MatrixPortal CircuitPython Animated Message Board

Created by Melissa LeBlanc-Williams

https://learn.adafruit.com/matrixportal-circuitpython-animated-message-board

Last updated on 2023-10-11 02:19:30 PM EDT
# Table of Contents

## Overview
- Parts

## 3D Printing
- Slicer Settings
- Parts Details

## Install CircuitPython
- Set up CircuitPython Quick Start!
- Further Information

## Code the Message Board
- Upload the Code, Fonts, Images and Libraries to the MatrixPortal S3
- Code Overview
- Main Classes
- Animation Classes

## Assembly
- MatrixPortal Mount Subassembly
- Connect the Matrices
- Top and Bottom Panel Assembly
- Side Panels and Front LED Acrylic
- Wiring
- Wire Management

## Usage
- Basic Usage
- Animations
- Examples

## bitmaptools Documentation
Overview

Perhaps you've seen those animated message board signs and perhaps you even wanted one. However, you may have noticed they tend to be fairly pricey and programming them can be complicated.

This project makes use of the powerful Adafruit MatrixPortal S3 and its large memory capacity to be able to pump out and blend graphics in real time using CircuitPython. It works by using CircuitPython's bitmaptools module along with displayio to blend, blit and buffer swap images and text together to produce smooth animations.

The code for this project is based on the OpenSign project, which was written for the Raspberry Pi, but the code was completely rewritten from scratch for this project to make use of strengths of CircuitPython. Certain features such as shadow and text stroke were omitted because of the lack of support at this time. Some library restructuring was also done to make it more expandable and avoid OpenSign's large library files for memory efficiency.

This project was featured on one of the livestreams for CircuitPython Day, which you can watch below.

Parts

This project requires the new MatrixPortal S3 for its speed and memory using four 16x32 RGB LED matrices.
Adafruit Matrix Portal S3 CircuitPython Powered Internet Display
Folks love our wide selection of RGB matrices and accessories for making custom colorful LED displays... and our RGB Matrix...
https://www.adafruit.com/product/5778

Medium 16x32 RGB LED matrix panel - 6mm Pitch
Bring a little bit of Times Square into your home with this 16 x 32 RGB LED matrix panel. These panels are normally used to make video walls, here in New York we see them on the sides...
https://www.adafruit.com/product/420

1 x GPIO Ribbon Cable 2x8 IDC Cable - 16 pins 12” long
Needed for reaching the first panel

For the LED acrylic, you can either purchase a 12"x12" sheet and cut it down to a size yourself. It will need to be cut to 3 equal width pieces approximately 101.6mm (4 inches) wide and the lengths of the pieces should add up to 776mm (30.55 inches).

Black LED Diffusion Acrylic Panel 12" x 12" - 0.1" / 2.6mm thick
A nice whoppin' slab of some lovely black acrylic to add some extra diffusion to your LED Matrix project. This material is 2.6mm (0.1") thick and is made of special cast...
https://www.adafruit.com/product/4594

Or you can get 1 large piece from TAP Plastics custom cut to size.
### Optional Parts

To assemble, you will need some metal or nylon 10mm M2.5 screws and washers, M3 screws, and zip ties for wire management.

<table>
<thead>
<tr>
<th>Item</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x M2.5 Nylon Screws</td>
<td><a href="https://www.adafruit.com/product/3299">https://www.adafruit.com/product/3299</a></td>
</tr>
<tr>
<td>Black Nylon Machine Screw and Stand-off Set – M2.5 Thread</td>
<td></td>
</tr>
<tr>
<td>1 x M3 Nylon Screws</td>
<td><a href="https://www.adafruit.com/product/4685">https://www.adafruit.com/product/4685</a></td>
</tr>
<tr>
<td>Black Nylon Machine Screw and Stand-off Set - M3 Thread</td>
<td></td>
</tr>
<tr>
<td>1 x Assorted Zip Ties</td>
<td><a href="https://amzn.to/3OOQmif">https://amzn.to/3OOQmif</a></td>
</tr>
<tr>
<td>1000 Pcs Black Cable Zip Ties Assorted Sizes</td>
<td></td>
</tr>
<tr>
<td>1 x USB A to USB C Adapter</td>
<td><a href="https://www.adafruit.com/product/4175">https://www.adafruit.com/product/4175</a></td>
</tr>
<tr>
<td>Optional if you don't have a USB-C Port on your computer.</td>
<td></td>
</tr>
</tbody>
</table>

### Extra Power Parts

If you need more power than the USB port can provide, you will want these parts.

<table>
<thead>
<tr>
<th>Item</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x 2 in 1 USB-C OTG Splitter</td>
<td><a href="https://amzn.to/3P59nhK">https://amzn.to/3P59nhK</a></td>
</tr>
<tr>
<td>Useful for providing more power than the USB port can supply</td>
<td></td>
</tr>
<tr>
<td>1 x 5V 4 Amp Power Supply</td>
<td><a href="https://www.adafruit.com/product/1466">https://www.adafruit.com/product/1466</a></td>
</tr>
<tr>
<td>5V 4A (4000mA) switching power supply</td>
<td></td>
</tr>
<tr>
<td>1 x 2.1mm 5VDC Barrel Jack to USB-C Adapter</td>
<td><a href="https://www.adafruit.com/product/4536">https://www.adafruit.com/product/4536</a></td>
</tr>
<tr>
<td>For connecting the Power Supply to USB-C</td>
<td></td>
</tr>
<tr>
<td>1 x USB Type A to Type C Cable</td>
<td><a href="https://www.adafruit.com/product/4474">https://www.adafruit.com/product/4474</a></td>
</tr>
<tr>
<td>USB Type A to Type C Cable - approx 1 meter / 3 ft long</td>
<td></td>
</tr>
</tbody>
</table>
This project will be assembled with quite a few 3D printed parts, described below, but many of them are duplicates. The parts are available in either the STL (older) or 3MF (newer) format. Most slicers support both. You'll need to print multiples of some parts:

- 6 x Connector Bracket.stl
- 7 x Top or Bottom Panel.stl
- 1 x Top Panel with Cutout.stl
- 2 x Side Panel.stl
- 1 x MatrixPortal Mount.stl

Slicer Settings

The specific setting values aren't critical with this case. Here are the settings used to print the case for this guide.
For the Connector Bracket and MatrixPortal Mount:

- 20% gyroid infill
- Supports on build plate enabled
- 0.3mm Layer Height

For the other parts:

- 20% gyroid infill
- 0.2mm Layer Height
- No Supports

Parts Details

Connector Bracket
Used to connect the matrices together, has a dovetail for the panels, and a zip tie slot.

Top or Bottom Panel
Dovetail panel for the top or bottom of the message board.
Top Panel with Cutout
Special version of the top panel with cutout holes for the MatrixPortal S3.

Side Panel
Panel for each side of the message board. These attach to the matrix, but also slot into the Top and Bottom Panels.

MatrixPortal Mount
Mount to hold the MatrixPortal S3 so that it lines up with the cutout holes. This attaches on top of the Matrix and one of the connector brackets.

Install CircuitPython
CircuitPython is a derivative of MicroPython designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the CIRCUITPY drive to iterate.
Set up CircuitPython Quick Start!

Follow this quick step-by-step for super-fast Python power :) 

This project requires the Absolute Newest version of CircuitPython.

Download the latest version of CircuitPython for this board via circuitpython.org

Further Information

For more detailed info on installing CircuitPython, check out Installing CircuitPython().

Click the link above and download the latest UF2 file.

Download and save it to your desktop (or wherever is handy).
Plug your MatrixPortal S3 into your computer using a known-good USB cable.

A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Click the Reset button (indicated by the green arrow) on your board. When you see the NeoPixel RGB LED (indicated by the magenta arrow) turn purple, press it again. At that point, the NeoPixel should turn green. If it turns red, check the USB cable, try another USB port, etc.

If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!

You will see a new disk drive appear called MATRXS3BOOT.

Drag the adafruit_circuitpython_etc.uf2 file over to MATRXS3BOOT.

The LED will flash. Then, the MATRXS3BOOT drive will disappear and a new disk drive called CIRCUITPY will appear.

That's it, you're done! :)
Code the Message Board

Once you've finished setting up your MatrixPortal S3 with CircuitPython, you can access the code, fonts, images and necessary libraries by downloading the Project Bundle.

To do this, click on the Download Project Bundle button in the window below. It will download as a zipped folder.

```python
# SPDX-FileCopyrightText: 2023 Melissa LeBlanc-Williams for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time
from adafruit_matrixportal.matrix import Matrix
from messageboard import MessageBoard
from messageboard.fontpool import FontPool
from messageboard.message import Message

matrix = Matrix(width=128, height=16, bit_depth=5)
messageboard = MessageBoard(matrix)
messageboard.set_background("images/background.bmp")
fontpool = FontPool()
fontpool.add_font("arial", "fonts/Arial-10.pcf")

# Create the message ahead of time
message = Message(fontpool.find_font("arial"), mask_color=0xFF00FF, opacity=0.8)
message.add_image("images/maskedstar.bmp")
message.add_text("Hello World!", color=0xFFFF00, x_offset=2, y_offset=2)

while True:
    # Animate the message
    messageboard.animate(message, "Scroll", "in_from_right")
    time.sleep(1)
    messageboard.animate(message, "Scroll", "out_to_left")
```

Upload the Code, Fonts, Images and Libraries to the MatrixPortal S3

After downloading the Project Bundle, plug your MatrixPortal S3 into the computer's USB port. You should see a new flash drive appear in the computer's File Explorer or Finder (depending on your operating system) called CIRCUITPY. Unzip the folder and copy the following items to the MatrixPortal S3's CIRCUITPY drive.

- lib folder
- fonts folder
- images folder
- code.py
- demo.py
Your MatrixPortal S3 CIRCUITPY drive should look like this after copying the fonts, images, and lib folders and the code.py and demo.py files.

![Directory Structure](image)

**Code Overview**

The majority of this code is written as a library. There are a couple of demos, which will be covered in the Usage page, but the overview will go over the library part. The code relies heavily on the CircuitPython `bitmaptools` and `displayio` modules in order to do the graphics work. The library classes can be broken up into a couple of different categories. You can find more information about `displayio` in the [CircuitPython Display Support Using displayio guide](https://circuitpython.org/tutorial/display-support/).
Main Classes

The main classes provide the primary functionality to the message board.

Double Buffering

Double Buffering is handled by the `DoubleBuffer` class. This class works by creating 2 sets of `displayio` Bitmaps, TileGrids, and Groups. It sets 1 of the bitmaps as the "active" buffer, meaning the buffer that is currently being drawn to. Once it is ready to display, the `show()` function is called, which will set the group of the active buffer to be displayed and then swap the buffers. By doing this, it avoids displaying small changes as they are drawn, which can lead to some flickering, and switches everything all at once. Both buffer bitmaps share the same shader, so it is best to keep consistent with the same bit depth of all bitmaps.

```python
# SPDX-FileCopyrightText: 2023 Melissa LeBlanc-Williams for Adafruit Industries
# SPDX-License-Identifier: MIT
import displayio

class DoubleBuffer:
    def __init__(self, display, width, height, shader=None, bit_depth=16):
        self._buffer_group = (displayio.Group(), displayio.Group())
        self._buffer = (displayio.Bitmap(width, height, 2**bit_depth - 1),
                         displayio.Bitmap(width, height, 2**bit_depth - 1),)
        self._x_offset = display.width - width
        self._y_offset = display.height - height
        self.display = display
        self._active_buffer = 0  # The buffer we are updating

        if shader is None:
            shader = displayio.ColorConverter()

        buffer0_sprite = displayio.TileGrid(  
            self._buffer[0],
            pixel_shader=shader,
            x=self._x_offset,
            y=self._y_offset,
        )
        self._buffer_group[0].append(buffer0_sprite)

        buffer1_sprite = displayio.TileGrid(  
            self._buffer[1],
            pixel_shader=shader,
            x=self._x_offset,
            y=self._y_offset,
        )
        self._buffer_group[1].append(buffer1_sprite)

    def show(self, swap=True):
        self.display.root_group = self._buffer_group[self._active_buffer]
        if swap:
            self.swap()
```

©Adafruit Industries
def swap(self):
    self._active_buffer = 0 if self._active_buffer else 1

@property
def active_buffer(self):
    return self._buffer[self._active_buffer]

@property
def shader(self):
    return self._buffer_group[0][0].pixel_shader

@shader.setter
def shader(self, shader):
    self._buffer_group[0][0].pixel_shader = shader
    self._buffer_group[1][0].pixel_shader = shader

Message

A **Message** is simply a class that uses a bitmap which contains one or more labels or images placed onto it. When displayed, it acts as the foreground which can be animated on the sign. The message's bitmap buffer is automatically enlarged as content is added to it so that it only takes up as much space as it needs. The **mask_color** and **opacity** can be set to control how it is blended into the background.

```python
import bitmaptools
import displayio
import adafruit_imageload
from adafruit_display_text import bitmap_label

class Message:
    def __init__(self, font, opacity=1.0, mask_color=0xFF00FF, blendmode=bitmaptools.BlendMode.Normal):
        self._current_font = font
        self._current_color = 0xFF0000
        self._buffer = displayio.Bitmap(0, 0, 65535)
        self._cursor = [0, 0]
        self.opacity = opacity
        self._blendmode = blendmode
        self._mask_color = 0
        self.mask_color = mask_color
        self._width = 0
        self._height = 0

    def _enlarge_buffer(self, width, height):
        """Resize the message buffer to grow as necessary""
        new_width = self._width
        if self._cursor[0] + width >= self._width:
            new_width = self._cursor[0] + width
        new_height = self._height
        if self._cursor[1] + height >= self._height:
            new_height = self._cursor[1] + height
```
new_height = self._cursor[1] + height

if new_width > self._width or new_height > self._height:
    new_buffer = displayio.Bitmap(new_width, new_height, 65535)
    if self._mask_color is not None:
        bitmaptools.fill_region(new_buffer, 0, 0, new_width, new_height, self._mask_color)
    bitmaptools.blit(new_buffer, self._buffer, 0, 0)
    self._buffer = new_buffer
    self._width = new_width
    self._height = new_height

def _add_bitmap(self, bitmap, x_offset=0, y_offset=0):
    new_width, new_height = (self._cursor[0] + bitmap.width + x_offset,
                              self._cursor[1] + bitmap.height + y_offset)
    # Resize the buffer if necessary
    self._enlarge_buffer(new_width, new_height)
    # Blit the image into the buffer
    source_left, source_top = 0, 0
    if self._cursor[0] + x_offset < 0:
        source_left = 0 - (self._cursor[0] + x_offset)
        x_offset = 0
    if self._cursor[1] + y_offset < 0:
        source_top = 0 - (self._cursor[1] + y_offset)
        y_offset = 0
    bitmaptools.blit(self._buffer, bitmap, self._cursor[0] + x_offset,
                     self._cursor[1] + y_offset, x1=source_left, y1=source_top)
    # Move the cursor
    self._cursor[0] += bitmap.width + x_offset

def add_text(self, text, color=None, font=None, x_offset=0, y_offset=0):
    if font is None:
        font = self._current_font
    if color is None:
        color = self._current_color
    color_565value = displayio.ColorConverter().convert(color)
    # Create a bitmap label and add it to the buffer
    bmp_label = bitmap_label.Label(font, text=text)
    color_overlay = displayio.Bitmap(bmp_label.bitmap.width, bmp_label.bitmap.height, 65535)
    color_overlay.fill(color_565value)
    mask_overlay = displayio.Bitmap(bmp_label.bitmap.width, bmp_label.bitmap.height, 65535)
    mask_overlay.fill(self._mask_color)
    bitmaptools.blit(color_overlay, bmp_label.bitmap, 0, 0, skip_source_index=1)
    bitmaptools.blit(color_overlay, mask_overlay, 0, 0, skip_dest_index=color_565value)
    bmp_label = None
    self._add_bitmap(color_overlay, x_offset, y_offset)
def add_image(self, image, x_offset=0, y_offset=0):
    # Load the image with imageload and add it to the buffer
    bmp_image, _ = adafruit_imageload.load(image)
    self._add_bitmap(bmp_image, x_offset, y_offset)

def clear(self):
    """Clear the canvas content, but retain all of the style settings""
    self._buffer = displayio.Bitmap(0, 0, 65535)
    self._cursor = [0, 0]
    self._width = 0
    self._height = 0

@property
def buffer(self):
    """Return the current buffer""
    if self._width == 0 or self._height == 0:
        raise RuntimeError("No content in the message")
    return self._buffer

@property
def mask_color(self):
    """Get or Set the mask color"
    return self._mask_color

@mask_color.setter
def mask_color(self, value):
    self._mask_color = displayio.ColorConverter().convert(value)

@property
def blendmode(self):
    """Get or Set the blendmode"
    return self._blendmode

@blendmode.setter
def blendmode(self, value):
    if value in bitmaptools.BlendMode:
        self._blendmode = value

Font Pool

The FontPool class is a simple font loader and dictionary that holds the loaded fonts so that they don't need to be duplicated for multiple messages.
MessageBoard

The **MessageBoard** class is the main container class and as the name implies, it controls the message board. The message board allows setting what the current background should be and will be used when the animate command is used. The `animate()` function is used to dynamically load and run an animation, which will perform an action on the text until it reaches the end of the function. The `_draw()` function is a callback that is passed into the animation class when it is instantiated and is the function primarily responsible for drawing each frame of the animation.

```python
import bitmaptools
import displayio
import adafruit_imageload
from .doublebuffer import DoubleBuffer
from .message import Message

class MessageBoard:
    def __init__(self, matrix):
        self.fonts = {}
        self.display = matrix.display
        self._buffer_width = self.display.width * 2
        self._buffer_height = self.display.height * 2
        self._dbl_buf = DoubleBuffer(
            self.display, self._buffer_width, self._buffer_height
        )
        self._background = None
        self.set_background()  # Set to black
        self._position = (0, 0)

    def set_background(self, file_or_color=0x000000):
        """The background image to a bitmap file."""
        if isinstance(file_or_color, str):  # its a filenme:
            background, bg_shader = adafruit_imageload.load(file_or_color)
            self._dbl_buf.shader = bg_shader
            self._background = background
        elif isinstance(file_or_color, int):
            # Make a background color fill
            bg_shader = displayio.ColorConverter(
                input_colorspace=displayio.Colorspace.RGB565
            )
            background = displayio.Bitmap(
                self.display.width, self.display.height, 65535
            )
            background.fill(displayio.ColorConverter().convert(file_or_color))
            self._dbl_buf.shader = bg_shader
            self._background = background
        else:
            raise RuntimeError("Unknown type of background")
```

©Adafruit Industries
def animate(self, message, animation_class, animation_function, **kwargs):
    anim_class = __import__(
        f"{self.__module__}.animations.{animation_class.lower()}"
    )
    anim_class = getattr(anim_class, "animations")
    anim_class = getattr(anim_class, animation_class)
    animation = anim_class(
        self.display, self._draw, self._position
    )  # Instantiate the class
    # Call the animation function and pass kwargs along with the message (positional)
    anim_func = getattr(animation, animation_function)
    anim_func(message, **kwargs)

def set_message_position(self, x, y):
    """Set the position of the message on the display""
    self._position = (x, y)

def _draw(
    self,
    image,
    x,
    y,
    opacity=None,
    mask_color=0xFF00FF,
    blendmode=bitmaptools.BlendMode.Normal,
    post_draw_position=None,
):  
    """Draws a message to the buffer taking its current settings into account.
    It also sets the current position and performs a swap."
    self._position = (x, y)
    buffer_x_offset = self._buffer_width - self.display.width
    buffer_y_offset = self._buffer_height - self.display.height

    # Image can be a message in which case its properties will be used
    if isinstance(image, Message):
        if opacity is None:
            opacity = image.opacity
        mask_color = image.mask_color
        blendmode = image.blendmode
        image = image.buffer
    if opacity is None:
        opacity = 1.0
    if mask_color > 65535:
        mask_color = displayio.ColorConverter().convert(mask_color)

    # Blit the background
    bitmaptools.blit(
        self._dbl_buf.active_buffer,
        self._background,
        buffer_x_offset,
        buffer_y_offset,
    )

    # If the image is wider than the display buffer, we need to shrink it
    if x + buffer_x_offset < 0:
        new_image = displayio.Bitmap(
            image.width - self.display.width, image.height, 65535
        )
        bitmaptools.blit(
            new_image,
            image,
            0,
            0,
x1=self.display.width,
y1=0, 
x2=image.width, 
y2=image.height, 
) 
  x += self.display.width 
  image = new_image 

  # If the image is taller than the display buffer, we need to shrink it 
  if y + buffer_y_offset < 0: 
    new_image = displayio.Bitmap( 
      image.width, image.height - self.display.height, 65535 
    ) 
    bitmaptools.blit( 
      new_image, 
      image, 
      0, 
      0, 
      x1=0, 
      y1=self.display.height, 
      x2=image.width, 
      y2=image.height, 
    ) 
    y += self.display.height 
    image = new_image 

  # Clear the foreground buffer 
  foreground_buffer = displayio.Bitmap( 
    self._buffer_width, self._buffer_height, 65535 
  ) 
  foreground_buffer.fill(mask_color) 
  bitmaptools.blit( 
    foreground_buffer, image, x + buffer_x_offset, y + buffer_y_offset 
  ) 

  # Blend the foreground buffer into the main buffer 
  bitmaptools.alphablend( 
    self._dbl_buf.active_buffer, 
    self._dbl_buf.active_buffer, 
    foreground_buffer, 
    displayio.Colorspace.RGB565, 
    1.0, 
    opacity, 
    blendmode=blendmode, 
    skip_source2_index=mask_color, 
  ) 
  self._dbl_buf.show() 

  # Allow for an override of the position after drawing (needed for split 
  effects) 
  if post_draw_position is not None and isinstance(post_draw_position, tuple): 
    self._position = post_draw_position 

Animation Classes

All of the animation classes are built on top of the base Animation class, which 
includes any functions that are shared with the different categories.
Scroll Animations

The **Scroll** animation class contains 8 different main animation functions plus 1 generic animation function, which the main functions call. The 8 functions include a scroll in and scroll out function for each of the 4 directions. Functions beginning with `out_to` cause the content to scroll offscreen and functions beginning with `in_from` cause the content to scroll towards the center of the screen.

The generic **scroll_from_to()** function allows for scrolling from one coordinate to another. It could be used to make text scroll at a diagonal angle.
start_time = time.monotonic()
current_x = start_x + round(i * increment_x)
current_y = start_y + round(i * increment_y)
self._draw(message, current_x, current_y)
if i <= steps:
    self._wait(start_time, duration / steps)

def out_to_left(self, message, duration=1):
    """Scroll a message off the display from its current position towards the
    left
    over a certain period of time.
    :param message: The message to animate.
    :param float duration: (optional) The period of time to perform the
    animation over in seconds. (default=1)
    """
    current_x, current_y = self._position
    self.scroll_from_to(
        message, duration, current_x, current_y, 0 - message.buffer.width,
        current_y)

def in_from_left(self, message, duration=1, x=0):
    """Scroll a message in from the left side of the display over a certain
    period of time. The final position is centered.
    :param message: The message to animate.
    :param float duration: (optional) The period of time to perform the
    animation over in seconds. (default=1)
    :param int x: (optional) The amount of x-offset from the center position
    (default=0)
    """
    center_x, center_y = self._get_centered_position(message)
    self.scroll_from_to(
        message, duration,
        0 - message.buffer.width,
        center_y,
        center_x + x,
        center_y,
    )

def in_from_right(self, message, duration=1, x=0):
    """Scroll a message in from the right side of the display over a certain
    period of time. The final position is centered.
    :param message: The message to animate.
    :param float duration: (optional) The period of time to perform the
    animation over in seconds. (default=1)
    :param int x: (optional) The amount of x-offset from the center position
    (default=0)
    """
    center_x, center_y = self._get_centered_position(message)
    self.scroll_from_to(
        message, duration, self._display.width - 1, center_y, center_x + x,
        center_y,
    )

def in_from_top(self, message, duration=1, y=0):
    """Scroll a message in from the top side of the display over a certain
    period of time. The final position is centered.
    :param message: The message to animate.
    :param float duration: (optional) The period of time to perform the
    animation over in seconds. (default=1)
    :param int y: (optional) The amount of y-offset from the center position
    (default=0)
    """
    center_x, center_y = self._get_centered_position(message)
    self.scroll_from_to(
        message, duration, 1, center_y, center_x + x,
        center_y,
from time import time. The final position is centered.

:param message: The message to animate.
:param float duration: (optional) The period of time to perform the animation over in seconds. (default=1)
:param int y: (optional) The amount of y-offset from the center position (default=0)
:type message: Message

```python
center_x, center_y = self._get_centered_position(message)
self.scroll_from_to(message, duration, center_x, 0 - message.buffer.height, center_x, center_y + y,
)
```

```python
def in_from_bottom(self, message, duration=1, y=0):
    """Scroll a message in from the bottom side of the display over a certain period of time. The final position is centered.

:param message: The message to animate.
:param float duration: (optional) The period of time to perform the animation over in seconds. (default=1)
:param int y: (optional) The amount of y-offset from the center position (default=0)
:type message: Message

```python
center_x, center_y = self._get_centered_position(message)
self.scroll_from_to(message, duration, center_x, self._display.height - 1, center_x, center_y + y,
)
```

```python
def out_to_right(self, message, duration=1):
    """Scroll a message off the display from its current position towards the right over a certain period of time.

:param message: The message to animate.
:param float duration: (optional) The period of time to perform the animation over in seconds. (default=1)
:type message: Message

```python
current_x, current_y = self._position
self.scroll_from_to(message, duration, current_x, current_y, self._display.width - 1, current_y,
)
```

```python
def out_to_top(self, message, duration=1):
    """Scroll a message off the display from its current position towards the top over a certain period of time.

:param message: The message to animate.
:param float duration: (optional) The period of time to perform the animation over in seconds. (default=1)
:type message: Message

```python
current_x, current_y = self._position
self.scroll_from_to(message, duration, current_x, current_y, self._display.width - 1, current_y,
)"""
The Loop animation class contains functions which are similar to the scroll animations, except they cause the content to wrap back in from the opposite side. There are 4 functions for each of the directions.

```
import time
import displayio
import bitmaptools
from . import Animation

class Loop(Animation):
    def _create_loop_image(self, image, x_offset, y_offset, mask_color):
        """Attach a copy of an image by a certain offset so it can be looped."""
        if 0 < x_offset < self._display.width:
            x_offset = self._display.width
        if 0 < y_offset < self._display.height:
            y_offset = self._display.height

        loop_image = displayio.Bitmap(
            image.width + x_offset, image.height + y_offset, 65535
        )
        loop_image.fill(mask_color)
        bitmaptools.blit(loop_image, image, 0, 0)
        bitmaptools.blit(loop_image, image, x_offset, y_offset)

        return loop_image

    def left(self, message, duration=1, count=1):
```
Loop a message towards the left side of the display over a certain period of time by a certain number of times. The message will re-enter from the right and end up back at the starting position.

:param message: The message to animate.
:param float count: (optional) The number of times to loop. (default=1)
:param float duration: (optional) The period of time to perform the animation over. (default=1)
:type message: Message

current_x, current_y = self._position
distance = max(message.buffer.width, self._display.width)
loop_image = self._create_loop_image(
    message.buffer, distance, 0, message.mask_color
)
for _ in range(count):
    for _ in range(distance):
        start_time = time.monotonic()
        current_x -= 1
        if current_x < 0 - message.buffer.width:
            current_x += distance
        self._draw(
            loop_image,
            current_x,
            current_y,
            message.opacity,
        )
        self._wait(start_time, duration / distance / count)

right(self, message, duration=1, count=1):
    """Loop a message towards the right side of the display over a certain period of time by a certain number of times. The message will re-enter from the left and end up back at the starting position.

:param message: The message to animate.
:param float count: (optional) The number of times to loop. (default=1)
:param float duration: (optional) The period of time to perform the animation over. (default=1)
:type message: Message

current_x, current_y = self._position
distance = max(message.buffer.width, self._display.width)
loop_image = self._create_loop_image(
    message.buffer, distance, 0, message.mask_color
)
for _ in range(count):
    for _ in range(distance):
        start_time = time.monotonic()
        current_x += 1
        if current_x > 0:
            current_x -= distance
        self._draw(
            loop_image,
            current_x,
            current_y,
            message.opacity,
        )
        self._wait(start_time, duration / distance / count)

up(self, message, duration=0.5, count=1):
    """Loop a message towards the top side of the display over a certain period of time by a certain number of times. The message will re-enter from the bottom and end..."""
up back a the
starting position.

:param message: The message to animate.
:param float count: (optional) The number of times to loop. (default=1)
:param float duration: (optional) The period of time to perform the
animation
:type message: Message

```python
current_x, current_y = self._position
distance = max(message.buffer.height, self._display.height)
loop_image = self._create_loop_image(
    message.buffer, 0, distance, message.mask_color
)
for _ in range(count):
    for _ in range(distance):
        start_time = time.monotonic()
        current_y += 1
        if current_y > 0:
            current_y -= distance
        self._draw(
            loop_image,
            current_x,
            current_y,
            message.opacity,
        )
        self._wait(start_time, duration / distance / count)
```

def down(self, message, duration=0.5, count=1):
    """Loop a message towards the bottom side of the display over a certain
    period of time by a
certain number of times. The message will re-enter from the top and end up
    back a the
    starting position.

    :param message: The message to animate.
    :param float count: (optional) The number of times to loop. (default=1)
    :param float duration: (optional) The period of time to perform the
animation
    :type message: Message

    ```python
current_x, current_y = self._position
distance = max(message.buffer.height, self._display.height)
loop_image = self._create_loop_image(
    message.buffer, 0, distance, message.mask_color
)
for _ in range(count):
    for _ in range(distance):
        start_time = time.monotonic()
        current_y += 1
        if current_y > 0:
            current_y -= distance
        self._draw(
            loop_image,
            current_x,
            current_y,
            message.opacity,
        )
        self._wait(start_time, duration / distance / count)
    ```
Split Animations

The **Split** animation class contains 4 functions as well. There are out functions and in functions, that cause the content to split out and join in respectively, for both the horizontal and vertical directions.

```python
import time
import displayio
import bitmaptools
from . import Animation

class Split(Animation):
    def out_horizontally(self, message, duration=0.5):
        """Show the effect of a message splitting horizontally over a certain period of time."
        :param message: The message to animate.
        :param float duration: (optional) The period of time to perform the animation over. (default=0.5)
        :type message: Message
        
        current_x, current_y = self._position
        image = message.buffer
        left_image = displayio.Bitmap(image.width // 2, image.height, 65535)
        bitmaptools.blit(
            left_image, image, 0, 0, x1=0, y1=0, x2=image.width // 2, y2=image.height
        )
        right_image = displayio.Bitmap(image.width // 2, image.height, 65535)
        bitmaptools.blit(
            right_image, image, 0, 0, x1=image.width // 2, y1=0, x2=image.width, y2=image.height,
        )

        distance = self._display.width // 2
        for i in range(distance + 1):
            start_time = time.monotonic()
            effect_buffer = displayio.Bitmap(
                self._display.width + image.width, image.height, 65535
            )
            effect_buffer.fill(message.mask_color)
            bitmaptools.blit(effect_buffer, left_image, distance - i, 0)
            bitmaptools.blit(
                effect_buffer, right_image, distance + image.width // 2 + i, 0
            )

            self._draw(
                effect_buffer, current_x - self._display.width // 2, current_y, message.opacity,
            )
```
def out_vertically(self, message, duration=0.5):
    """Show the effect of a message splitting vertically over a certain period of time.

    :param message: The message to animate.
    :param float duration: (optional) The period of time to perform the animation over. (default=0.5)
    :type message: Message
    """
    current_x, current_y = self._position
    image = message.buffer

    top_image = displayio.Bitmap(image.width, image.height // 2, 65535)
    bitmaptools.blit(top_image, image, 0, 0, x1=0, y1=0, x2=image.width, y2=image.height // 2)

    bottom_image = displayio.Bitmap(image.width, image.height // 2, 65535)
    bitmaptools.blit(bottom_image, image, 0, 0, x1=0, y1=image.height // 2, x2=image.width, y2=image.height,

    distance = self._display.height // 2
    effect_buffer_width = self._display.width
    if current_x < 0:
        effect_buffer_width -= current_x
    for i in range(distance + 1):
        start_time = time.monotonic()
        effect_buffer = displayio.Bitmap(
            effect_buffer_width, self._display.height + image.height, 65535
        )
        effect_buffer.fill(message.mask_color)
        bitmaptools.blit(effect_buffer, top_image, 0, distance - i)
        bitmaptools.blit(effect_buffer, bottom_image, 0, distance + image.height // 2 + i + 1)

        self._draw(
            effect_buffer,
            current_x,
            current_y - self._display.height // 2,
            message.opacity,
            post_draw_position=(current_x, current_y - self._display.height // 2),
        )
        self._wait(start_time, duration / distance)

def in_horizontally(self, message, duration=0.5):
    """Show the effect of a split message joining horizontally over a certain period of time.

    :param message: The message to animate.
    :param float duration: (optional) The period of time to perform the animation over. (default=0.5)
    :type message: Message
    """
current_x = int(self._display.width // 2 - message.buffer.width / 2)
current_y = int(self._display.height // 2 - message.buffer.height / 2)
image = message.buffer
left_image = displayio.Bitmap(image.width // 2, image.height, 65535)
bitmaptools.blit(
    left_image, image, 0, 0, x1=0, y1=0, x2=image.width // 2,
y2=image.height
)

right_image = displayio.Bitmap(image.width // 2, image.height, 65535)
bitmaptools.blit(
    right_image, image, 0, 0, x1=image.width // 2,
y1=0, x2=image.width, y2=image.height,
)

distance = self._display.width // 2
effect_buffer = displayio.Bitmap(
    self._display.width + image.width, image.height, 65535
)
effect_buffer.fill(message.mask_color)
for i in range(distance + 1):
    start_time = time.monotonic()
    bitmaptools.blit(effect_buffer, left_image, i, 0)
    bitmaptools.blit(
        effect_buffer, right_image,
        self._display.width + image.width // 2 - i + 1,
        0,
    )
    self._draw(
        effect_buffer,
        current_x = self._display.width // 2,
        current_y,
        message.opacity,
        post_draw_position=(current_x, current_y),
    )
    self._wait(start_time, duration / distance)

def in_vertically(self, message, duration=0.5):
    """Show the effect of a split message joining vertically
    over a certain period of time.
    
    :param message: The message to animate.
    :param float duration: (optional) The period of time to perform the
    animation over. (default=0.5)
    
    :type message: Message
    """
current_x = int(self._display.width / 2 - message.buffer.width / 2)
current_y = int(self._display.height / 2 - message.buffer.height / 2)
image = message.buffer
top_image = displayio.Bitmap(image.width, image.height // 2, 65535)
bitmaptools.blit(
    top_image, image, 0, 0, x1=0, y1=0, x2=image.width, y2=image.height // 2
)
bottom_image = displayio.Bitmap(image.width, image.height // 2, 65535)
bitmaptools.blit(
    bottom_image, image, 0,
The Static animation class is more of a miscellaneous collection of functions that don't provide any motion. It includes functions to show, hide, blink, flash, fade_in, and fade_out content. The static class will always display the message at the position of the last message to be displayed and default to (0,0).

```python
import time
from . import Animation

class Static(Animation):
    def show(self, message):
        """Show the message at its current position.

        :param message: The message to show.
        :type message: Message
        """
        x, y = self._position
        self._draw(message, x, y)

    def hide(self, message):
        """Hide the message at its current position.

        :param message: The message to hide.
        :type message: Message
        """
        x, y = self._position
```
def blink(self, message, count=3, duration=1):
    """Blink the foreground on and off a certain number of times over a certain period of time.

    :param message: The message to animate.
    :param float count: (optional) The number of times to blink. (default=3)
    :param float duration: (optional) The period of time to perform the animation
                          over. (default=1)
    :type message: Message
    ""
    delay = duration / count / 2
    for _ in range(count):
        start_time = time.monotonic()
        self.hide(message)
        start_time = self._wait(start_time, delay)
        self.show(message)
        self._wait(start_time, delay)

def flash(self, message, count=3, duration=1):
    """Fade the foreground in and out a certain number of times over a certain period of time.

    :param message: The message to animate.
    :param float count: (optional) The number of times to flash. (default=3)
    :param float duration: (optional) The period of time to perform the animation
                           over. (default=1)
    :type message: Message
    ""
    delay = duration / count / 2
    steps = 50 // count
    for _ in range(count):
        self.fade_out(message, duration=delay, steps=steps)
        self.fade_in(message, duration=delay, steps=steps)

def fade_in(self, message, duration=1, steps=50):
    """Fade the foreground in over a certain period of time by a certain number of steps. More steps is smoother, but too high of a number may slow down the animation too much.

    :param message: The message to animate.
    :param float duration: (optional) The period of time to perform the animation
                           over. (default=1)
    :param float steps: (optional) The number of steps to perform the animation. (default=50)
    :type message: Message
    ""
    current_x = int(self._display.width / 2 - message.buffer.width / 2)
    current_y = int(self._display.height / 2 - message.buffer.height / 2)
    delay = duration / (steps + 1)
    for opacity in range(steps + 1):
        start_time = time.monotonic()
        self._draw(message, current_x, current_y, opacity=opacity / steps)
        self._wait(start_time, delay)

def fade_out(self, message, duration=1, steps=50):
    """Fade the foreground out over a certain period of time by a certain number of steps. More steps is smoother, but too high of a number may slow down the animation too much.

    :param message: The message to animate.
    :param float duration: (optional) The period of time to perform the animation
                           over. (default=1)
    :param float steps: (optional) The number of steps to perform the animation
                           over. (default=50)
    :type message: Message
    """
Assembly

Assembly of the matrixPortal is fairly straightforward and the only tools you will need are a screwdriver or two and some flush cutters. You may wish to clean up the 3D prints prior to starting assembly.
For mounting the MatrixPortal S3, you will need:

- MatrixPortal S3
- MatrixPortal Mount
- 2 x M2.5 x 10mm Screws
- 2 x M2.5 Hex Nut

Insert the MatrixPortal so that the 2x10 IDC connector goes into the rectangle. It should be inserted from the side with the raised arms so that the holes line up. Insert the M2.5 screws through the MatrixPortal and out the holes in the mount.
Attach the M2.5 Hex Nuts and tighten snugly.

Connect the Matrices

For this part, start with four 16x32 matrices.
You will want to lay them out side-by-side so that the input connector is to the right and the text appears upside down.

For connecting the matrices, you will need:

6 x Connector Bracket
10 x M3 x 10mm Screws
Starting from the left, place a connector so that it sits on the alignment peg and is aligned to the screw holes. Install 2 screws to hold it in place. The screws should be snug, but not overtightened.

Do the same for the other left connector bracket and both middle connector brackets.
For the two right connector brackets, only install a screw in the right hole. The holes without screws will be used for the MatrixPortal S3 Mount.

Now you should have all 6 connector brackets attached with the exception of the right two brackets missing their left screw.

To connect the MatrixPortal Mount Subassembly, two of the screws can be anywhere from 10mm to 16mm. You will need the following:

MatrixPortal Mount Subassembly
2 x M3 x 10mm Screws
2 x M3 x 10mm-16mm Screws
Place the MatrixPortal Mount Subassembly so that the 2 raised arms go into the holes without the screws.

Install the shorter screws into the part of the subassembly that covers the center holes of the matrix.

Install the longer screws into the holes above the connector bracket.
Top and Bottom Panel Assembly

Start with the panel with the cutout holes. You will also need 7 panels without holes further down.

With the front side of the matrices facing towards you, place the panel so that it's almost in its final place and slide the right side onto the connector bracket. You may need to flex the panel a bit to get it past the buttons.
Continue sliding the panel to the right until the panel's left side is able to clear the connector bracket underneath it and slide it onto its respective bracket.

Go ahead and center it so that the holes all line up.
The regular panels are easier to slide on from one end or the other until they sit flush in the middle.

When sliding them on, be sure that the lip of the panel faces the same direction as the one with cutout holes.

Continue sliding on the remainder of the panels until all 8 are on the message board.
Side Panels and Front LED Acrylic

To install the sides, you will need the following:

2 x Side Panels
4 x 10mm Screws
LED Acrylic Piece or Pieces

Start by connecting one side. To get it past the alignment peg, you may need to flex parts slightly. Install 2 screws once it is in place.
Slide any acrylic pieces you have into the message board from the side without a panel. The acrylic should be installed with the shiny side facing towards the matrices and the matte side facing towards the outside.

Install the second side panel. It's easiest to get it over the alignment peg first and then get the acrylic into the slot. Install 2 more screws.
Wiring

For the wiring, you will be using 3 of the shorter ribbon cables and 2 of the power cables that came with the matrices.

Start by connecting the 4 panels together using the 3 shorter ribbon cables.
When connecting the wires to the MatrixPortal S3, place the flatter sides of the spade connectors together and insert them into the 5V screw terminal. Make sure the screw is fairly tight.

Connect the ground wires to the other terminal.
Connect the 4 power plugs to the power inlet of each panel. You may have to try a few different ends until you find one that reaches the furthest plug well enough.
Connect the longer black ribbon cable between the MatrixPortal and the input of the right-most panel.

Wire Management

The connector brackets have built-in zip tie holders designed to fit zip with a width of about 3mm or smaller.
When zipping down wires using the connectors, leave the zip tie loose for now. You will be feeding a larger zip tie between the smaller ones.

Place a second zip tie across from the first, even if you aren't holding any wires in place.
You will want to carefully fold the cable back and forth until it sits neatly between the 2 connectors.

Take a larger zip tie (8 inches or longer) and loosely connect it between the smaller zip ties so that it looks a bit like a chain.
Tighten the smaller zip ties first.

Then tighten the larger zip tie, which will hold the ribbon cable in place.

Clip off the ends of the zip ties with flush cutters.

Do this for the other sets of connectors.
You may want to carefully fold and secure the ribbon cable around the MatrixPortal so that it doesn't stick out the back.

You can also use the arms of the MatrixPortal Mount for securing wires.

That's it! You should now be done.

Usage

Basic Usage

To create a message and display it, you can use the following minimal example:
import time
from adafruit_matrixportal.matrix import Matrix
from messageboard import MessageBoard
from messageboard.fontpool import FontPool
from messageboard.message import Message

matrix = Matrix(width=128, height=16, bit_depth=5)
messageboard = MessageBoard(matrix)
fontpool = FontPool()

message = Message(fontpool.find_font("terminal"))
message.add_text("Hello World")
messageboard.animate(message, "Static", "show")
time.sleep(5)

You will need to import Matrix, MessageBoard, FontPool, and Message. Matrix uses the MatrixPortal library. See the Creating MatrixPortal Projects with CircuitPython guide() for more details on that library. The matrix object is the only thing needed to pass in to create a MessageBoard object.

When FontPool is initialized, the terminal font is automatically added in, so that's available without doing anything further.

Finally, a message is created using the terminal font. The text "Hello World" is added, and the show animation is called. The message displays for 5 seconds before finishing the code.

Animations

There are quite a few animations available. To call an animation function, the animate function of the messageboard object is called, along with the Class name and function name of the animation. For instance, looking at the basic usage, the show function of the Static class is called. Here is a list of animations included:

Static Animations

Static animations keep a message in its last position, which is why this group is called static. These are used to show or hide text in various ways. If you would like to set the position manually, you can call messageboard.set_message_position(x, y) before animating the text.
Call `messageboard.animate(message, "Static", "show")` to show a message at its current position instantly.

Call `messageboard.animate(message, "Static", "hide")` to hide a message instantly.

Call `messageboard.animate(message, "Static", "blink", count=3, duration=1)` to blink a message on and off a certain number of times expressed by count (default of 3) over the duration time (default of 1 second).
Call `messageboard.animate(message, "Static", "fade_in", duration=1)` to fade a message in over the duration time (default of 1 second).

Call `messageboard.animate(message, "Static", "fade_out", duration=1)` to fade a message in over the duration time (default of 1 second).

Call `messageboard.animate(message, "Static", "flash", count=3, duration=1)` to fade a message in and out a certain number of times expressed by count (default of 3) over the duration time (default of 1 second).

Scroll Animations

Scroll animations scroll the text into or out of frame.
Call `messageboard.animate(message, "Scroll", "in_from_left")` to scroll a message into frame from the left.

Call `messageboard.animate(message, "Scroll", "in_from_right")` to scroll a message into frame from the right.

Call `messageboard.animate(message, "Scroll", "in_from_top")` to scroll a message into frame from the top.
Call `messageboard.animate(message, "Scroll", "in_from_bottom")` to scroll a message into frame from the bottom.

Call `messageboard.animate(message, "Scroll", "out_to_left")` to scroll a message out of frame to the left.

Call `messageboard.animate(message, "Scroll", "out_to_right")` to scroll a message out of frame to the right.
Call `messageboard.animate(message, "Scroll", "out_to_top")` to scroll a message out of frame towards the top.

Call `messageboard.animate(message, "Scroll", "out_to_from")` to scroll a message out of frame towards the bottom.

Loop Animations

Similar to the scroll animations, but the message wraps around to the other side of the message board.

Call `messageboard.animate(message, "Loop", "left")` to loop a message out of frame to the left and back in from the right.
Call `messageboard.animate(message, "Loop", "right")` to loop a message out of frame to the right and back in from the left.

Call `messageboard.animate(message, "Loop", "up")` to loop a message out of frame to the top and back in from the bottom.

Call `messageboard.animate(message, "Loop", "down")` to loop a message out of frame to the bottom and back in from the top.

**Split Animations**

These animations are for splitting up or joining together the message either horizontally or vertically.
Call `messageboard.animate(message, "Split", "out_horizontally")` to split a message out of frame horizontally.

Call `messageboard.animate(message, "Split", "in_horizontally")` to join a message into frame horizontally.

Call `messageboard.animate(message, "Split", "out_vertically")` to split a message out of frame vertically.
Call `messageboard.animate(message, "Split", "in_vertically")` to join a message into frame vertically.

Examples

There are a couple of examples that demonstrate how to use the included messageboard library.

code.py

The one included in code.py is only slightly more complicated than the basic usage code. It also loads the Arial font, a background, and an image.

```python
import time
def import_time:
    from adafruit_matrixportal.matrix import Matrix
    from messageboard import MessageBoard
    from messageboard.fontpool import FontPool
    from messageboard.message import Message

    matrix = Matrix(width=128, height=16, bit_depth=5)
    messageboard = MessageBoard(matrix)
    messageboard.set_background("images/background.bmp")
    fontpool = FontPool()
    fontpool.add_font("arial", "fonts/Arial-10.pcf")

    # Create the message ahead of time
    message = Message(fontpool.find_font("arial"), mask_color=0xFF00FF, opacity=0.8)
    message.add_image("images/maskedstar.bmp")
    message.add_text("Hello World!", color=0xFFFF00, x_offset=2, y_offset=2)

    for the animation, it resides in the main program loop and scrolls the message both into and out of the frame.

    while True:
        # Animate the message
        messageboard.animate(message, "Scroll", "in_from_right")
        time.sleep(1)
        messageboard.animate(message, "Scroll", "out_to_left")
```

Here is that example code in its entirety:
demo.py

The demo.py example is a little more complex and shows some of the capabilities using four different messages. Like the previous example, it starts off with the same imports and sets up the `messageboard` with a background image.

```python
import time
from adafruit_matrixportal.matrix import Matrix
from messageboard import MessageBoard
from messageboard.fontpool import FontPool
from messageboard.message import Message

matrix = Matrix(width=128, height=16, bit_depth=5)
messageboard = MessageBoard(matrix)
messageboard.set_background("images/background.bmp")
fontpool = FontPool()
fontpool.add_font("arial", "fonts/Arial-10.pcf")

# Create the message ahead of time
message = Message(fontpool.find_font("arial"), mask_color=0xFF00FF, opacity=0.8)
message.add_image("images/maskedstar.bmp")
message.add_text("Hello World!", color=0xFFFF00, x_offset=2, y_offset=2)

while True:
    # Animate the message
    messageboard.animate(message, "Scroll", "in_from_right")
    time.sleep(1)
    messageboard.animate(message, "Scroll", "out_to_left")
```

After that, a `FontPool` object is created and two fonts are added to it, which will be reused.

```python
fontpool = FontPool()
fontpool.add_font("comic", "fonts/Comic-10.pcf")
fontpool.add_font("dejavu", "fonts/DejaVuSans-10.pcf")
```

Next, the messages are set up. Note that the first message doesn't have any content added to it at this time because the intent is to change it inside the main loop.

```python
message1 = Message(fontpool.find_font("dejavu"))
message2 = Message(fontpool.find_font("comic"), mask_color=0x00FF00)
print("add blinka")
```
In the main loop, message1 has content added and is animated. To demonstrate how the messages can be dynamically changed, this is show three more times. This would be useful for updating the message with the current time or from data on the internet.

Here is the animation of the remaining messages. For message2, use of an alternate mask color is demonstrated and for message3, the background is changed to green prior to displaying the message and changed back to the image after.

Here is that example code in its entirety:
import time
from adafruit_matrixportal.matrix import Matrix
from messageboard import MessageBoard
from messageboard.fontpool import FontPool
from messageboard.message import Message

matrix = Matrix(width=128, height=16, bit_depth=5)
messageboard = MessageBoard(matrix)
messageboard.set_background("images/background.bmp")

fontpool = FontPool()
fontpool.add_font("comic", "fonts/Comic-10.pcf")
fontpool.add_font("dejavu", "fonts/DejaVuSans-10.pcf")

message1 = Message(fontpool.find_font("dejavu"))
message2 = Message(fontpool.find_font("comic"), mask_color=0x00FF00)
print("add blinka")
message2.add_image("images/maskedblinka.bmp")
print("add text")
message2.add_text("CircuitPython", color=0xFFFF00, y_offset=-2)

message3 = Message(fontpool.find_font("dejavu"))
message3.add_text("circuitpython.com", color=0xFF0000)

message4 = Message(fontpool.find_font("arial"))
message4.add_text("Buy Electronics", color=0xFFFF00)

while True:
    # Set message 1 content and animate
    message1.clear()
    message1.add_text("Scroll Text In", color=0xFF0000)
    messageboard.animate(message1, "Scroll", "in_from_left")
    time.sleep(1)

    # Change message 1 content and animate
    message1.clear()
    message1.add_text("Change Messages")
    messageboard.animate(message1, "Static", "show")
    time.sleep(1)

    # Change message 1 content again and animate
    message1.clear()
    message1.add_text("And Scroll Out")
    messageboard.animate(message1, "Static", "show")
    messageboard.animate(message1, "Scroll", "out_to_right")
    time.sleep(1)

    # Change message 1 content a final time and animate
    message1.clear()
    message1.add_text("Or more effects like looping ", color=0xFFFF00)
    messageboard.animate(message1, "Split", "in_vertically")
    messageboard.animate(message1, "Loop", "left")
    messageboard.animate(message1, "Static", "flash", count=3)
    messageboard.animate(message1, "Split", "out_vertically")
    time.sleep(1)

    messageboard.animate(message2, "Static", "fade_in")
    time.sleep(1)
    messageboard.animate(message2, "Static", "fade_out")
    messageboard.set_background(0x00FF00)
    messageboard.animate(message3, "Scroll", "in_from_top")
    time.sleep(1)
    messageboard.animate(message3, "Scroll", "out_to_bottom")
    messageboard.set_background("images/background.bmp")
messageboard.animate(message4, "Scroll", "in_from_right")
time.sleep(1)