Nanolux LED Strip Simulator v3

Technical Documentation

Version 1.0

Last updated 7-30-2024

Overview

The purpose of the Nanolux LED Strip Simulator (henceforth referred to as "simulator") is to provide an easy way for new developers to view device output without having to own a compatible LED strip.

The application is written in Python using the open-source CustomTkinter library for the GUI. It uses the following external dependencies:

```
pip install customtkinter
pip install PIL
pip install numpy
pip install opencv-python
```

The previous simulator revision utilized PySimpleGUI as a frontend, which became a paid product after development of the application was finished. This version recycles much of the code used then, but is better structured and commented.

Running the Simulator

1. Enable simulator output on the Nanolux web application.
2. Launch the application's .EXE or .py executable.
3. Select the COM port of the device.
4. Hit "Connect."

If disconnecting,

1. Hit the "Disconnect" button.
2. Close the application or select a new COM port.

Class Structure

The v3 Simulator uses a basic class structure. There is one main class, called App, which runs the frontend functions and backend thread. It calls a second class, named NanoluxSerial, which manages all communication with the device. Additionally, it also creates the images displayed on the UI for the virtual LED strips.

App Class

The App class is contained inside the file main.py. The file manages all setup for the App object in the following function, at the end of the file.

```
if __name__ == "__main__":
    app = App()
    app.stop()
```

When App() is instantiated, the program does not continue inside of this If statement until the UI has been closed. When this happens, app.stop() stops the backend thread through state control.

main.py also contains an enumerator for managing state:

```
class State(Enum):
    DISCONNECTED = 0
    CONNECTING = 1
    CONNECTED = 2
    EXITING = 3
```

This is utilized by the App class to enable switching between different functions (aka case logic).

Inside of App's __init()__:

```
self.sm = [
    self.__disconnected_state,
    self.__connecting_state,
    self.__connected_state,
]
```

Then, inside of the following function:

```
# Function to run the state machine and all state functions.
def __state_machine(self):
    if self.state.value > len(State):
        self.__null_state()
    else:
        self.sm[self.state.value]()
```

Essentially, based on the current state stored in the class variable self.state, the state machine automatically runs the function stored at that state's position in the state array. Note that the state array does not contain an explicit state for EXITING: it is covered under the "null state." This is to enable better scaleability for future states should the application need them.

The images used to represent the LED strip are loaded in during the connected state, and stored until the UI update task can push them to the UI:

```
def __connected_state(self):
    if self.ser.read(self.slider_value):
        self.hsv_image = self.ser.get_hsv()
        self.bgr_image = self.ser.get_bgr()
        self.rgb_image = self.ser.get_rgb()
    else:
        self.state = State.DISCONNECTED
        self.__start_scan()
```

The App class also has a variety of "flags" that can be raised by the UI:

```
# Flags
```

```
self.connect_trigger = False
self.stop_trigger = False
self.mode_trigger = False
self.is_dark_mode = True
```

This makes communication between the frontend and the backend easier, and allows for a design pattern where the backend cannot directly control the frontend. This design pattern was chosen to prevent flickering caused by updating CustomTkinter objects outside of scheduled function calls.

For example, the theme of the application is controlled by the "is_dark_mode" flag. This particular flag is observed inside of the ui_update() function:

```
# If the mode flag is raised, switch to the other visual mode (light/dark)
if self.is_dark_mode and self.mode_trigger:
    ct.set_appearance_mode("light")
    ct.set_default_color_theme("blue")
    self.toggle_mode_button.configure(text="Toggle Dark Mode")
elif not self.is_dark_mode and self.mode_trigger:
    ct.set_appearance_mode("dark")
    ct.set_default_color_theme("dark-blue")
    self.toggle_mode_button.configure(text="Toggle Light Mode")

# Reset the mode flag and change the current mode boolean
if self.mode_trigger:
    self.is_dark_mode = not self.is_dark_mode
    self.mode_trigger = False
```

The flag is raised using a simple function that sets the flag to "true" and is called via the button:

```
self.toggle_mode_button = ct.CTkButton(master=self.options_frame, text="Toggle Light Mode", command=self.__raise_mode_trigger)
```

NanoluxSerial Class

The NanoluxSerial class is responsible for a few things:

- Connecting to the Audiolux's serial port
- Reading incoming serial data
- Processing that data into a CustomTkinter image

The class uses the serial library to do the first two tasks, and the image generation is done via a few steps:

1. Parse the serial data into a list.
2. Fill a Numpy array with the colors from the generated list.
3. Convert the Numpy array into an image using PIL.
4. Create and return a CTkImage object with the appropriate size using the PIL image.

Essentially, the process converts raw data to a list to an array to an image to a GUI element. These processes are performed by the following lines of code:

```
# Step 1
self.serial_data = self.serial.readline().decode("utf-8").split()

# Step 2
self.__fill_bgr_array(scaling, w)

# Step 3
self.bgr = cv2.resize(self.bgr, (850, 100))

# Step 4
temp = PIL.Image.fromarray(self.bgr)
return ct.CTkImage(light_image=temp, dark_image=temp, size=(850,100))
```

Step 4 is done exclusively during return functions (get_hsv(), get_bgr(), get_rgb()).

The HSV and RGB images are generated using the existing BGR data sent over serial. Effectively, Step 3 is done for the BGR image, and then an additional line converts it:

```
self.hsv, self.rgb = bgr_to_hsv_rgb(self.bgr)
```