                infos, an array of arrays. Each sub array should have the parameter
list for the creation of a slider.
"""
class SliderFrames:
    def __init__(self, frame: Frame, infos: list[list]):
        #Make a frame for the slider.
        self.frame = Frame(frame)
        self.frame.pack(anchor = W)

        #Make it easy to retrieve the frames and scales created.
        self.listofscaleFrame_sels = [[], []]
        self.stripFrames = []

        #Create strips of frames, with each frame containing LabelFrames of sliders.
        for i in range(len(infos)):
            #Create a frame to put sliders in (horizontally)
            self.stripFrames.append(Frame(self.frame))
            self.stripFrames[i].pack(anchor = W)

            for j in range(len(infos[i])):
                #Create the individual slider in the strip.
                infos[i][j].insert(0, self.stripFrames[i])
                self.listofscaleFrame_sels[i].append(self.scaleSetup(infos[i][j]))
```

```
    """
    Make a scale based on the parameters given in info.
    parameters: info, a list of parameters in the format selFrame (Frame class),
title (string),
        lo (int), hi (int), variable (needs a get function), command (function
handle).
    returns: references to frames and sliders created (list).
    """
    def scaleSetup(self, info: list) -> list:
        #Parse info.
        selFrame, title, from_, to, var, command = info

        #Create a LabelFrame for the slider.
        scaleFrame = LabelFrame(selFrame, text = title)
        scaleFrame.pack(side = LEFT, anchor=W)

        #Create a slider based on the info parameter.
        sel = Scale(scaleFrame, from_=from_, to=to, command = command,
orient=HORIZONTAL)

        #Set the default value.
        sel.set(var.get())
        sel.pack(side = LEFT)

        return [scaleFrame, sel]
```

## Appendix A.3: pythonCOMport.py

```
# File:       pythonCOMport.py
# Author:     Henry Gillespie
# Date:       May 25, 2021
# Purpose:    This program defines the serial connection to the arduino by searching
the available COM ports

import serial.tools.list_ports
from tkinter import messagebox

# This function loops through the available COM ports to find the Arduino Nanos
provided by Tekbots.
# When it has found the device, the function returns its COM port as a string.
def getPort(description: str, showBox: bool) -> str:
    serList = serial.tools.list_ports.comports()
    descriptions = []
    for i in range(len(serList)):
        descriptions.append(serList[i].description) # append description
    try:
        idx = descriptions.index(description)
        if showBox:
            messagebox.showinfo(f"{serList[idx].name} Selected", "Please click okay
or close the window to continue.")
        return serList[idx].name # this will return COMX
    except:
        messagebox.showerror(f"{description} was unavailble", "Please ensure you have
the desired com inserted." \
                "\nThis will allow you to play animations on the Arduino.\n" \
                "\nNote: You can still make custom animation and view them" \
                "\n\t in the demo tab, you just cannot send them.")
        return ""

# This function returns a list of the descriptions of the available COM ports
def getDescriptions() -> list:
    serList = serial.tools.list_ports.comports()
    descriptions = []
    for i in range(len(serList)):
        descriptions.append(serList[i].description)
    return descriptions
```

## Appendix A.4: Matrix.py

```
"""
Author: John Burns
Last Modified: 5/15/2021
OSU Email Address: burnsjo@oregonstate.edu
Course Number ECE 342
Project: LED Visualizer Group 2           Due Date: 5/28/2021
"""

"""
The definition for how a LED matrix can be defined.
    Primarily as a x, y, z list of colors. This particular implementation also has a limit on
    how high the value can be set.
There are two modes to call this class's constructor.
    The default LEDMatrix() with default size of 5x5x7 with a limit on the value of 63.
    The full parameter list can also be entered thanks to *args.
        LEDMatrix(x (int), y (int), z (int), lim (int)) can be called to make a custom matrix.
parameters: *args, either 4 parameters long or 0 parameters long. Otherwise nothing is set.
"""
class LEDMatrix:
    def __init__(self, *args):
        #full parameter list
        if len(args) == 4:
            #Parse expected parameters.
            x, y, z, lim = args
            #Make a matrix of the requested side.
            self.matrix = [[[0 for k in range(z)] for j in range(y)] for i in range(x)]
            #set values.
            self.x = x
            self.y = y
            self.z = z
            self.lim = lim
        #Default parameter list.
        elif len(args) == 0:
            #Make a matrix of the default 5x5x7 size.
            self.matrix = [[[0 for k in range(7)] for j in range(5)] for i in range(5)]
            #Set the values to the default values.
            self.x = 5
```

```
            self.y = 5
            self.z = 7
            self.lim = 63

    #method for getting the x value.
    def getxlen(self) -> int:
        return self.x

    #method for getting the y value.
    def getylen(self) -> int:
        return self.y

    #method for getting the z value.
    def getzlen(self) -> int:
        return self.z

    #method for getting the lim value.
    def getlim(self) -> int:
        return self.lim

    #method for getting the matrix.
    def getMatrix(self) -> list[list[list[int]]]:
        return self.matrix

    """
    Sets a value in the matrix.
    Note: implementing the color of a bulb is implented in getRowValue.
    parameters: x, the x value on the layer (int).
                y, the y value on the layer (int).
                z, the layer (int).
                val, the value of the corresponding bulb (int).
    returns: wasProb, whether the value was valid (boolean).
    """
    def setVal(self, x: int, y: int, z: int, val: int) -> bool:
        #Allow for problem detection.
        wasProb = False

        #Try to prevent users from entering bogus values.
        if val <= self.lim and val >= 0:
            self.matrix[x][y][z] = val
        else:
            wasProb = True

        return wasProb
```

```
    """
    Retrieve te value from the matrix.
    paramaters: x, the x value on the layer (int).
                y, the y value on the layer (int).
                z, the layer (int).
    returns: value, the value stored in the matrix at that location.
    """
    def getVal(self, x: int, y: int, z: int) -> int:
        return self.matrix[x][y][z]

    """
    Provide a way to clear the value. Same as setting the value to 0.
    paramaters: x, the x value on the layer (int).
                y, the y value on the layer (int).
                z, the layer (int).
    """
    def clearVal(self, x: int, y: int, z: int) -> None:
        self.matrix[x][y][z] = 0

    """
    Provide a way to clear the entire matrix. Rebuilds the matrix by setting every
value to 0.
    """
    def clearMatrix(self) -> None:
        self.matrix = [[[0 for k in range(self.z)] for j in range(self.y)] for i in
range(self.x)]

    """
    Provide a way to get an intelligent string of the matrix.
        Matrices are printed as layers from bottom to top.
    returns: outStr, the string to be used (string).
    """
    def toString(self) -> str:
        outStr = ""
        for z in range(self.z):
            for y in range(self.y):
                for x in range (self.x):
                    outStr += str(self.matrix[x][y][z])
                    #Don't put strings at the end of printout.
                    if x < self.x - 1:
                        outStr += "   "
                #Put a new line at the end of every row.
                outStr += "\n"
```

```
            #Put a new line at the end of every layer except the last one.
            if z < self.z - 1:
                outStr += "\n"
        return outStr

    #Overwriting the default address print statement.
    def __str__(self) -> str:
        return self.toString()

    #Written by Henry Gillespie for getting his binary numbers.
    def getRowValue(self, row: int, color: int) -> int:
        # This function calculates the values that will fill the header file. This
creates the correct value (in binary) to push to the LEDs
        # Each value is multiplied by a factor of 2^(3i) to set it to a different
value. The color variable allows the writeFile function to
        # get switch between / and % from the values that are in the Colors
dictionary in the main visualizer.py file.
        layer = int(row / self.y)
        horizontalRow = row % self.y
        value = 0

        for i in range(self.x):
            if color == 0:
                value += int(self.matrix[horizontalRow][i][layer] / 8) * 2**(3*i)
            else:
                value += (self.matrix[horizontalRow][i][layer] % 8) * 2**(3*i)
        return value
```

## Appendix A.5: Layer.py

```
"""
Author: John Burns
Last Modified: 5/15/2021
OSU Email Address: burnsjo@oregonstate.edu
Course Number ECE 342
Project: LED Visualizer Group 2           Due Date: 5/28/2021
"""

from tkinter import *
from Colors import *
from Matrix import LEDMatrix
import numpy as np

"""
Built for the display and edit of the layer values of a given layer and matrix.
   It is implemented in such a way that buttons are only selectable if a color is
selected.
   The buttons will also change color in accordance with the color chosen.
paramaters: master, the frame that button layer will be placed into (Frame).
            matrix, the LED layer is within (Matrix).
            colorChosen, the variable assigned for changing a button's color (IntVar)
            layerChosen, the variable assigned for keeping track of the desired layer
(IntVar).
"""
class LEDLayer:
   def __init__(self, master: Frame, matrix: LEDMatrix, colorChosen: IntVar,
layerChosen: IntVar):
      #Make a label frame that tells the user how to use it.
      self.frame = LabelFrame(master, text = 'Select LEDs Below')
      self.frame.pack(anchor = W)

      #Make the parameters easily accessable elsewhere.
      self.matrix = matrix
      self.layerChosen = layerChosen
      #Get the row and column of a set layer, using numpys easy splicing.
      self.layer = np.array(self.matrix.getMatrix())[:, :,
self.layerChosen.get()].tolist()
      self.colorChosen = colorChosen

      #Make the buttons available to the class.
      self.buttonLst = []

      counter = 0
```

```
      #Create strips of buttons.
      stripFrames = []
      for y in range(len(self.layer[0])):
         stripFrames.append(Frame(self.frame))
         stripFrames[y].pack(anchor = W, pady = 10)

         for x in range(len(self.layer)):
            #Make the button unusable if no color has been selected.
            state = NORMAL if colorChosen.get() != -1 else DISABLED

            #Make a button with the bg color of that layer's value, name it and
define it based on its positioning.
            btn = Button(stripFrames[y], bg = '#' + str(getHex(self.layer[x][y])),
width = 5, state = state, command = self.defAction(x, y, counter))
            btn.pack(side = LEFT, padx = 10)
            #Put the button in the list.
            self.buttonLst.append(btn)
            counter += 1

   """
   Make it easy to remove the layer so that it can be easily replaced.
   """
   def remove(self) -> None:
      self.frame.pack_forget()
      self.frame.destroy()

   """
   Define the action for an active button in the button layer being clicked.
   parameters: x, the column number in the button layer (int).
               y, the row number in the button layer (int).
               btnRef, the direct reference to the button in the grid created
(Button).
   returns: function handle, a preset function handle that can be run whenever an
even is read (lambda).
   """
   def defAction(self, x: int, y: int, btnRef: Button):
      return lambda: self.changeMatrix(x, y, btnRef)

   """
   Change the matrix based on the actions of the user.
      When a button is clicked (that is clickable), the color of that button should
change to the chosen color
      and the matrix value should also change.
   parameters: x, the column number in the button layer (int).
```

```
                y, the row number in the button layer (int).
                btnRef, the direct reference to the button in the grid created
(Button).
    """
    def changeMatrix(self, x: int, y: int, btnRef: Button) -> None:
        #Update the layer based on the user's action.
        self.matrix.setVal(x, y, self.layerChosen.get(), self.colorChosen.get())
        self.layer = np.array(self.matrix.getMatrix())[:, :,
self.layerChosen.get()].tolist()

        #Set the color of the button based on the user's selection.
        self.buttonLst[btnRef]['bg'] = '#' + str(getHex(self.colorChosen.get()))
```

## Appendix A.6: Display.py

```
"""
Author: John Burns
Last Modified: 5/25/2021
OSU Email Address: burnsjo@oregonstate.edu
Course Number ECE 342
Project: LED Visualizer Group 2           Due Date: 5/28/2021
"""

from tkinter import *
from Animation import Animation
from Colors import *
import time
import sys

"""
Class made for displaying animations on a window.
    Animates at the speed of the animation.
    Play button and back button is disabled while animation is playing.
    Pause button pauses the animation in frame (togglable for easy quick clicking,
disabled whne nothing is playing)
    Stop button enables play button and back button as well as restarts the animation
(disabled when nothing is playing).
parameters: window, a window for displaying a canvas on (Toplevel)
            animation, the animation to display (Animation)
            backBtnSettings, a way to get back to another window (txt, command)
"""
class Display:
    def __init__(self, window: Toplevel, animation: Animation, backBtnSettings:
list):
        #Save window for later.
        self.window = window
        self.window.protocol("WM_DELETE_WINDOW", sys.exit)

        #Is the program to be played, or are we paused?
        self.play = BooleanVar(value = True)

        #Make a frame for the buttons.
        self.frame = Frame(self.window)
        self.frame.pack(anchor = W)

        #Keep track of time.
        self.t = None
```

```
        #Save settings for later.
        self.backBtnSettings = backBtnSettings
        #Create the buttons.
        self.createBtns(self.frame)

        #Create a canvas for the display.
        self.canvas = Canvas(self.window, width = 500, height = 700, bg="#000000")
        self.canvas.pack()

        #Save animation features for later.
        self.animation = animation
        self.speed = int(1000/self.animation.getSpeed())

        #Provide a way to count within the class.
        self.counter = 0

    """
    Creates the buttons used for controlling the animation.
        One button for starting the animation.
        One button for going back to the previous window.
        One button for pausing the animation.
        One button for stopping the animation.
    parameters: frame, the frame to create the buttons on (Frame).
    """
    def createBtns(self, frame: Frame) -> None:
        #Create a fresh button for a nice layout.
        #Create the play button.
        self.playButton = Button(frame, text = "Play Animation", command =
self.playAnimation)
        self.playButton.pack(side = LEFT)

        #Create the back button. Account for the user not having a command (no back
button functionality)
        if len(self.backBtnSettings) == 1:
            self.backBtnSettings.append(None)
        self.backButton = Button(frame, text = self.backBtnSettings[0], command =
self.backBtnSettings[1])
        self.backButton.pack(side = LEFT)

        #Create a togglable pause button.
        self.pauseButton = Checkbutton(frame, text="Pause Animation", onvalue=False,
offvalue=True, variable=self.play, command = self.pauseBehavior,
activeforeground="black")
        self.pauseButton.pack(side = LEFT)
```

```
        #Create a stop button.
        self.stopButton = Button(frame, text = "Stop Animation", command =
self.stopAnimation)
        self.stopButton.pack(side = LEFT)

        #Disable the pause and stop button (nothing is playing right now).
        self.pauseButton['state'] = DISABLED
        self.stopButton['state'] = DISABLED

    """
    Define the behavior of clicking pause.
    """
    def pauseBehavior(self) -> None:
        #If we are playing, pause otherwise do nothing.
        if self.play.get():
            #If the animation is done, reenable the play and back button.
            if self.counter == self.animation.getNumFrames():
                #Clear animation if on last frame.
                self.resetCanvas()
            #If the play button is distabled, the animation must be paused.
            elif self.playButton['state'] == DISABLED:
                #Continue animation.
                self.drawCircles()

    """
    Helpfull function to create a circle with the oval function.
    parameters: x, the x coordinate (int)
                y, the y coordinate (int)
                r, the radius (int)
                kwargs, extra arguments for create oval.
    """
    def createCircle(self, x: int, y: int, r: int, **kwargs) -> None:
        self.canvas.create_oval(x-r, y-r, x+r, y+r, **kwargs)

    """
    Resets the canvas.
    """
    def resetCanvas(self) -> None:
        # If playing go ahead, otherwise return (prevents infinite recursion).
        if self.play.get():
            #clear the canvas.
            self.canvas.destroy()
```

```
            #Make a new canvas.
            self.canvas = Canvas(self.window, width = 500, height = 700,
bg="#000000")
            self.canvas.pack()

            #Draw the next frame.
            self.window.after(0, self.drawCircles)

            #If the animation is done, reenable the play and back button.
            #   Disable the pause button and stop button.
            if self.counter == self.animation.getNumFrames():
                self.playButton['state'] = NORMAL
                self.backButton['state'] = NORMAL
                self.pauseButton['state'] = DISABLED
                self.stopButton['state'] = DISABLED

    """
    Draw a frames of the animation.
    """
    def drawCircles(self) -> None:
        #only draw a frame when the time has elapsed.
        if  not (not self.play.get() and self.counter !=
self.animation.getNumFrames()) \
                and self.counter < self.animation.getNumFrames() \
                and int((time.perf_counter() - self.t) * 1000) >= self.speed:
            #Record time.
            self.t = time.perf_counter()

            #Get the matrix.
            matrix = self.animation.getFrame(self.counter)
            for x in range(matrix.getxlen()):
                for z in range(matrix.getzlen()):
                    for y in range(matrix.getylen()):
                        #Create a 3D animation. Going back into the top right.
                        self.createCircle(x*100+y*10+25, 700-z*100-y*10-25, 5, fill =
"#"+str(getHex(matrix.getVal(x, matrix.getylen() - 1 - y, z))))

            #Keep track of frame.
            self.counter += 1

            #Wait the preset amount of time and clear canvas for next animation.
            self.window.after(self.speed, self.resetCanvas)

    """
```

```
    Rest the counter every time an animation is played.
    """
    def playAnimation(self) -> None:
        #Reset the animation.
        self.counter = 0

        #Disable the play and back button, but activate the pause and stop button.
        self.playButton['state'] = DISABLED
        self.backButton['state'] = DISABLED
        self.pauseButton['state'] = NORMAL
        self.stopButton['state'] = NORMAL

        #Set time to the time that would have the program start.
        self.t = ((time.perf_counter() * 1000) - self.speed)/1000

        #Show the animation.
        self.drawCircles()

    """
    Stops the animation. This enables the play and stop button. It also enables the
play
        button. If paused, it resets that as well.
    """
    def stopAnimation(self) -> None:
        #Put the counter at the end, so no currently waiting calls will run.
        self.counter = self.animation.getNumFrames()

        #If there is stuff disabled, the animation must be paused.
        # Otherwise it isn't. Prevents the play button from getting changed just by
the stop button.
        if self.playButton['state'] == DISABLED:
            self.play.set(True)

            self.playButton['state'] = NORMAL
            self.backButton['state'] = NORMAL
            self.resetCanvas()
```

## Appendix A.7: Colors.py

```
"""
Author: John Burns
Last Modified: 5/15/2021
OSU Email Address: burnsjo@oregonstate.edu
Course Number ECE 342
Project: LED Visualizer Group 2           Due Date: 5/28/2021
Description: The colors used in the visualizer project. Makes it easy to use in
multiple programs.
"""

"""
Colors that are used for naming the tabs.
"""
Colors = {'Off': 0, 'Dim Blue': 1, 'Dim Green': 2, 'Dim Cyan': 3, \
            'Dim Red': 4, 'Dim Purple': 5, 'Dim Yellow': 6, 'Dim White': 7, \
            'Blue': 9, 'Blue-Cyan': 11, 'Blue-Purple': 13, 'Baby Blue': 15, \
            'Green': 18, 'Green-Cyan': 19, 'Green-Yellow': 22, 'Green-White': 23, \
            'Cyan': 27, 'Cyan-White': 31, 'Red': 36, 'Red-Purple': 37, \
            'Orange': 38, 'Pink': 39, 'Purple': 45, 'Purple-White': 47, \
            'Yellow': 54, 'Yellow-White': 55, 'White': 63}

"""
Corresponding hex values for the above colors. Retrieved from analysis done on
possible colors.
"""
HexColors = {'000000': 0, '00007f': 1, '007f00': 2, '007f7f': 3, \
             '7f0000': 4, '7f007f': 5, '7f7f00': 6, '7f7f7f': 7, \
             '0000ff': 9, '007fff': 11, '7f00ff': 13, '7f7fff': 15, \
             '00ff00': 18, '00ff7f': 19, '7fff00': 22, '7fff7f': 23, \
             '00ffff': 27, '7fffff': 31, 'ff0000': 36, 'ff007f': 37, \
             'ff7f00': 38, 'ff7f7f': 39, 'ff00ff': 45, 'ff7fff': 47, \
             'ffff00': 54, 'ffff7f': 55, 'ffffff': 63}

"""
A method for getting the key based on the value of a dictionary.
params: val, the value to search with.
returns: key, the value being looked for.
"""
def getHex(val: int) -> str:
    hxVals = list(HexColors.keys())
    numVals = list(HexColors.values())
    indexOfVal = numVals.index(val)
    return hxVals[indexOfVal]
```

## Appendix A.8: CheckMarkList.py

```
"""
Author: John Burns
Last Modified: 5/25/2021
OSU Email Address: burnsjo@oregonstate.edu
Course Number ECE 342
Project: LED Visualizer Group 2           Due Date: 5/28/2021
"""

from tkinter import *
from Colors import *

"""
Setup the check buttons for setting colors. The panel of checks is colored and named
based on the colors the checks represent.
If no color is selected the listOfButtons will become disabled.
parameters: master, the frame the color pannel will be put on (Frame)
            listOfButtons, a list of the dependent buttons (list(Button))
            text, the label used for the color selection (string).
            var, the variable that is changed with clicks (IntVar())
            command, the command to be used (function handle)
            commandlen, the number of commands available (int)
            txtLst, a list of txt for each command (list(string))
            badVal, the off value to avoid (int)
"""
class CheckButtonBar:
   def __init__(self, master: Frame, listOfButtons: list, text: str, var: IntVar,
command=None, commandlen = 0, txtLst = [], badVal = -1):
      #Make parameters accessible within the class.
      self.var = var
      self.command = command
      self.commandlen = commandlen
      self.txtLst = txtLst

      #prepare a list.
      self.checkList = []
      self.dependentButtons = listOfButtons

      #Make a labelFrame for describing the pannel.
      self.frame = LabelFrame(master, text=text)
      self.frame.pack()

      self.badVal = badVal
```

```
      #Setup the disabling and reenabling the dependant buttons.
      self.setupChecks()

   """
   Make a list of CheckButtons that are off at -1 and on at the given numeric value.
      the variable, makes it so that the value will be accessible elsewhere without
having to check it (auto updates).
   """
   def setupChecks(self) -> None:
      #Make a list of CheckButtons and put them in the named pannel.
      #The length of the for loop is dependent on if there are commands expected.
      length = len(list(Colors.values())) if not self.command else self.commandlen

      for i in range(length):
         #Handle the case of black colors.
         fg = "white" if list(HexColors.keys())[i] == "000000" else "black"
         selectColor = "black" if list(HexColors.keys())[i] == "000000" else "white"

         #If there is a given command, then it isn't a colored list.
         if not self.command:
            #Creates a button, with the interest of allowing black to be seen.
            self.checkList.append(Checkbutton(
               self.frame,
               text = list(Colors.keys())[i],
               onvalue = list(Colors.values())[i],
               offvalue = -1,
               variable = self.var,
               command = self.activateSelection,
               activebackground='#' + str(list(HexColors.keys())[i]),
               activeforeground="black",
               fg = fg,
               selectcolor = selectColor,
               width = 11,
               bg = '#' + str(list(HexColors.keys())[i]),
               anchor = W
            ))
         else:
            #Make a generic check button list.
            self.checkList.append(Checkbutton(
               self.frame,
               text = self.txtLst[i],
               onvalue = i,
               offvalue = -1,
               variable = self.var,
```

```
                command = self.activateCommand
            ))
        #Pack the buttons as they come in.
        self.checkList[i].pack(anchor = W)

    """
    Define the functionality for selecting a box.
       If a box is enabled use the command, otherwise do nothing.
       If an unreachable badValue is given, then command is executed every time the
check is toggled.
    """
    def activateCommand(self) -> None:
       if self.var.get() != self.badVal:
          self.command()

    """
    Define the functionality for selecting a box.
       If a box is selected then the dependent buttons
       are enabled, otherwise the dependent buttons are disabled.
    """
    def activateSelection(self) -> None:
       if self.var.get() != -1:
          for i in range(len(self.dependentButtons[0])):
             self.dependentButtons[0][i]['state'] = NORMAL

       elif self.var.get() == -1:
          for i in range(len(self.dependentButtons[0])):
             self.dependentButtons[0][i]['state'] = DISABLED
```

## Appendix A.9: ArduinoInterface.py

```
# File:       ArduinoInterface.py
# Author:     Henry Gillespie
# Date:       May 25, 2021
# Purpose:    This program defines functions that communicate with the Arduino

import os
from Animation import Animation
from tkinter import messagebox
import pythonCOMport as com

def sendValToArd(val:int, port:str) -> None:
    # This function sends the input value to Serial Port COM3
    portNum = com.getPort(port, False)
    if len(portNum) > 0:
        print(f'Animation {val+1} was sent.')
        setMode = f'mode {portNum} BAUD=9600 PARITY=n DATA=8'
        os.system(r"{}".format(setMode))
        # os.system(r'mode COM3 BAUD=9600 PARITY=n DATA=8')

        # The following lines make a raw string with a variable in it. 'string' is
parsed as an f-string with special characters and a variable
        # Then it is converted to a raw string with the variable in place

        string = f'set /p x="{val}" <nul >\\\\.\{portNum}' # in f-strings, '\' is a
special character, so '\\' becomes '\'
        os.system(r"{}".format(string)) # raw strings ignore otherwise special
characters

        #Show a window that tells them what they can expect.
        messagebox.showinfo("Animation Sent", f"Animation #{val+1} was
selected.\nEnjoy.")
    else:
        print(f'Animation {val+1} could not be sent.')

def animationsForHeader(animations: list[Animation], port:str) -> None:
    # This function writes a header file and then uploads the arduino code, which
calls the header file

    portNum = com.getPort(port, False)
    if len(portNum) > 0:
        for i in range(len(animations)):
            animation = animations[i]
            frameCnt = animation.getNumFrames()
```

```
            speed = animation.getSpeed()

            print(f'Animation {i+1} has {frameCnt} frame{"s" if frameCnt > 1 else ""}
and will be played at {speed} frame{"s" if speed > 1 else ""} per second.\n')

            print('Animation sent:')
            print(animation)

            if i < len(animations) - 1:
                print(":---------------:\n")

        writeFile(animations)

        uploadStr = f'arduino --upload --port {portNum}
JuniorDesign_PreProgrammedAnimations\JuniorDesign_PreProgrammedAnimations.ino'
        # os.system(r'arduino --upload
JuniorDesign_PreProgrammedAnimations\JuniorDesign_PreProgrammedAnimations.ino')
        os.system(r"{}".format(uploadStr))

        messagebox.showinfo("Animations uploaded", "Animations 4, 5, and 6 have been
uploaded.\nTo play, go to the main menu and select an animation.")
    else:
        print('Animations could not be updated.')

def writeFile(animations: list[Animation]) -> None:
    headerFile = open("JuniorDesign_PreProgrammedAnimations\headerFile.h", 'w')
    headerFile.write("#ifndef headerFile\n#define headerFile\n\n")
    # in f-strings, '{' is a special character to open variables, so '{{' will become
'{' in the header file
    headerFile.write(f"const PROGMEM uint16_t customLengths[3] =
{{{animations[0].getNumFrames()}, {animations[1].getNumFrames()},
{animations[2].getNumFrames()}}};\n")
    headerFile.write(f"const PROGMEM unsigned long customSpeeds[3] =
{{{framesToMicros(animations[0].getSpeed())},
{framesToMicros(animations[1].getSpeed())},
{framesToMicros(animations[2].getSpeed())}}};\n\n")
    headerFile.write("const PROGMEM uint16_t colorArr[3][2][30][35] = {")


    for i in range(len(animations)): # for each animation
            headerFile.write("\n{") # open animation array
            for color in range(2): # for each of the two color values to be passed
                headerFile.write("{") # open color array
                for frame in range(30): # for each frame
```

```
                    headerFile.write("{") # open frame array
                    for LED_row in range(35):
                        if frame < animations[i].getNumFrames(): # if this frame
exists

headerFile.write(str(int(animations[i].getFrame(frame).getRowValue(LED_row,color))))
                        else:
                            headerFile.write("0") # if frame > number of frames, fill
it with 0
                        if LED_row != 34:
                            headerFile.write(', ') # do not use a ',' after the last
value in the array
                    # close frame arrays
                    if frame != 29:
                        headerFile.write('}, ')
                    else:
                        headerFile.write('}')
                # close color arrays
                if color != 1:
                    headerFile.write('},\n ')
                else:
                    headerFile.write('}')
            # close animation arrays
            if i != len(animations)-1:
                headerFile.write('}, ')
            else:
                headerFile.write('}')

    headerFile.write("};\n\n#endif") # close complete array
    headerFile.close()

def framesToMicros(fps: int) -> int:
    # This function converts a frames per second value to microseconds per frame
    return round(1000000 / fps)
```

## Appendix A.10: Animation.py

```
"""
Author: John Burns
Last Modified: 5/27/2021
OSU Email Address: burnsjo@oregonstate.edu
Course Number ECE 342
Project: LED Visualizer Group 2           Due Date: Due Date: 5/28/2021
"""

from Matrix import LEDMatrix
import random
from Colors import *

"""
The definition for how an Animation can be defined.
    Animations are lists of LEDMatrices with an adjustable frame count and speed.
    You can add animation filler functions at the end and then add them to the
special Animations list.
3 constructors-
    Animation(x, y, z, lim, frameCnt, speed)
        x, number of columns in an LEDMatrix (int).
        y, number of rows in an LEDMatrix (int).
        z, number of layers in an LEDMatrix (int).
        lim, maximum allowable value (int).
        frameCnt, number of frames (int).
        speed, speed of the animation in fps (int).
    Animation(frameCnt, speed)
        Default Matrix() constructor (5x5x7) with a lim of 63.
        frameCnt, number of frames (int).
        speed, speed of the animation in fps (int).
    Animation()
        Default Matrix() constructor (5x5x7) with a lim of 63.
        Default frameCnt of 30.
        Default speed of 20.
"""
class Animation:
    def __init__(self, *args):
        #Full parameter list.
        if len(args) == 6:
            x, y, z, lim, frameCnt, speed = args
            self.animation = [LEDMatrix(x, y, z, lim) for frame in range(frameCnt)]
            self.frameCnt = frameCnt
            self.viewable = self.frameCnt
            self.speed = speed
```

```
        #Default matrix with custom speed and frame count.
        elif len(args) == 2:
            frameCnt, speed = args
            self.animation = [LEDMatrix() for frame in range(frameCnt)]
            self.frameCnt = frameCnt
            self.viewable = self.frameCnt
            self.speed = speed
        #Default animation.
        elif len(args) == 0:
            self.animation = [LEDMatrix() for frame in range(30)]
            self.frameCnt = 30
            self.viewable = self.frameCnt
            self.speed = 30

        #Easy way for coder to add more custom animations.
        self.specialAnimations = []
        self.specialAnimations.append(["Sequential", self.seq])
        self.specialAnimations.append(["Random", self.rand])
        self.specialAnimations.append(["Set Color", self.solid])
        self.specialAnimations.append(["Clear", self.clear])

    """
    Method for getting number of frames.
    returns: viewable, the effective number of available frames an animation has
(int).
    """
    def getNumFrames(self) -> int:
        return self.viewable

    """
    Method for changing the speed of an animation.
    parameters: speed, the speed to run the animation at (in fps, int).
    """
    def setSpeed(self, speed: int) -> None:
        self.speed = speed

    """
    Method for getting the Animation's speed.
    returns: speed, the Animations current speed (int).
    """
    def getSpeed(self) -> int:
        return self.speed

    """
```

```
    Method for adjusting the number of frames.
    parameters: frame, the matrix to edit (int).
                matrix, the matrix to replace the current matrix with (Matrix).
    """
    def setFrame(self, frame: int, matrix: LEDMatrix) -> None:
        self.animation[frame] = matrix

    """
    Method for getting a frame from the Animation.
    parameters: frame, the frame of interest (int).
    returns: matrix, the Matrix at the given frame (Matrix).
    """
    def getFrame(self, frame: int) -> LEDMatrix:
        return self.animation[frame]

    """
    Method for adjusting the number of frames in an Animation.
    parameters: newFrameCnt, the desired frame count for the Animation (int).
    """
    def changeFrameCnt(self, newFrameCnt: int) -> None:
        self.viewable = newFrameCnt

    """
    Method for clearing the animation.
        set effected part's LEDs to off.
    parameters: args -> frame, layer, val (layer effect, val doesn't matter)
                        frame, val (frame effect, val doesn't matter)
                        val (whole animation, val doesn't matter)
    """
    def clear(self, *args) -> None:
        args = list(args)
        args[-1] = 0
        self.fill(args)

    """
    Making sequential colors on animation.
        set effected part to colors in a sequential order.
    parameters: args -> frame, layer, val (layer effect, val doesn't matter)
                        frame, val (frame effect, val doesn't matter)
                        val (whole animation, val doesn't matter)
    """
    def seq(self, *args) -> None:
        args = list(args)
```

```
        #give special arguement value, unreachable otherwise.
        args[-1] = -3

        self.fill(args)

    """
    Customized method for adding random colors.
        set effected part to random color values.
    parameters: args -> frame, layer, val (layer effect, val doesn't matter)
                        frame, val (frame effect, val doesn't matter)
                        val (whole animation, val doesn't matter)
    """
    def rand(self, *args) -> None:
        args = list(args)

        #give special arguement value, unreachable otherwise.
        args[-1] = -2

        self.fill(args)

    """
    Custom method for adding solid color.
        set effected part to the decided color.
    parameters: args -> frame, layer, color (layer effect)
                        frame, color (frame effect)
                        color (whole animation)
    """
    def solid(self, *args) -> None:
        self.fill(list(args))

    """
    Modular function for setting
    parameters: args -> frame, layer, val (layer effect)
                        frame, val (frame effect)
                        val (whole animation)
        val may be of the color dictionary or a specific value that has meaning
            -2: random
            -3: sequential
    """
    def fill(self, args:list) -> None:
        #Choose frame length based on the parameter list.
        frameLen = self.getNumFrames()
        if len(args) > 1:
            frameLen = args[0]
```

```
        counter = 0

        lower = frameLen if len(args) > 1 else 0
        upper = frameLen+1 if len(args) > 1 else frameLen
        for frame in range(lower, upper):
            matrix = self.getFrame(frame)

            #Choose layer count based on the parameter list.
            layerCnt = matrix.getzlen()
            if len(args) > 2:
                layerCnt = args[1]

            lower = layerCnt if len(args) > 2 else 0
            upper = layerCnt+1 if len(args) > 2 else layerCnt
            for layer in range(lower, upper):
                for y in range(matrix.getylen()):
                    for x in range(matrix.getxlen()):
                        #Choose method of filling based on specialized parameter
values.
                        #-1 means no color has been selected (bad state).
                        if args[-1] == -2:
                            matrix.setVal(x, y, layer,
random.choice(list(HexColors.values())))

                        elif args[-1] == -3:
                            matrix.setVal(x, y, layer,
list(HexColors.values())[counter])

                            #Reset the counter when it hits max.
                            if counter < len(list(HexColors.values())) - 1:
                                counter += 1
                            else:
                                counter = 0

                        elif args[-1] >= 0:
                            matrix.setVal(x, y, layer, args[-1])

    """
    Intelligent stringification of the class.
        Shows frame by frame of the Matrices which are layer by layer.
    returns: outStr, the out come of the work (string).
    """
    def toString(self) -> str:
```

```
        outStr = ""

        for frame in range(self.viewable):
            outStr += self.animation[frame].toString()
            if frame < self.viewable - 1:
                outStr += ''.join(['-' for i  in range(4 *
(self.animation[frame].getxlen()-1) + 1)])
                outStr += "\n"

        return outStr

    """
    Overwrite the default string for the class, by showing good information.
    """
    def __str__(self) -> str:
        return self.toString()
```

## Appendix B.1: JuniorDesign_PreProgrammedAnimations.ino

```
#include <SoftwareSerial.h>
#include "headerFile.h"
SoftwareSerial btConnection(5, 6); // RX, TX. Arduino digital PWM pins are 3, 5, 6, 9, 10, 11

int SER = 11;      // serial data to rows
int RCLK = 12;     // storage register clock
int SRCLK = 13;    // shift register clock for serial input
int GNDRCLK = 8;
int GNDSER = 9;
int GNDSRCLK = 10;

unsigned long t = 0;

int arr[15];

byte val_lsb;
byte val_msb;
uint16_t value;
unsigned long t1;
int count;

int btReceived = 0;
uint16_t animationLength;
unsigned long animationSpeed;

void setup() {

  /* Set pins to correct setting */
  pinMode(SER, OUTPUT);
  pinMode(RCLK, OUTPUT);
  pinMode(SRCLK, OUTPUT);
  pinMode(GNDSER, OUTPUT);
  pinMode(GNDRCLK, OUTPUT);
  pinMode(GNDSRCLK, OUTPUT);

  Serial.begin(9600);
  btConnection.begin(9600);
}


void loop() {
```

```
  /* Ground the LED for the first cycle */
  count = 0;
  do {
    if (btConnection.available() > 0) {
      btReceived = btConnection.read() - 48; // 48 is character '0'
      count = 0;
    }
    if (Serial.available() > 0) {
      btReceived = Serial.read() - 48;
      count = 0;
    }
    if (count == 0 && (btReceived < 0 || btReceived > 5)) {
      btConnection.println("\nYou entered an invalid animation number");
      count++;
    }
  } while (btReceived < 0 || btReceived > 5);
  switch (btReceived) {
    case 0:
      colorCycle();
      break;

    case 1:
      changeLayers();
      break;

    case 2:
      swirl();
      break;

    case 3:
    case 4:
    case 5:
      customAnimation(btReceived - 3);
      break;
  }
}

void customAnimation(int btValue) {
  animationLength = pgm_read_word(&customLengths[btValue]);
  animationSpeed = pgm_read_dword(&customSpeeds[btValue]);

  for (int i = 0; i < animationLength; i++) { // for each animation frame

    t1 = micros();
```

```
    while ((micros() - t1) < animationSpeed) { // slow the animation down by
repeating the same frame
      resetGND();
      for (int j = 0; j < 35; j++) { // for each row


        /*
            Running nextRow() right before the next pushArr function ensures
            that the 2nd color from the previous row will be on during the
            time-consuming Serial.read() functions, meaning that the two
            colors will be on for equal time intervals. However, it should
            only run this after the first iteration, or else the 1st row will
            only be grounded for one of its colors.
        */


        pushBlankArr();
        if (j != 0)
          nextRow();

        value = pgm_read_word(&colorArr[btValue][0][i][j]);

        num2arr(arr, value);
        pushArr(arr);

        /* Read and combine the bytes, then display them */

        value = pgm_read_word(&colorArr[btValue][1][i][j]);
        num2arr(arr, value);
        pushArr(arr);
      }
      pushBlankArr();
    }
  }
}

void colorCycle() {
  uint16_t colors[6] = {18724, 28086, 9362, 14043, 4681, 23405}; // rainbow colors
  for (int frame = 0; frame < 6; frame++) {
    value = colors[frame];
    num2arr(arr, value);
    for (int repeat = 0; repeat < 500; repeat++) {
      //t1 = micros();
      resetGND();
```

```
      for (int row = 0; row < 35; row++) {
        if (row != 0)
          nextRow();
        pushArr(arr);
      }
      nextRow();
    }
  }
}

void changeLayers() {
  uint16_t white = 32767;
  for (int frame = 0; frame < 7; frame++) { // for each layer
    for (int repeat = 0; repeat < 100; repeat++) {
      //t1 = micros();
      resetGND();
      for (int row = 0; row < 35; row++) {
        pushBlankArr();
        if (row != 0)
          nextRow();

        if (row < (frame + 1) * 5 && row >= frame * 5) {
          num2arr(arr, white);
          pushArr2(arr);
        }
        else {
          value = 0;
          num2arr(arr, value);
          pushArr2(arr);
        }
      }
      pushBlankArr();
    }
  }
}

void swirl() {
  for (int frame = 0; frame < 16; frame++) {
    for (int repeat = 0; repeat < 50; repeat++) {
      t1 = micros();
      value = 0;
      num2arr(arr, value);
      pushArr(arr);
      resetGND();
```

```
      for (int layer = 0; layer < 7; layer++) {
        for (int row = 0; row < 5; row++) {
          if (frame < 5 && row == frame) { // front part
            value = 7;
          }
          else if (frame < 9 && frame > 4 && row == 4) { // one side
            value = 7 * (1 << (3 * (frame - 4)));
          }
          else if (frame < 13 && frame > 8 && row == 12 - frame) { // back part
            value = 28672;
          }
          else if (frame > 12 && row == 0) {
            value = 7 * (1 << (3 * (16 - frame))); // second side
          }
          else {
            value = 0;
          }

          if (row != 0 || layer != 0) {
            pushBlankArr();
            nextRow();
          }
          num2arr(arr, value);

          while (micros() - t1 < 150) {}

          pushArr(arr);
        }
      }
    }
  }
}

void resetGND() {
  /*
      This function sets all the first ground pin LOW and the rest
      to HIGH, preparing the cube for another frame.
  */
  digitalWrite(GNDRCLK, LOW);
  digitalWrite(GNDSER, HIGH); // HIGH values
  for (int i = 0; i < 34; i++) { // Cycle ground serial clock 34 times
    digitalWrite(GNDSRCLK, LOW);
    delayMicroseconds(1);
    digitalWrite(GNDSRCLK, HIGH);
```

```
    delayMicroseconds(1);
  }
  /* Push one LOW value */
  digitalWrite(GNDSER, LOW);
  digitalWrite(GNDSRCLK, LOW); // Cycle ground serial clock once
  delayMicroseconds(1);
  digitalWrite(GNDSRCLK, HIGH);
  delayMicroseconds(1);

  digitalWrite(GNDRCLK, HIGH); // Push output values through storage registers
}



void nextRow() {
  /* This function cycles the ground clock, moving the program to the next row */
  digitalWrite(GNDSRCLK, LOW); // shift register clock low
  digitalWrite(GNDSER, HIGH);
  delayMicroseconds(1);
  digitalWrite(GNDRCLK, LOW); // storage register clock low
  digitalWrite(GNDSRCLK, HIGH); // cycle shift registers
  delayMicroseconds(1);
  digitalWrite(GNDRCLK, HIGH); // update output through storage registers
}

void nextRow_bitManipulation() {
  PORTB = B00111011;
  delayMicroseconds(1);
  PORTB = B00111110;
  delayMicroseconds(1);
  PORTB = B00111111;
}



void pushArr(int arr[15]) {
  /*
      This function pushes all 15 bits from the input array into the shift
      registers and then cycles the register clock, updating the color pins
      on the cube. Functionally, it is the same as pushArr1() below, which is easier
      to read.
  */
  bitClear(PORTB, 4);
  t = micros();

  /* Feed the array into the shift registers 'backwards' and cycle the clock */
```

```
  for (int i = 14; i >= 0; i--) {

    PORTB = B00000111;
    if (arr[i] == 0) {
      PORTB = B00000111;
      delayMicroseconds(1);
      PORTB = B00100111;
    }
    else {
      PORTB = B00001111;
      delayMicroseconds(1);
      PORTB = B00101111;
    }
    delayMicroseconds(1);
  }
  bitSet(PORTB, 4);
}

void pushArr1(int arr[15]) {
  /*
      This function pushes all 15 bits from the input array into the shift
      registers and then cycles the register clock, updating the color pins
      on the cube.
  */
  digitalWrite(RCLK, LOW);
  t = micros();

  /* Feed the array into the shift registers 'backwards' and cycle the clock */
  for (int i = 14; i >= 0; i--) {
    digitalWrite(SRCLK, LOW);
    if (arr[i] == 0)
      digitalWrite(SER, LOW);
    else
      digitalWrite(SER, HIGH);
    delayMicroseconds(1);
    digitalWrite(SRCLK, HIGH);
    delayMicroseconds(1);
  }
  digitalWrite(RCLK, HIGH);
}

void pushArr2(int arr[15]) {
  /*
      This function pushes all 15 bits from the input array into the shift
```

```
      registers and then cycles the register clock, updating the color pins
      on the cube.
  */
  bitClear(PORTB, 4);
  t = micros();

  /* Feed the array into the shift registers 'backwards' and cycle the clock */
  for (int i = 14; i >= 0; i--) {

    PORTB = B00000110;
    if (arr[i] == 0) {
      PORTB = B00000110;
      delayMicroseconds(1);
      PORTB = B00100110;
      delayMicroseconds(1);
      PORTB = B00110111;
    }
    else {
      PORTB = B00001110;
      delayMicroseconds(1);
      PORTB = B00101110;
      delayMicroseconds(1);
      PORTB = B00111111;
    }
    delayMicroseconds(1);
  }
}

void pushBlankArr(){
  /*
  This function simply pushes a blank array to the color pins. This
  is needed to prevent colors from 'bleeding' into adjacent rows
  because when the ground is switched, any remaining color will show
  up before the next color is sent.
  */
  num2arr(arr, 0);
  pushArr(arr);
}

void num2arr(int arr1[15], uint16_t value) {
  /*
     This function converts the value into an array of length 15,
     in which every value is a 0 or 1. It does so by converting
     the value to binary
```

```
  */
  for (int i = 0; i < 15; i++) {
    //arr1[i] = value % 2;
/*
      The two following lines implement a faster modulus operation, which works for
      factors of 2
*/
    arr1[i] = value - ((value >> 1) << 1);
    value = value >> 1;
  }
}
```