Improved AMG8833 PyGamer Thermal Camera

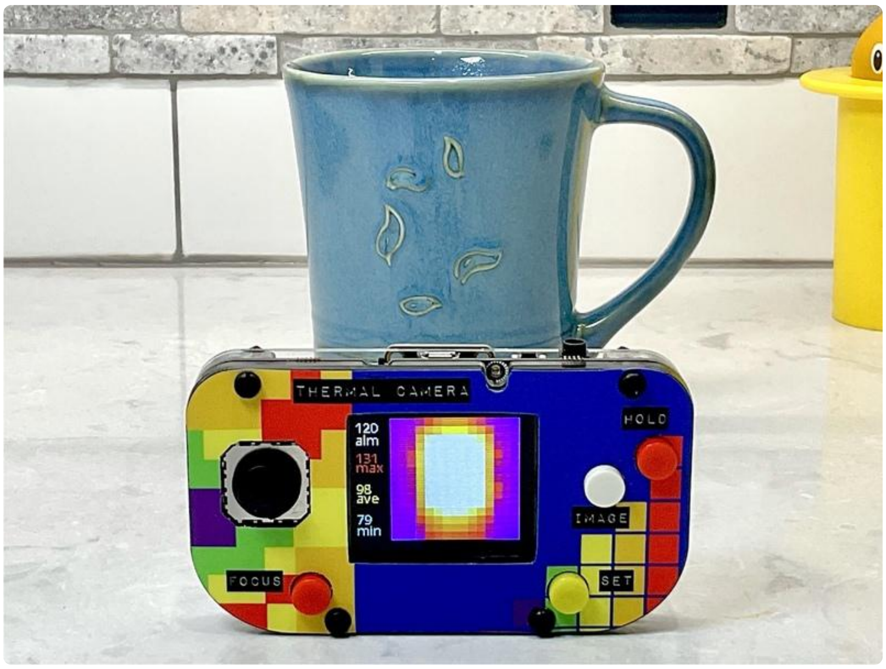
Created by Jan Goolsbey

https://learn.adafruit.com/improved-amg8833-pygamer-thermal-camera

Last updated on 2021-12-21 05:15:19 PM EST

Table of Contents

Overview



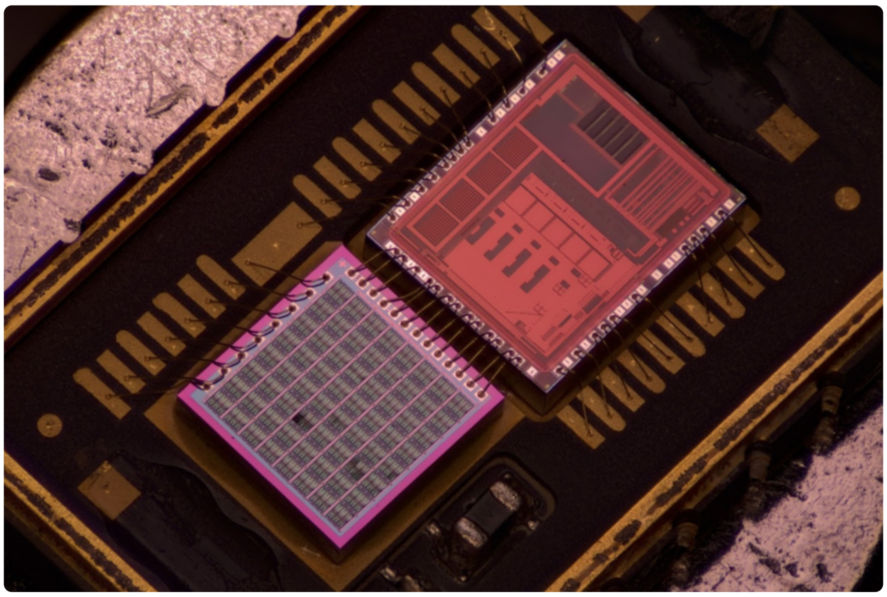
As with the original PyGamer Thermal Camera (https://adafru.it/Tb4), this portable thermal camera project combines an AMG8833 IR Thermal Camera FeatherWing with a PyGamer. The upgraded CircuitPython code used in this version increases the camera resolution from 64 pixels (8 x 8) to 225 pixels (15 x 15) and deepens the color depth from 8 colors to 100 colors, all without hardware modifications.

The new code improves the camera's ability to visualize thermal images to help discern heating and air conditioning ventilation issues, to evaluate the quality of your home's insulation, and to avoid the sleeping cat when heading to the kitchen in the middle of the night.

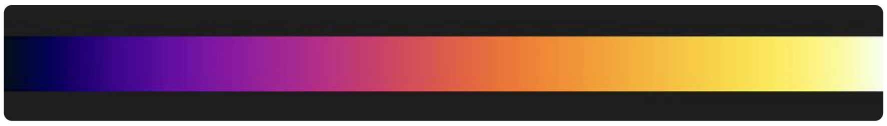
Increasing the display's resolution required changes to the original camera's code to maintain a useful image frame display rate. As a result, performance monitoring was built-in to the new CircuitPython code as a series of time markers with a summary performance report printed to the serial port at the end of each frame update. See the section on Performance Monitoring for more information.

Thermal Camera Features

The camera's thermal image can be frozen or focused at the touch of a button. The focus feature fine-tunes the display's temperature range to match the current image's maximum and maximum measurements, improving the detail of the image. To get a statistical view of an object's heat, switch to histogram mode. A settable alarm flashes lights and beeps when the camera sees a temperature at or above the threshold. The setup function is used to set the temperature display range and the alarm threshold. An editable configuration file contains the camera's power-up settings for the default temperature range and camera sensor direction.



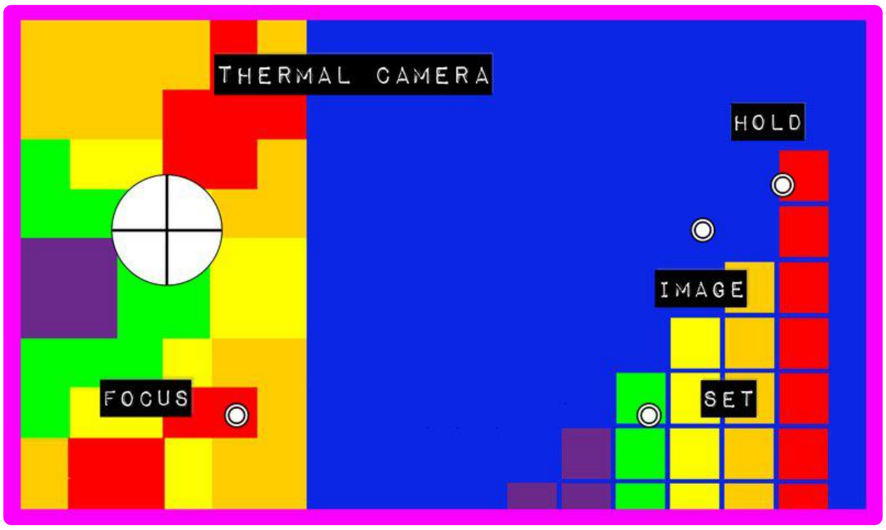
The camera's thermal imaging sensor is an 8 by 8 thermopile array that reads temperatures from 32°F to 176°F (0°C to 80°C) with an absolute accuracy of +- 4.5°F (2.5°C) and resolution of 0.9°F (0.5°C). To improve object recognition, the camera software algorithmically enlarges the number of imaged elements from 64 to 225 by calculating the in-between values using a technique called bilinear interpolation. See the guide section 1-2-3s of Bilinear Interpolation for more detail about the technique.



Temperatures are represented in the displayed image as colors in a spectrum, ranging from a cold blue to white-hot. The color spectrum is based on a frequently-used palette similar to the range of colors seen when an iron bar is heated -- a technique a blacksmith might use to gauge the malleability of metal.

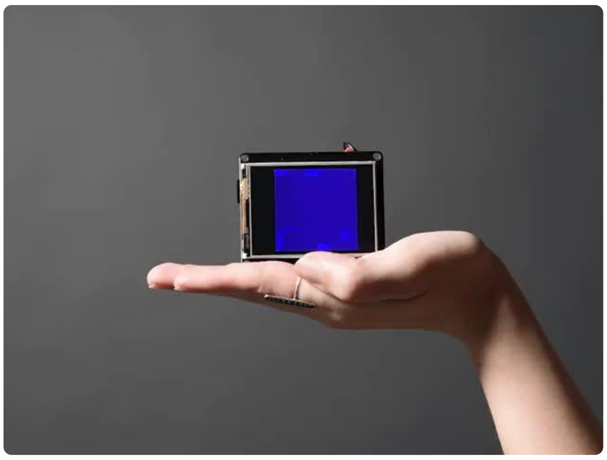
The camera's numeric temperature values are displayed as degrees Fahrenheit. Converting the values to Celsius is possible but is left as an exercise.

The PyGamer Thermal Camera's custom cover skin was produced by a commercial on-line sticker printing service using the image file below.



CAUTION: The AMG8833 sensor used in this project is not accurate or stable enough to be used for health or safety purposes.

Parts



Adafruit AMG8833 IR Thermal Camera FeatherWing

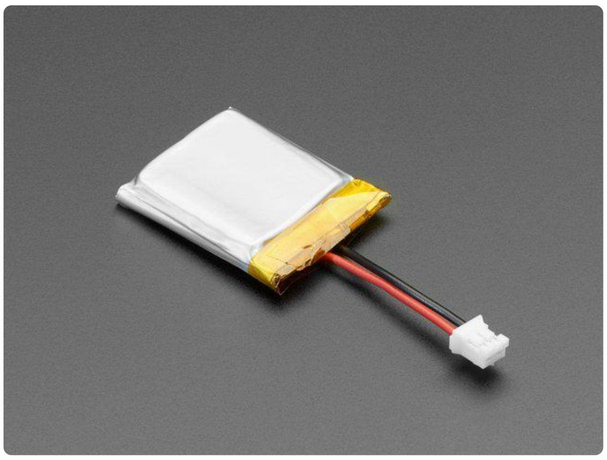
A Feather board without ambition is a Feather board without FeatherWings! This is the Thermal Camera FeatherWing: thanks to the Panasonic AMG8833 8x8 GridEYE sensor,...

https://www.adafruit.com/product/3622

Adafruit PyGamer for MakeCode Arcade, CircuitPython or Arduino

What fits in your pocket, is fully Open Source, and can run CircuitPython, MakeCode Arcade or Arduino games you write yourself? That's right, it's the Adafruit...

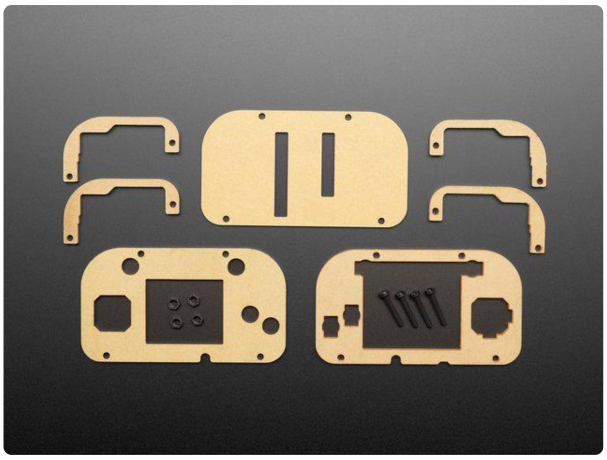
https://www.adafruit.com/product/4242



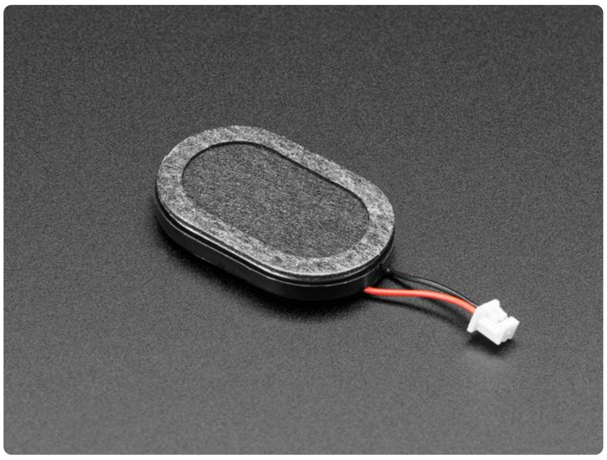
Lithium Ion Polymer Battery with Short Cable - 3.7V 350mAh

Lithium-ion polymer (also known as 'lipo' or 'lipoly') batteries are thin, light, and powerful. The output ranges from 4.2V when completely charged to 3.7V. This...

https://www.adafruit.com/product/4237



Adafruit PyGamer Acrylic Enclosure Kit
You've got your PyGamer, and you're ready to start jammin' on your favorite arcade games. You gaze adoringly at the charming silkscreen designed by Ada-friend...
https://www.adafruit.com/product/4238



Mini Oval Speaker with Short Wires - 8 Ohm 1 Watt
Hear the good news! This wee speaker is a great addition to any audio project where you need 8 ohm impedance and 1W or less of power. We particularly like...
https://www.adafruit.com/product/4227

Plastic Button Caps For Square Top (10-pack) - 8mm Diameter
These Reese's Piece's lookin' bits fit perfectly on top of tactile buttons with 2.4mm square tops and give a satisfying 8mm diameter surface area for your fingers to...
https://www.adafruit.com/product/4228

Other than the AMG8833 Thermal Camera FeatherWing, the following kit contains the PyGamer parts for this project including a nifty carrying case.

Adafruit PyGamer Starter Kit
Please note: you may get a royal blue or purple case with your starter kit (they're both lovely colors)What fits in your pocket, is fully Open...
https://www.adafruit.com/product/4277

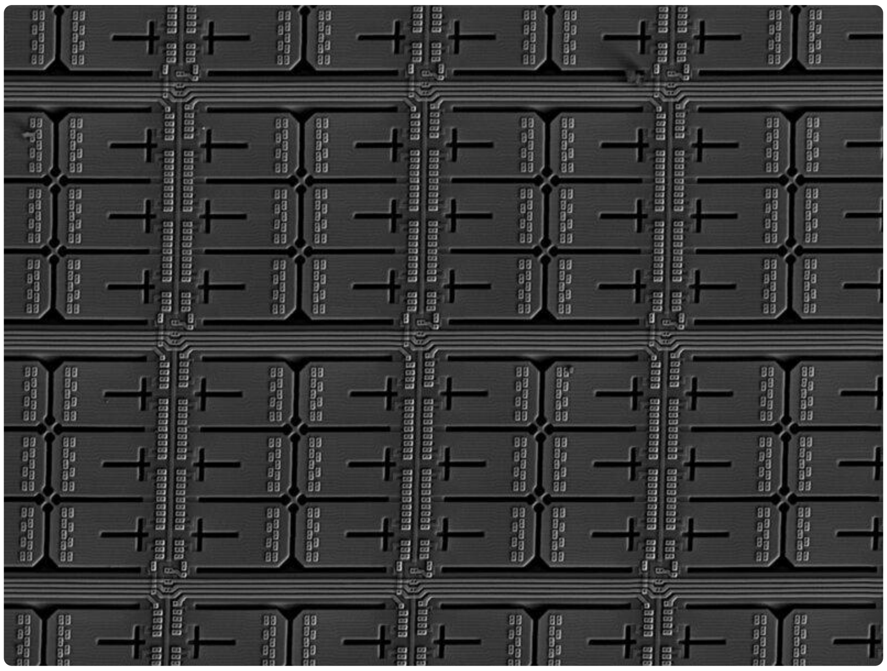
Acknowledgements

Thank you to Adam McCombs for the highly detailed optical and electron microscope photographs of the de-capped AMG8833 sensor. It's fascinating to see how it operates under the covers.

Special thanks to David Glaude and Zoltán Vörös for the ulab-based bilinear interpolation helper. Array calculations using CircuitPython's integral ulab (micro lab) library are amazingly fast and efficient!

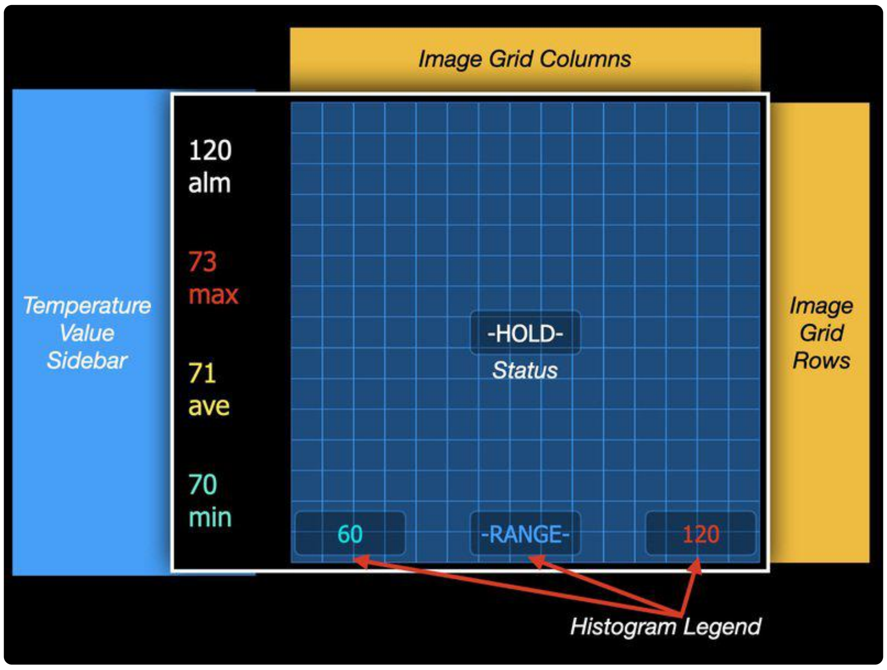
For more information about ulab, check out Jeff Epler's ulab: Crunch Numbers Fast in CircuitPython (https://adafru.it/KaJ) learning guide.

Features and Operation



The Thermal Camera's controls are used to switch display modes, take snapshots, automatically increase or decrease image temperature gradient detail, and facilitate setting alarm and maximum/minimum display range parameters. The display shows the image or histogram and the currently measured maximum, minimum, and average temperature values in Fahrenheit.

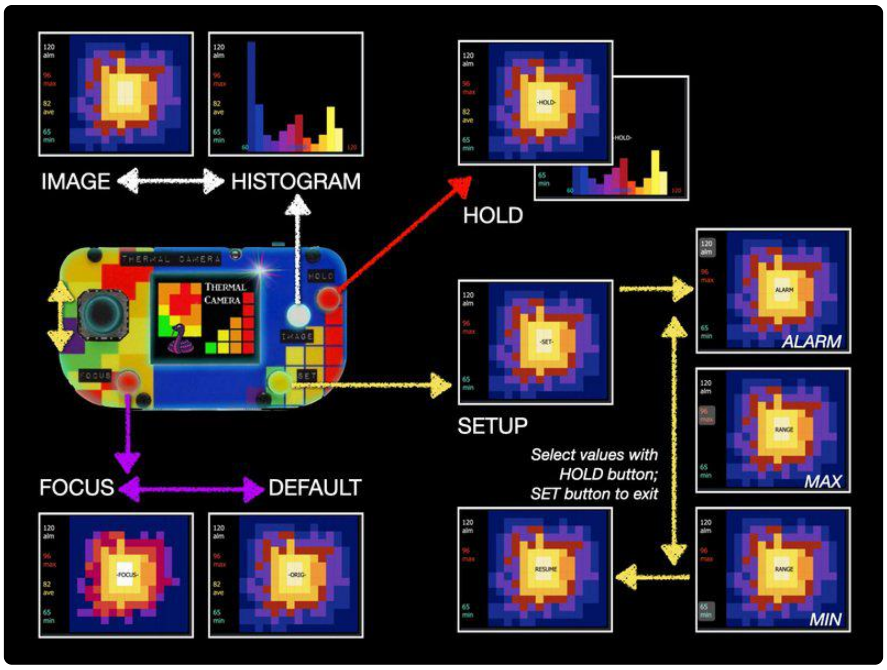
Display Layout



The camera's display is divided into four zones. The temperature value sidebar is used to display the alarm (alm) threshold setting, the measured maximum temperature (max), the average temperature calculation (ave), and the measured minimum temperature (min). The sidebar continuously displays measured values during normal operation. When in the Setup mode, the sidebar indicates the current alarm threshold, the maximum display range, and the minimum display range.

The image grid area consists of 225 blocks in a 15 column by 15 row array. The image grid is used to display a thermal sensor image or histogram.

Superimposed over the display grid are the status message area (centered in the image array area) and the histogram legend area (near the bottom of the image array area). The status message area indicates various operational states including Hold, Fo cus, and the Setup mode. The histogram legend area shows the current minimum and maximum display range settings when viewing a histogram



Hold Mode

The HOLD button (PyGamer BUTTON_A) freezes and releases the image or histogram display contents. Press the button once to hold the display; press it again to resume normal operation. The IMAGE and FOCUS buttons continue to operate normally regardless of whether or not the display is held.

Image / Histogram Mode

The IMAGE (PyGamer BUTTON_B) is used to toggle between a temperature gradient image and a temperature distribution representation of the thermopile sensor's measurements. The IMAGE button is operational when in Hold mode to allow analysis of held measurements.

Focus Range / Default Range

The FOCUS button (PyGamer BUTTON_SELECT) automatically changes the minimum and maximum display range values to provide increased or decreased detail based on the currently measured maximum and minimum temperatures. Press FOCUS once to change the current display range from the current setting to a range that matches the measured minimum and maximum values. Press it again to return to the original display range settings. Focus mode is useful when looking for increased temperature gradient detail or when the temperature of the object is outside of the default display range.

Setup Function

Pressing the SET button (PyGamer BUTTON_START) will stop normal operation and enter the Setup mode to adjust the alarm threshold and maximum/minimum display range. Use the joystick or the PyBadge D-Pad buttons to highlight the parameter to change, then press the HOLD button to select. Use the joystick to increase or decrease the parameter value. Press the HOLD button to select the new value. To exit the Setup mode, press the SET button.

The newly selected values will go into effect when exiting Setup mode, but will not be preserved if the camera's power is turned off. To change power-on parameter values, edit the thermal_cam_config.py file with mu or your favorite text editor.

Build the Camera

It's time to get the PyGamer ready by installing CircuitPython and its libraries, plug in the speaker and battery, and put it into an elegant enclosure. Once the enclosure is in place, we'll attach the AMG8833 FeatherWing and load the Thermal Camera code.

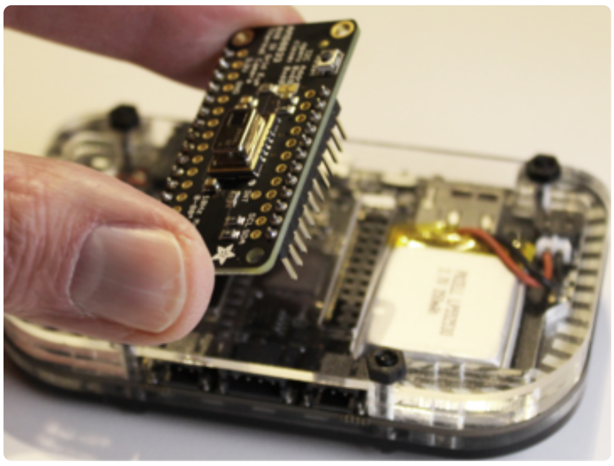
You can build the Thermal Camera from individual components or from the PyGamer Starter Kit (https://adafru.it/IAh). Add the AMG8833 FeatherWing (https://adafru.it/IAi) and you'll be ready to go.

Assembling the PyGamer

The PyGamer Introduction (https://adafru.it/pygamer) will guide you through the process of setting up the PyGamer to include the case, battery, and speaker.

You may follow the Starter Kit enclosure instructions (https://adafru.it/IAj) for installing the speaker and battery, even if you don't plan to use the enclosure.

Prepare the FeatherWing



Solder the included male headers onto the AMG8833 FeatherWing and attach it through the acrylic back panel into the PyGamer's Feather connector. Refer to the soldering guide (https://adafru.it/dxy) if this is your first time with a soldering iron.

Preparing the PyGamer with CircuitPython, Libraries, and Accessories

The PyGamer Introduction (https://adafru.it/pygamer) guide also has the information needed to install CircuitPython (https://adafru.it/FoA) and its libraries (https://adafru.it/IAk).

Software Setup

This project uses CircuitPython, a user friendly version of Python for microcontrollers. The files are just text files and are copied over to the PyGamer to the flash drive CIRCUITPY which appears when the PyGamer is attached to a computer via a USB cable.

Preparing the PyGamer with CircuitPython and Software Libraries

The PyGamer Introduction (https://adafru.it/pygamer) guide also has the information needed to install CircuitPython (https://adafru.it/FoA) and its libraries (https://adafru.it/IAk).

See the following pages on how to perform these operations.

CircuitPython

CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) 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 flash drive to iterate.

The following instructions will show you how to install CircuitPython. If you've already installed CircuitPython but are looking to update it or reinstall it, the same steps work for that as well!

Set up CircuitPython Quick Start!

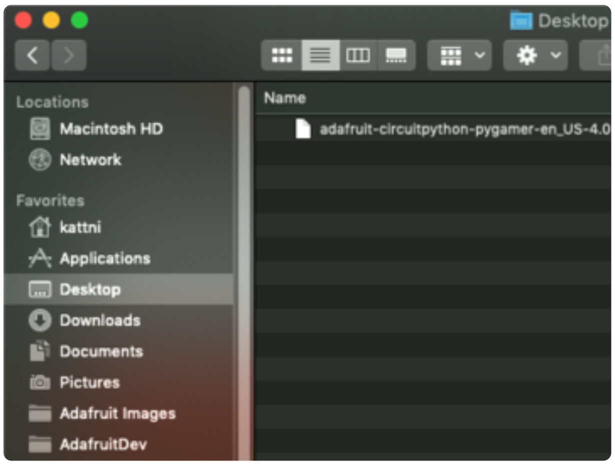
Follow this quick step-by-step for super-fast Python power :)

Download the latest version of CircuitPython for PyGamer via circuitpython.org

https://adafru.it/FxM

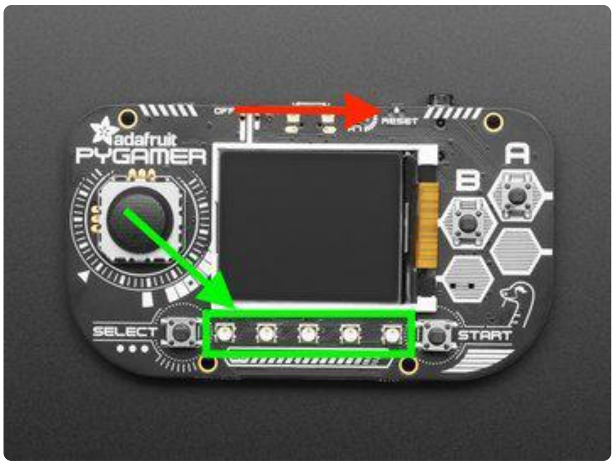
Further Information

For more detailed info on installing CircuitPython, check out Installing CircuitPython (https://adafru.it/Amd).



Click the link above and download the latest UF2 file.

Download and save it to your desktop (or wherever is handy).

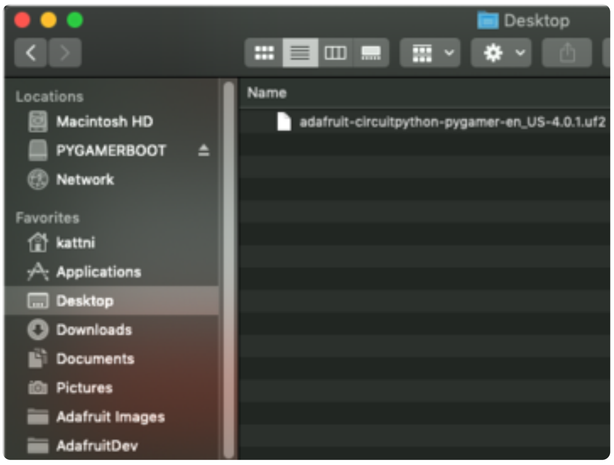


Plug your PyGamer into your computer using a known-good USB cable.

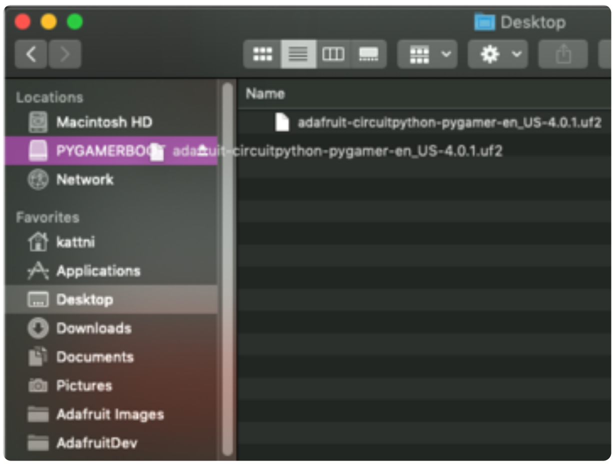
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Double-click the Reset button on the top of your board (indicated by the red arrow in the first image). You will see an image on the display instructing you to drag a UF2 file to your board, and the row of NeoPixel RGB LEDs on the front will turn green (indicated by the green arrow and square in the image). If they turn red, check the USB cable, try another USB port, etc.

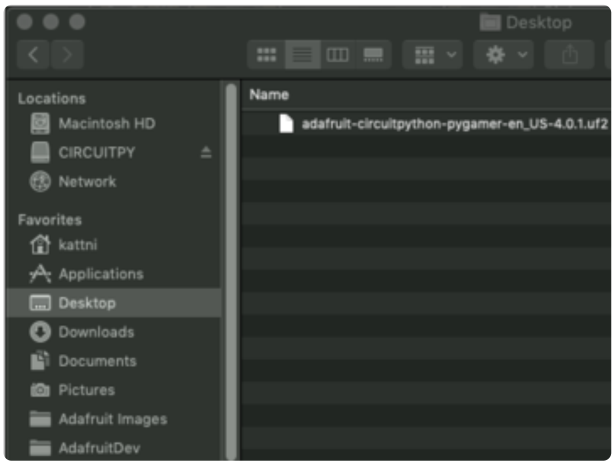
If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!



You will see a new disk drive appear called PYGAMERBOOT.



Drag the adafruit_circuitpython_etc.uf2 file to PYGAMERBOOT.



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

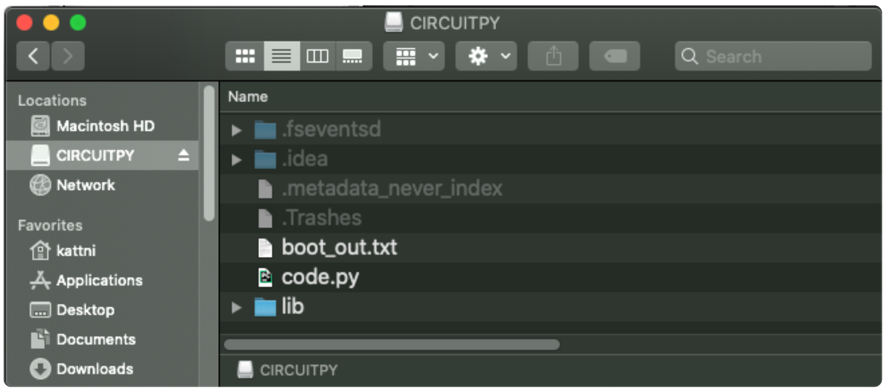
That's it, you're done! :)

CircuitPython Libraries

As CircuitPython development continues and there are new releases, Adafruit will stop supporting older releases. Visit https://circuitpython.org/downloads to download the latest version of CircuitPython for your board. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update CircuitPython and then visit https://circuitpython.org/libraries to download the latest Library Bundle.

Each CircuitPython program you run needs to have a lot of information to work. The reason CircuitPython is so simple to use is that most of that information is stored in other files and works in the background. These files are called libraries. Some of them are built into CircuitPython. Others are stored on your CIRCUITPY drive in a folder called lib. Part of what makes CircuitPython so great is its ability to store code separately from the firmware itself. Storing code separately from the firmware makes it easier to update both the code you write and the libraries you depend.

Your board may ship with a lib folder already, it's in the base directory of the drive. If not, simply create the folder yourself. When you first install CircuitPython, an empty lib directory will be created for you.



CircuitPython libraries work in the same way as regular Python modules so the Python docs (https://adafru.it/rar) are an excellent reference for how it all should work. In Python terms, you can place our library files in the lib directory because it's part of the Python path by default.

One downside of this approach of separate libraries is that they are not built in. To use them, one needs to copy them to the CIRCUITPY drive before they can be used. Fortunately, there is a library bundle.

The bundle and the library releases on GitHub also feature optimized versions of the libraries with the .mpy file extension. These files take less space on the drive and have a smaller memory footprint as they are loaded.

Due to the regular updates and space constraints, Adafruit does not ship boards with the entire bundle. Therefore, you will need to load the libraries you need when you begin working with your board. You can find example code in the guides for your board that depends on external libraries.

Either way, as you start to explore CircuitPython, you'll want to know how to get libraries on board.

The Adafruit CircuitPython Library Bundle

Adafruit provides CircuitPython libraries for much of the hardware they provide, including sensors, breakouts and more. To eliminate the need for searching for each library individually, the libraries are available together in the Adafruit CircuitPython Library Bundle. The bundle contains all the files needed to use each library.

Downloading the Adafruit CircuitPython Library Bundle

You can download the latest Adafruit CircuitPython Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

Match up the bundle version with the version of CircuitPython you are running. For example, you would download the 6.x library bundle if you're running any version of CircuitPython 6, or the 7.x library bundle if you're running any version of CircuitPython 7, etc. If you mix libraries with major CircuitPython versions, you will get incompatible mpy errors due to changes in library interfaces possible during major version changes.

Click to visit circuitpython.org for the latest Adafruit CircuitPython Library Bundle

https://adafru.it/ENC

Download the bundle version that matches your CircuitPython firmware version. If you don't know the version, check the version info in boot_out.txt file on the CIRCUITPY drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.x library bundle.

There's also a py bundle which contains the uncompressed python files, you probably don't want that unless you are doing advanced work on libraries.

The CircuitPython Community Library Bundle

The CircuitPython Community Library Bundle is made up of libraries written and provided by members of the CircuitPython community. These libraries are often written when community members encountered hardware not supported in the Adafruit Bundle, or to support a personal project. The authors all chose to submit these libraries to the Community Bundle make them available to the community.

These libraries are maintained by their authors and are not supported by Adafruit. As you would with any library, if you run into problems, feel free to file an issue on the GitHub repo for the library. Bear in mind, though, that most of these libraries are supported by a single person and you should be patient about receiving a response. Remember, these folks are not paid by Adafruit, and are volunteering their personal time when possible to provide support.

Downloading the CircuitPython Community Library Bundle

You can download the latest CircuitPython Community Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

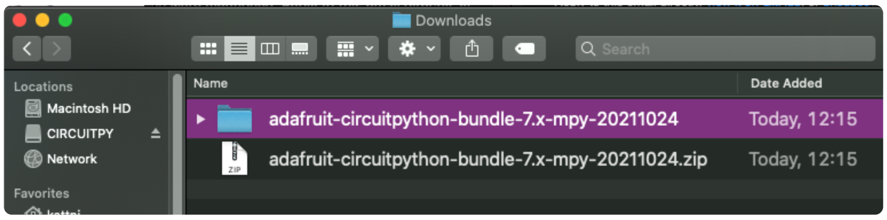
Click for the latest CircuitPython Community Library Bundle release

https://adafru.it/VCn

The link takes you to the latest release of the CircuitPython Community Library Bundle on GitHub. There are multiple versions of the bundle available. Download the bundle version that matches your CircuitPython firmware version. If you don't know the version, check the version info in boot_out.txt file on the CIRCUITPY drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.x library bundle.

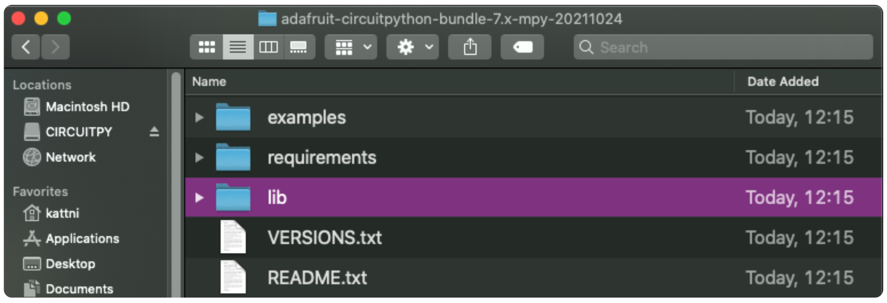
Understanding the Bundle

After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.



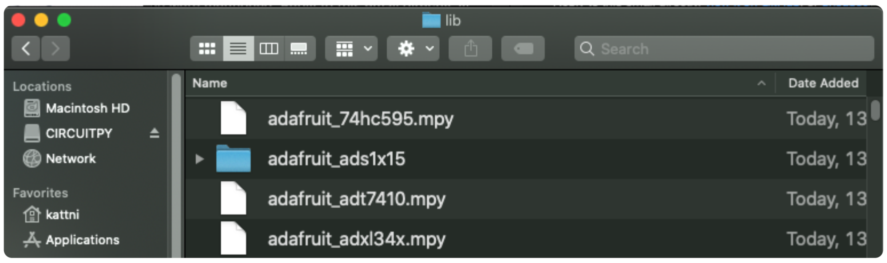
Name	Date Added
adafruit-circuitpython-bundle-7.x-mpy-20211024	Today, 12:15
adafruit-circuitpython-bundle-7.x-mpy-20211024.zip	Today, 12:15

Open the bundle folder. Inside you'll find two information files, and two folders. One folder is the lib bundle, and the other folder is the examples bundle.



Name	Date Added
examples	Today, 12:15
requirements	Today, 12:15
lib	Today, 12:15
VERSIONS.txt	Today, 12:15
README.txt	Today, 12:15

Now open the lib folder. When you open the folder, you'll see a large number of .mpy files, and folders.

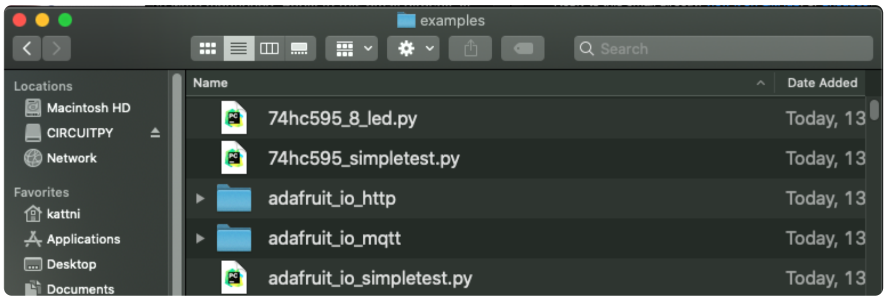


Name	Date Added
adafruit_74hc595.mpy	Today, 13
adafruit_ads1x15	Today, 13
adafruit_adt7410.mpy	Today, 13
adafruit_adxl34x.mpy	Today, 13

Example Files

All example files from each library are now included in the bundles in an examples directory (as seen above), as well as an examples-only bundle. These are included for two main reasons:

- Allow for quick testing of devices.
- Provide an example base of code, that is easily built upon for individualized purposes.



74hc595_8_led.py Today, 13
74hc595_simpletest.py Today, 13
adafruit_io_http Today, 13
adafruit_io_mqtt Today, 13
adafruit_io_simpletest.py Today, 13

Copying Libraries to Your Board

First open the lib folder on your CIRCUITPY drive. Then, open the lib folder you extracted from the downloaded zip. Inside you'll find a number of folders and .mpy files. Find the library you'd like to use, and copy it to the lib folder on CIRCUITPY.

If the library is a directory with multiple .mpy files in it, be sure to copy the entire folder to CIRCUITPY/lib.

This also applies to example files. Open the examples folder you extracted from the downloaded zip, and copy the applicable file to your CIRCUITPY drive. Then, rename it to code.py to run it.

If a library has multiple .mpy files contained in a folder, be sure to copy the entire folder to CIRCUITPY/lib.

Understanding Which Libraries to Install

You now know how to load libraries on to your CircuitPython-compatible microcontroller board. You may now be wondering, how do you know which libraries you need to install? Unfortunately, it's not always straightforward. Fortunately, there is an obvious place to start, and a relatively simple way to figure out the rest. First up: the best place to start.

When you look at most CircuitPython examples, you'll see they begin with one or more `import` statements. These typically look like the following:

- `import library_or_module`

However, `import` statements can also sometimes look like the following:

- `from library_or_module import name`

- `from library_or_module.subpackage import name`
- `from library_or_module import name as local_name`

They can also have more complicated formats, such as including a `try` / `except` block, etc.

The important thing to know is that an `import` statement will always include the name of the module or library that you're importing.

Therefore, the best place to start is by reading through the `import` statements.

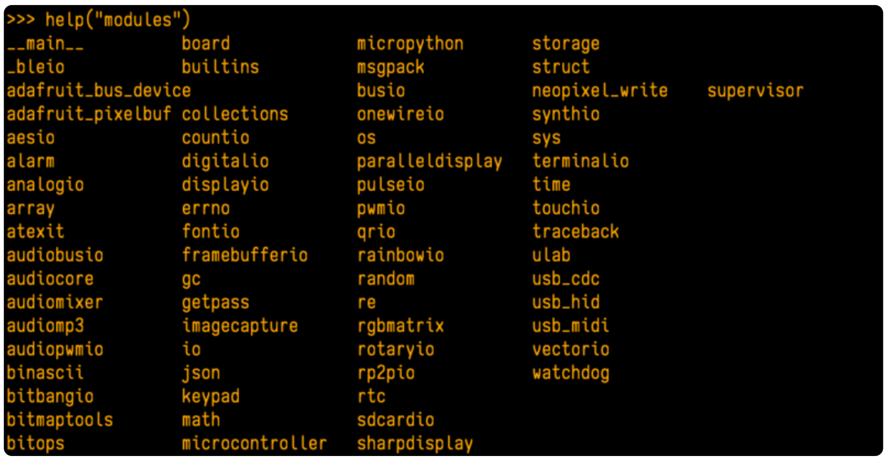
Here is an example import list for you to work with in this section. There is no setup or other code shown here, as the purpose of this section involves only the import list.

```
import time
import board
import neopixel
import adafruit_lis3dh
import usb_hid
from adafruit_hid.consumer_control import ConsumerControl
from adafruit_hid.consumer_control_code import ConsumerControlCode
```

Keep in mind, not all imported items are libraries. Some of them are almost always built-in CircuitPython modules. How do you know the difference? Time to visit the REPL.

In the Interacting with the REPL section (https://adafru.it/Awz) on The REPL page (https://adafru.it/Awz) in this guide, the `help("modules")` command is discussed. This command provides a list of all of the built-in modules available in CircuitPython for your board. So, if you connect to the serial console on your board, and enter the REPL, you can run `help("modules")` to see what modules are available for your board. Then, as you read through the `import` statements, you can, for the purposes of figuring out which libraries to load, ignore the statement that import modules.

The following is the list of modules built into CircuitPython for the Feather RP2040. Your list may look similar or be anything down to a significant subset of this list for smaller boards.



```
>>> help("modules")
__main__          board             micropython       storage
_bleio            builtins          msgpack           struct
adafruit_bus_device                 busio             neopixel_write    supervisor
adafruit_pixelbuf collections       onewireio         synthio
aesio             countio           os                sys
alarm             digitalio         paralleldisplay   terminalio
analogio          displayio         pulseio           time
array             errno             pwmio             touchio
atexit            fontio            qrio              traceback
audiobusio        framebufferio     rainbowio         ulab
audiocore         gc                random            usb_cdc
audiomixer        getpass           re                usb_hid
audiomp3          imagecapture      rgbmatrix         usb_midi
audiopwmio        io                rotaryio          vectorio
binascii          json              rp2pio            watchdog
bitbangio         keypad            rtc
bitmaptools       math              sdcardio
bitops            microcontroller   sharpdisplay
```

Now that you know what you're looking for, it's time to read through the import statements. The first two, `time` and `board`, are on the modules list above, so they're built-in.

The next one, `neopixel`, is not on the module list. That means it's your first library! So, you would head over to the bundle zip you downloaded, and search for neopixel. There is a neopixel.mpy file in the bundle zip. Copy it over to the lib folder on your CI RCUITPY drive. The following one, `adafruit_lis3dh`, is also not on the module list. Follow the same process for adafruit_lis3dh, where you'll find adafruit_lis3dh.mpy, and copy that over.

The fifth one is `usb_hid`, and it is in the modules list, so it is built in. Often all of the built-in modules come first in the import list, but sometimes they don't! Don't assume that everything after the first library is also a library, and verify each import with the modules list to be sure. Otherwise, you'll search the bundle and come up empty!

The final two imports are not as clear. Remember, when `import` statements are formatted like this, the first thing after the `from` is the library name. In this case, the library name is `adafruit_hid`. A search of the bundle will find an adafruit_hid folder. When a library is a folder, you must copy the entire folder and its contents as it is in the bundle to the lib folder on your CIRCUITPY drive. In this case, you would copy the entire adafruit_hid folder to your CIRCUITPY/lib folder.

Notice that there are two imports that begin with `adafruit_hid`. Sometimes you will need to import more than one thing from the same library. Regardless of how many times you import the same library, you only need to load the library by copying over the adafruit_hid folder once.

That is how you can use your example code to figure out what libraries to load on your CircuitPython-compatible board!

There are cases, however, where libraries require other libraries internally. The internally required library is called a dependency. In the event of library dependencies, the easiest way to figure out what other libraries are required is to connect to the serial console and follow along with the `ImportError` printed there. The following is a very simple example of an `ImportError`, but the concept is the same for any missing library.

Example: `ImportError` Due to Missing Library

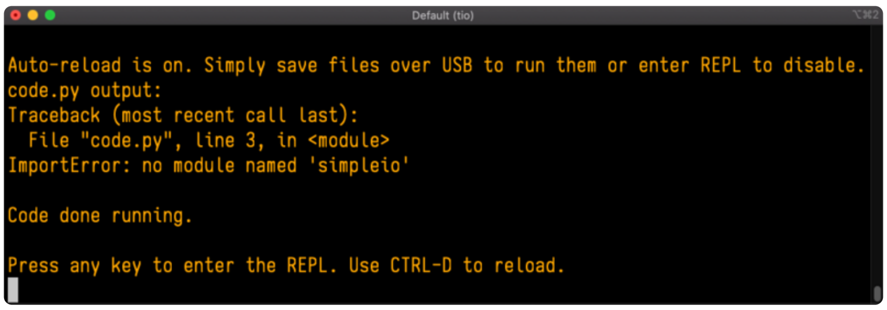
If you choose to load libraries as you need them, or you're starting fresh with an existing example, you may end up with code that tries to use a library you haven't yet loaded. This section will demonstrate what happens when you try to utilise a library that you don't have loaded on your board, and cover the steps required to resolve the issue.

This demonstration will only return an error if you do not have the required library loaded into the lib folder on your CIRCUITPY drive.

Let's use a modified version of the Blink example.

```
import board
import time
import simpleio

led = simpleio.DigitalOut(board.LED)

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

Save this file. Nothing happens to your board. Let's check the serial console to see what's going on.

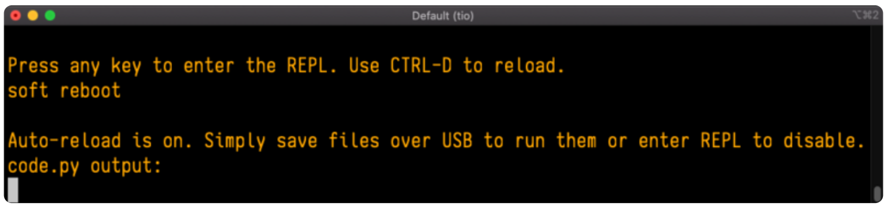


```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Traceback (most recent call last):
  File "code.py", line 3, in <module>
ImportError: no module named 'simpleio'

Code done running.

Press any key to enter the REPL. Use CTRL-D to reload.
```

You have an `ImportError`. It says there is `no module named 'simpleio'`. That's the one you just included in your code!

Click the link above to download the correct bundle. Extract the lib folder from the downloaded bundle file. Scroll down to find simpleio.mpy. This is the library file you're looking for! Follow the steps above to load an individual library file.

The LED starts blinking again! Let's check the serial console.



```
Press any key to enter the REPL. Use CTRL-D to reload.
soft reboot

Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
```

No errors! Excellent. You've successfully resolved an `ImportError`!

If you run into this error in the future, follow along with the steps above and choose the library that matches the one you're missing.

Library Install on Non-Express Boards

If you have an M0 non-Express board such as Trinket M0, Gemma M0, QT Py M0, or one of the M0 Trinkeys, you'll want to follow the same steps in the example above to install libraries as you need them. Remember, you don't need to wait for an `ImportError` if you know what library you added to your code. Open the library bundle you downloaded, find the library you need, and drag it to the lib folder on your CIRCUITPY drive.

You can still end up running out of space on your M0 non-Express board even if you only load libraries as you need them. There are a number of steps you can use to try to resolve this issue. You'll find suggestions on the Troubleshooting page (https://adafru.it/Den).

Updating CircuitPython Libraries and Examples

Libraries and examples are updated from time to time, and it's important to update the files you have on your CIRCUITPY drive.

To update a single library or example, follow the same steps above. When you drag the library file to your lib folder, it will ask if you want to replace it. Say yes. That's it!

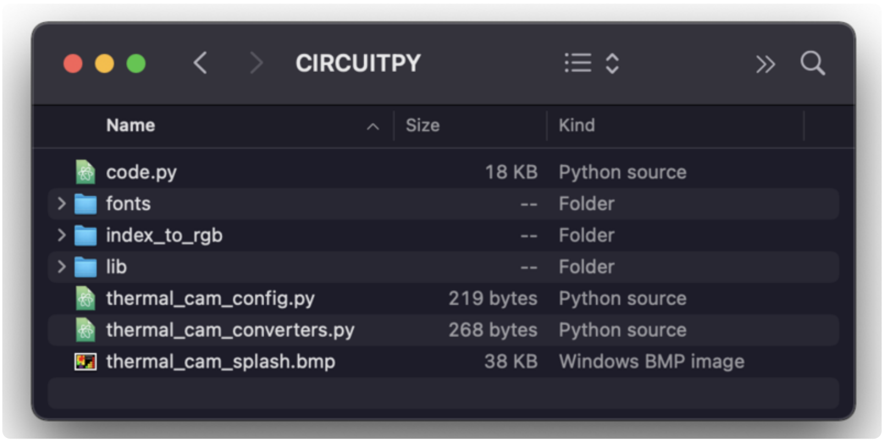
A new library bundle is released every time there's an update to a library. Updates include things like bug fixes and new features. It's important to check in every so often to see if the libraries you're using have been updated.

CircuitPython Code

PyGamer Thermal Camera Source Code

Download the project's source files and copy them to the PyGamer's CIRCUITPY root directory, including the fonts and index_to_rgb folders.

In the code window below, click the link Download Project Bundle. This will download a zip file containing the code (4 .py files), the index_to_rgb folder, needed library files and the font folder.



Name	Size	Kind
code.py	18 KB	Python source
fonts	--	Folder
index_to_rgb	--	Folder
lib	--	Folder
thermal_cam_config.py	219 bytes	Python source
thermal_cam_converters.py	268 bytes	Python source
thermal_cam_splash.bmp	38 KB	Windows BMP image

The zip folder contains the following folders and files:

- fonts folder
    - OpenSans-9.bdf font file
- index_to_rgb folder
    - iron_spectrum.py color converter method file
- code.py main thermal camera code
- thermal_cam_config.py start-up default settings
- thermal_cam_converters.py temperature converter helpers
- thermal_cam_splash.bmp startup screen graphic

- A lib folder containing these required libraries:
    - adafruit_amg88xx
    - adafruit_bitmap_font
    - adafruit_display_text
    - adafruit_display_shapes
    - adafruit_register
    - neopixel

Here's the main CircuitPython code for the Thermal Camera. It's contained in the project zip folder as code.py. Copy this to the main (root) folder of the CIRCUITPY drive that appears when your PyGamer is connected to your computer via a known good USB cable.

```
# SPDX-FileCopyrightText: 2021 Jan Goolsbey for Adafruit Industries
# SPDX-License-Identifier: MIT

# Thermal_Cam_v70_PyBadge_code.py
# 2021-12-21 v7.0  # CircuitPython v7.x compatible

import time
import board
import busio
import gc
import ulab
import displayio
import neopixel
from analogio import AnalogIn
from digitalio import DigitalInOut
from simpleio import map_range, tone
from adafruit_display_text.label import Label
from adafruit_bitmap_font import bitmap_font
from adafruit_display_shapes.rect import Rect
import adafruit_amg88xx
from gamepadshift import GamePadShift
from index_to_rgb.iron_spectrum import index_to_rgb
from thermal_cam_converters import celsius_to_fahrenheit, fahrenheit_to_celsius
from thermal_cam_config import ALARM_F, MIN_RANGE_F, MAX_RANGE_F, SELFIE

# Instantiate display, joystick, speaker, and neopixels
display = board.DISPLAY
# Load the text font from the fonts folder
font_0 = bitmap_font.load_font("/fonts/OpenSans-9.bdf")

if hasattr(board, "JOYSTICK_X"):
    has_joystick = True  # PyGamer with joystick
    joystick_x = AnalogIn(board.JOYSTICK_X)
    joystick_y = AnalogIn(board.JOYSTICK_Y)
else:
    has_joystick = False  # PyBadge with buttons

speaker_enable = DigitalInOut(board.SPEAKER_ENABLE)
speaker_enable.switch_to_output(value=True)

pixels = neopixel.NeoPixel(board.NEOPIXEL, 5, pixel_order=neopixel.GRB)
pixels.brightness = 0.25  # Set NeoPixel brightness
pixels.fill(0x000000)  # Clear all NeoPixels
```

```
# Define and instantiate front panel buttons
BUTTON_LEFT = 0b10000000
BUTTON_UP = 0b01000000
BUTTON_DOWN = 0b00100000
BUTTON_RIGHT = 0b00010000
BUTTON_SELECT = 0b00001000
BUTTON_START = 0b00000100
BUTTON_A = 0b00000010
BUTTON_B = 0b00000001

panel = GamePadShift(
    DigitalInOut(board.BUTTON_CLOCK),
    DigitalInOut(board.BUTTON_OUT),
    DigitalInOut(board.BUTTON_LATCH),
)

# Establish I2C interface for the AMG8833 Thermal Camera
i2c = busio.I2C(board.SCL, board.SDA, frequency=400000)
amg8833 = adafruit_amg88xx.AMG88XX(i2c)

# Display splash graphics
splash = displayio.Group(scale=display.width // 160)
bitmap = displayio.OnDiskBitmap("/thermal_cam_splash.bmp")
splash.append(displayio.TileGrid(bitmap, pixel_shader=bitmap.pixel_shader))
board.DISPLAY.show(splash)
time.sleep(0.1)  # Allow the splash to display

# Set up ulab arrays
n = 8  # Thermal sensor grid axis size; AMG8833 sensor is 8x8
sensor_data = ulab.numpy.array(range(n * n)).reshape((n, n))  # Color index narray
grid_data = ulab.numpy.zeros(((2 * n) - 1, (2 * n) - 1))  # 15x15 color index narray
histogram = ulab.numpy.zeros((2 * n) - 1)  # Histogram accumulation narray

# Convert default alarm and min/max range values from config file
ALARM_C = fahrenheit_to_celsius(ALARM_F)
MIN_RANGE_C = fahrenheit_to_celsius(MIN_RANGE_F)
MAX_RANGE_C = fahrenheit_to_celsius(MAX_RANGE_F)

# The board's integral display size
WIDTH = display.width
HEIGHT = display.height

GRID_AXIS = (2 * n) - 1  # Number of cells along the grid x or y axis
GRID_SIZE = HEIGHT  # Maximum number of pixels for a square grid
GRID_X_OFFSET = WIDTH - GRID_SIZE  # Right-align grid with display boundary
CELL_SIZE = GRID_SIZE // GRID_AXIS  # Size of a grid cell in pixels

PALETTE_SIZE = 100  # Number of colors in spectral palette (must be > 0)

# Default colors for temperature value sidebar
BLACK = 0x000000
RED = 0xFF0000
YELLOW = 0xFFFF00
CYAN = 0x00FFFF
BLUE = 0x0000FF
WHITE = 0xFFFFFF

# Text colors for setup helper's on-screen parameters
param_colors = [("ALARM", WHITE), ("RANGE", RED), ("RANGE", CYAN)]

# ### Helpers ###
def play_tone(freq=440, duration=0.01):
    tone(board.A0, freq, duration)
    return


def flash_status(text="", duration=0.05):  # Flash status message once
    status_label.color = WHITE
    status_label.text = text
    time.sleep(duration)
    status_label.color = BLACK
    time.sleep(duration)
    status_label.text = ""
    return


def spectrum():  # Load a test spectrum into the grid_data array
    for row in range(0, GRID_AXIS):
        for col in range(0, GRID_AXIS):
            grid_data[row][col] = ((row * GRID_AXIS) + col) * 1 / 235
    return


def update_image_frame(selfie=False):  # Get camera data and update display
    for row in range(0, GRID_AXIS):
        for col in range(0, GRID_AXIS):
            if selfie:
                color_index = grid_data[GRID_AXIS - 1 - row][col]
            else:
                color_index = grid_data[GRID_AXIS - 1 - row][GRID_AXIS - 1 - col]
            color = index_to_rgb(round(color_index * PALETTE_SIZE, 0) /
PALETTE_SIZE)
            if color != image_group[((row * GRID_AXIS) + col)].fill:
                image_group[((row * GRID_AXIS) + col)].fill = color
    return


def update_histo_frame():  # Calculate and display histogram
    min_histo.text = str(MIN_RANGE_F)  # Display histogram legend
    max_histo.text = str(MAX_RANGE_F)

    histogram = ulab.numpy.zeros(GRID_AXIS)  # Clear histogram accumulation array
    for row in range(0, GRID_AXIS):  # Collect camera data and calculate histo
        for col in range(0, GRID_AXIS):
            histo_index = int(map_range(grid_data[col, row], 0, 1, 0, GRID_AXIS -
1))
            histogram[histo_index] = histogram[histo_index] + 1

    histo_scale = ulab.numpy.max(histogram) / (GRID_AXIS - 1)
    if histo_scale <= 0:
        histo_scale = 1

    for col in range(0, GRID_AXIS):  # Display histogram
        for row in range(0, GRID_AXIS):
            if histogram[col] / histo_scale > GRID_AXIS - 1 - row:
                image_group[((row * GRID_AXIS) + col)].fill = index_to_rgb(
                    round((col / GRID_AXIS), 3)
                )
            else:
                image_group[((row * GRID_AXIS) + col)].fill = BLACK
    return


def ulab_bilinear_interpolation():  # 2x bilinear interpolation
    # Upscale sensor data array; by @v923z and @David.Glaude
    grid_data[1::2, ::2] = sensor_data[:-1, :]
    grid_data[1::2, ::2] += sensor_data[1:, :]
    grid_data[1::2, ::2] /= 2
    grid_data[::, 1::2] = grid_data[::, :-1:2]
    grid_data[::, 1::2] += grid_data[::, 2::2]
    grid_data[::, 1::2] /= 2
    return


def setup_mode():  # Set alarm threshold and minimum/maximum range values
    status_label.color = WHITE
    status_label.text = "-SET-"
    ave_label.color = BLACK  # Turn off average label and value display
    ave_value.color = BLACK

    max_value.text = str(MAX_RANGE_F)  # Display maximum range value
    min_value.text = str(MIN_RANGE_F)  # Display minimum range value

    time.sleep(0.8)  # Show SET status text before setting parameters
    status_label.text = ""  # Clear status text

    param_index = 0  # Reset index of parameter to set

    # Select parameter to set

    buttons = panel.get_pressed()
    while not buttons & BUTTON_START:
        buttons = panel.get_pressed()
        while (not buttons & BUTTON_A) and (not buttons & BUTTON_START):
            up, down = move_buttons(joystick=has_joystick)
            if up:
                param_index = param_index - 1
            if down:
                param_index = param_index + 1
            param_index = max(0, min(2, param_index))
            status_label.text = param_colors[param_index][0]
            image_group[param_index + 226].color = BLACK
            status_label.color = BLACK
            time.sleep(0.25)
            image_group[param_index + 226].color = param_colors[param_index][1]
            status_label.color = WHITE
            time.sleep(0.25)
            buttons = panel.get_pressed()

        buttons = panel.get_pressed()
        if buttons & BUTTON_A:  # Hold (button A) pressed
            play_tone(1319, 0.030)  # E6
        while buttons & BUTTON_A:  # Wait for button release
            buttons = panel.get_pressed()
            time.sleep(0.1)

        # Adjust parameter value
        param_value = int(image_group[param_index + 230].text)
        buttons = panel.get_pressed()
        while (not buttons & BUTTON_A) and (not buttons & BUTTON_START):
            up, down = move_buttons(joystick=has_joystick)
            if up:
                param_value = param_value + 1
            if down:
                param_value = param_value - 1
            param_value = max(32, min(157, param_value))
            image_group[param_index + 230].text = str(param_value)
            image_group[param_index + 230].color = BLACK
            status_label.color = BLACK
            time.sleep(0.05)
            image_group[param_index + 230].color = param_colors[param_index][1]
            status_label.color = WHITE
            time.sleep(0.2)
            buttons = panel.get_pressed()

        buttons = panel.get_pressed()
        if buttons & BUTTON_A:  # Button A pressed
            play_tone(1319, 0.030)  # E6
        while buttons & BUTTON_A:  # Wait for button release
            buttons = panel.get_pressed()
            time.sleep(0.1)

    # Exit setup process
    buttons = panel.get_pressed()
    if buttons & BUTTON_START:  # Start button pressed
        play_tone(784, 0.030)  # G5
    while buttons & BUTTON_START:  # Wait for button release
        buttons = panel.get_pressed()
        time.sleep(0.1)

    status_label.text = "RESUME"
    time.sleep(0.5)
    status_label.text = ""

    # Display average label and value
    ave_label.color = YELLOW
    ave_value.color = YELLOW
    return int(alarm_value.text), int(max_value.text), int(min_value.text)


def move_buttons(joystick=False):  # Read position buttons and joystick
    move_u = move_d = False
    if joystick:  # For PyGamer: interpret joystick as buttons
        if joystick_y.value < 20000:
            move_u = True
        elif joystick_y.value > 44000:
            move_d = True
    else:  # For PyBadge read the buttons
        buttons = panel.get_pressed()
        if buttons & BUTTON_UP:
            move_u = True
        if buttons & BUTTON_DOWN:
            move_d = True
    return move_u, move_d


play_tone(440, 0.1)  # A4
play_tone(880, 0.1)  # A5

# ### Define the display group ###
t0 = time.monotonic()  # Time marker: Define Display Elements
image_group = displayio.Group(scale=1)

# Define the foundational thermal image grid cells; image_group[0:224]
#   image_group[#] = image_group[ (row * GRID_AXIS) + column ]
for row in range(0, GRID_AXIS):
    for col in range(0, GRID_AXIS):
        cell_x = (col * CELL_SIZE) + GRID_X_OFFSET
        cell_y = row * CELL_SIZE
        cell = Rect(
            x=cell_x,
            y=cell_y,
            width=CELL_SIZE,
            height=CELL_SIZE,
            fill=None,
            outline=None,
            stroke=0,
        )
        image_group.append(cell)

# Define labels and values
status_label = Label(font_0, text="", color=None)
status_label.anchor_point = (0.5, 0.5)
status_label.anchored_position = ((WIDTH // 2) + (GRID_X_OFFSET // 2), HEIGHT // 2)
image_group.append(status_label)  # image_group[225]

alarm_label = Label(font_0, text="alm", color=WHITE)
alarm_label.anchor_point = (0, 0)
alarm_label.anchored_position = (1, 16)
image_group.append(alarm_label)  # image_group[226]

max_label = Label(font_0, text="max", color=RED)
max_label.anchor_point = (0, 0)
max_label.anchored_position = (1, 46)
image_group.append(max_label)  # image_group[227]
```

```
min_label = Label(font_0, text="min", color=CYAN)
min_label.anchor_point = (0, 0)
min_label.anchored_position = (1, 106)
image_group.append(min_label)  # image_group[228]

ave_label = Label(font_0, text="ave", color=YELLOW)
ave_label.anchor_point = (0, 0)
ave_label.anchored_position = (1, 76)
image_group.append(ave_label)  # image_group[229]

alarm_value = Label(font_0, text=str(ALARM_F), color=WHITE)
alarm_value.anchor_point = (0, 0)
alarm_value.anchored_position = (1, 5)
image_group.append(alarm_value)  # image_group[230]

max_value = Label(font_0, text=str(MAX_RANGE_F), color=RED)
max_value.anchor_point = (0, 0)
max_value.anchored_position = (1, 35)
image_group.append(max_value)  # image_group[231]

min_value = Label(font_0, text=str(MIN_RANGE_F), color=CYAN)
min_value.anchor_point = (0, 0)
min_value.anchored_position = (1, 95)
image_group.append(min_value)  # image_group[232]

ave_value = Label(font_0, text="---", color=YELLOW)
ave_value.anchor_point = (0, 0)
ave_value.anchored_position = (1, 65)
image_group.append(ave_value)  # image_group[233]

min_histo = Label(font_0, text="", color=None)
min_histo.anchor_point = (0, 0.5)
min_histo.anchored_position = (GRID_X_OFFSET, 121)
image_group.append(min_histo)  # image_group[234]

max_histo = Label(font_0, text="", color=None)
max_histo.anchor_point = (1, 0.5)
max_histo.anchored_position = (WIDTH - 2, 121)
image_group.append(max_histo)  # image_group[235]

range_histo = Label(font_0, text="-RANGE-", color=None)
range_histo.anchor_point = (0.5, 0.5)
range_histo.anchored_position = ((WIDTH // 2) + (GRID_X_OFFSET // 2), 121)
image_group.append(range_histo)  # image_group[236]

# ###--- PRIMARY PROCESS SETUP ---###
t1 = time.monotonic()  # Time marker: Primary Process Setup
fm1 = gc.mem_free()  # Monitor free memory
display_image = True  # Image display mode; False for histogram
display_hold = False  # Active display mode; True to hold display
display_focus = False  # Standard display range; True to focus display range
orig_max_range_f = 0  # Establish temporary range variables
orig_min_range_f = 0

# Activate display and play welcome tone
display.show(image_group)
spectrum()
update_image_frame()
flash_status("IRON", 0.75)
play_tone(880, 0.010)  # A5

# ###--- PRIMARY PROCESS LOOP ---###
while True:
    t2 = time.monotonic()  # Time marker: Acquire Sensor Data
    if display_hold:
        flash_status("-HOLD-", 0.25)
    else:
        sensor = amg8833.pixels  # Get sensor_data data
    sensor_data = ulab.numpy.array(sensor)  # Copy to narray

    t3 = time.monotonic()  # Time marker: Constrain Sensor Values
    for row in range(0, 8):
        for col in range(0, 8):
            sensor_data[col, row] = min(max(sensor_data[col, row], 0), 80)

    # Update and display alarm setting and max, min, and ave stats
    t4 = time.monotonic()  # Time marker: Display Statistics
    v_max = ulab.numpy.max(sensor_data)
    v_min = ulab.numpy.min(sensor_data)
    v_ave = ulab.numpy.mean(sensor_data)

    alarm_value.text = str(ALARM_F)
    max_value.text = str(celsius_to_fahrenheit(v_max))
    min_value.text = str(celsius_to_fahrenheit(v_min))
    ave_value.text = str(celsius_to_fahrenheit(v_ave))

    # Normalize temperature to index values and interpolate
    t5 = time.monotonic()  # Time marker: Normalize and Interpolate
    sensor_data = (sensor_data - MIN_RANGE_C) / (MAX_RANGE_C - MIN_RANGE_C)
    grid_data[::2, ::2] = sensor_data  # Copy sensor data to the grid array
    ulab_bilinear_interpolation()  # Interpolate to produce 15x15 result

    # Display image or histogram
    t6 = time.monotonic()  # Time marker: Display Image
    if display_image:
        update_image_frame(selfie=SELFIE)
    else:
        update_histo_frame()

    # If alarm threshold is reached, flash NeoPixels and play alarm tone
    if v_max >= ALARM_C:
        pixels.fill(RED)
        play_tone(880, 0.015)  # A5
        pixels.fill(BLACK)

    # See if a panel button is pressed
    buttons = panel.get_pressed()
    if buttons & BUTTON_A:  # Toggle display hold (shutter)
        play_tone(1319, 0.030)  # E6
        display_hold = not display_hold

        while buttons & BUTTON_A:
            buttons = panel.get_pressed()
            time.sleep(0.1)

    if buttons & BUTTON_B:  # Toggle image/histogram mode (display image)
        play_tone(659, 0.030)  # E5
        display_image = not display_image
        while buttons & BUTTON_B:
            buttons = panel.get_pressed()
            time.sleep(0.1)

        if display_image:
            min_histo.color = None
            max_histo.color = None
            range_histo.color = None
        else:
            min_histo.color = CYAN
            max_histo.color = RED
            range_histo.color = BLUE

    if buttons & BUTTON_SELECT:  # Toggle focus mode (display focus)
        play_tone(698, 0.030)  # F5
        display_focus = not display_focus
        if display_focus:
            # Set range values to image min/max for focused image display
            orig_min_range_f = MIN_RANGE_F
            orig_max_range_f = MAX_RANGE_F
            MIN_RANGE_F = celsius_to_fahrenheit(v_min)
            MAX_RANGE_F = celsius_to_fahrenheit(v_max)
            # Update range min and max values in Celsius
            MIN_RANGE_C = v_min
            MAX_RANGE_C = v_max
            flash_status("FOCUS", 0.2)
        else:
            # Restore previous (original) range values for image display
            MIN_RANGE_F = orig_min_range_f
            MAX_RANGE_F = orig_max_range_f
            # Update range min and max values in Celsius
            MIN_RANGE_C = fahrenheit_to_celsius(MIN_RANGE_F)
            MAX_RANGE_C = fahrenheit_to_celsius(MAX_RANGE_F)
            flash_status("ORIG", 0.2)

        while buttons & BUTTON_SELECT:
            buttons = panel.get_pressed()
            time.sleep(0.1)

    if buttons & BUTTON_START:  # Activate setup mode
        play_tone(784, 0.030)  # G5
        while buttons & BUTTON_START:
            buttons = panel.get_pressed()
            time.sleep(0.1)

        # Invoke startup helper; update alarm and range values
        ALARM_F, MAX_RANGE_F, MIN_RANGE_F = setup_mode()
        ALARM_C = fahrenheit_to_celsius(ALARM_F)
        MIN_RANGE_C = fahrenheit_to_celsius(MIN_RANGE_F)
        MAX_RANGE_C = fahrenheit_to_celsius(MAX_RANGE_F)

    t7 = time.monotonic()  # Time marker: End of Primary Process
    gc.collect()
    fm7 = gc.mem_free()
    print("*** PyBadge/Gamer Performance Stats ***")
    print(f"    define displayio:      {(t1 - t0):6.3f} sec")
    print(f"    startup free memory: {fm1/1000:6.3} Kb")
    print("")
    print(
        f" 1) data acquisition: {(t4 - t2):6.3f}      rate:  {(1 / (t4 - t2)):
5.1f} /sec"
    )
    print(f" 2) display stats:    {(t5 - t4):6.3f}")
    print(f" 3) interpolate:      {(t6 - t5):6.3f}")
    print(f" 4) display image:    {(t7 - t6):6.3f}")
    print(f"                     =======")
    print(
        f"total frame:          {(t7 - t2):6.3f} sec  rate:  {(1 / (t7 - t2)):
5.1f} /sec"
    )
    print(f"                           free memory: {fm7/1000:6.3} Kb")
    print("")
```

The Thermal Camera needs some helpers to convert back and forth between Celsius and Fahrenheit units. This file is contained in the project zip folder as thermal_cam_c onverters.py.

```python
# thermal_cam_converters.py

def celsius_to_fahrenheit(deg_c=None):  # convert C to F; round to 1 degree C
    return round(((9 / 5) * deg_c) + 32)

def fahrenheit_to_celsius(deg_f=None):  # convert F to C; round to 1 degree F
    return round((deg_f - 32) * (5 / 9))
```

The color spectrum is calculated by the `index_to_rgb` helper of the iron_spectrum.py file within the index_to_rgb folder. The helper calculates a 24-bit red, green, and blue (RGB) color value from an input value of 0 to 1.0.

```python
# iron_spectrum.py
# 2021-05-27 version 1.2
# Copyright 2021 Cedar Grove Studios
# Temperature Index to Iron Pseudocolor Spectrum RGB Converter Helper

def map_range(x, in_min, in_max, out_min, out_max):
    """
    Maps and constrains an input value from one range of values to another.
    (from adafruit_simpleio)
    :return: Returns value mapped to new range
    :rtype: float
    """
    in_range = in_max - in_min
    in_delta = x - in_min
    if in_range != 0:
        mapped = in_delta / in_range
    elif in_delta != 0:
        mapped = in_delta
    else:
        mapped = 0.5
    mapped *= out_max - out_min
    mapped += out_min
    if out_min <= out_max:
        return max(min(mapped, out_max), out_min)
    return min(max(mapped, out_max), out_min)

def index_to_rgb(index=0, gamma=0.5):
    """
    Converts a temperature index to an iron thermographic pseudocolor spectrum
    RGB value. Temperature index in range of 0.0 to 1.0. Gamma in range of
    0.0 to 1.0 (1.0=linear), default 0.5 for color TFT displays.
    :return: Returns a 24-bit RGB value
    :rtype: integer
    """

    band = index * 600  # an arbitrary spectrum band index; 0 to 600

    if band < 70:  # dark gray to blue
        red = 0.1
        grn = 0.1
        blu = (0.2 + (0.8 * map_range(band, 0, 70, 0.0, 1.0))) ** gamma
    if band >= 70 and band < 200:  # blue to violet
        red = map_range(band, 70, 200, 0.0, 0.6) ** gamma
        grn = 0.0
        blu = 1.0 ** gamma
    if band >= 200 and band < 300:  # violet to red
        red = map_range(band, 200, 300, 0.6, 1.0) ** gamma
        grn = 0.0
        blu = map_range(band, 200, 300, 1.0, 0.0) ** gamma
    if band >= 300 and band < 400:   # red to orange
        red = 1.0 ** gamma
        grn = map_range(band, 300, 400, 0.0, 0.5) ** gamma
        blu = 0.0
    if band >= 400 and band < 500:  # orange to yellow
        red = 1.0 ** gamma
        grn = map_range(band, 400, 500, 0.5, 1.0) ** gamma
        blu = 0.0
    if band >= 500:  # yellow to white
        red = 1.0 ** gamma
        grn = 1.0 ** gamma
        blu = map_range(band, 500, 580, 0.0, 1.0) ** gamma

    return (int(red * 255) << 16) + (int(grn * 255) << 8) + int(blu * 255)
```

Finally, the power-up alarm threshold, temperature display range settings, and camera orientation are contained in the thermal_cam_config.py file. All values are in degrees Fahrenheit. A `SELFIE` value of `True` adjusts the image for a front-facing camera ori entation; `False` is used for cameras facing away from the viewer.

```
# thermal_cam_config.py
# ### Alarm and range default values in Farenheit ###
ALARM_F     = 120
MIN_RANGE_F =  60
MAX_RANGE_F = 120

# ### Display characteristics
SELFIE = False  # Rear camera view; True for front view
```

After copying all the project files to the PyGamer, you'll see the camera's splash graphics and a sample of the iron color spectrum. After a couple of beeps, the thermal image will appear.

The next section shows the features of the camera and how it operates.

Because of changes to ulab with the release of CircuitPython v7.x, the thermal camera code is no longer compatible with CircuitPython version 6.3.0 or earlier.

CircuitPython Code Details

The CircuitPython code for the Thermal Camera project is contained in four files:

- code.py, the main code module,
- thermal_cam_converters.py the temperature unit conversion helper,
- iron_spectrum.py (in the index_to_rgb folder), the iron pseudocolor spectrum conversion helper, and
- thermal_cam_config.py, the start-up default parameter file.

Code Details

Let's take a walk through the code and look in more detail how each section works starting with code.py.

The main module, code.py, prepares and operates the Thermal Camera. It consists of the following major sections:

- Import and Initialize: Libraries, Devices, and Welcome Screen
- Constants: Display, Min/Max, and Alarm Threshold Values
- Helpers: Display, Buttons, and Setup Functions
- Display: Define Group Layers
- Primary Process: Setup and Loop

Things are started with importing libraries, establishing devices, and saying hello, all of which are described on the following pages.

Because of changes to ulab with the release of CircuitPython v7.x, the thermal camera code is no longer compatible with CircuitPython version 6.3.0 or earlier.

Import and Initialize

When the PyGamer's power is turned on, the code.py module first imports all the required libraries. That includes the thermal_cam_converters and index_to_rgb help er files that we'll review later.

```
import time
import board
import busio
import gc
import ulab
import displayio
import neopixel
from analogio import AnalogIn
from digitalio import DigitalInOut
from simpleio import map_range, tone
from adafruit_display_text.label import Label
from adafruit_bitmap_font import bitmap_font
from adafruit_display_shapes.rect import Rect
import adafruit_amg88xx
from gamepadshift import GamePadShift
from index_to_rgb.iron_spectrum import index_to_rgb
from thermal_cam_converters import celsius_to_fahrenheit, fahrenheit_to_celsius
from thermal_cam_config import ALARM_F, MIN_RANGE_F, MAX_RANGE_F, SELFIE
```

After importing libraries, the display and default font are instantiated, the speaker is enabled, and the on-board NeoPixels are defined.

If the PyGamer's joystick is present, the `has_joystick` flag is set to `True`. If not, then the host device is probably a PyBadge or EdgeBadge. This allows the code to work for those devices in addition to the PyGamer, interpreting the Badge D-Pad buttons like the Gamer's joystick.

```
# Instantiate display, joystick, speaker, and neopixels
display = board.DISPLAY
# Load the text font from the fonts folder
font_0 = bitmap_font.load_font("/fonts/OpenSans-9.bdf")

if hasattr(board, "JOYSTICK_X"):
    has_joystick = True  # PyGamer with joystick
    joystick_x = AnalogIn(board.JOYSTICK_X)
    joystick_y = AnalogIn(board.JOYSTICK_Y)
else:
    has_joystick = False  # PyBadge with buttons

speaker_enable = DigitalInOut(board.SPEAKER_ENABLE)
speaker_enable.switch_to_output(value=True)

pixels = neopixel.NeoPixel(board.NEOPIXEL, 5, pixel_order=neopixel.GRB)
pixels.brightness = 0.25  # Set NeoPixel brightness
pixels.fill(0x000000)  # Clear all NeoPixels
```

The PyGamer and PyBadge control buttons are connected to a hardware shift register chip that is controlled by the GamePadShift class. This section of the code defines each button's bit position within the shift register. The local GamePadShift class, panel, will be used to read the buttons in the primary process loop and setup helper.

```
# Define and instantiate front panel buttons
BUTTON_LEFT = 0b10000000
BUTTON_UP = 0b01000000
BUTTON_DOWN = 0b00100000
BUTTON_RIGHT = 0b00010000
BUTTON_SELECT = 0b00001000
BUTTON_START = 0b00000100
BUTTON_A = 0b00000010
BUTTON_B = 0b00000001

panel = GamePadShift(
    DigitalInOut(board.BUTTON_CLOCK),
    DigitalInOut(board.BUTTON_OUT),
    DigitalInOut(board.BUTTON_LATCH),
)
```

Now it's time to connect to and instantiate the AMG8833 thermal camera FeatherWing using the I2C bus connection. The I2C serial bus speed is increased from the default 100K to 400K bits per second to improve data acquisition speed and ultimately the display frame rate.

This section of the code will also work if the AMG8833 thermal camera STEMMA breakout is used in place of the FeatherWing version. STEMMA cable length may impact sensor performance, so if you encounter issues with the breakout version, try reducing the I2C bus speed to the default 100K bits per second rate.

```
# Establish I2C interface for the AMG8833 Thermal Camera
i2c = busio.I2C(board.SCL, board.SDA, frequency=400000)
amg8833 = adafruit_amg88xx.AMG88XX(i2c)
```

Next, the welcome graphics screen, thermal_cam_splash.bmp is displayed. The size of the image is scaled to fit the size of the PyGamer's display.

```
# Display splash graphics
splash = displayio.Group(scale=display.width // 160)
bitmap = displayio.OnDiskBitmap("/thermal_cam_splash.bmp")
splash.append(displayio.TileGrid(bitmap, pixel_shader=bitmap.pixel_shader))
board.DISPLAY.show(splash)
time.sleep(0.1)  # Allow the splash to display
```

Finally, the ulab (micro lab) arrays needed to hold the normalized 8x8 sensor index and the transformed 15x15 display grid index are defined. In addition, an array to hold histogram statistical data is established.

An array defined for ulab use is an `narray` type; a format different than other CircuitPython arrays. The special `narray` array type (named after a close cousin, the numpy array) is designed to support rapid array calculation and transformation.

```
# Set up ulab arrays
n = 8  # Thermal sensor grid axis size; AMG8833 sensor is 8x8
sensor_data = ulab.numpy.array(range(n * n)).reshape((n, n))  # Color index narray
grid_data = ulab.numpy.zeros(((2 * n) - 1, (2 * n) - 1))  # 15x15 color index narray
histogram = ulab.numpy.zeros((2 * n) - 1)  # Histogram accumulation narray
```

A series of numerical constants are needed to set boundaries and limits for thermal camera calculations. We'll talk about those next.

Constants

After setting up the hardware devices and saying hello, a few commonly used constants and variables are defined. These include the alarm threshold, display minimum/maximum, and camera orientation settings that were previously loaded from the thermal_cam_config.py file. Default values in Celsius are converted to Fahrenheit where needed.

```
# Convert default alarm and min/max range values from config file
ALARM_C = fahrenheit_to_celsius(ALARM_F)
MIN_RANGE_C = fahrenheit_to_celsius(MIN_RANGE_F)
MAX_RANGE_C = fahrenheit_to_celsius(MAX_RANGE_F)
```

Display width and height are retrieved from the PyGamer's board definitions. The size of individual grid image cells is calculated from the display's height and the number of displayed cells along one grid access.

By calculating the grid and cell dimensions in this manner, the code becomes adaptable for use with displays larger than the PyGamer's integral TFT display.

```
# The board's integral display size
WIDTH = display.width
HEIGHT = display.height

GRID_AXIS = (2 * n) - 1  # Number of cells along the grid x or y axis
GRID_SIZE = HEIGHT  # Maximum number of pixels for a square grid
GRID_X_OFFSET = WIDTH - GRID_SIZE  # Right-align grid with display boundary
CELL_SIZE = GRID_SIZE // GRID_AXIS  # Size of a grid cell in pixels
```

Color values are defined next. `PALETTE_SIZE` is used to select the maximum number of display colors across the iron spectrum to map to temperature values. The palette size of `100` colors was selected empirically as a value that balanced the sensor resolution of 0.5°C with the ability to visually discern objects. Increasing the number of colors beyond 160 does not improve readability and can slow the display frame rate. Fewer than `80` palette colors significantly decreases visual object detection.

The remaining color definitions are used for the various text labels and measured values in the display's sidebar. The `param_colors` list is used by the setup helper that we'll discuss in the next section.

```
PALETTE_SIZE = 100  # Number of colors in spectral palette (must be &gt; 0)

# Default colors for temperature value sidebar
BLACK = 0x000000
RED = 0xFF0000
YELLOW = 0xFFFF00
CYAN = 0x00FFFF
BLUE = 0x0000FF
WHITE = 0xFFFFFF

# Text colors for setup helper's on-screen parameters
param_colors = [("ALARM", WHITE), ("RANGE", RED), ("RANGE", CYAN)]
```

Helpers

Helpers for Display, Buttons, and Setup Functions

Helpers are used to simplify the primary loop code. The helpers:

- Play a tone to signify a button press or alert;
- Display a status message in the center of the image area;
- Load a set of test color spectrum values into the display array;
- Display and refresh the sensor image;
- Display and refresh the histogram image;
- Enlarge the 8x8 sensor data into a 15x15 display array;
- Change default parameters for temperature range and alarm threshold;
- Convert joystick movement to simulate up, down, left, and right button presses to support use with PyGamer or PyBadge boards.

`play_tone()` Helper

Using the `tone()` helper that's contained in the simpleio library, the thermal camera's `play_tone()` helper plays a musical note through the PyGamer's speaker. The frequency in Hertz and duration in seconds are passed to the helper as the parameters `freq` and `duration`.

```
# ### Helpers ###
def play_tone(freq=440, duration=0.01):
    tone(board.A0, freq, duration)
    return
```

`flash_status()` Helper

The `flash_status()` helper accepts a text string and displays it in the status area of the display. The text appears as white letters for a time specified by `duration` then as black letters for `duration` length in seconds. This is very useful for flashing a message that can be seen regardless of the background colors, especially handy while displaying a sensor image.

```
def flash_status(text="", duration=0.05):  # Flash status message once
    status_label.color = WHITE
    status_label.text = text
    time.sleep(duration)
    status_label.color = BLACK
    time.sleep(duration)
    status_label.text = ""
    return
```

`spectrum()` Helper

Initially used to help debug the iron spectrum conversion code, the `spectrum()` helper loads the image grid array with a sequence of index values that sweep through the colors of the thermal camera's visual pseudocolor spectrum. This helper is used in conjunction with the `update_image_frame()` helper to display a sample of the default spectrum upon startup.

```
def spectrum():  # Load a test spectrum into the grid_data array
    for row in range(0, GRID_AXIS):
        for col in range(0, GRID_AXIS):
            grid_data[row][col] = ((row * GRID_AXIS) + col) * 1 / 235
    return
```

`update_image_frame()` Helper

The `update_image_frame()` helper looks through a list of 225 indexed color values stored by row and column in the `grid_data` list. The helper converts the color index into a displayable RGB color value and updates the fill color of the corresponding display cell.

To save processing time and improve image frame rate, a cell is only updated if the calculated RGB value has changed from one frame to the next.

```
def update_image_frame(selfie=False):  # Get camera data and update display
    for row in range(0, GRID_AXIS):
        for col in range(0, GRID_AXIS):
            if selfie:
                color_index = grid_data[GRID_AXIS - 1 - row][col]
            else:
                color_index = grid_data[GRID_AXIS - 1 - row][GRID_AXIS - 1 - col]
            color = index_to_rgb(round(color_index * PALETTE_SIZE, 0) /
PALETTE_SIZE)
            if color != image_group[((row * GRID_AXIS) + col)].fill:
                image_group[((row * GRID_AXIS) + col)].fill = color
    return
```

`update_histo_frame()` Helper

The `update_histo_frame()` helper collects a distribution of 15 temperature sub-ranges within the current temperature display range (one for each color) and displays a histogram of relative temperature values. The helper scans all 225 sensor color index values in the `grid_data` array and counts the number of times a value falls within one of 15 sub-ranges.

When invoked, the helper displays the histogram range legend values and clears the `histogram` array used to accumulate the 15 sub-range values. After collecting the histogram data from the array, the largest sub-range value is stored in the `histo_scale` variable is used to scale the results when the histogram is displayed.

The second part of the helper updates the image area to display the histogram as a series of vertical bars with height proportional to the accumulated sub-range value. The display update starts at the upper left of the display's image area and works down to the lower right. Each cell is filled with a color that corresponds to the color index value. The remainder of boxes in the histogram display area are colored black if not used to build a histogram bar.

```
def update_histo_frame():  # Calculate and display histogram
    min_histo.text = str(MIN_RANGE_F)  # Display histogram legend
    max_histo.text = str(MAX_RANGE_F)

    histogram = ulab.numpy.zeros(GRID_AXIS)  # Clear histogram accumulation array
    for row in range(0, GRID_AXIS):  # Collect camera data and calculate histo
        for col in range(0, GRID_AXIS):
            histo_index = int(map_range(grid_data[col, row], 0, 1, 0, GRID_AXIS - 
1))
            histogram[histo_index] = histogram[histo_index] + 1

    histo_scale = ulab.numpy.max(histogram) / (GRID_AXIS - 1)
    if histo_scale &lt;= 0:
        histo_scale = 1

    for col in range(0, GRID_AXIS):  # Display histogram
        for row in range(0, GRID_AXIS):
            if histogram[col] / histo_scale &gt; GRID_AXIS - 1 - row:
                image_group[((row * GRID_AXIS) + col)].fill = index_to_rgb(
                    round((col / GRID_AXIS), 3)
                )
            else:
                image_group[((row * GRID_AXIS) + col)].fill = BLACK
    return
```

`ulab_bilinear_interpolation()` Helper

The `ulab_bilinear_interpolation()` helper utilizes ulab array calculations to find values for cells in the 225-cell grid_data array that fall between the 64 known sensor element values. First, the even rows are scanned, assigning the average of the adjacent known cells to each unknown cell. Next, odd rows are scanned, assigning the average of the values above and below to every cell in the row. See the section, 1-2-3s of Bilinear Interpolation for the details of the interpolation method.

```
def ulab_bilinear_interpolation():  # 2x bilinear interpolation
    # Upscale sensor data array; by @v923z and @David.Glaude
    grid_data[1::2, ::2] = sensor_data[:-1, :]
    grid_data[1::2, ::2] += sensor_data[1:, :]
    grid_data[1::2, ::2] /= 2
    grid_data[::, 1::2] = grid_data[::, :-1:2]
    grid_data[::, 1::2] += grid_data[::, 2::2]
    grid_data[::, 1::2] /= 2
    return
```

`setup_mode()` Helper

The `setup_mode()` helper pauses normal operation and collects user input to set alarm threshold and display range min/max values. During the Setup mode, the display's average value and label are blanked.

The joystick or PyBadge D-Pad is used to select the parameter to change and to increase or decrease the parameter value. The HOLD button acts as the parameter select button. Pressing the SET button at any time during the Setup mode will exit back to the primary process loop.

The first task is to temporarily display a status message that indicates the camera is in the Setup mode. The display's average value and label are blanked and the measured maximum and minimum values are replaced with the current maximum and minimum display range values (`MAX_RANGE_F` and `MIN_RANGE_F`).

After waiting a bit for the status message to be read and prior to watching for button and joystick changes, the index pointer (`param_index`) is reset to point to the alarm threshold parameter.

```
def setup_mode():  # Set alarm threshold and minimum/maximum range values
    status_label.color = WHITE
    status_label.text = "-SET-"

    ave_label.color = BLACK  # Turn off average label and value display
    ave_value.color = BLACK

    max_value.text = str(MAX_RANGE_F)  # Display maximum range value
    min_value.text = str(MIN_RANGE_F)  # Display minimum range value

    time.sleep(0.8)  # Show SET status text before setting parameters
    status_label.text = ""  # Clear status text

    param_index = 0  # Reset index of parameter to set
```

The following is the meat of the setup process. Before moving on to choosing which parameter to set, the process waits until the SET button has been released.

As long as the HOLD (select) or the SET (setup mode exit) buttons have not been pressed, the code loops. During the loop, the joystick is watched using the `move_buttons()` helper. If the joystick is moved down, the parameter index is incremented, pointing to the next parameter. If moved up, the index will point to the previous parameter. The parameter label text flashes black and white, indicating which parameter is ready to be changed.

In the `image_group` list (that is defined later in the display portion of the code just before the primary process loop), the three parameter text labels for alarm, maximum, and minimum are sequentially positioned in the list:

- Alarm text label --> `image_group[226]`
- Maximum text label --> `image_group[227]`
- Minimum text label --> `image_group[228]`

Using an indexed position in `image_group` for the parameters makes it simpler to sequentially step from one parameter to the next.

```
# Select parameter to set

buttons = panel.get_pressed()
while not buttons &amp; BUTTON_START:
    buttons = panel.get_pressed()
    while (not buttons &amp; BUTTON_A) and (not buttons &amp; BUTTON_START):
        up, down = move_buttons(joystick=has_joystick)
        if up:
            param_index = param_index - 1
        if down:
            param_index = param_index + 1
        param_index = max(0, min(2, param_index))
        status_label.text = param_colors[param_index][0]
        image_group[param_index + 226].color = BLACK
        status_label.color = BLACK
        time.sleep(0.25)
        image_group[param_index + 226].color = param_colors[param_index][1]
        status_label.color = WHITE
        time.sleep(0.25)
        buttons = panel.get_pressed()
```

After the HOLD button is pressed and released, the selected parameter, represented by the value of `param_index`, can be changed.

The selected parameter value is incrementally changed by the joystick's up and down movements as provided to this helper from the `move_buttons()` helper. The new value is checked against and limited to the sensor's factory min/max limits (`MIN_SENSOR_F`, `MAX_SENSOR_F`).

In the `image_group` list the three parameter value labels for alarm, maximum, and minimum are sequentially positioned in the list:

- Alarm value label --> `image_group[230]`
- Maximum value label --> `image_group[231]`
- Minimum value label --> `image_group[232]`

The value label for the selected parameter is changed and displayed.

Meanwhile, a flashing status message indicates which type of parameter is being changed, either the alarm or one of the range values.

When the desired value is reached and the HOLD (select) button is pressed, the Setup process continues back to the parameter select mode.

```
buttons = panel.get_pressed()
if buttons &amp; BUTTON_A:  # Hold (button A) pressed
    play_tone(1319, 0.030)  # E6
while buttons &amp; BUTTON_A:  # Wait for button release
    buttons = panel.get_pressed()
    time.sleep(0.1)

# Adjust parameter value
param_value = int(image_group[param_index + 230].text)
buttons = panel.get_pressed()
while (not buttons &amp; BUTTON_A) and (not buttons &amp; BUTTON_START):
    up, down = move_buttons(joystick=has_joystick)
    if up:
        param_value = param_value + 1
    if down:
        param_value = param_value - 1
    param_value = max(32, min(157, param_value))
    image_group[param_index + 230].text = str(param_value)
    image_group[param_index + 230].color = BLACK
    status_label.color = BLACK
    time.sleep(0.05)
    image_group[param_index + 230].color = param_colors[param_index][1]
    status_label.color = WHITE
    time.sleep(0.2)
    buttons = panel.get_pressed()

buttons = panel.get_pressed()
if buttons &amp; BUTTON_A:  # Button A pressed
    play_tone(1319, 0.030)  # E6
while buttons &amp; BUTTON_A:  # Wait for button release
    buttons = panel.get_pressed()
    time.sleep(0.1)
```

If SET is pressed instead of HOLD, the Setup process prepares to exit back to the primary process loop.

```
# Exit setup process
buttons = panel.get_pressed()
if buttons &amp; BUTTON_START:  # Start button pressed
    play_tone(784, 0.030)  # G5
while buttons &amp; BUTTON_START:  # wait for button release
    buttons = panel.get_pressed()
    time.sleep(0.1)
```

Before exiting, a resumption status message is displayed and the display of the average label and value are restored.

Finally, the text strings that may have changed during the Setup process are converted to integer numeric values and returned to the primary process loop.

```
status_label.text = "RESUME"
time.sleep(0.5)
status_label.text = ""

# Display average label and value
ave_label.color = YELLOW
ave_value.color = YELLOW
return int(alarm_value.text), int(max_value.text), int(min_value.text)
```

`move_buttons()` Helper

The `move_buttons()` helper first resets the variables that indicate joystick movement or D-Pad button presses. If the `joystick` argument is `True`, the joystick movements beyond set thresholds are represented as button depressions. For example, a value for `panel.joystick[1]` of less than 20000 means that the joystick was moved upwards; greater than 44000 indicates downward movement.

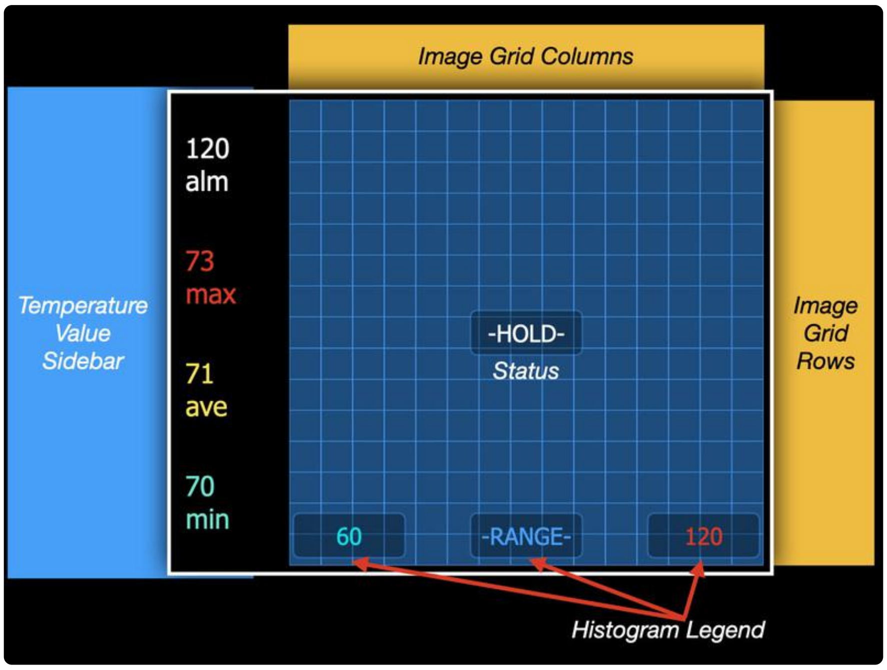
If the `joystick` argument is `False`, instead of watching the joystick, the D-Pad buttons are checked to see if any are depressed.

Finally, the movement indicating values are returned to the calling module.

```
def move_buttons(joystick=False):  # Read position buttons and joystick
    move_u = move_d = False
    if joystick:  # For PyGamer: interpret joystick as buttons
        if joystick_y.value &lt; 20000:
            move_u = True
        elif joystick_y.value &gt; 44000:
            move_d = True
    else:  # For PyBadge read the buttons
        buttons = panel.get_pressed()
        if buttons &amp; BUTTON_UP:
            move_u = True
        if buttons &amp; BUTTON_DOWN:
            move_d = True
    return move_u, move_d
```

After the helpers are defined, we move on to specifying the text and graphic features of the display.

Display



Define Display Group Layers

Within CircuitPython's `displayio` library, a display group is a list of label or graphic attributes that are defined for each object of the display. This section of the Thermal Camera's primary process module defines the `image_group` display group that the camera will use to show measured values, the sensor image or histogram, status message, and the histogram legend.

The camera's display group, `image_group`, consists of layered objects. The first 225 objects of the display make up the colored cells used for the image grid area.

The status message label comes next, followed by the values and labels in the display sidebar area. Finally, objects that make up the histogram legend top off the stack of display objects in `image_group`.

The objects and their attributes are appended to the `image_group` list one-at-a-time when first defined. When appended to `image_group`, the attributes of each object are defined. For example, the alarm label alm is defined as a Label object with attributes that include the label's font, text contents, text color, and maximum character count (glyphs):

```
status_label = Label(font_0, text="", color=None)
status_label.anchor_point = (0.5, 0.5)
status_label.anchored_position = ((WIDTH // 2) + (GRID_X_OFFSET // 2), HEIGHT // 2)
image_group.append(status_label)  # image_group[225]
```

The anchored_position (x/y coordinates) and anchor_point (left/right/center justification) of the alarm label on the PyGamer's display screen are calculated to appear in the center of the image grid area. Other display label and value positions were determined and fine-tuned empirically. After defining the display object's attributes, it is appended to the `image_group` display group.

This process is repeated, starting from the back of the display and progressing towards the front, as each new object is appended to the display group.

Define the Image Group

After playing a couple of musical tones and storing the elapsed time to mark the beginning of the display definition process, the `image_group` definition list for display group objects comes next. The `scale` argument adjusts image group element position and size parameters. For the PyGamer's display, no adjustment is required so `scale=1`.

The time marker `t0` along with seven other process time markers will be reported at the end of each displayed frame to calculate thermal camera code performance. This marker establishes the time that the display group definition phase began.

```
play_tone(440, 0.1)  # A4
play_tone(880, 0.1)  # A5

# ### Define the display group ###
t0 = time.monotonic()  # Time marker: Define Display Elements
image_group = displayio.Group(scale=1)
```

Define the Thermal Image Display Group Layers

Next, the 225 square cells used to represent sensor array temperatures are defined and appended to `image_group`. Two `for` loops are used to step through each column and row of cells. Each square is defined as a rectangle with width and height equal to `CELL_SIZE`. No color attribute is defined for the cell, making it transparent -- for now.

```
# Define the foundational thermal image grid cells; image_group[0:224]
#   image_group[#]=(row * GRID_AXIS) + column
for row in range(0, GRID_AXIS):
    for col in range(0, GRID_AXIS):
        cell_x = (col * CELL_SIZE) + GRID_X_OFFSET
        cell_y = row * CELL_SIZE
        cell = Rect(
            x=cell_x,
            y=cell_y,
            width=CELL_SIZE,
            height=CELL_SIZE,
            fill=None,
            outline=None,
            stroke=0,
        )
        image_group.append(cell)
```

Define the Text Label Display Group Layers

Finally, the remaining text objects that display legends and values are defined and appended to the `image_group` display group.

For each object, the label name is defined along with the font, text contents, font color, and the maximum number of characters (glyphs) to display. Next, the object's `anchor_point` (justification) and `anchored_position` (x/y coordinates) attributes are defined. After the attributes are defined, each object is appended to `image_group`.

```
# Define labels and values
status_label = Label(font_0, text="", color=None)
status_label.anchor_point = (0.5, 0.5)
status_label.anchored_position = ((WIDTH // 2) + (GRID_X_OFFSET // 2), HEIGHT // 2)
image_group.append(status_label)  # image_group[225]

alarm_label = Label(font_0, text="alm", color=WHITE)
alarm_label.anchor_point = (0, 0)
alarm_label.anchored_position = (1, 16)
image_group.append(alarm_label)  # image_group[226]

max_label = Label(font_0, text="max", color=RED)
max_label.anchor_point = (0, 0)
max_label.anchored_position = (1, 46)
image_group.append(max_label)  # image_group[227]

min_label = Label(font_0, text="min", color=CYAN)
min_label.anchor_point = (0, 0)
min_label.anchored_position = (1, 106)
image_group.append(min_label)  # image_group[228]

ave_label = Label(font_0, text="ave", color=YELLOW)
ave_label.anchor_point = (0, 0)
ave_label.anchored_position = (1, 76)
image_group.append(ave_label)  # image_group[229]

alarm_value = Label(font_0, text=str(ALARM_F), color=WHITE)
alarm_value.anchor_point = (0, 0)
alarm_value.anchored_position = (1, 5)
image_group.append(alarm_value)  # image_group[230]
```

```
max_value = Label(font_0, text=str(MAX_RANGE_F), color=RED)
max_value.anchor_point = (0, 0)
max_value.anchored_position = (1, 35)
image_group.append(max_value)  # image_group[231]

min_value = Label(font_0, text=str(MIN_RANGE_F), color=CYAN)
min_value.anchor_point = (0, 0)
min_value.anchored_position = (1, 95)
image_group.append(min_value)  # image_group[232]

ave_value = Label(font_0, text="---", color=YELLOW)
ave_value.anchor_point = (0, 0)
ave_value.anchored_position = (1, 65)
image_group.append(ave_value)  # image_group[233]

min_histo = Label(font_0, text="", color=None)
min_histo.anchor_point = (0, 0.5)
min_histo.anchored_position = (GRID_X_OFFSET, 121)
image_group.append(min_histo)  # image_group[234]

max_histo = Label(font_0, text="", color=None)
max_histo.anchor_point = (1, 0.5)
max_histo.anchored_position = (WIDTH - 2, 121)
image_group.append(max_histo)  # image_group[235]

range_histo = Label(font_0, text="-RANGE-", color=None)
range_histo.anchor_point = (0.5, 0.5)
range_histo.anchored_position = ((WIDTH // 2) + (GRID_X_OFFSET // 2), 121)
image_group.append(range_histo)  # image_group[236]
```

Whew. We've imported libraries, listed the essential constants, established some helpers, and defined the elements of the display. After a quick aside to talk about how a display group can be accessed, it'll be time to bring it all together in the thermal camera's primary process.

Fun Facts about Display Group Objects

Objects and their attributes in the display group can be accessed in two ways. The most commonly-used method is to assign a name attribute to the object. For example, the text of the status message label can be set to display the text WELCOME in this manner:

```
status_label.text = "WELCOME"
```

Objects in `image_group` can also be accessed by their indexed position in the display group. An index of 0 is the back-most object in the display group; the highest index value is front-most. The status message text can also be changed using the index:

```
image_group[225].text = "WELCOME"
```

The Thermal Camera uses both techniques. Named display objects are used whenever possible to clearly identify which object is being changed. For efficiency, however, the index position method is used when stepping through a sequence of `image_group` objects, as when displaying the 225 colored cells for the sensor image. The index position method is also used by the `setup_mode()` helper when moving on-screen to select the alarm, maximum, or minimum parameter.

Primary Process

We've finally arrived at the portion of the code that controls the Thermal Camera's primary process. Before getting started in the main process, we need to define a couple of things to get the camera ready for looping.

After taking a performance time stamp at the beginning with the time marker variable `t1`, the default display mode flags and the initial ranges values are established. Next, the `image_group` display group is activated, a sample of the iron spectrum colors is displayed for 0.75 seconds, and a "ready" tone is sounded.

```
# ###--- PRIMARY PROCESS SETUP ---###
t1 = time.monotonic()  # Time marker: Primary Process Setup
fm1 = gc.mem_free()  # Monitor free memory
display_image = True  # Image display mode; False for histogram
display_hold = False  # Active display mode; True to hold display
display_focus = False  # Standard display range; True to focus display range
orig_max_range_f = 0  # Establish temporary range variables
orig_min_range_f = 0

# Activate display and play welcome tone
display.show(image_group)
spectrum()
update_image_frame()
flash_status("IRON", 0.75)
play_tone(880, 0.010)  # A5
```

Primary Process Loop, Part I

Because of its complexity, the primary process loop is divided into two sections to make it easier to understand. The first section fetches the image sensor's data, analyzes and displays the sensor data as an image or histogram, and checks to see if any of the sensor elements have exceeded the alarm threshold. The second section looks at the buttons and joystick to select display modes and to run the Setup helper.

Retrieve Sensor Data, Display Image or Histogram, Check Alarm Threshold

At time `t2`, the image sensor's 64 data elements are moved into the `sensor` list when the `display_hold` flag is false; otherwise a "-HOLD-" status message is displayed. To allow the sensor's temperature data to be used by the ultra fast ulab interpolation helper, the sensor list is copied into a ulab-compatible array, `sensor_data`.

Starting at time marker `t3`, the temperature value of each element of the the `sensor_data` array is constrained to the valid temperature range of the AMG8833 sensor, 0°C to 80°C.

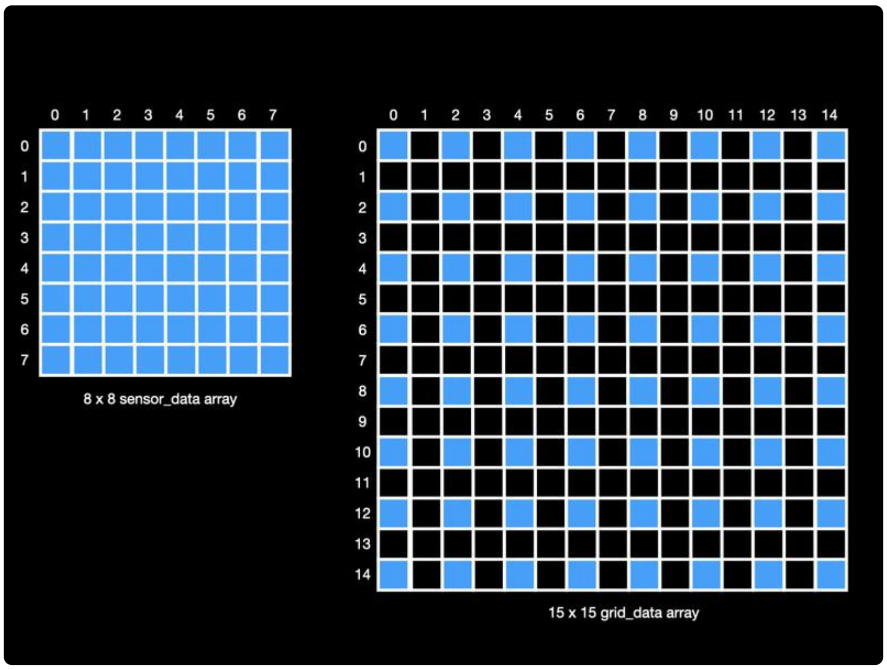
```
# ###--- PRIMARY PROCESS LOOP ---###
while True:
    t2 = time.monotonic()  # Time marker: Acquire Sensor Data
    if display_hold:
        flash_status("-HOLD-", 0.25)
    else:
        sensor = amg8833.pixels  # Get sensor_data data
    sensor_data = ulab.numpy.array(sensor)  # Copy to narray

    t3 = time.monotonic()  # Time marker: Constrain Sensor Values
    for row in range(0, 8):
        for col in range(0, 8):
            sensor_data[col, row] = min(max(sensor_data[col, row], 0), 80)
```

Before the temperature data in the `sensor_data` array is altered for the interpolation process, the minimum, maximum, and average values of the array are captured in the variables `v_min`, `v_max`, and `v_ave`. The display is then updated with the Fahrenheit values of the current alarm setting as well as the converted minimum, maximum, and average Fahrenheit values. This section starts at time marker `t4`.

```
# Update and display alarm setting and max, min, and ave stats
t4 = time.monotonic()  # Time marker: Display Statistics
v_max = ulab.numpy.max(sensor_data)
v_min = ulab.numpy.min(sensor_data)
v_ave = ulab.numpy.mean(sensor_data)

alarm_value.text = str(ALARM_F)
max_value.text = str(celsius_to_fahrenheit(v_max))
min_value.text = str(celsius_to_fahrenheit(v_min))
ave_value.text = str(celsius_to_fahrenheit(v_ave))
```

It's time to use bilinear interpolation to enlarge the sensor's 64 elements (8x8) into a grid of 225 elements (15x 15). The interpolation process commences at time marker `t5` and begins by converting each of the 64 temperature values in the `sensor_data` array to a normalized value that ranges from 0.0 to 1.0 depending on the recorded temperature value as compared to the currently displayed temperature range. For example, a normalized value of 0.0 represents a temperature at the minimum of the currently displayed range, `MIN_RANGE_C;` a normalized value of 1.0 represents the maximum of the range, `MAX_RANGE_C`. Normalizing the temperature values makes it easier to display a full range of pseudocolors for any temperature range that the FOCUS mode may invoke.

After normalization, the known values from the `sensor_data` array are copied into the `grid_data` array, starting at [0, 0], the upper left corner, and placed into the cells of the even columns. The odd rows in the `grid_data` array are initially left blank. Once the grid_data array is filled, the missing values are replaced with the results of the interpolation helper, `ulab_bilinear_interpolation()`. Refer to the 1-2-3s of Bilinear Interpolation section for details of the image enlargement process.



```
# Normalize temperature to index values and interpolate
t5 = time.monotonic()  # Time marker: Normalize and Interpolate
sensor_data = (sensor_data - MIN_RANGE_C) / (MAX_RANGE_C - MIN_RANGE_C)
grid_data[::2, ::2] = sensor_data  # Copy sensor data to the grid array
ulab_bilinear_interpolation()  # Interpolate to produce 15x15 result
```

This section checks the `display_image` flag to see whether to display a sensor image or histogram. If `display_image` is `True`, the `update_image_frame()` helper is used to display the data contained in the `grid_data` array as an thermal image. When `False`, `update_histo_frame()` displays the data as a histogram distribution.

```
# Display image or histogram
t6 = time.monotonic()  # Time marker: Display Image
if display_image:
    update_image_frame(selfie=SELFIE)
else:
    update_histo_frame()
```

The next step in the primary process loop checks the returned maximum value against the current alarm threshold (`ALARM_C`). If the threshold is met or exceeded, the NeoPixels flash red and a warning tone is played through the speaker.

```
# If alarm threshold is reached, flash NeoPixels and play alarm tone
if v_max &gt;= ALARM_C:
    pixels.fill(RED)
    play_tone(880, 0.015)  # A5
    pixels.fill(BLACK)
```

Primary Process Loop, Part II

The second portion of the primary process loop checks to see if any buttons have been pressed and sets the appropriate flags to select camera functions. This section also watches the SET button to activate the `setup_mode()` helper to permit changing camera parameters. Finally, all the time markers are analyzed and a code performance report is printed.

Watch the Buttons and Change Parameters

First the HOLD button (the PyGamer's BUTTON_A) is checked. If pressed, an acknowl edgment tone is sounded and the boolean `display_hold` parameter is toggled to the opposite state. After `display_hold` is toggled, the code waits until the button is released.

```
# See if a panel button is pressed
buttons = panel.get_pressed()
if buttons &amp; BUTTON_A:  # Toggle display hold (shutter)
    play_tone(1319, 0.030)  # E6
    display_hold = not display_hold

    while buttons &amp; BUTTON_A:
        buttons = panel.get_pressed()
        time.sleep(0.1)
```

If the IMAGE button (BUTTON_B) is pressed, a tone is played and the boolean `displ ay_image` value is toggled to the opposite state. After waiting until the button is released, the variable `display_image` is checked. If `True` (display image), the histogram legend colors are disabled. If False (display histogram), the histogram legend colors are enabled.

```
if buttons &amp; BUTTON_B:  # Toggle image/histogram mode (display image)
    play_tone(659, 0.030)  # E5
    display_image = not display_image
    while buttons &amp; BUTTON_B:
        buttons = panel.get_pressed()
        time.sleep(0.1)

    if display_image:
        min_histo.color = None
        max_histo.color = None
        range_histo.color = None
    else:
        min_histo.color = CYAN
        max_histo.color = RED
        range_histo.color = BLUE
```

When the FOCUS button (PyGamer BUTTON_SELECT) is pressed, a tone is played and the boolean display_focus value is toggled to the opposite state.

If display_focus is True, the default display range values MIN_RANGE_F and MAX_RANGE_F are stored in temporary variables orig_min_range_f and orig_max_range_f. The display range is then updated with the current minimum and maximum values v_min and v_max. This change causes the color spectrum of the display to conform to the new range. The status "FOCUS" is then flashed on the display.

If display_focus is False, the previously stored range variables become the current display range. The display range reverts to the original default values and the colors match the original range. The display flashes the "ORIG" status message.

Finally, the code waits until the button is released before moving on.

```
if buttons &amp; BUTTON_SELECT:  # Toggle focus mode (display focus)
    play_tone(698, 0.030)  # F5
    display_focus = not display_focus
    if display_focus:
        # Set range values to image min/max for focused image display
        orig_min_range_f = MIN_RANGE_F
        orig_max_range_f = MAX_RANGE_F
        MIN_RANGE_F = celsius_to_fahrenheit(v_min)
        MAX_RANGE_F = celsius_to_fahrenheit(v_max)
        # Update range min and max values in Celsius
        MIN_RANGE_C = v_min
        MAX_RANGE_C = v_max
        flash_status("FOCUS", 0.2)
    else:
        # Restore previous (original) range values for image display
        MIN_RANGE_F = orig_min_range_f
        # Update range min and max values in Celsius
        MIN_RANGE_C = fahrenheit_to_celsius(MIN_RANGE_F)
        MAX_RANGE_C = fahrenheit_to_celsius(MAX_RANGE_F)
        flash_status("ORIG", 0.2)

    while buttons &amp; BUTTON_SELECT:
        buttons = panel.get_pressed()
        time.sleep(0.1)
```

When the SET button (BUTTON_START) is pressed, a tone is played and the code waits for the button to be released. The `setup_mode()` helper is then executed returning new values for `ALARM_F`, `MAX_RANGE_F`, and `MIN_RANGE_F` that are promptly converted to Celsius.

```
if buttons &amp; BUTTON_START:  # Activate setup mode
    play_tone(784, 0.030)  # G5
    while buttons &amp; BUTTON_START:
        buttons = panel.get_pressed()
        time.sleep(0.1)

    # Invoke startup helper; update alarm and range values
    ALARM_F, MAX_RANGE_F, MIN_RANGE_F = setup_mode()
    ALARM_C = fahrenheit_to_celsius(ALARM_F)
    MIN_RANGE_C = fahrenheit_to_celsius(MIN_RANGE_F)
    MAX_RANGE_C = fahrenheit_to_celsius(MAX_RANGE_F)
```

Before looping to the start of the primary process loop to display the next image, the time marker `t7` is used to record the time at the end of the code loop. The code performance time markers are analyzed and performance results printed to the REPL's serial output.

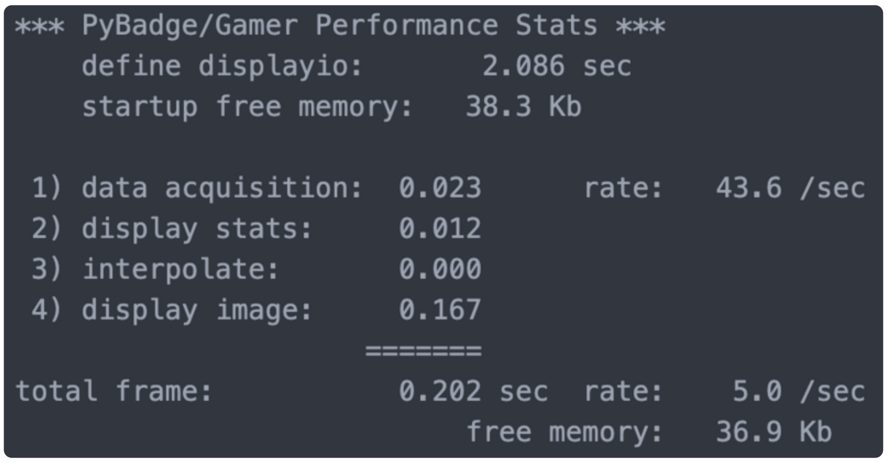
```
t7 = time.monotonic()  # Time marker: End of Primary Process
    gc.collect()
    fm7 = gc.mem_free()
    print("*** PyBadge/Gamer Performance Stats ***")
    print(f"    define displayio:      {(t1 - t0):6.3f} sec")
    print(f"    startup free memory: {fm1/1000:6.3} Kb")
    print("")
    print(
        f" 1) data acquisition: {(t4 - t2):6.3f}      rate:  {(1 / (t4 - t2)):
5.1f} /sec"
    )
    print(f" 2) display stats:    {(t5 - t4):6.3f}")
    print(f" 3) interpolate:      {(t6 - t5):6.3f}")
    print(f" 4) display image:    {(t7 - t6):6.3f}")
    print(f"                     =======")
    print(
        f"total frame:          {(t7 - t2):6.3f} sec  rate:  {(1 / (t7 - t2)):
5.1f} /sec"
    )
    print(f"                           free memory: {fm7/1000:6.3} Kb")
    print("")
```

Each phase of code execution is calculated from the time markers:

- define displayio: the time in seconds to define the displayio elements before the primary loop begins
- 1) data acquisition: elapsed time and the calculated ideal rate for acquiring and conditioning sensor data

- 2) display stats: update the on-screen alarm, min, max, and average values
- 3) interpolate: normalize the 8 x 8 sensor data and enlarge the image to a 15 x 15
- 4) display image: using displayio, refresh the screen image
- total frame: the elapsed time to generate a frame (steps 1 through 4); also includes the frame-per-second rate of the just-displayed frame

The PyGamer displays approximately 4 to 5 image frames per second depending on the quantity of changed display grid elements from one frame to the next. Here's a screen shot of a typical performance report:



```
*** PyBadge/Gamer Performance Stats ***
    define displayio:       2.086 sec
    startup free memory:   38.3 Kb

 1) data acquisition:  0.023      rate:   43.6 /sec
 2) display stats:     0.012
 3) interpolate:       0.000
 4) display image:     0.167
                     =======
total frame:           0.202 sec  rate:    5.0 /sec
                           free memory:   36.9 Kb
```

Does performance reporting slow performance? Yes, but not significantly. Capturing the time markers and printing the performance report adds less than 0.005 seconds to each frame. That's a frame rate performance impact of approximately 2.5%.

Refer to the Performance Monitoring section for a further discussion of the method used and comparison of the Thermal Camera code performance on a variety of Adafruit development boards.

Other Modules

Startup Configuration

When imported, the thermal_cam_config.py file provides the Thermal Camera's initial power-up alarm threshold as well as minimum and maximum display range values. The power-up configuration parameters can be changed by editing the file with your favorite text editor.

Values are in degrees Fahrenheit.

```
# thermal_cam_config.py
# ### Alarm and range default values in Farenheit ###
ALARM_F = 120
MIN_RANGE_F = 60
MAX_RANGE_F = 120

# ### Display characteristics
SELFIE = False  # Rear camera view; True for front view
```

Converter Helpers

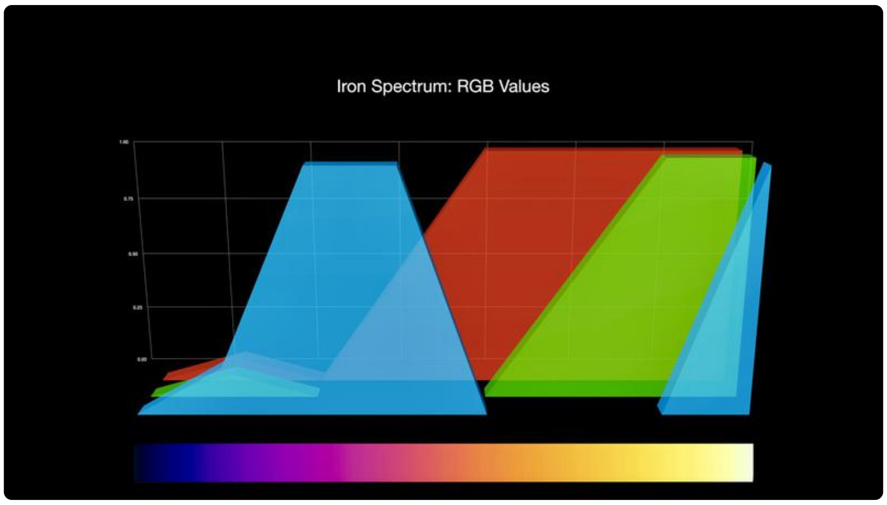
The thermal_cam_converters.py module consists of two temperature converters, one for Celsius to Fahrenheit and the other for Fahrenheit to Celsius. The value to be converted is passed as an argument to the appropriate helper. Because the Thermal Camera's sensor has limited accuracy, a rounded integer value is returned.

```
# thermal_cam_converters.py


def celsius_to_fahrenheit(deg_c=None):  # convert C to F; round to 1 degree C
    return round(((9 / 5) * deg_c) + 32)


def fahrenheit_to_celsius(deg_f=None):  # convert F to C; round to 1 degree F
    return round((deg_f - 32) * (5 / 9))
```

Pseudocolor Spectrum Converter

Showing a visual image of temperatures requires the use of a spectrum of gradual color changes that correspond to the range of temperatures to be displayed. Since the colors are representative of the measured temperature, the collection of colors is called a pseudocolor spectrum. The pseudocolor spectrum of heated iron was used for this thermographic imaging project.

The color of a heated iron bar as a temperature scale originated with blacksmiths to determine when the metal can be shaped, joined, or hardened. Cold iron starts as a blueish color that changes to purple, red, orange, yellow, and eventually glows white-hot as the temperature increases.



For this project, a helper was created that converts a temperature index value of 0.0 to 1.0 to the RGB values needed to create the iron pseudocolor spectrum on the PyGamer's color TFT display.

Within the `iron_spectrum.py` file are two helpers, the `primary index_to_rgb()` code that converts the index to the corresponding RGB value and a `map_range()` helper used to calculate values within the i`ndex_to_rgb()` helper.

map_range()

The `map_range()` helper accepts an input value inside of a specified input range and returns a proportional value constrained by a specified output range. The input value `x` is contained in the range `in_min` to `in_max`. The returned output value is constrained to the range `out_min` and `out_max`.

```
def map_range(x, in_min, in_max, out_min, out_max):
    """
    Maps and constrains an input value from one range of values to another.
    (from adafruit_simpleio)
    :return: Returns value mapped to new range
    :rtype: float
    """
    in_range = in_max - in_min
    in_delta = x - in_min
    if in_range != 0:
        mapped = in_delta / in_range
    elif in_delta != 0:
        mapped = in_delta
    else:
        mapped = 0.5
    mapped *= out_max - out_min
    mapped += out_min
    if out_min &lt;= out_max:
        return max(min(mapped, out_max), out_min)
    return min(max(mapped, out_max), out_min)
```

index_to_rgb()

The `index_to_rgb` helper accepts an `index` input value from 0.0 to 1.0, returning a 24-bit RGB color value. Within this helper, the input value is converted to an internal spectrum, represented by the `band` variable. The spectrum band value ranges from 0 to 600, arbitrarily selected to provide a simple way to understand the gradual color shifting within the spectrum.

Within each sub-band, the red, green, and blue components are established and calculated. In the red to orange sub-band (300 to 399) for example, the red value is held at 1.0, blue at 0.0, where green changes proportionally from 0.0 to 0.5 as the band value increases.

```
if band &gt;= 300 and band &lt; 400:  # red to orange
    red = 1.0 ** gamma
    grn = map_range(band, 300, 400, 0.0, 0.5) ** gamma
    blu = 0.0
```

The `gamma` parameter is applied to improve the visual perception of the color spectrum, improving the continuity or smoothness of the spectrum, helping to compensate for the differences between human color perception and the source display's rendition of color. For the PyGamer's TFT display, a gamma value of 0.5 works nicely. A gamma value of 1.0 seems to work the best with the MatrixPortal's 32 x 64 RGB LED display.

Finally, the resulting 0.0 to 1.0 values for red, green, and blue determined within a sub-band are used to calculate the returned 24-bit RGB value.

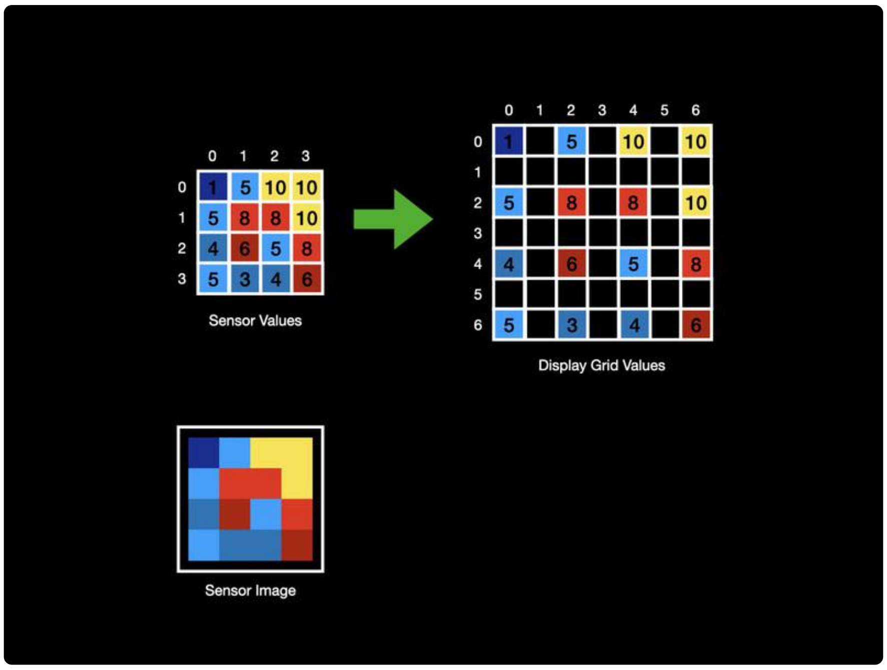
```
def index_to_rgb(index=0, gamma=0.5):
    """
    Converts a temperature index to an iron thermographic pseudocolor spectrum
    RGB value. Temperature index in range of 0.0 to 1.0. Gamma in range of
    0.0 to 1.0 (1.0=linear), default 0.5 for color TFT displays.
    :return: Returns a 24-bit RGB value
    :rtype: integer
    """

    band = index * 600  # an arbitrary spectrum band index; 0 to 600

    if band &lt; 70:  # dark gray to blue
        red = 0.1
        grn = 0.1
        blu = (0.2 + (0.8 * map_range(band, 0, 70, 0.0, 1.0))) ** gamma
    if band &gt;= 70 and band &lt; 200:  # blue to violet
        red = map_range(band, 70, 200, 0.0, 0.6) ** gamma
        grn = 0.0
        blu = 1.0 ** gamma
    if band &gt;= 200 and band &lt; 300:  # violet to red
        red = map_range(band, 200, 300, 0.6, 1.0) ** gamma
        grn = 0.0
        blu = map_range(band, 200, 300, 1.0, 0.0) ** gamma
    if band &gt;= 300 and band &lt; 400:  # red to orange
        red = 1.0 ** gamma
        grn = map_range(band, 300, 400, 0.0, 0.5) ** gamma
        blu = 0.0
    if band &gt;= 400 and band &lt; 500:  # orange to yellow
        red = 1.0 ** gamma
        grn = map_range(band, 400, 500, 0.5, 1.0) ** gamma
        blu = 0.0
    if band &gt;= 500:  # yellow to white
        red = 1.0 ** gamma
        grn = 1.0 ** gamma
        blu = map_range(band, 500, 580, 0.0, 1.0) ** gamma

    return (int(red * 255) &lt;&lt; 16) + (int(grn * 255) &lt;&lt; 8) + int(blu *
255)
```

1-2-3s of Bilinear Interpolation

With just 64 display elements, the thermal camera can only display blocky object shapes. It's surprising how visual recognition improves when a bilinear interpolation technique is applied to the thermal sensor data to increase the resolution of the image.

Interpolation is a technique for enhancing limited data sets by estimating "in-between" values. It's often used for enlarging images to make patterns and objects easier to discern. Two varieties of interpolation are commonly used for images, bilinear and bicubic. The bilinear method, based on a linear equation like y = mx + b, is the simplest and the least computationally intensive. By comparison, bicubic interpolation is computationally more complicated, usually involving a polynomial function of the second degree or higher such as the quadratic form $y = ax^2 + bx + c$. The bicubic method can produce enlarged images with smoother and clearer object edges than the bilinear method -- but at the price of increased computational power and elapsed processing time.

Given the computational power of the PyGamer's SAMD-51 processor, the simpler bilinear approach was chosen to enlarge the thermal camera's image. The AMG8833 sensor's image is enlarged from 8 x 8 (64 elements) to a display grid of 15 x 15 (225 cells). Let's talk about how that is done. For the sake of simplicity, the following conceptual example of the method is limited to a 4 x 4 sensor array (16 elements) and 7 x 7 display grid (49 cells).

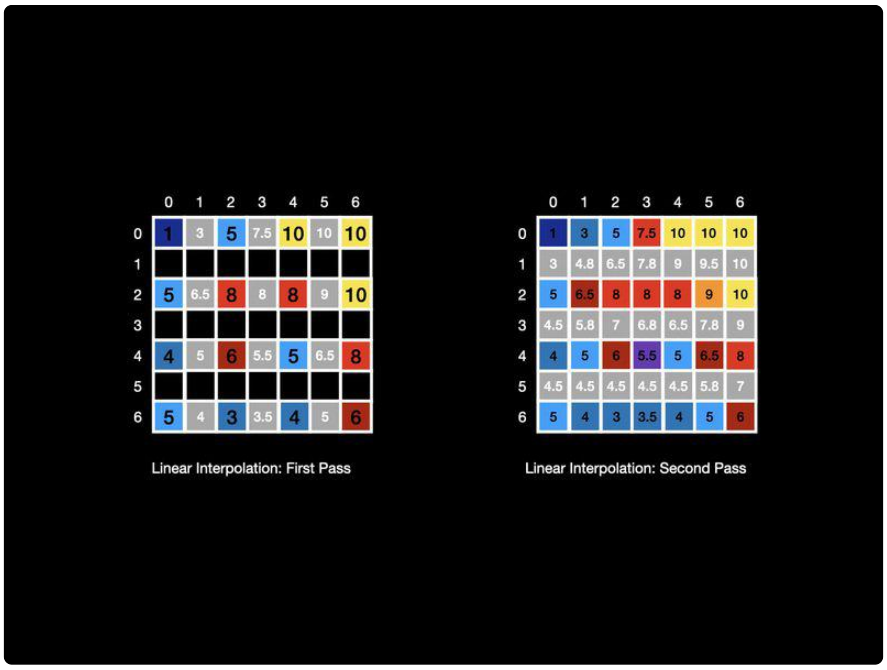


	0	1	2	3
0	1	5	10	10
1	5	8	8	10
2	4	6	5	8
3	5	3	4	6

	0	1	2	3	4	5	6
0	1		5		10		10
1							
2	5		8		8		10
3							
4	4		6		5		8
5							
6	5		3		4		6

The first step in the bilinear interpolation process is to copy the contents of the sensor value array into an image grid array of (2n - 1) rows and (2n - 1) columns. In this example n = 4, so the display grid array will have 7 rows and columns.

The first row of the display grid array contains the contents of the first row of the sensor value array with a blank element between each known sensor value. A row is skipped and the next row of sensor data is copied into the display grid array. After the sensor data is placed in the display grid array, the interpolation will replace the unknown cells with a calculated value from the closest known cells using a two-pass process.


Linear Interpolation: First Pass
Linear Interpolation: Second Pass

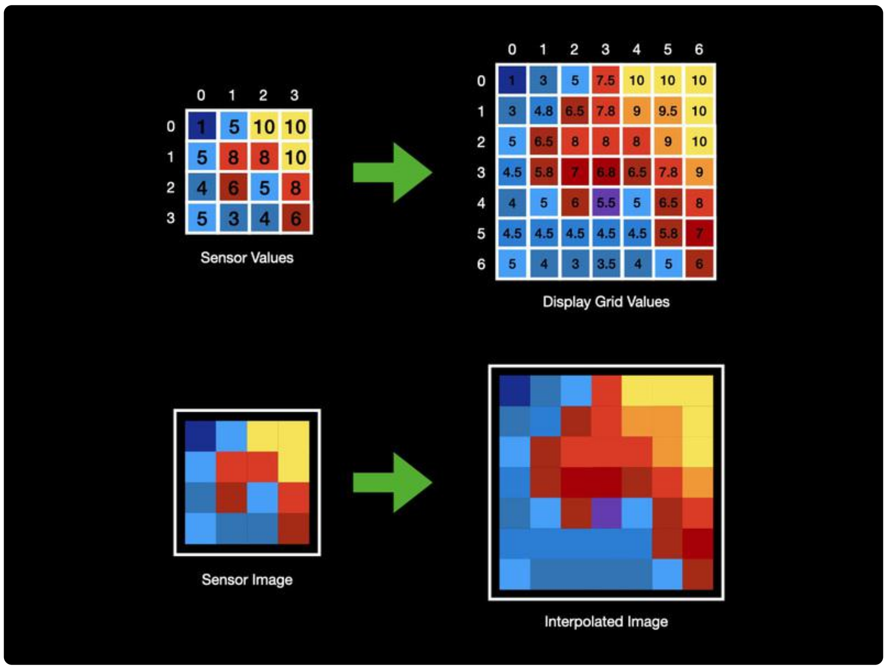
	0	1	2	3	4	5	6
0	1	3	5	7.5	10	10	10
1							
2	5	6.5	8	8	8	9	10
3							
4	4	5	6	5.5	5	6.5	8
5							
6	5	4	3	3.5	4	5	6

	0	1	2	3	4	5	6
0	1	3	5	7.5	10	10	10
1	3	4.8	6.5	7.8	9	9.5	10
2	5	6.5	8	8	8	9	10
3	4.5	5.8	7	6.8	6.5	7.8	9
4	4	5	6	5.5	5	6.5	8
5	4.5	4.5	4.5	4.5	4.5	5.8	7
6	5	4	3	3.5	4	5	6

The first pass starts with the first unknown cell and calculates its value from the preceding and following known cells in that row. For example, the cell between columns 0 and 1 of row 0 is calculated using an average of the two adjacent cells. The unknown cell is updated with the value of 3. The process continues to calculate the remaining unknown cells in the evenly numbered rows. The missing values in the odd numbered rows will be calculated next.

The second pass starts by processing the unknown cells of row 1, calculating the value from the average of the cells directly above and below. For example, the first cell of row 1 is updated with the value 3. The second pass continues to update the remaining unknown cells in the display grid array.

Because an unknown cell is flanked by known cells, its calculated value is the average of the known cells -- the unknown cell is halfway between the known cells. If the target image is enlarged to create two unknown cells between known cells, then the value of each unknown cell is calculated based on its proportional display distance from each of the two known cells.



	0	1	2	3
0	1	5	10	10
1	5	8	8	10
2	4	6	5	8
3	5	3	4	6

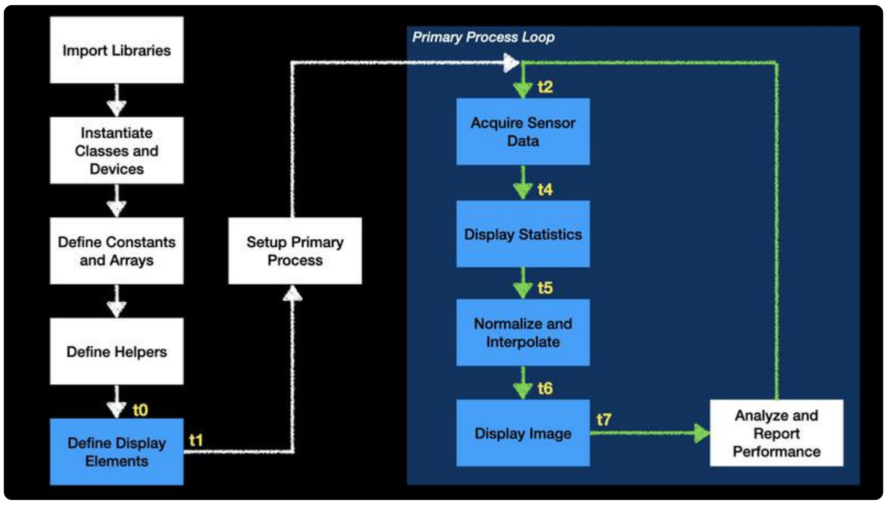
Sensor Values

	0	1	2	3	4	5	6
0	1	3	5	7.5	10	10	10
1	3	4.8	6.5	7.8	9	9.5	10
2	5	6.5	8	8	8	9	10
3	4.5	5.8	7	6.8	6.5	7.8	9
4	4	5	6	5.5	5	6.5	8
5	4.5	4.5	4.5	4.5	4.5	5.8	7
6	5	4	3	3.5	4	5	6

Display Grid Values

Here's the finished product, colored to roughly represent each cell's value. Compared to the original 4 x 4 sensor image, the newly interpolated 7 x 7 image has added detail with color gradients that can help to identify the object in the field of view.

Performance Monitoring



Because some significant timing and memory challenges were anticipated from the start, the structure of the improved thermal camera CircuitPython code was instrumented to measure its performance. Five functional areas were identified that would provide obvious hints as to where architectural or speed issues may live. Not only did the performance monitoring help solve some tricky timing and memory allocation issues, the resulting structure of the code allowed it to be easily adapted for testing on other development boards. The five code performance areas were:

Define Display Elements

> The one-time display definitions for rectangles, labels, and on-screen status. This process uses large blocks of memory to build the displayio group of display element attributes.

Acquire Sensor Data

> The first portion of the repeating primary loop that acquires and conditions the thermal sensor data. The acquisition process uses I2C input/output resources, the AMG88xx sensor library, and creates two large arrays in memory to hold and process the data. Floating point calculations constrain the sensor data to a valid temperature range.

Display Statistics

> Updates the on-screen alarm, min, max, and average values. This process manipulates display element attributes in memory, requires floating point calculations to support displayio, utilizes SPI input/output resources, and uses ul ab to quickly determine min, max, and average.

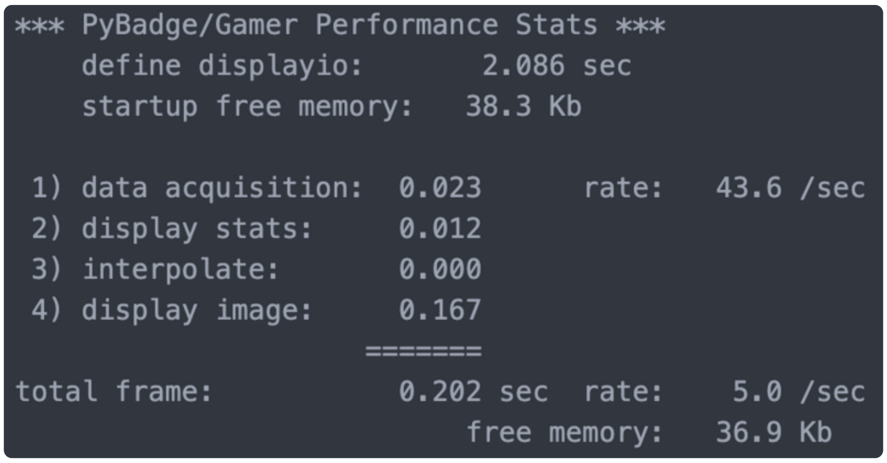
Normalize and Interpolate

> Normalizes the 8 x 8 sensor data, copies it to the display grid array, and interpolates the values within the 15 x 15 display grid array. ulab is used for all calculations.

Display Image

> Scans elements in the display grid array, calculates the iron spectrum color, and uses displayio to update an on-screen rectangle if the color has changed from the previous frame. After updating the image, this code segment checks the operational controls and modifies the display mode as selected. This segment heavily uses floating point, memory, and SPI input/output for the displayio functionality.

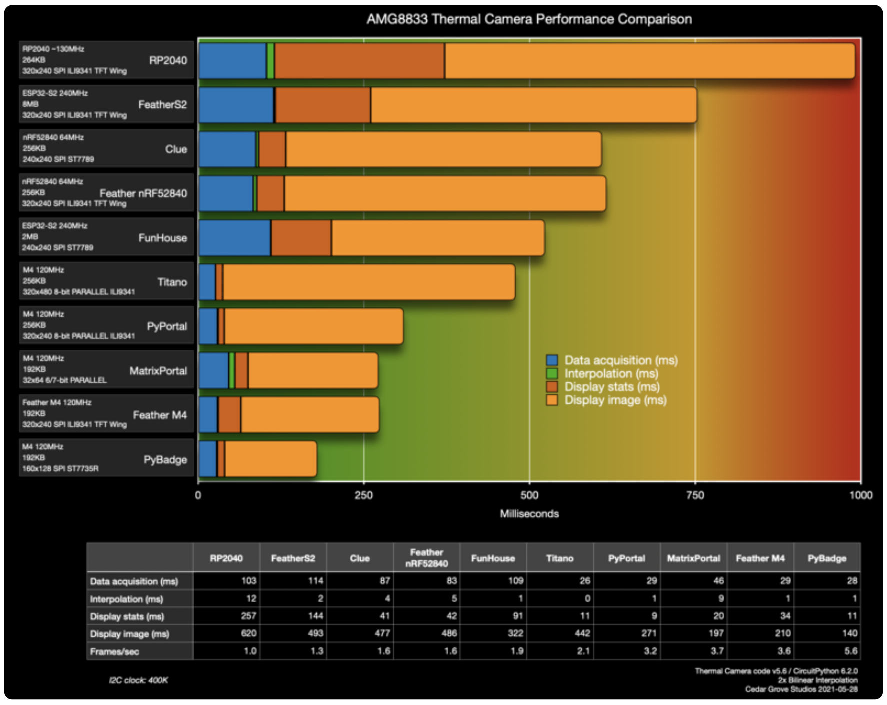
An elapsed time marker is stored at the beginning of each code segment. At the end of the primary process loop, the markers are analyzed and a report is printed to the serial output to be viewed via the REPL. Here's a screen shot of the performance report:



```
*** PyBadge/Gamer Performance Stats ***
    define displayio:       2.086 sec
    startup free memory:   38.3 Kb

 1) data acquisition:  0.023      rate:   43.6 /sec
 2) display stats:     0.012
 3) interpolate:       0.000
 4) display image:     0.167
                     =======
total frame:           0.202 sec  rate:    5.0 /sec
                           free memory:   36.9 Kb
```

After improvements, the code was ported to run on 9 other development boards in the workshop inventory ranging from SAMD-51 (M4) boards to ESP32-S2, nRF52840, and the RP2040. All of the PyGamer code was left intact except where specific display or button interface requirements were needed. For example, since the

PyPortal has no hardware buttons, its touch screen was used to implement button-like controls. Similarly, the Setup helper code was removed if memory capacity issues were identified for a particular development board.



	RP2040	FeatherS2	Clue	Feather nRF52840	FunHouse	Titano	PyPortal	MatrixPortal	Feather M4	PyBadge
Data acquisition (ms)	103	114	87	83	109	26	29	46	29	28
Interpolation (ms)	12	2	4	5	1	0	1	9	1	1
Display stats (ms)	257	144	41	42	91	11	9	20	34	11
Display image (ms)	620	493	477	486	322	442	271	197	210	140
Frames/sec	1.0	1.3	1.6	1.6	1.9	2.1	3.2	3.7	3.6	5.6

The PyGamer platform performed the best in this comparison. Generally, development boards that use the M4 (SAMD-51) processor performed well, beating the 2 frames-per-second performance threshold.

Since the thermal camera code uses a unique combination of resources suited for displaying temperature images, the comparison of thermal camera performance on the different platforms should not be construed as revealing intractable flaws of a particular development board or processor architecture. Instead, the comparison helps to point out performance bottlenecks unique to the thermal camera application that could benefit from further code refinement.

Many factors from processor architecture to the board's TFT display bus could impact thermal camera performance. For example, the current version of the thermal camera depends heavily on floating point calculations for almost everything, from normalizing and constraining sensor data to the internal calculations of CircuitPython displayio functions when it positions objects and justifies on-screen text. Development boards that use the SAMD-51 (M4) have an integral floating point processor in hardware that makes calculations a breeze -- so much so that little attention is given to tuning the code to calculate with integers when floating point math really isn't needed. Development boards such as those with the RP2040 processor do not have integral hardware floating point. Can you see where this is going?

So this comparison wasn't a completely fair test. The thermal camera's code was written to work best with the M4 architecture, not to take advantage of the RP2040's faster clock speed and huge memory capacity (and low cost!). What would it take to modify the code to work better with the RP2040? Are CircuitPython displayio and AMG8833 libraries tuned to take advantage of the RP2040's talents? We're going to have to add that project to the list.