Adapting the Raspberry Pi LED Animation System for WS2811 Addressable LEDs

Overview of the Existing System and New Requirements

The current system uses a Raspberry Pi 3 B+ running a Flask web GUI and a Python-based animation program to drive a 64×64 RGB LED matrix (HUB75 panel) via the rpi-rgb-led-matrix library. The user can upload or select animations (GIFs, PNG image sequences, etc.) which are then displayed on the HUB75 panel. Now, we want to **add support for an alternate display**: a lightbox with approximately 100 WS2811 addressable RGB LEDs arranged in a 2D grid (with serpentine wiring). In this new mode, the same animations should play on the WS2811 LED grid instead of (or interchangeable with) the HUB75 panel, with minimal changes to the code and interface.

Key challenges and differences:

- **LED Technology**: The HUB75 matrix is a parallel-drive LED panel (directly controlled rows/columns with high refresh rate), whereas WS2811 LEDs (aka NeoPixels) are individually addressable LEDs daisy-chained and controlled by a single-wire serial data protocol. This means the method of driving the LEDs is fundamentally different. The HUB75 uses many GPIO pins in parallel with a specialized library, while WS2811 requires precise timing on one data line.

- **Resolution and Mapping**: The HUB75 panel is 64×64 = 4096 LEDs; the new WS2811 grid is ~100 LEDs (e.g. 10×10 or similar). Animations designed for 64×64 will need to be scaled or adapted to a much lower resolution. Also, the WS2811 LEDs are wired in a serpentine (zig-zag) pattern, so the mapping from 2D coordinates to the 1D LED index is non-linear (alternate rows reversed). We must handle this mapping in software.

- **Unified Control**: We want to reuse the same Flask web interface and Python codebase to control both output types. Ideally, the user can select the output mode (HUB75 matrix or WS2811 lightbox) and the system will route the animation frames to the appropriate hardware. This calls for a flexible software architecture where the rendering logic is shared, and only the output "driver" differs.

- **Hardware Constraints**: WS2811 LEDs require precise timing (800 kHz signal with sub-microsecond bit widths) and a stable 5V power supply. The Raspberry Pi is not a real-time microcontroller, so driving these LEDs directly from the Pi can be tricky without a special library. Fortunately, libraries exist to handle the timing via DMA (Direct Memory Access) on the Pi. Alternatively, an external microcontroller (like an ESP32) can be used to offload LED driving. We will explore both approaches.

In the sections below, we detail how to adapt the system for WS2811 output, including hardware options (direct Pi control vs. using a microcontroller), necessary changes to the software, mapping of animations to the new grid, integration with the web GUI, and practical considerations for wiring, power, and performance.

Hardware Options: Raspberry Pi Direct Control vs. Microcontroller Offloading

One of the first decisions is how to drive the WS2811 LED string. We have two primary options:

1. **Use the Raspberry Pi's GPIO directly to drive the WS2811 LEDs**, using a library that handles the precise timing (such as the rpi_ws281x library or Adafruit's NeoPixel library).

2. **Use a dedicated microcontroller (e.g. ESP32, ESP8266, Arduino)** to drive the LEDs, and have the Raspberry Pi communicate with this controller (over USB, serial, or Wi-Fi) to update the LED frames.

Each approach has trade-offs in terms of complexity, performance, and integration:

- Raspberry Pi Direct Control: It is entirely possible to drive WS2811/WS2812 LEDs directly from the Pi's GPIO using a proper library. The widely used solution is to employ PWM hardware along with DMA on the Pi to generate the WS2811 signal without CPU intervention. For example, the rpi_ws281x C library (and its Python wrapper) uses DMA to feed the PWM hardware with the exact bitstream needed for the LEDs, so the Linux OS timing jitter is not an issue. Adafruit's CircuitPython NeoPixel library builds on this, allowing Python to control NeoPixels on the Pi by leveraging the underlying rpi_ws281x driver. In short, with the right setup, the Pi can reliably drive the ~100 WS2811 LEDs at a good frame rate, despite not being a real-time system. Many projects have done this successfully (even with hundreds or thousands of LEDs, using multiple output channels) by using the Pi's hardware peripherals (PWM, PCM, or SPI) in combination with DMA.

    - *Advantages*: No additional microcontroller hardware needed – the existing Pi handles everything. The integration is simpler since the Python code can directly set LED values just as it did for the matrix (just using a different library/API). The Flask server and animation code remain on the Pi, maintaining a unified system. Also, the Pi has plenty of processing power to handle image processing (down-scaling animations, etc.) before pushing pixel data out.

    - *Considerations*: The Raspberry Pi's GPIO is 3.3V, while WS2811 data expects ~5V logic. Often a level-shifter is needed to reliably reach logic HIGH threshold (see Wiring section below). The Pi's PWM method by default uses a specific GPIO (typically GPIO18 for PWM0) and it may conflict with audio output (PWM0/1 are used for analog audio). If analog audio is not needed, this isn't a problem, or one can use the SPI method (which outputs on MOSI pin) to avoid PWM conflict . Running the LED driver may require root privileges (accessing /dev/mem for PWM/DMA), though some libraries and the SPI method can work as non-root. In terms of reliability, while the DMA-driven signal is very precise, heavy CPU loads on the Pi *could* in theory interfere if not using DMA – but with DMA the LED updates are actually quite stable. As a caution, one expert notes that the Pi "isn't realtime" and driving NeoPixels on it can be less reliable than using a microcontroller, which is why many people offload to an ESP32. In practice, the DMA solution mitigates most issues, and 100 LEDs is a light load. Still, if the Pi will be doing a lot of multitasking, it's worth keeping this in mind. We can also mitigate risk by updating the LEDs in a single DMA burst per frame (which the library handles) and not doing other heavy I/O simultaneously.

- External Microcontroller (ESP32, etc.): Another approach is to use a dedicated microcontroller to handle all WS2811 LED driving. The Raspberry Pi would then send the animation data to this controller, which outputs it to the LEDs. Popular choices are ESP8266/ESP32 (which have built-in Wi-Fi and can easily interface with the Pi over network) or an Arduino-compatible board (which could interface via USB serial or other means). The microcontroller would run firmware specifically for driving addressable LEDs. For example, one could use FastLED or Adafruit NeoPixel library on an Arduino/ESP to set up a 10×10 LED matrix. The communication from Pi to microcontroller can happen in a few ways:

    - *Wi-Fi networking*: If using an ESP32 running a firmware like WLED, the Pi can send commands or stream frames over Wi-Fi (or Ethernet) to the ESP. WLED is an open-source firmware that turns ESP8266/ESP32 into a flexible LED controller with a web API. It's "a fast and feature-rich implementation of an ESP32/8266 webserver to control NeoPixel (WS2812B/WS2811…) LEDs". WLED supports multiple control interfaces – for integration, notably HTTP/JSON requests, MQTT, and standardized lighting protocols like E1.31 (sACN), Art-Net, and a raw UDP streaming mode. This means our Pi software could, for example, open a socket and stream raw RGB frame data to the ESP32 running WLED in realtime (WLED's "UDP realtime" mode) or send each frame as an E1.31 DMX packet which WLED will display. The ESP, in turn, drives the LEDs with perfect timing. If not using WLED, one could write custom firmware for the ESP32 to listen for UDP or serial data and update the LEDs using FastLED – essentially building a simple protocol.

    - *Serial/USB connection*: For an Arduino or even an ESP32 connected via USB, the Pi could send frames over a serial link. A simple protocol might send a start byte and then the raw color bytes for all LEDs each frame. The microcontroller (running a custom sketch) reads this and updates the LED strip. This is straightforward, though the UART bandwidth limits how fast you can stream frames (at 115200 baud, ~30 fps for 100 LEDs is possible; using higher baud rates or a binary compact protocol can improve this). USB CDC (virtual COM port) on an ESP32 or a Pi Pico (RP2040) could handle higher rates reliably. There's also the possibility of using SPI or I2C to an MCU for higher throughput, but that adds complexity.

    - *Advantages*: The microcontroller approach offloads all real-time signal generation from the Pi. The micro will handle the WS2811 timing, which it can do easily (ESP32 for instance has an RMT peripheral specifically for driving LEDs, and AVR microcontrollers can use cycle-accurate code). This means the Pi's Linux scheduler can never glitch the LED timing – the LEDs will update very consistently. It also frees the Pi to focus on the web server and image processing. If the installation ever grows (say to many more LEDs or multiple strips), an ESP32 can handle thousands of LEDs and multiple outputs, reducing the Pi's load. In addition, WLED or similar firmware provides a lot of out-of-the-box functionality (lighting effects, brightness control, OTA updates) that

could be leveraged.

- *Trade-offs*: Introducing a secondary controller complicates the system architecture. You now have two devices to configure and maintain (Pi and microcontroller). Communication and synchronization become considerations – e.g. ensuring the frame data arrives in time and possibly acknowledging it. There will be a slight latency (likely a few milliseconds) in sending data over Wi-Fi or serial, but for moderate frame rates (30-60 FPS) and 100 pixels this is negligible. The Flask GUI on the Pi would need to know how to send data to the microcontroller. If using WLED, the Pi could send HTTP requests (simple but overhead per frame is large, not ideal for animation) or use UDP streaming (better for continuous animation). Using a microcontroller also means any *interactive* features (like if the GUI wants to get the current LED colors or status) require a back-channel from the micro. However, WLED provides APIs for status as well. In terms of coding, using a micro may require writing some code on that micro (unless using stock WLED). If we aim for minimal rewriting, the direct-Pi method is actually simpler, since we can reuse the Python code and just swap out the LED driver library. Using a micro might require writing a new module to interface with that micro (e.g. a Python class to send UDP or serial data). It's still quite feasible, and abstracting this in the code can hide the differences from the rest of the system.

- *Reliability*: The ESP32 approach is very robust for LED driving. Many users choose it because "the Raspberry Pi isn't real-time, so Neopixels on a Pi aren't super reliable… most people use an ESP32 or similar". In our case, with proper libraries the Pi can be reliable too, but if the Pi is expected to multitask or if absolute stability is required, the microcontroller is a safe bet. Additionally, if the WS2811 lightbox is physically separate or remote from the Pi, a wireless ESP32 could be placed with the lightbox and controlled over network, which adds placement flexibility (whereas running long wires from the Pi's GPIO to the LEDs is not ideal beyond short distances).

Recommendation: For the simplest integration with the existing system, using the Raspberry Pi directly with a WS2811 library is the quickest path. It requires only a software update (installing a NeoPixel library and writing a mapping function) and some wiring adjustments (connecting the LED strip to the Pi with a level shifter and external power). This keeps all animation logic in one place. However, we will design the software to be flexible so that switching to an external controller later is possible without a major rewrite. Essentially, we can abstract the LED output so that whether it's a local GPIO or a remote ESP32, the rest of the code doesn't care. In the next section, we'll outline the software changes and how to achieve this abstraction.

(Figure below illustrates the two hardware approaches: the Pi can drive the WS2811 LEDs directly via a GPIO pin using the rpi_ws281x library, or the Pi can send frame data to an ESP32 controller (running something like WLED or custom FastLED code) which then drives the LEDs). The original HUB75 panel output (driven by many GPIOs via the rpi-rgb-led-matrix library) remains an option as well.

Architecture options for integrating a WS2811 LED grid. The Raspberry Pi can either drive the LEDs directly using its GPIO (via PWM/SPI and a level-shifter), or delegate to an external microcontroller like an ESP32 (communicating over Wi-Fi or serial). In both cases, the goal is to reuse the same Python animation code and provide a unified web interface.

Software Adaptation: Reusing the Python Codebase and Flask GUI

To support the new WS2811 lightbox output while preserving the existing functionality, the software should be refactored to handle multiple output targets. Key steps include abstracting the LED panel control into a pluggable module, handling differences in resolution and mapping, and providing a way for the user (via the GUI) to choose the active output.

1. Output Abstraction (Driver Classes): We can introduce an interface (or simply a conditional in code) for the two output modes. For example, we might have a base class or set of functions like LEDOutputDevice with methods such as clear(), draw_frame(frame_data), set_brightness(level), etc. Then implement two versions: HUB75OutputDevice and WS2811OutputDevice. The Flask routes or animation loop can decide which output to use based on a configuration setting (perhaps stored in a config file or selected in the web UI). This way, the logic that loads image frames and decides what to display calls a generic method, and under the hood it goes to the appropriate hardware.

- *HUB75OutputDevice*: This would wrap calls to the existing rpi-rgb-led-matrix library. For example, if the current code uses matrix.SetImage(image) or matrix.SetPixel(x,y,color), that would be encapsulated here. The initialization would configure the matrix dimensions (64x64) and options (such as hardware multiplexing, PWM bits, etc., usually provided to the RGBMatrix constructor).

- *WS2811OutputDevice*: This new class would initialize the NeoPixel/WS2811 driver. Using the rpi_ws281x Python binding or Adafruit NeoPixel, it would set up the LED strip length (e.g. 100 LEDs) and the GPIO pin (e.g. 18). If using Adafruit's library, for instance:

```
None
import board, neopixel

pixels = neopixel.NeoPixel(board.D18, num_leds, auto_write=False)
```

- If using the lower-level library:

```
None
from rpi_ws281x import PixelStrip, Color

strip = PixelStrip(num_leds, gpio_pin, brightness=255)

strip.begin()
```

- The driver class would also need to know the geometry (width, height of the LED grid) and the mapping order (serpentine) so it can translate x,y coordinates to strip indices. We will discuss mapping in detail in the next section.
- *External Controller Option*: If we plan to support an ESP32 output as well, we could also implement, say, ESP32OutputDevice that handles network communication. For example, it might have an IP address and port configured and on draw_frame it will format the pixel data into a UDP packet or HTTP request. This can be added later without affecting the rest of the code, if the abstraction is done right. Initially, focusing on the Pi direct method is fine, but keeping this possibility in mind is useful.

2. Handling Resolution and Content Scaling: The software must deal with the different panel sizes. A 64×64 image cannot directly display on a ~10×10 LED grid meaningfully without down-sampling. There are a couple of approaches here:

- We can restrict or define specific animations for the small panel. For example, perhaps the user will upload separate animations that are already designed for 10×10 (or whatever the grid size is). This would yield the best visual result since an artist could optimize the content for low resolution.
- More generally, we can programmatically scale down or sample larger images to the grid resolution. The Python Pillow library (PIL) makes this easy. We can resize each frame of a GIF to (width, height) = (10,10) and then push those pixels. In fact, the existing code might already be using PIL to handle images for the 64×64 matrix (for instance, converting GIF frames to an image buffer). We can insert a resize step. Example:

```
None
from PIL import Image

img = Image.open(frame_file).convert('RGB')  # load frame

img_small = img.resize((grid_width, grid_height), Image.ANTIALIAS)  # downscale
```

- If the aspect ratio differs (say the LED grid is 10×10 but source is 64×64, which is 1:1 so that's fine; but if it were a non-square matrix like 8×12, one might need to handle aspect ratio or cropping appropriately). For smooth scaling, ANTIALIAS (or now called Image.LANCZOS) can be used. However, for pixel-art style GIFs, sometimes NEAREST (nearest-neighbor) might be better to avoid blending colors. This can be a user or design choice.

- The system could also allow choosing a portion of a larger image to display on the 10×10, but that likely complicates the UI. Simpler is to uniformly scale.

- Color depth remains the same (24-bit color), so no change needed there. But note that the HUB75 library often uses its own parallel PWM to achieve brightness levels. WS2811 LEDs have built-in PWM and accept 8-bit per color values. So the color rendering should be comparable. If anything, the WS2811 might appear "smoother" at low brightness because each LED is static until updated, whereas the HUB75 is constantly refreshing and multiplexing (though too fast to notice most of the time). For our purposes, sending 8-bit RGB to the strip is fine. If the original system used gamma correction tables or brightness adjustments for the panel, we might want to apply similar correction to the LED strip output to maintain visual consistency. The rpi_ws281x library does not automatically gamma-correct, but we can manually adjust the values if needed (or some libraries let you set a gamma LUT).

3. Implementing the Serpentine Mapping: The WS2811 LEDs in the lightbox are connected in a zig-zag pattern – meaning electrically the LEDs are in one long string (from LED #0 to LED #99, for example), but physically they form rows that alternate direction. For instance, on a 10×10:

- Row 0 might go left-to-right (LED indices 0-9),
- Row 1 goes right-to-left (indices 10-19),
- Row 2 left-to-right (20-29), and so on.

We need a function to map a coordinate (x,y) in the grid to the correct index in the LED strip. This is straightforward math:

```
None
def led_index(x, y, width):
    if y % 2 == 0:
        # even row: left to right
        return y * width + x
    else:
        # odd row: right to left
        return y * width + (width - 1 - x)
```

So if we have width=10, led_index(0,1,10) (x=0, y=1 which is second row) would give 1*10 + (10-1-0) = 19, meaning the LED at (0,1) is index 19 in the chain – which makes sense because the second row is reversed and ends at index 10 on the far right, then 11 at (9,1) on the left, up to 19 at (0,1). This mapping must be applied whenever we write to the LEDs.

If the grid isn't a perfect rectangle or has a different snake pattern, we would adjust accordingly. (Some installations snake in columns instead of rows, but given the description, row serpentine is likely.) We can encapsulate this in the driver class so that higher-level code can still think in terms of x,y. For example, the WS2811OutputDevice.draw_frame(frame) might accept frame as a 2D array or image and internally loop y then x, do the mapping, and set the LED colors.

This need for coordinate remapping on serpentine NeoPixel matrices has been encountered by others; one project noted *"the wiring does not follow a typical Cartesian grid, but rather uses a weaving serpentine pattern"* and thus required overriding the coordinate mapping in their library. Our approach above is exactly to handle that.

4. Integrating with Flask GUI: The Flask web interface will need an option for the user to select the output mode (Matrix vs Lightbox). This could be a simple toggle button or a dropdown. For example, a dropdown "Output Target" with values {HUB75 Matrix, WS2811 Lightbox}. When the user selects one, the server could either:

- Immediately switch the mode (if an animation is currently running, it could stop on one device and start on the other).
- Or require the user to confirm and reload the animation in the new mode.

The simplest implementation is a setting stored server-side (in a config or even in memory) that the Flask routes check. For instance, if there's a route that starts an animation, it will check a global CURRENT_OUTPUT and choose the corresponding driver. We must ensure that the drivers don't conflict; if both devices were active, they might both try to use resources. But since we are doing alternate mode, we can ensure only one is active at a time. (If running both simultaneously was desired – e.g. mirror the animation on both – that's another scenario, but the question suggests alternate outputs, not concurrent, so we can keep it one at a time.)

From a code perspective, on startup we could initialize both drivers (if hardware permits) but it might be safer to initialize on demand. For example, if default mode is HUB75, we set that up. If user switches to WS2811, we de-init the matrix (freeing GPIOs) and init the WS2811 strip. There are some GPIO pin overlaps to consider: HUB75 matrices typically use a bunch of GPIOs (and specific ones if using an add-on HAT). The WS2811 might want GPIO18. If the HUB75 uses GPIO18 for something (it might, depending on pin mapping), that could conflict. Usually, HUB75 panels don't use GPIO18 because that's PCM or PWM pin which might be avoided due to audio, but it depends on configuration. If there is a conflict, we may need to physically disconnect the panel when using the strip, or at least ensure the matrix library isn't driving while the strip is. In practice, it might be easiest to not initialize both at the same time, and clearly power down or disable the unused device when switching (to avoid ghost signals).

From the Flask UI viewpoint, after switching mode, the content might need to be adjusted (for example, if an animation was loaded for 64x64, we might want to automatically rescale it for 10x10, or possibly have separate content prepared – this can be handled behind the scenes by the code as mentioned). We should inform the user of the resolution difference, perhaps in the UI (so they know a detailed image will be extremely pixelated on the 100-LED display).

5. Minimal Code Changes Emphasis: We want to reuse as much code as possible:

- The animation loop/timer that advances frames can remain the same.
- Image decoding (using PIL to get frames from a GIF, etc.) remains the same except for adding a resize step.
- Any double-buffering logic or gamma correction from the HUB75 code can be repurposed for the WS2811 if needed. Note that rpi-rgb-led-matrix often did its own double-buffering to prevent flicker on the panel; for NeoPixels, you don't really need double buffering because updating 100 LEDs is nearly instantaneous (a few milliseconds) and the LEDs hold their state until the next update. But if the code was written assuming a double-buffer mechanism, we can either strip that out or simulate it easily (e.g. build an off-screen pixel array, then output it at once with show() which is effectively double buffering).
- Brightness control: The matrix library had brightness settings (HUB75 panels can adjust PWM intensity). The WS2811 drivers also often allow setting a global brightness. For instance, PixelStrip(..., brightness=some_value) or Adafruit NeoPixel(..., brightness=0.5) as a fraction. We can expose a single brightness control on the GUI that affects whichever output is active. This again can be unified in the driver interface (e.g. both drivers implement set_brightness()).
- Example Code Snippet: Below is a simplified illustration of how one might integrate the WS2811 output in Python (assuming use of Adafruit's NeoPixel library for clarity). It includes the mapping function and frame drawing loop:

```
None
import board, neopixel


# Configuration
```

```
WIDTH, HEIGHT = 10, 10   # dimensions of the LED grid
NUM_LEDS = WIDTH * HEIGHT
LED_PIN = board.D18      # data output pin (GPIO18)
pixels = neopixel.NeoPixel(LED_PIN, NUM_LEDS, auto_write=False)

def led_index(x, y):
    """Map (x,y) on the serpentine grid to the LED strip index."""
    if y % 2 == 0:  # even row
        return y * WIDTH + x
    else:           # odd row (reversed)
        return y * WIDTH + (WIDTH - 1 - x)

def display_frame(pixel_colors_2d):
    """Display a 2D list of RGB tuples (height x width) on the LED strip."""
    for y in range(HEIGHT):
        for x in range(WIDTH):
            idx = led_index(x, y)
            r, g, b = pixel_colors_2d[y][x]
            pixels[idx] = (r, g, b)
    pixels.show()

# Example usage: clear all pixels (set to black)
black_frame = [[(0,0,0) for x in range(WIDTH)] for y in range(HEIGHT)]
display_frame(black_frame)
```

In the above snippet, pixel_colors_2d could be obtained by taking a PIL image of size 10×10 and doing list(image.getdata()) then reshaping it into [height][width]. The mapping function ensures the image's [0,0] pixel goes to LED index 0, [1,0] to LED 1, etc., in the correct serpentine order.

If using the lower-level rpi_ws281x library, the logic is similar except you use strip.setPixelColor(i, Color(r,g,b)) and call strip.show(). The important part is that all LEDs are updated for each frame, whether by setting the pixels[] array and calling .show() or by writing to a PixelStrip and calling .show(). This is effectively **double buffered** because you prepare all pixel values then latch them to the strip at once, avoiding tearing.

6. Integrating an ESP32 Option in Software (optional): For completeness, if we wanted the code to optionally output to an ESP32 (running WLED or custom firmware), we could implement the display_frame to send data over network instead. For example, if using WLED's E1.31 support, one could construct an E1.31 packet (512 bytes universe data, where each LED uses 3 bytes). Or using WLED's UDP realtime, simply send a packet of 3*N bytes (R,G,B for each LED in order) to WLED's UDP port (default 21324). WLED will listen and immediately update the LEDs with that data. The code for that could look like:

```
None
import socket

UDP_IP = "192.168.0.50"   # ESP32 running WLED

UDP_PORT = 21324

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)


def send_frame_udp(pixel_colors_2d):

    data = bytearray()

    for y in range(HEIGHT):

        for x in range(WIDTH):

            idx = led_index(x,y)

            r, g, b = pixel_colors_2d[y][x]

            data += bytes([r, g, b])

    sock.sendto(data, (UDP_IP, UDP_PORT))
```

This simple approach assumes the LED order in WLED is configured the same as our led_index (we might actually want to send in natural order and let WLED handle serpentine mapping via its segment settings, or just wire it consistently and send in our computed order). WLED also expects a specific 2-byte header (0x0000) for "Realtime" packets in some versions – but as of now, sending raw RGB should work by default (it interprets any packet of the correct length as a frame). If using a custom microcontroller code, we could define any protocol we like (even the same as above).

The Flask GUI would need a way to specify the target IP or COM port for the external controller. This could be another config field in the UI (e.g. a text box for "Controller IP" if network-based). However, if the system is typically used all on one Pi, this might be overkill unless we specifically plan for remote installations.

In summary, the software changes focus on adding a new output driver and managing different frame sizes. By keeping the architecture modular, we ensure minimal disruption to existing functionality. The user will experience it as simply having an extra display option.

Software Libraries for WS2811 LEDs and Integration

When adapting to WS2811 LEDs, we will leverage existing libraries and tools to avoid reinventing the wheel for LED control and to maintain high performance:

- rpi_ws281x (Python) – This is the underlying driver library (originally written by Jeremy Garff) for controlling WS2811/WS2812 LEDs on the Raspberry Pi. It includes a C library that uses the Pi's PWM or PCM hardware with DMA to precisely time the pulses, as discussed. The Python wrapper allows easy use from our Flask application. We can install it (e.g. via pip as rpi_ws281x or adafruit-circuitpython-neopixel which wraps it). Using this library, we can control a single strip (or even multiple strips with different channels, if needed) of NeoPixels. The Adafruit guide notes that this library

"does all the hard work of setting up PWM and DMA to drive NeoPixels... so the data signal can be generated without being interrupted by the OS". This is ideal for our use-case on the Pi. In code, one would create a PixelStrip object, call begin(), then set pixel colors and call show() to update as shown earlier.

- Adafruit CircuitPython NeoPixel – This is a higher-level Python library provided by Adafruit. Internally, it uses the above rpi_ws281x (via a compatibility layer called Blinka). The advantage of this library is a very Pythonic interface (you treat the LED strip like a list of tuples), and it abstracts away some details like color object creation. We have used it in the code snippet for clarity. Both this and direct rpi_ws281x are viable. Adafruit's NeoPixel library also supports features like setting a pixels.brightness property (0.0 to 1.0) to globally dim the LEDs in software, which can be convenient.

- FastLED (for microcontrollers) – If we go the microcontroller route and decide to write custom firmware, FastLED is a powerful Arduino/C++ library supporting many LED chip types (including WS2811). It provides high-level functions for controlling LEDs, color palettes, dithering, etc. We could write an Arduino sketch that uses FastLED to set up a 100-LED CRGB array and updates it via data received (over serial or WiFi). For example, on an ESP32 we might use FastLED's show() to output on a pin configured for I2S or RMT for smooth timing. FastLED also has a *Neopixel matrix* extension (e.g. Marc Merlin's FastLED_NeoMatrix library) which can handle matrix mapping and even offers an Adafruit GFX compatible API for drawing text and shapes. However, using FastLED means writing/compiling C++ code for the microcontroller – which is fine but outside the Python environment of our main system. If minimal changes are desired, keeping logic on the Pi side is simpler.

- WLED – This is not a library but a firmware, as mentioned. Running WLED on an ESP32 gives a ready-to-use solution that supports multiple control methods. According to its documentation, WLED can receive control data via HTTP requests (e.g. send a JSON with an array of LED colors), MQTT messages, or lighting protocols like E1.31, Art-Net, and DDP. DDP (Distributed Display Protocol) in particular is a very efficient binary protocol for LED data that WLED supports; it has lower overhead than E1.31 for large data. For 100 LEDs, any of these would work. WLED's advantage is that it also includes its own web UI and tons of effects, so in a pinch, you could even bypass the Flask GUI to test patterns on the lightbox. But in our integration, we'd likely have the Flask app tell WLED exactly what to show (essentially using WLED as a slave display device). WLED is confined to ESP8266/ESP32 hardware (it cannot run on the Pi), so it's only applicable for the external controller scenario.

- Other Tools:

    - *Pigpio / PWM via hardware*: Another way to drive WS2811 on Pi is using the pigpio daemon, which can generate waveforms with DMA as well. But since rpi_ws281x is already tuned for NeoPixels, we prefer that.

    - *LED Matrix libraries*: Since our original system used rpi-rgb-led-matrix for the HUB75, one might wonder if that library could also drive NeoPixels (since it had convenient canvas drawing functions). The answer is generally no – it's specifically for parallel RGB panels, not serial LEDs. Developers on its forum have recommended using NeoPixel-specific libraries instead. So we treat the two outputs differently at the driver level.

    - *BiblioPixel / LEDila / others*: There are some Python frameworks like BiblioPixel and LED Arcade, etc., which aim to be universal LED controllers supporting both matrices and strips. These could theoretically provide a unified API for both types of outputs. However, integrating those into our existing system might be heavy and unnecessary. Since we already have a working pipeline for images to HUB75, extending it with a few functions for WS2811 is straightforward without a large framework.

    - *Open Pixel Control (OPC)*: OPC is a protocol that FadeCandy and some other LED controllers use (FadeCandy is a USB LED driver for NeoPixels with built-in dithering). We likely don't need OPC here, but it's worth noting if the user ever chose to use a FadeCandy board to drive the 100 LEDs (FadeCandy can handle up to 512 LEDs and connects via USB). In that case, the Pi would send pixel data over USB in OPC protocol. That said, FadeCandy is somewhat legacy at this point and an ESP32 with WLED is more flexible for modern use.

Software Architecture Summary: The system will initialize the appropriate library depending on mode. For HUB75, continue to use rpi-rgb-led-matrix (with its Python binding). For WS2811, initialize rpi_ws281x/NeoPixel on a chosen GPIO. Both will be fed frames from the same animation logic. The Flask web app will allow mode switching. By maintaining a clean separation, the bulk of the code (frame generation, web interface, etc.) remains unchanged. This approach prioritizes code reuse and minimal rewriting: essentially, we're adding functionality rather than changing the existing functionality.

Mapping Animations to the WS2811 LED Grid

As noted, the WS2811 LED arrangement is a **2D serpentine matrix**. We must map 2D animation frames onto this physical layout correctly:

- **Coordinate System**: We treat the LED grid as having (0,0) at one corner (say top-left for consistency with typical images). The "serpentine" means that one row goes left-to-right, the next goes right-to-left. We already provided a mapping function led_index(x,y) above. We will use that whenever writing to the LEDs. For clarity, if the grid is 10 columns wide:
    - LEDs 0-9 correspond to row 0 (x=0..9 left to right),
    - LEDs 10-19 correspond to row 1 but reversed (x=0 corresponds to LED19, x=9 to LED10),
    - etc.
- **Image Scaling**: If the source animation is larger (like 64×64), we create a downsampled 10×10 version (or whatever the grid size is). The quality of the downscaling can be decided. Using Pillow, one might do:

```
None
frame = frame_image.convert('RGB')

small_frame = frame.resize((WIDTH, HEIGHT), Image.NEAREST)
```

- Nearest-neighbor (Image.NEAREST) will pick the dominant color in each block of the original image. This might be best if the original is pixel art. If the original has gradients, using a filtered resize (Image.BILINEAR or Image.LANCZOS) might produce a smoother but possibly dimmer result. We can experiment and possibly allow the user to choose scaling method if needed. Since the LED grid is so low-res, even text might be unreadable unless the font is designed for that size. So likely the animations will be simple shapes, icons, or text scrolling. One idea is to implement a **scrolling text** or marquee mode for the WS2811 (which might not have been needed on the 64×64 since it could show a lot at once). With only 10 columns, to display a message you'd have to scroll it. We can reuse code or fonts if we have any for the matrix. This, however, goes beyond just showing existing animations – it's an additional feature. So unless needed, we can leave text rendering aside.
- **Frame Data Structure**: After resizing, we have a small image (e.g. 10×10). We can get its pixel data as a list of (R,G,B) tuples. For example, list(img_small.getdata()) would return 100 tuples. To map this to the LED strip, two methods:
    1. Compute each LED's tuple by mapping its x,y. We can do nested loops as in display_frame above.
    2. Pre-compute a mapping list, i.e. a list of length 100 where each entry is (x,y) for that LED index. Then simply iterate over index and pick the corresponding pixel from the image. For example:

```
None
mapping = []

for y in range(HEIGHT):

    for x in range(WIDTH):

        mapping.append((x,y) if y % 2 == 0 else (WIDTH-1-x, y))

# Now mapping[i] gives the (x,y) that corresponds to LED i in the chain.
```

2. Then:

```
None
pixels = img_small.load()

for i, (x,y) in enumerate(mapping):

    r,g,b = pixels[x,y]

    strip.setPixelColor(i, Color(r,g,b))

strip.show()
```

2. This might be slightly faster by avoiding two nested loops in Python for each frame (we flatten it into one loop). But with only ~100 pixels, performance is not a concern – it will be extremely fast either way (0.1 ms or so per frame to do the mapping in Python, negligible compared to the 3 ms it takes to actually send out 100 LED data).

- Double Buffering and Synchronization: On the HUB75 panel, tearing or flicker could be an issue if not synchronized with the panel's refresh (hence SwapOnVSync calls in that library). With NeoPixels, the LEDs latch new data only when the full stream is sent followed by a reset pulse. So if we update them in one go with strip.show(), all LEDs update nearly simultaneously. There is no need for an explicit sync; just ensure we don't call show() in the middle of updating a frame. In our code, we set all pixel values then call show(), so that's handled. If we were doing something super clever like streaming partial data, it would complicate things, but we're not.

- Animation Playback Rate: If the original animations were timed for a certain frame rate (e.g. a GIF that is 20 FPS), our system should honor that. Likely the existing code uses a loop with a delay or uses the GIF's frame durations. We should keep that consistent. The WS2811 can easily handle typical animation speeds. We just need to be careful not to run frames faster than the original or overwhelm the network if using Wi-Fi. But 100 pixels * 30 FPS is only 9,000 RGB bytes per second, which is trivial for any network.

- Memory Considerations: Storing a 64×64 animation vs 10×10 is a non-issue on the Pi. We likely process frames on the fly, not store them all, but either way, 100 pixels per frame is tiny. If anything, using PIL to downscale will use negligible CPU compared to what it took to handle 64×64. We should actually consider whether we want to keep using full 64×64 source frames at all in WS2811 mode or if we should directly demand lower-res content. But a dynamic approach (resizing on the fly) makes the system more flexible (you can throw any image at it and it will do its best).

- Color Order: Note that WS2811 LEDs often use GRB order (the protocol expects the first byte after reset to be green, then red, then blue for the first LED). The rpi_ws281x library and Adafruit library typically take care of this (defaulting to GRB). The Color(r,g,b) function in rpi_ws281x will arrange it correctly. In Adafruit's neopixel.NeoPixel, by default it also assumes GRB and will swap your tuple accordingly unless you specify a different order. So as long as we use these libraries normally, we can treat colors as RGB in our code. If we notice colors are swapped (like blue and green switched), we may need to specify the pixel order. For instance, neopixel.NeoPixel(board.D18, num_leds, pixel_order=neopixel.RGB) if the strip is actually RGB order. Most WS2811 are GRB though.

In summary, mapping the content to the LED string involves resizing the content and sending it out in the serpentine sequence. We have the tools and methods for that, and it integrates smoothly with the existing content pipeline.

Electrical Wiring and Power Considerations

Driving WS2811 LEDs requires proper wiring to ensure signal integrity and sufficient power delivery. The new lightbox has ~100 LEDs, which at full brightness white can draw a significant current, and the data signal from the controller must be within spec. Here's how to wire and power the system safely:

Wiring diagram for connecting a Raspberry Pi to a WS2811/WS2812 LED strip (or matrix) with a level-shifting chip and external power supply.

- Data Line Connection: Connect the data input of the first WS2811 LED in the chain to the Raspberry Pi's GPIO pin that will output the signal (e.g. GPIO 18 if using PWM, or another if configured). Do not connect it directly without considering voltage levels. The Pi's GPIO is 3.3V, while WS2811 inputs are 5V logic. Often, a 3.3V signal can be *just barely* recognized as HIGH by some WS2811 pixels, but it's not guaranteed (the WS2811 datasheet's threshold is around 0.7*Vdd ≈ 3.5V for HIGH). To be reliable, use a level shifter to shift 3.3V up to 5V . A recommended chip is the 74AHCT125 (a quad level buffer) or 74HCT245. In the diagram above, a 74AHCT125 is used: Pi's GPIO18 connects to the level shifter input, and the output goes to the LED DIN. If a level shifter chip isn't available, some have used a simple MOSFET or even just tried direct connection. *If you run into issues, add the level shifter, as it often fixes strange flickering or non-responsive LEDs.* Another trick is to ensure the first LED shares ground and is very close to the Pi so the signal hasn't degraded much – sometimes the first LED will accept 3.3V and regenerate it to 5V for the rest, but again, it's not guaranteed across all LED strips/batches.

- Grounding: The ground reference must be common between the Raspberry Pi and the LED strip's power supply. Connect the Pi's GND to the LED strip GND, and also to the negative terminal of the 5V power supply. In the diagram, all grounds (Pi GND, level shifter GND, LED GND, PSU negative) are tied together. This common ground is crucial for data signal integrity; without it, the data line has no reference and the LEDs will not latch the data correctly.

- Power Supply for LEDs: Do NOT power the 100 LEDs directly from the Raspberry Pi's 5V pin. The Pi's 5V output (from its GPIO header) is limited by the Pi's regulator and/or polyfuse and is only meant for very small loads. A strip of 100 WS2811 LEDs at full brightness white could draw up to ~100 * 60 mA = 6000 mA (6 A)! Typically, you won't have them all full white, but even a few bright colors can pull a couple of amps. The Pi cannot source this from its USB power input safely. Use an external 5V DC power supply rated for the LED load (for example, a 5V 10A supply would be plenty for 100 LEDs and leaves headroom). Connect the +5V from this supply to the LED string's +5V line. It's a good idea to also connect the +5V to the level shifter's VCC (if using a 74AHCT125, power it at 5V so its outputs drive 5V).

    - If the LED matrix is physically small (10×10), you can likely feed power at one point. If it were a long strip, sometimes you inject power at both ends to reduce voltage drop. For 100 LEDs, feeding at one end might result in a slight drop (the farthest LED might see maybe 4.5V if the wire is thin and all LEDs are on white). To be safe, you can inject power at both ends of the string (and even middle) by running 5V and GND wires to those points from the supply. This evens out the voltage. Monitor the brightness of white across the matrix; if the far end looks reddish (voltage sag causing blue/green to dim first), you need to feed power in more places or use thicker wires.

    - Use appropriately thick wiring for the power lines. 6A is quite a bit of current – AWG 18 or 20 wires are recommended to carry that safely over short distances. Avoid long thin jumper wires for power.

- Protection Components: The diagram shows a diode (1N4001) from the power supply 5V to the LED strip 5V . This is sometimes added to prevent the LED strip from back-feeding power to the Pi. For instance, if the Pi is off but the LED PSU is on, current could flow from the LED's 5V into the Pi's 5V rail through unintended paths. A diode (stripe side toward LEDs) will allow current *to* the LEDs but block reverse current. It also drops a bit of voltage (~0.7V) – in some Adafruit setups, they did this because their supply was slightly above 5V or to reduce the voltage to exactly 5.0 from 5.2. It's an optional safety measure. If used, ensure orientation is correct (cathode to LED side).

  Another protection is a small resistor (50–300Ω) in series with the data line right before the LED DIN. This can dampen ringing on the data line and is recommended by Adafruit's NeoPixel best practices. In many cases it works fine without it, but if you see sporadic issues, adding a 220Ω resistor on the data line close to the LED input can help. It's a cheap addition.

  And of course, never connect or disconnect the LEDs while power is applied. Hot-plugging can send voltage spikes that kill the first LED. Always power down before rewiring.

- Powering the Raspberry Pi: The Pi will still need its own 5V supply via USB or GPIO input (or however it was powered for the original project). Make sure the Pi and LED supply share ground. If the LED supply is very robust (5V 10A), you could theoretically power the Pi from it as well (e.g. splitting off a line to the Pi's 5V input). This is fine as long as the supply is stable and within 5V ±5%. Often for convenience, people do use one supply for both Pi and LEDs, connecting 5V to Pi's 5V pin. Caution: the Pi has a polyfuse and recommended max current draw of about 2A on its 5V line, so if you do this, ensure the LEDs are not drawing through the Pi (the diode or separate wiring as above takes care of that). Essentially, wire the 5V in a star: PSU to Pi, PSU to LED (with diode perhaps), not Pi 5V -> LED. That way the heavy LED current bypasses the Pi.

- ESP32/Microcontroller Wiring (if used): If using an ESP32 instead:

    - The ESP32 would connect to the LED data line (often on pin GPIO2, 5, 18, or others that support RMT for NeoPixel – WLED by default uses GPIO2 or GPIO16 depending on board). The same level shifting considerations apply. However, many ESP32 dev boards output a 3.3V signal that is often *just* enough for short runs to a WS2811. It's better to level-shift here too for reliability, but lots of hobbyists run small strips directly from ESP32 3.3V data and get away with it. If doing a permanent installation, use a level shifter.

    - The ESP32 must also share ground with the LED supply. And if the ESP32 is powered via USB or its own regulator, its ground must tie to LED ground.

    - Powering the ESP: If using the same 5V supply, you can feed the ESP32's 5V (VIN) pin from it (many ESP dev boards like NodeMCU or Feather ESP32 have a 5V input that goes through a regulator to 3.3V). The ESP draws nowhere near as much as the LEDs (maybe 200 mA peak with WiFi), so it's fine to share the supply as long as it's sized for total current.

    - An ESP8266 or Arduino would similarly connect. Arduino Uno, for instance, is 5V logic already, so it could drive the LED directly without level shifter (but still common ground and separate power supply for LEDs).

    - If using an ESP and WLED, the only wiring difference is instead of Pi's GPIO -> LED, it's ESP's GPIO -> LED. The Pi in that case only connects to the ESP via network or serial, not direct electrical connection (except ground if using serial).

- Mounting and Noise: WS2811 LEDs, if in a lightbox, presumably are either individual nodes or a strip arranged in rows. Ensure the wiring between rows (for serpentine) is correct and sturdy. Sometimes twisting the data line with ground can help reduce interference if the cable run is long. For 100 LEDs in close proximity, interference isn't likely a big issue. If the data line runs longer than, say, 50 cm from Pi to first LED, consider using a twisted pair (data and ground) or even a differential driver for very long runs (RS485 converters) – but for a lightbox, likely the Pi or ESP will be right next to it.

- Testing and Safety: After wiring, test with a simple script (like a color wipe or blinking one LED) to verify that the LEDs respond correctly. Always start at low brightness or with a few LEDs to ensure your wiring can handle it. Then you can run full brightness tests to see if the supply holds up (voltage should stay near 5.0, and nothing should overheat). It's good to have a fuse or some protection on the power supply output as well, especially if this is a permanent installation (a short in one LED could otherwise draw huge current; many bench supplies have current limiting or you can add an inline automotive fuse of appropriate rating).

In summary, use a proper 5V power source, common ground, and level shifting to adapt the Pi (or ESP) to the WS2811 strip. These precautions will ensure the WS2811 lightbox runs reliably without harming the Pi or the LEDs. The Adafruit guide emphasizes not powering too many LEDs from the Pi and the importance of a level shifter. Following those guidelines, our hardware setup will be robust.

Performance Considerations and Syncing with Animations

Frame Rate and Throughput: WS2811 LEDs have a fixed timing: each LED takes 24 bits, and the data stream for N LEDs is 24*N bits sent at about 800 kHz, plus a reset pulse (>50 µs) to latch. For 100 LEDs, one full frame update takes ~100 * 24 / 800,000 = 0.003 seconds (3 ms) plus ~0.05 ms reset ≈ 3.05 ms. This theoretical max is about 327 frames per second if we could send continuously. In practice, if we're doing calculations or if using Python, we won't hit that, but achieving 30 or 60 FPS is trivial; 60 FPS would use only ~18% of the LED bus bandwidth. On the Raspberry Pi, the rpi_ws281x library often defaults to 800kHz and can achieve ~full bandwidth. Python overhead to prepare data for 100 LEDs is minimal. So we anticipate no issues achieving smooth animation on the WS2811 grid. The HUB75 panel, by contrast, might have been limited by how fast the Pi can toggle those pins (though that library is highly optimized in C++). In any event, the WS2811 is not the bottleneck here.

If using an external microcontroller, the bottleneck might be the communication channel:

- Wi-Fi UDP can easily handle 100*3 bytes * 60 fps = 18,000 bytes/sec, which is nothing for Wi-Fi (even 802.11n can do millions of bytes per sec). Latency of a UDP packet on a local network is usually < 5 ms, so real-time animation works fine. If network congestion occurs, one might drop a frame, but for a dedicated setup it's okay. If reliability is paramount, one could use TCP or ensure sequence numbers in UDP, but that's likely overkill here.

- Serial at 2 Mbaud could also handle it (2,000,000 bits/sec ~ 200,000 bytes/sec, enough for hundreds of FPS at 300 bytes/frame). So performance is fine.

CPU Usage: On the Pi, decoding images and resizing them might use more CPU than the LED output. But since it already handled 64×64 images, doing 10×10 is lighter. The rpi_ws281x library's DMA approach means the CPU is free while the LED data is clocking out – it just needs to fill a buffer and start the transfer, which is quick. Thus, the Pi can likely drive the LEDs *and* handle the Flask server simultaneously without issues. If the Pi was close to its limit with the HUB75 (that library can be CPU-intensive especially if doing high refresh or multiple panels), then turning that off and using WS2811 should actually be *less* CPU load. If we ever attempted to run both outputs at once (mirroring content), the Pi would be doing more work – but still probably within reason, as the LED stuff would be in C and parallel. However, since we plan alternate mode, that's moot.

Synchronization with Animation Timing: If the animations are time-based (some GIFs have specific frame delays, e.g. 100 ms per frame), our code should maintain those delays for the WS2811 as well. The HUB75 library sometimes can run uncapped if not careful, but presumably the existing code accounts for frame timing. We should reuse that timing. If the user can adjust speed in the GUI, that should apply equally to both outputs.

Brightness and Power Limits: Running 100 LEDs at full white 100% brightness will draw ~6A. It's unlikely an animation does that continuously (unless it's a strobe or something). But to be safe, it might be wise to cap the brightness in software. For example, one could set the brightness property of the NeoPixel library to, say, 0.8 (80%) or even lower, to avoid hitting the absolute max current. Many LED projects intentionally run at 50% or less, because the perceived brightness is still quite high and it gives a safety margin. We could expose a brightness slider in the GUI if not already present. This would be useful since the HUB75 panel might have needed a different brightness than the LED strip. The user can dial it in for each environment.

Gamma Correction: Often LED colors look better if gamma-corrected (because our eyes don't perceive brightness linearly). The HUB75 library might be doing some gamma correction internally (not sure if it does; some do 8-bit to 16-bit expansion for smoother gradients). The WS2811 will show whatever we send. If the colors appear too harsh or not matching between the two displays, we could apply a gamma curve to the pixel values for the WS2811. A common approach is raising color values to 2.2 gamma or using a lookup table. Adafruit's NeoPixel library does not automatically gamma correct; it leaves it to the user. If the existing system wasn't doing gamma correction for the matrix (maybe it didn't need to), we might not bother. However, since the HUB75 has a different visual output (discrete LEDs but densely packed) versus possibly diffused WS2811 in a lightbox (maybe the lightbox has a diffuser panel?), the visual perception might differ. For example, at low values, the WS2811 might still seem bright if not gamma adjusted. This is something to fine-tune during testing.

Multi-Tasking and LED Timing: If driving WS2811 directly from Pi, one potential pitfall is if some other process or interrupt blocks the DMA for too long (rare). The known conflict is with audio: on older Pi, the PWM for audio and PWM for NeoPixel can conflict. Solution: use PCM or SPI as output for NeoPixels. The rpi_ws281x library lets you choose the driver mode. It has PWM (PCM) or SPI. SPI mode requires you wire the LED to MOSI (GPIO10) and clock out a specific pattern. They found SPI mode is stable and has the benefit that you don't need root if /dev/spidev is accessible . In our case, we might stick to PWM on GPIO18 and simply not use analog audio (if sound is needed, perhaps route audio through HDMI or USB to avoid that conflict). Also, running the Flask server and some Python threads shouldn't interfere with the DMA, since DMA is hardware. However, if the CPU is maxed out, it might delay preparing the next frame slightly. Given our low frame data, that's unlikely. But if we ever drove, say, 1000 LEDs, the preparation might start to matter. At 100, we're fine.

Memory on Microcontrollers: If using an Arduino Uno (unlikely, but suppose), 100 LEDs * 3 bytes = 300 bytes, which is fine for an Uno's RAM (2 kB). For an ESP32, 300 bytes is nothing. So no issues there. Just note if using a very small micro (ATtiny, etc.), one must ensure enough RAM for buffer, but that's beyond our scope.

Comparison of Visuals: The HUB75 matrix has 4096 LEDs but each LED (like those P3/P4 panels) might be smaller and less bright individually, and they are multiplexed (each LED is actually lit a fraction of the time but persistence of vision makes it appear steady). The WS2811 LEDs are typically fairly bright (each LED continuously lit when on). If the lightbox uses diffusers, it may appear as a grid of soft "pixels." The brightness difference might mean that an animation that looked good on the big panel could be blinding on the small one at full power. This is where that brightness control is important. Also, color balance might differ – sometimes LED strips are not perfectly balanced (some have a cooler white, etc.). If needed, color calibration can be done (WLED for example allows per-channel brightness calibration). This is an advanced step that we likely won't automate, but just be aware.

Synchronization Between Devices: If one were to output to both the HUB75 and WS2811 at the same time (maybe for debugging or because you want two displays), we'd have to synchronize frame updates. It's easiest if one code loop updates both almost back-to-back. The HUB75's SwapOnVSync() provides a way to wait for the panel's refresh boundary to swap buffers without tearing. For the WS2811, show() returns as soon as data is written (or after a reset), which is very quick. We could update the WS2811 immediately after the HUB75 swap. They'll be close in time (< a few ms apart). If needed, one could also intentionally delay one to line up, but again this is only if doing concurrent mirror, which we aren't focusing on. Alternate mode simplifies things.

Testing Performance: It would be wise to test the system with a variety of content:

- Static images (makes sure mapping is correct).
- Fast animations (to ensure we hit desired FPS and nothing crashes).
- All pixels on (white) to test power and stability.
- Rapid color changes to see if any tearing or delay occurs.

Given the small scale, it should pass with flying colors. The WS2811 protocol is robust as long as timing is right – if the timing were off, you'd see glitched LEDs, but with the proven library, that's handled.

If using WLED, one performance tip: WLED's default UDP realtime will automatically drop back to its own effects if it stops receiving data for a while (like 500ms by default). That's fine, but if you see it flipping to another effect, you might increase the "realtime timeout" in WLED settings or continuously send data even if nothing changes (maybe at lower rate). Also, WLED can handle up to ~1000 LEDs at 30 FPS via E1.31 on ESP32 easily, so 100 is trivial.

Conclusion on Performance: The adapted system will comfortably drive the 100-pixel WS2811 matrix with smooth animation and minimal latency. The Raspberry Pi 3 B+ is more than capable of the necessary computation and data output, especially with the hardware-accelerated LED driving. If an external microcontroller is used, it offloads the Pi even further, basically guaranteeing LED timing integrity. Either way, the user experience should be of equal quality: fluid animations on the new WS2811 lightbox, with timing and color fidelity similar to the original HUB75 display (acknowledging the resolution difference).

Example Workflow and Instructions for Integration

To tie everything together, here's a step-by-step outline of how one would implement and use this adapted system:

1. Setup the Hardware: Mount the ~100 WS2811 LEDs in the desired grid layout (ensuring the wiring is serpentine from one row to the next). Connect the LED's 5V and GND to a capable external 5V power supply. Connect the LED data input to Raspberry Pi's GPIO18 through a level shifter. Tie all grounds together. Double-check the wiring matches the diagram and that the first LED in the chain is connected to the Pi (DIN) and the last LED's DOUT is left unconnected (or goes to the next chain if there was one). If using an ESP32 instead, wire that in and connect it to network/power accordingly.
2. Install Required Software Libraries: On the Raspberry Pi (Raspbian OS):
    - Install the NeoPixel library. For example, run pip3 install rpi_ws281x adafruit-circuitpython-neopixel. This will get the necessary dependencies. Alternatively, if preferring the lower-level usage, clone the rpi_ws281x GitHub and build it, but pip is easier.
    - Ensure the rpi-rgb-led-matrix library (for HUB75) remains installed as before.
    - (If using an ESP32 with WLED, flash the WLED firmware to the ESP32 using its instructions and configure the LED count to ~100, set LED type to WS281x, set pin, etc. Note the device's IP address on your network.)
3. Code Integration: Modify the Python Flask application:
    - Define configuration for the two display modes, e.g. DISPLAY_MODE = "HUB75" or "WS2811" (maybe read from a config file or an environment variable for default).
    - Initialize both drivers or on-demand. Example initialization for WS2811:

```
None
import board, neopixel

WS2811_WIDTH = 10
```

```
WS2811_HEIGHT = 10

NUM_PIXELS = WS2811_WIDTH * WS2811_HEIGHT

pixels = neopixel.NeoPixel(board.D18, NUM_PIXELS, auto_write=False)
```

- If using rpi_ws281x directly:

```
None
from rpi_ws281x import PixelStrip, Color

strip = PixelStrip(NUM_PIXELS, 18, brightness=255, strip_type=ws.WS2811_STRIP_GRB)

strip.begin()
```

-

- Implement the mapping function as shown earlier. Possibly precompute map list for speed.
- Wherever in the code an animation frame is drawn to the matrix, add a branch: if mode is WS2811, call the function to output to the LED strip instead. For example:

```
None
def output_frame(image_frame):

    if mode == "HUB75":

        # Convert PIL image to the matrix's format and display

        matrix.SetImage(image_frame)

    elif mode == "WS2811":

        # Resize image_frame to (WS2811_WIDTH, WS2811_HEIGHT)

        small = image_frame.resize((WS2811_WIDTH, WS2811_HEIGHT),
   Image.NEAREST).convert('RGB')

        # Transfer to LED strip

        pixel_data = list(small.getdata())

        for i, (r,g,b) in enumerate(pixel_data):

            # Map i (if pixel_data is row-major in normal order) to serpentine order:

            # Compute x,y from i: x = i % width, y = i // width
```

```
            x = i % WS2811_WIDTH
            y = i // WS2811_WIDTH
            # Map (x,y) to index j
            j = y * WS2811_WIDTH + (x if y % 2 == 0 else (WS2811_WIDTH - 1 - x))
            pixels[j] = (r, g, b)
        pixels.show()
```

- (This code inlines the mapping for clarity. It assumes pixel_data is in row-major order starting top-left. If using getdata(), that's true.)
- Adjust any frame timing code if necessary (likely not).
- Add a route or form input in Flask for switching mode. For example, the settings page could have a radio button or a toggle. When changed, you might call a small function to de-initialize the current output and init the other. In practice, you might not need to fully de-init; you can keep the matrix ready but just not updating it, and vice versa. However, freeing resources (like releasing the LED DMA or matrix threads) can be good. The rpi_ws281x doesn't spawn threads; it just uses DMA. The HUB75 library by default may spawn a C thread for refreshing. It might keep running even if we stop sending frames. Ideally, the library has a method to clear or terminate (or we can drop brightness to 0 on the matrix when not in use to effectively turn it off).
- If using an ESP32/WLED: instead of pixels.show(), you would send the UDP packet or HTTP request in output_frame. Possibly, maintain an open socket for UDP for performance.

4. Testing: Start the Flask app and test with a simple known image (like a red square) to ensure the WS2811 grid lights up correctly (e.g. top-left LED is red, and if you choose something like a diagonal line image, verify the mapping matches). You may find the image is rotated or mirrored depending on how you oriented the physical panel vs the coordinate system. You can adjust in software by swapping x,y or reversing columns, etc., until it matches a logical orientation (like the same orientation as the HUB75 display, if that's the goal).
    - Test the mode switch: display something on HUB75, then switch to WS2811, ensure one turns off and the other turns on showing the content.
    - Try an actual animation (GIF) on the WS2811 mode. Check for smooth playback and correct colors.
    - Test brightness control if available. Perhaps add a simple slider to the web UI to send a value to set brightness (you can call pixels.brightness = val and pixels.show() to apply new brightness on Adafruit lib; on rpi_ws281x you have to scale values manually or reinitialize with a brightness parameter because that library's brightness is a fixed global scaling).
    - Observe the power supply and LEDs for any signs of stress (overheating wires, etc.) when running bright patterns.
5. Optimization & Cleanup: Once it's working, you can refine:
    - If the frame rate is lower than expected, see if adding time.sleep() calls somewhere. Possibly the original code slowed down for 64×64; you might remove delays for the smaller grid if you want it faster, but likely it's tied to content.
    - If using an ESP32 and the latency is noticeable (shouldn't be, but if say Wi-Fi is laggy), you could consider using a wired Ethernet for the ESP32 (some boards have it) or even using an RS-485 wired link. But again, for 100 LEDs, Wi-Fi should be fine.

    - Memory usage on Pi: the NeoPixel library will allocate an array for 100 LEDs (300 bytes) – trivial. The matrix library allocates for 4096 LEDs – also not huge (maybe some MBs for internal buffers). So memory is fine.

    - When both libraries are loaded, watch out for any potential conflict in low-level resources: one possible conflict is if both try to use PWM hardware. rpi-rgb-led-matrix uses a lot of GPIO but not the Pi's PWM hardware (it bit-bangs via CPU). So it shouldn't conflict with PWM0 used by NeoPixels. If using the SPI method for NeoPixels, that uses SPI channel, which the matrix lib also doesn't use. So they can coexist installed, just not active on same pins.

6. Documentation for the User: Update any user manual or on-screen instructions in the Flask app to explain the new mode. For example, if the GUI had a section showing the LED panel status, include the WS2811 status. In the best case, the user should be able to just choose the output and everything else (uploading an animation, playing it) is the same process.

By following these steps, the WS2811 LED lightbox becomes a drop-in alternative output for all the cool animations originally made for the LED matrix. The two displays offer different experiences (the matrix is higher resolution, the lightbox might be more portable or have a certain aesthetic), and the system can cater to both without maintaining separate codebases.

This approach underscores code reuse and unified control: we did not rewrite the animation logic, we simply extended the output capabilities. The Flask interface remains the single point of control for the user, maintaining a seamless experience whether they are lighting up the large HUB75 panel or the smaller WS2811 lightbox.

Conclusion

Adapting the Raspberry Pi-based LED animation system to support a WS2811 addressable LED grid is achievable with a few targeted modifications and careful planning of hardware integration. By introducing an abstraction for different output devices, we maintain one unified codebase and GUI that can drive either the original HUB75 64×64 matrix or the new WS2811 lightbox. The key technical tasks – ensuring proper signal timing (solved by using the Pi's PWM/DMA hardware or an external controller), mapping the 2D animations to the serpentine LED layout, and providing adequate power – have well-established solutions:

- We leverage the rpi_ws281x/NeoPixel library on the Pi to generate the WS2811 signal reliably, or use an ESP32 with WLED as an offload if preferred, which speaks common protocols.

- We implement coordinate mapping in software so that images and animations display correctly on the zig-zag LED arrangement.

- We follow best practices for wiring: common ground, level shifting the data line, and using a proper 5V supply for the LEDs to avoid stressing the Pi.

The result is a flexible system where, with a click in the web interface, you can choose to output an animation on the big LED panel or on the smaller WS2811-powered lightbox. Both modes can show the same kinds of content (albeit scaled to different sizes), and both are controlled from the same Flask web app. This maximizes code reuse and minimizes the effort to add the new hardware.

By addressing the points above – from hardware trade-offs to software design, library choices, and wiring – we have a clear roadmap to implement the WS2811 support. The approach emphasizes **modularity** (separating the device-specific details), **reliability** (using proven libraries and proper electrical practices), and **maintainability** (the unified GUI and code mean we don't fork the project for a new LED type).

With the WS2811 lightbox integrated, the project gains an alternate visual output which could be useful for demos, wearable displays, or installations where the 64×64 panel isn't suitable. Yet, everything remains controlled under one umbrella. This demonstrates how a well-structured Python/Flask LED controller can be extended to new devices with minimal fuss – truly write once, display anywhere!

Sources:

- Adafruit Industries. *NeoPixels on Raspberry Pi (Overview).* Explains using PWM and DMA on Pi for NeoPixel timing, and wiring requirements (level shifting, external power).

- Raspberry Pi LED Matrix Forum. Discussions on driving NeoPixels from Pi vs microcontroller (noting real-time issues on Pi).
- WLED Project Documentation. Details on WLED capabilities for ESP32 (webserver control of WS2811 LEDs, supported protocols for external control).
- J. Lu Blog. *Fun with a Raspberry Pi and WS281x LED Matrix.* Describes handling a serpentine-wired NeoPixel matrix and the need for coordinate mapping.
- The Geek Pub. *Wiring WS2812b to Raspberry Pi.* Provides wiring diagrams with level shifter and power considerations , reinforcing the importance of proper 5V supply and logic level conversion.