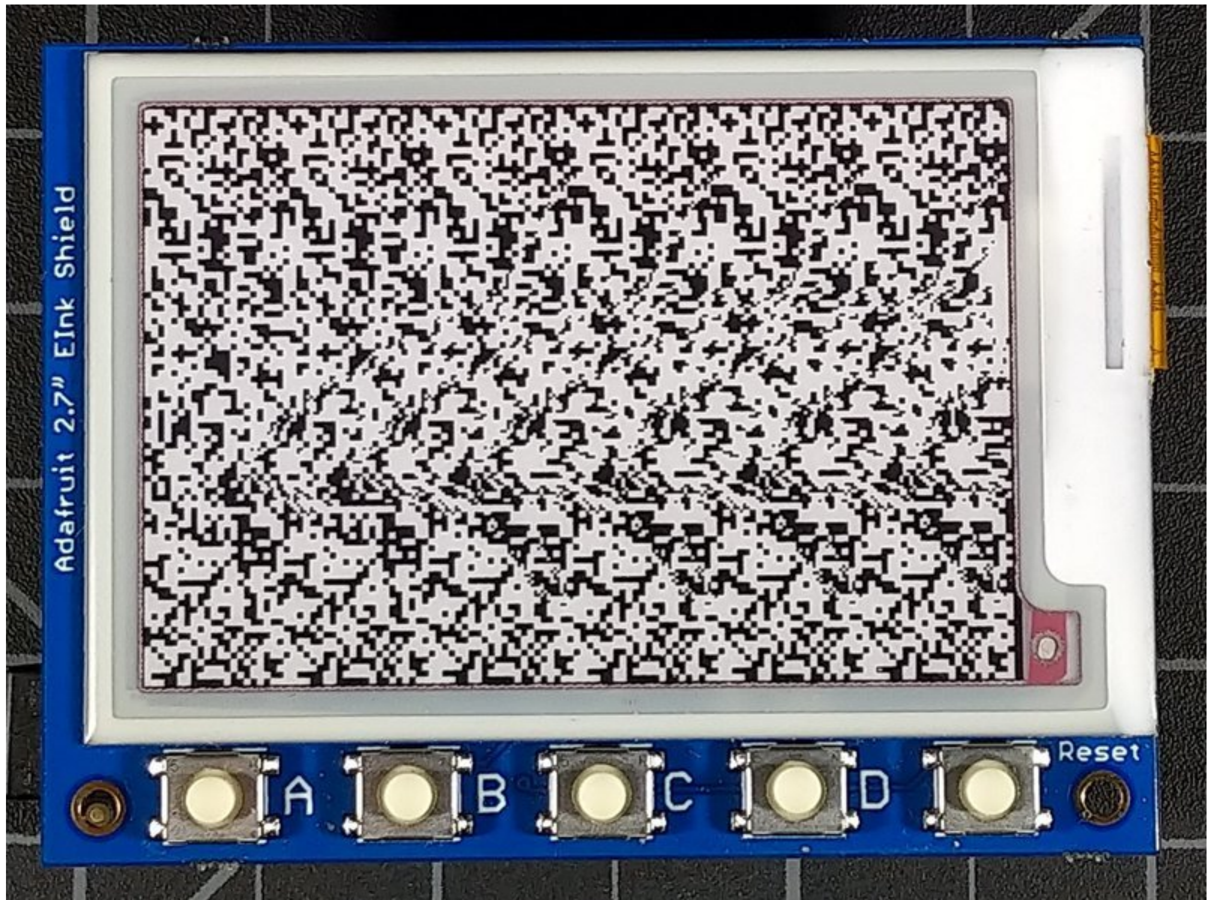
Tiny Autostereogram Construction Kit

Created by Dan Cogliano



https://learn.adafruit.com/tiny-autostereogram-construction-kit

Last updated on 2024-03-08 03:32:41 PM EST

Table of Contents

Overview 3

- Parts
- Parts

How to View an Autostereogram 5

How It Works 6

Installing the Code 6

Creating an Autostereogram 13

- Job Files
- The config.json File
- Creating an Autostereogram
- Slideshow Mode
- Finding Images to Convert to Autostereograms

Overview

There was a buzz created in the 1990s when the series of "Magic Eye" books were first published. These books displayed 3D images on plain paper without the need of any special head gear like red-green, Polaroid or LCD shutter glasses. These 3D images could be seen with the naked eye. The secret was the use of a horizontal repeating series of dots with a slight variation in the repeating series to give the illusion of depth in the 2D image. These images have been called a few names, including autostereograms, single image random dot stereograms (SIRDS), and single image stereograms (SIS). We will call them autostereograms in this learning guide.

With the Tiny Autostereogram Construction Kit (or TACK for short), you can create your own autostereograms using CircuitPython and an Adafruit Metro M4 Express and an Adafruit ePaper shield. It creates these 3D images from black and white or grayscale BMP images and display them on an Adafruit ePaper shield. You can even save them to a file and post them on social media. They are small enough be displayed on a smart phone and still retain the 3D effect. This project will require you to create and copy files onto the Metro M4 Express device. You will also need the help of a paint program that can resize images and save them to either 2 color or grayscale BMP files.

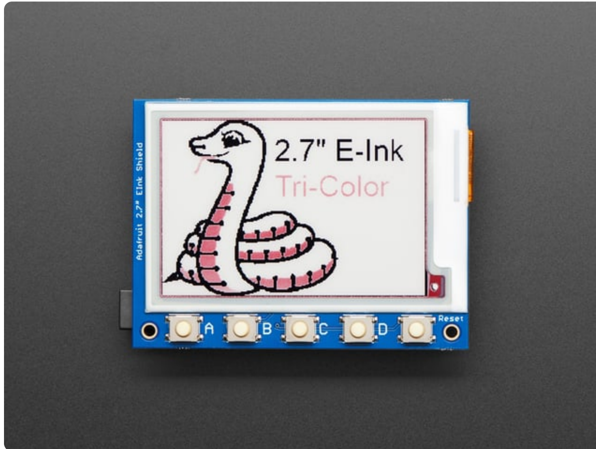
Parts

Building this project requires no soldering and uses just two parts: the Adafruit Metro M4 Express and the Adafruit 2.7" Tri-Color eInk / ePaper Shield with SRAM. Either the Metro M4 or Metro M4 Airlift Express can be used.

If you are interested in ePaper displays for other projects, check out the entire line of Adafruit's ePaper displays (https://adafru.it/ExU).

If you like this no solder project, you may also like the ePaper Maze Maker (https://adafru.it/FBL), the eInk / ePaper Weather Station (https://adafru.it/FBM), and the ePaper Calendar (https://adafru.it/FBN) projects, which use the same components as this project: the ePaper shield and either the Metro M4 Express or Metro M4 Express Airlift light board. Since this project does not require a WiFi connection either Metro M4 Express board will work.

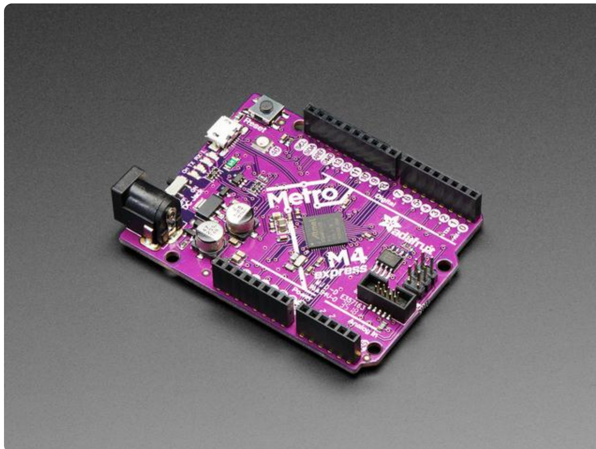
Parts



[Adafruit 2.7" Tri-Color eInk / ePaper Shield with SRAM](https://www.adafruit.com/product/4229)

Easy e-paper finally comes to microcontrollers, with this breakout that's designed to make it a breeze to add a tri-color eInk display. Chances are you've seen one of those...

https://www.adafruit.com/product/4229



[Adafruit Metro M4 feat. Microchip ATSAMD51](https://www.adafruit.com/product/3382)

Are you ready? Really ready? Cause here comes the fastest, most powerful Metro ever. The Adafruit Metro M4 featuring the Microchip ATSAMD51. This...

https://www.adafruit.com/product/3382

[Adafruit Metro M4 Express AirLift (WiFi) - Lite](https://www.adafruit.com/product/4000)

Give your next project a lift with AirLift - our witty name for the ESP32 co-processor that graces this Metro M4. You already know about the Adafruit Metro...

https://www.adafruit.com/product/4000

[USB cable - USB A to Micro-B](https://www.adafruit.com/product/592)

This here is your standard A to micro-B USB cable, for USB 1.1 or 2.0. Perfect for connecting a PC to your Metro, Feather, Raspberry Pi or other dev-board or...

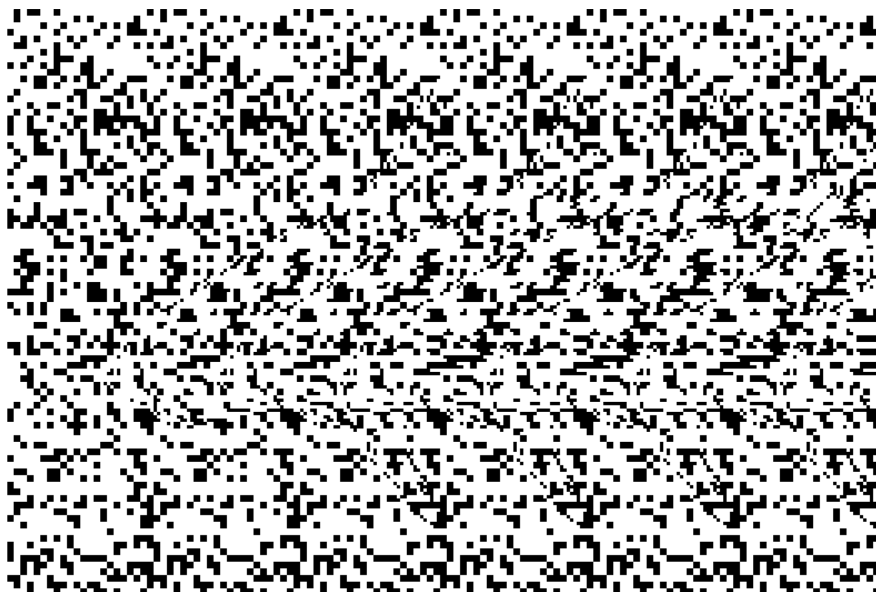
https://www.adafruit.com/product/592

How to View an Autostereogram

Viewing an autostereogram can be difficult at first and some people simply can not see the 3D effect. However, once you can see your first image then other 3D images will be easier to see.

The key is to focus your eyes beyond the screen. One technique is to use a flashlight (or flashlight app) and shine it on the screen displaying the autostereogram. Focus on the light beyond the plane of the display instead of the display itself. Slowly move the light towards the screen and hopefully your eyes will be converge to a plane beyond the screen and see the 3D effect.

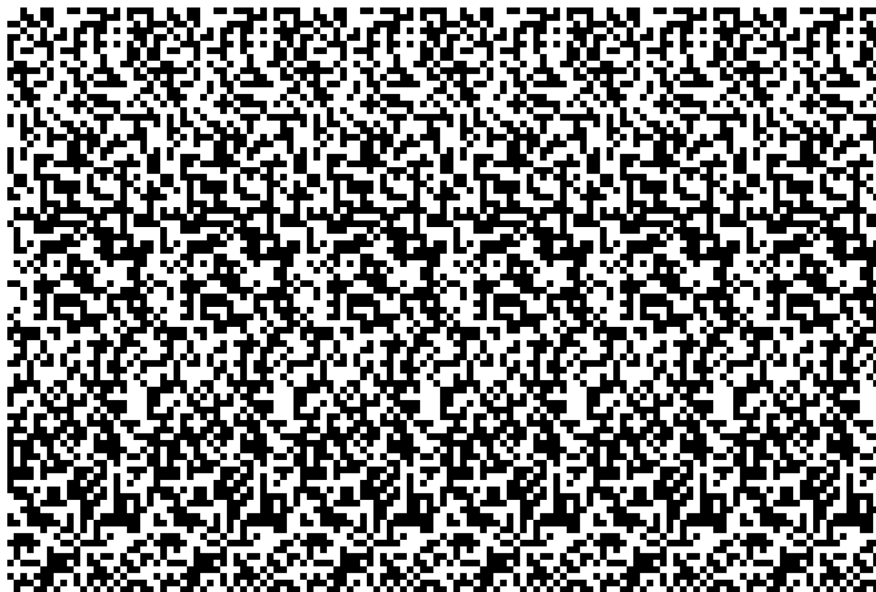
Here is an image to try out that was created from the TACK.



How It Works

Autostereograms trick the eye by using a repeating horizontal series of random dots. Because of the repeating nature of the autostereogram, you can not only focus on the dots directly but also on one of the other repeating series. What makes it cool is that the series of random dots can be shifted slightly to simulate depth. This is similar to how a ViewMaster and other stereo photo viewers work, where a slight difference in images can simulate an object as being closer or further away. A computer is required to generate an autostereogram since each dot must be calculated. With our ePaper display, that means there are over 46,000 dots, or pixels that must be calculated.

The animation below shows how the pixels move when creating an autostereogram of a square. The repeating series is shifted to make it appear the square is in front of the background. It is also interesting to note that although the moving pixels are on the right side of the image, the 3D image is centered since the stationary pixels are combined with the moved pixels to create the 3D image.



You can find a more detailed explanation on how autostereograms work and their history from this Wikipedia page (https://adafru.it/FBO).

Installing the Code

Download the project files from the GitHub repo (https://adafru.it/FBP) linked to the green button below.

Project files on GitHub

https://adafru.it/FBP

In any of the code boxes below you can get all the files in one step by clicking "Download: Zip".

In addition to the **code.py** file, this project also requires several supporting folders and files. Create these folders, if you do not have them:

- **asgfiles**
- **backgrounds**
- **images**
- **jobs**

You will also need to copy the background files, sample jobs and images, and the **config.json** and **boot.py** files from the Github repository.

The **config.json** file is used to define which jobs to run and folders to use. Here is the **config.json** that comes with the project:

```
{
    "jobs": ["shark.json","adafruit.json"]
    "asgfolder": "/asgfiles"
    "jobfolder": "/jobs"
    "bkfolder": "/backgrounds"
    "imagefolder": "/images"
    "slidefolder": "/asgfiles"
}
```

The **boot.py** file allows the project to create autostereogram BMP files and write to the CircuitPython file system by pressing a button at boot time. Normally, CircuitPython can not write to the file system since the host operating system has that access by default. Using this boot file and pressing a button during the boot process reverses this access, giving CircuitPython read/write access and read only access to the host operating system. Rebooting without a button press reverts back CircuitPython's access to read only. You can see which boot mode you are currently using by looking at the boot_out file on the file system. When you press a button during the boot process you will be able to create BMP files and the boot_out file will include this line:

> CircuitPython has write access to drive

Here is the **boot.py** code:

```
# SPDX-FileCopyrightText: 2019 Anne Barela for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import board
import storage
```

```
from analogio import AnalogIn

def read_buttons():
    with AnalogIn(board.A3) as ain:
        reading = ain.value / 65535
        if reading > 0.75:
            return None
        if reading > 0.4:
            return 4
        if reading > 0.25:
            return 3
        if reading > 0.13:
            return 2
        return 1

readonly = True
# if a button is pressed while booting up, CircuitPython can write to the drive
button = read_buttons()
if button != None:
    readonly = False
if readonly:
    print("OS has write access to CircuitPython drive")
else:
    print("CircuitPython has write access to drive")
storage.remount("/", readonly=readonly)
```

The code.py contains the code to create the autostereograms as well as display them on the ePaper shield and save them to a BMP file.

```
# SPDX-FileCopyrightText: 2019 Mike Cogliano for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import os
import time
import json
import digitalio
import busio
import board
import displayio
import adafruit_imageload
from analogio import AnalogIn
from adafruit_epd.epd import Adafruit_EPD
from adafruit_epd.il91874 import Adafruit_IL91874

spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)
ecs = digitalio.DigitalInOut(board.D10)
dc = digitalio.DigitalInOut(board.D9)
srcs = digitalio.DigitalInOut(board.D8)  # can be None to use internal memory
led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

print("Creating display")
display = Adafruit_IL91874(
    176,
    264,
    spi,  # 2.7" Tri-color display
    cs_pin=ecs,
    dc_pin=dc,
    sramcs_pin=srcs,
    rst_pin=None,
    busy_pin=None,
)

# read buttons from ePaper shield
def read_buttons():
```

```
    btn = 1
    with AnalogIn(board.A3) as ain:
        reading = ain.value / 65535
        if reading > 0.75:
            btn = None
        if reading > 0.4:
            btn = 4
        if reading > 0.25:
            btn = 3
        if reading > 0.13:
            btn = 2
        btn = 1
    return btn

#pylint: disable-msg=too-many-nested-blocks
#pylint: disable-msg=too-many-branches
#pylint: disable-msg=too-many-statements
# display bitmap file
def display_bitmap(epd, filename):
    try:
        f = open("/" + filename, "rb")
        print("File opened")
        if f.read(2) != b"BM":  # check signature
            raise BMPError("Not BitMap file")
        read_le(f.read(4)) # File size
        f.read(4)  # Read & ignore creator bytes
        bmpImageoffset = read_le(f.read(4))  # Start of image data
        headerSize = read_le(f.read(4))
        bmpWidth = read_le(f.read(4))
        # convert width to 8 pixels per byte width
        bmpWidth = (bmpWidth + 7) // 8
        bmpHeight = read_le(f.read(4))
        # convert unsigned int to signed int in case there is a negative height
        if bmpHeight > 0x7FFFFFFF:
            bmpHeight = bmpHeight - 4294967296
        flip = True
        if bmpHeight < 0:
            bmpHeight = abs(bmpHeight)
            flip = False
        print(
            "Image offset: %d\nHeader size: %d"
            % (bmpImageoffset, headerSize)
        )
        print("Width: %d\nHeight: %d" % (bmpWidth, bmpHeight))
        if read_le(f.read(2)) != 1:
            raise BMPError("Not singleplane")
        if read_le(f.read(2)) != 1: # bits per pixel
            raise BMPError("Not 1-bit")
        if read_le(f.read(4)) != 0:
            raise BMPError("Compressed file not supported")
        read_le(4)  # SizeImage
        read_le(4)  # biXPelsPerMeter
        read_le(4)  # biYPelsPerMeter
        read_le(4)  # biClrUsed
        read_le(4)  # biClrImportant
        blkpixel = 1
        if read_le(4) != 0:
            blkpixel = 0
        print("black pixel is ", blkpixel)
        print("Image OK! Drawing...")
        rowSize = (bmpWidth + 3) & ~3  # 32-bit line boundary
        for row in range(bmpHeight):  # For each scanline...
            # blink the LED
            if row % 2 == 0:
                led.value = False
            else:
                led.value = True
            if flip:  # Bitmap is stored bottom-to-top order (normal BMP)
                f.seek(bmpImageoffset + (bmpHeight - 1 - row) * rowSize)
```

```
            else:  # Bitmap is stored top-to-bottom
                f.seek(bmpImageoffset + row * rowSize)

            rowdata = f.read(bmpWidth)
            for col in range(bmpWidth):
                for b in range(8):
                    if (rowdata[col] & (0x80 >> b) != 0 and blkpixel == 0) or (
                            rowdata[col] & (0x80 >> b) == 0 and blkpixel == 1):
                        epd.pixel(col * 8 + b, row, Adafruit_EPD.BLACK)
    except (ValueError) as e:
        display_message("Error: " + e.args[0])
    except BMPError:
        display_message("Error: unsupported BMP file " + filename)
    except OSError:
        display_message("Error: Couldn't open file " + filename)
    finally:
        f.close()
    print("Finished drawing")
    return


def read_le(s):
    result = 0
    shift = 0
    for byte in bytearray(s):
        result += byte << shift
        shift += 8
    return result


class BMPError(Exception):
    pass


# alternate bitmap display method using imageload library
def display_bitmap_alternate(epd, filename):
    image, _ = adafruit_imageload.load(
        filename, bitmap=displayio.Bitmap, palette=displayio.Palette
    )
    for y in range(display.height):
        # blink the LED
        if y % 2 == 0:
            led.value = True
        else:
            led.value = False
        for x in range(display.width):
            if image[x, y] == 0:
                epd.pixel(x, y, Adafruit_EPD.BLACK)
    return


# display message both on the screen and the serial port
def display_message(message):
    print(message)
    display.rotation = 1
    display.fill_rect(0, 10, 264, 20, Adafruit_EPD.WHITE)
    display.text(message, 10, 10, Adafruit_EPD.BLACK)
    display.display()
    return


# slide show routine
def show_files():
    try:
        filelist = os.listdir(config["slidefolder"])
        display.rotation = 1
        led.value = True
        while True:
            for file in filelist:
```

```
                starttime = time.monotonic()
                display.fill(Adafruit_EPD.WHITE)
                print("displaying file", config["slidefolder"] + "/" + file)
                display_bitmap(display, config["slidefolder"] + "/" + file)
                endtime = time.monotonic()
                minutes = (endtime - starttime) // 60
                seconds = int(endtime - starttime) - minutes * 60
                print("update time:", minutes, "minutes", seconds, "seconds")
                print("updating display")
                display.display()
                print("done")
                time.sleep(5)
    except OSError:
        display_message("Error: Couldn't display file " + file)
    led.value = False
    return

#pylint: disable-msg=too-many-branches
#pylint: disable-msg=too-many-statements
#pylint: disable-msg=too-many-locals
# run specified job
def run_job(jobfile):
    try:
        print("running job " + jobfile)
        fpr = open(config["jobfolder"] + "/" + jobfile, mode="r")
        job = json.load(fpr)
        fpr.close()
        print("image: ", job["image"])
        starttime = time.monotonic()
        pixelsize = job["bkpixelsize"]
        whitepct = job["bkratio"]
        panelcount = 6
        led.value = True

        display.rotation = 1
        display.fill(Adafruit_EPD.WHITE)
        print("ePaper display size:", display.width, display.height)
        print(config["imagefolder"] + "/" + job["image"])
        image, _ = adafruit_imageload.load(
            config["imagefolder"] + "/" + job["image"],
            bitmap=displayio.Bitmap,
            palette=displayio.Palette,
        )
        inv = False
        print("image size:", image.width, image.height)
        if image[0] == 1:
            inv = True
            print("using inv image")
        panelwidth = display.width // panelcount
        createfile = True
        try:
            out = open(config["asgfolder"] + "/asg" + job["image"], mode="wb")
            print("writing to file asg" + job["image"])
        except OSError:
            # readonly filesystem, do not create file
            createfile = False
        if createfile:  # == True
            # BMP files are all the same dimensions, just different bitmaps,
            #   writing hardcoded headers here
            # write file header (14 bytes)
            out.write(
                bytearray(
                    [
                        0x42, 0x4D, 0xFE, 0x18, 0, 0, 0, 0, 0, 0, 0x3E, 0, 0, 0
                    ]
                )
            )
            # write image header (40 bytes)
            out.write(
```

```
                bytearray(
                    [
                        0x28, 0, 0, 0, 0x8, 0x1, 0, 0, 0x50, 0xFF,
                        0xFF, 0xFF, 0x1, 0, 0x1, 0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    ]
                )
            )
            # write 2 item color table (8 bytes)
            out.write(bytearray([0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0]))
        with open(
            config["bkfolder"] + "/background-" + str(whitepct)
            + "-" + str(pixelsize) + ".dat", "rb") as fp:
            bkdata = fp.read()
            canvas = list(bkdata)
            for y in range(0, display.height):
                # blink the LED
                if y % 2 == 0:
                    led.value = True
                else:
                    led.value = False
                tcanvas = [0 for i in range(display.width + panelwidth)]
                tpanel = [0 for i in range(panelwidth)]
                for x in range(panelwidth):
                    bytepos = ((x % panelwidth) // 8) + (
                        y // pixelsize * (panelwidth + 7) // 8
                    )
                    bitpos = x % 8
                    pixel = canvas[bytepos] & 1 << (bitpos)
                    if pixel != 0:
                        tpanel[x] = 1
                for x in range(display.width + panelwidth):
                    pixel = tpanel[x % panelwidth]
                    if pixel != 0:
                        tcanvas[x] = 1
                for x in range(0, display.width):
                    if (x % panelwidth) == 0 and x > 0:
                        for x2 in range(x, x + panelwidth):
                            tcanvas[x2] = tcanvas[x2 - panelwidth]
                        for x2 in range(panelwidth):
                            tpanel[x2] = tcanvas[x + x2 - panelwidth]
                    offset = 0
                    if (x >= 22 and
                            x < (image.width + panelwidth // 2)
                            and y < image.height):
                        if (
                                (image[x - panelwidth // 2, y] != 0 and not inv
                                ) or (
                                    image[x - panelwidth // 2, y] == 0 and inv)
                            ):
                            # offset = 4
                            if job["imagegrayscale"] == 0:
                                offset = job["imageheight"]
                            else:
                                offset = (
                                    image[x - panelwidth // 2, y]
                                    * job["grayscalecolors"]
                                    // 255
                                )
                    if offset != 0:
                        for x2 in range(x, display.width, panelwidth):
                            tcanvas[x2] = tcanvas[x2 + offset]
                    for x3 in range(0, display.width):
                        # write line to eink display
                        if tcanvas[x3] != 0:
                            display.pixel(x3, y, Adafruit_EPD.BLACK)
                        # else:
                        #    display.pixel(x,y,Adafruit_EPD.WHITE)
```

```
                    if createfile:
                        count = 0
                        for x4 in range(0, display.width + 7, 8):
                            value = 0
                            for b in range(8):
                                value |= (tcanvas[x4 + b] << 7) >> b
                            out.write(bytes([value]))
                            count += 1
                        # add padding to end of line
                        padding = (4 - (count % 4)) % 4
                        for x4 in range(padding):
                            out.write(bytes([0]))
        if createfile:
            out.close()
        endtime = time.monotonic()
        minutes = (endtime - starttime) // 60
        seconds = int(endtime - starttime) - minutes * 60
        print("completion time:", minutes, "minutes", seconds, "seconds")
        print("updating display")
        display.display()
        print("done")
    except (ValueError) as e:
        display_message("Error: " + e.args[0])
    led.value = False
    return

# main routine
display.fill(Adafruit_EPD.WHITE)
with open("/config.json") as fpx:
    config = json.load(fpx)
    fpx.close()
print("waiting for button press")
while True:
    button = read_buttons()
    if button:
        # button pressed, waiting for button release
        while read_buttons():
            time.sleep(0.1)
    if not button:
        continue
    print("Button #%d pressed" % button)
    if button == 1:
        for jobname in config["jobs"]:
            run_job(jobname)
    if button == 2:
        show_files()
    time.sleep(0.01)
```

Creating an Autostereogram

Job Files

The program uses JSON files to create autostereograms. Job files, typically in the /jobs folder, contain the information on creating an autostereogram image. This includes various parameters and the name of a BMP file containing the image to use for creating the autostereogram.

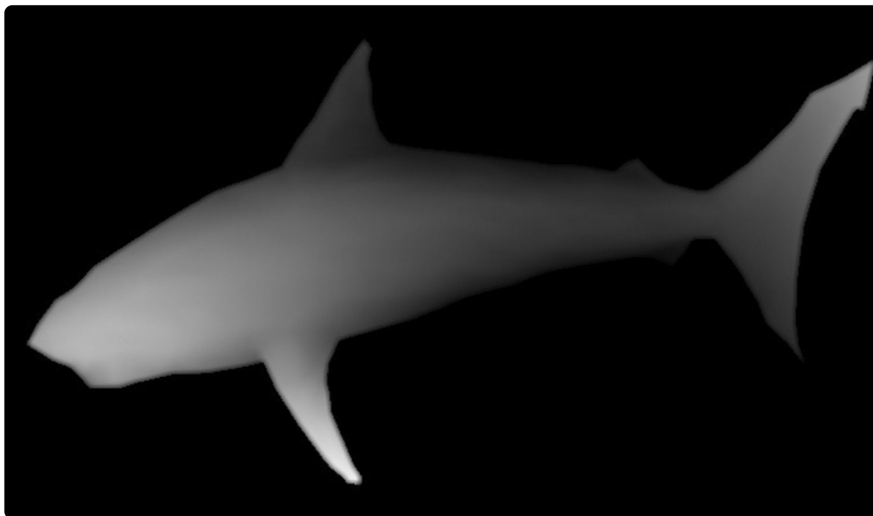
Here is an example job file:

```
{
    "image": "shark.bmp"
    "imagegrayscale": 1
    "grayscalecolors": 16
    "bkpixelsize": 2
    "bkratio": 7
}
```

The "image" option specifies the BMP file to use, typically in the "images" folder. The code supports both monochrome and grayscale BMP files. Monochrome BMP files create a 3D image with a foreground image and background image. Grayscale BMP files use the grayscale color to indicate the depth of the image, with white being closest to the viewer and black the furthest. You can find these images on the internet using search terms like "grayscale depth map" or "3d depth map". Use 1 for "imagegrayscale" if using a grayscale BMP file, otherwise, use 0.

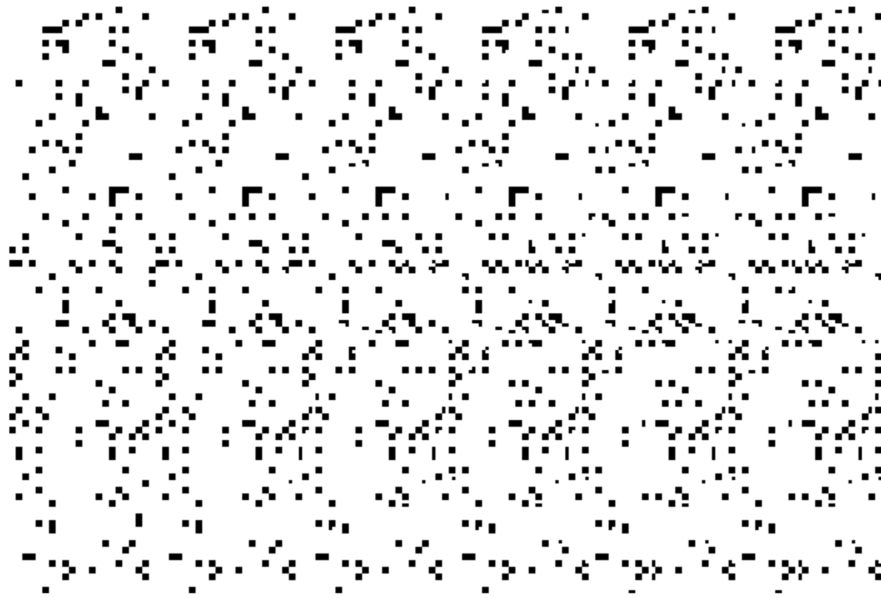
Images should match the ePaper shield size of 264 by 176 pixels, since no scaling is done to the images.

For monochrome images, the "imageheight" option determines how high in the Z direction to display the image. A value of 4 works well here. For grayscale images, the "grayscalecolors" option determines how many planes to use to display the image. The maximum # is 20. You may wish to use a lower number if you find the image is disproportionately large in the Z direction.



The "bkpixelsize" and "bkratio" specify the pixel size and white to black pixel ratio to use for the background. The bkpixelsize can range from 1 to 4, specifying a pixel from 1x1 dots in size to 4x4 dots in size, respectively. The bkratio can range from 1 to 9 with 1 specifying 1 out of 10 pixels are white (a dark background), and 9 specifying 9 out of 10 pixels are white (a light background). The random pixels at the various ratios and sizes have been precalculated to save time in creating the autostereograms and are typically located in the /backgrounds folder. Most images look best with a bkratio

between 4 to 7, however, some simple images can work well using a 1,2, 8 or 9 bkratio value. Here is a monochrome image that works quite well with a bkratio of 9.



The config.json File

The **config.json** file contains information on where files are located, as well as which job(s) to run. Here is a typical **config.json** file:

```
    {
    "jobs": ["adafruitlogo.json"]
    "asgfolder": "/asgfiles"
    "jobfolder": "/jobs"
    "bkfolder": "/backgrounds"
    "imagefolder": "/images"
    "slidefolder": "/slides"
}
```

Here is a description of the options in the config.json file:

- **jobs**: one or more jobs to run to create the autostereograms. Job files are located in the job folder. Separate multiple jobs with commas.
- **asgfolder**: This is where autostereograms are saved if writing them to disk (explained in more detail below)
- **jobfolder:** The location of the job files
- **bkfolder**: the location of the background pixel files used by the "bkpixelsize" and "bkratio" options in the job file
- **imagefolder**: the location of the image files used to create the autostereograms.
- **slidefolder**: the location of image files used for displaying a slideshow (explained in more detail below).

Creating an Autostereogram

Once the job file has been defined and the **config.cfg** file has been updated to include the job name, the job can then be run. Pressing the "A" button on the ePaper display generates the autostereogram from the specified job file and displays it on the ePaper display. You will notice the LED on the Metro M4 Express will blink while it is being generated.

Autostereograms can be saved to a file for sharing on social media, but an additional step is needed to accomplish that. By default, CircuitPython does not have write access to files on its drive, instead the host computer has write access, which is needed to copy images the modify job files, for example. This is to prevent the host computer and CircuitPython possibly opening the same file for writing and corrupting a file. In order for CircuitPython to be able to write to its drive (and make the drive read only to the host computer), press the **Reset** button then quickly press and hold the "A" button. Leave the "A" button pressed for a few seconds to allow the device to boot up. Once the device has booted up, released the "A" button. The device is now running the job as before (with the LED flashing), but this time it is writing an autostereogram file to the **/asgfolder.** The name of the autostereogram will be the same as the image file name with the prefix "asg" in the name. Once the job is completed and the image is displayed, press the reset button to switch back to the original host computer read write mode, where you can then copy the newly created file in the **/asgfolder.**

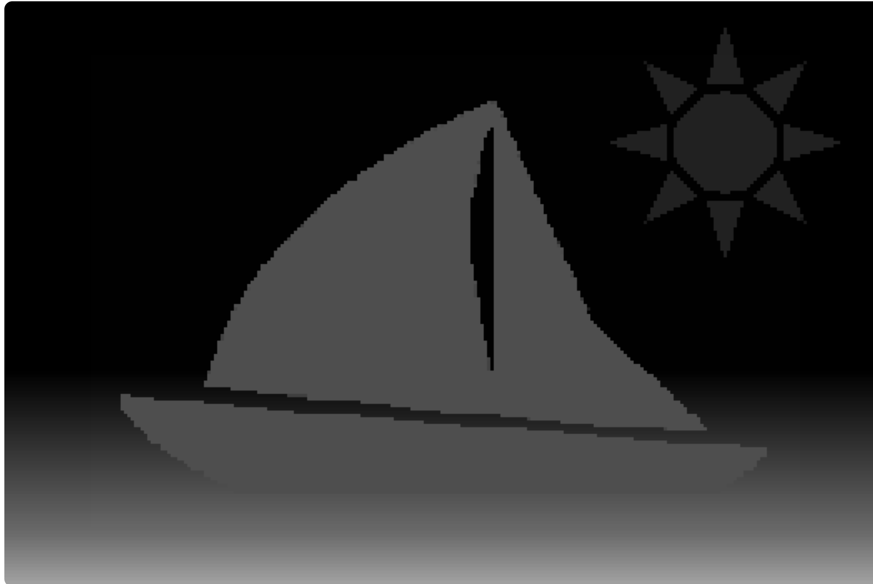
Slideshow Mode

There is a slide show mode, where you can display previously created autostereogram files. Copy the files you wish to include in the slideshow from the **/asgfolder** to **/slidefolder**. Once you have your slideshow images ready, press the "B" button to start the show. The show will continue indefinitely until you power off the device or press the reset button.

Finding Images to Convert to Autostereograms

As mentioned earlier, you can find 3D images to convert on the internet using search terms like "grayscale depth map" or "3d depth map". Remember that these images need to be scaled to 264x176 pixels, so simple images without detail work best. Text is usually not a good choice unless you are converting large block lettering. You can also create images from 2 color bitmaps or even create a grayscale image using a paint program. Images must be saved as either 2 color or grayscale BMP files. Create

a 3D image by creating a grayscale image, black being the background and using white for the closest object. A black to white gradient can be used to create a floor, or ocean waves as shown here:



And here is the image converted to an autostereogram: