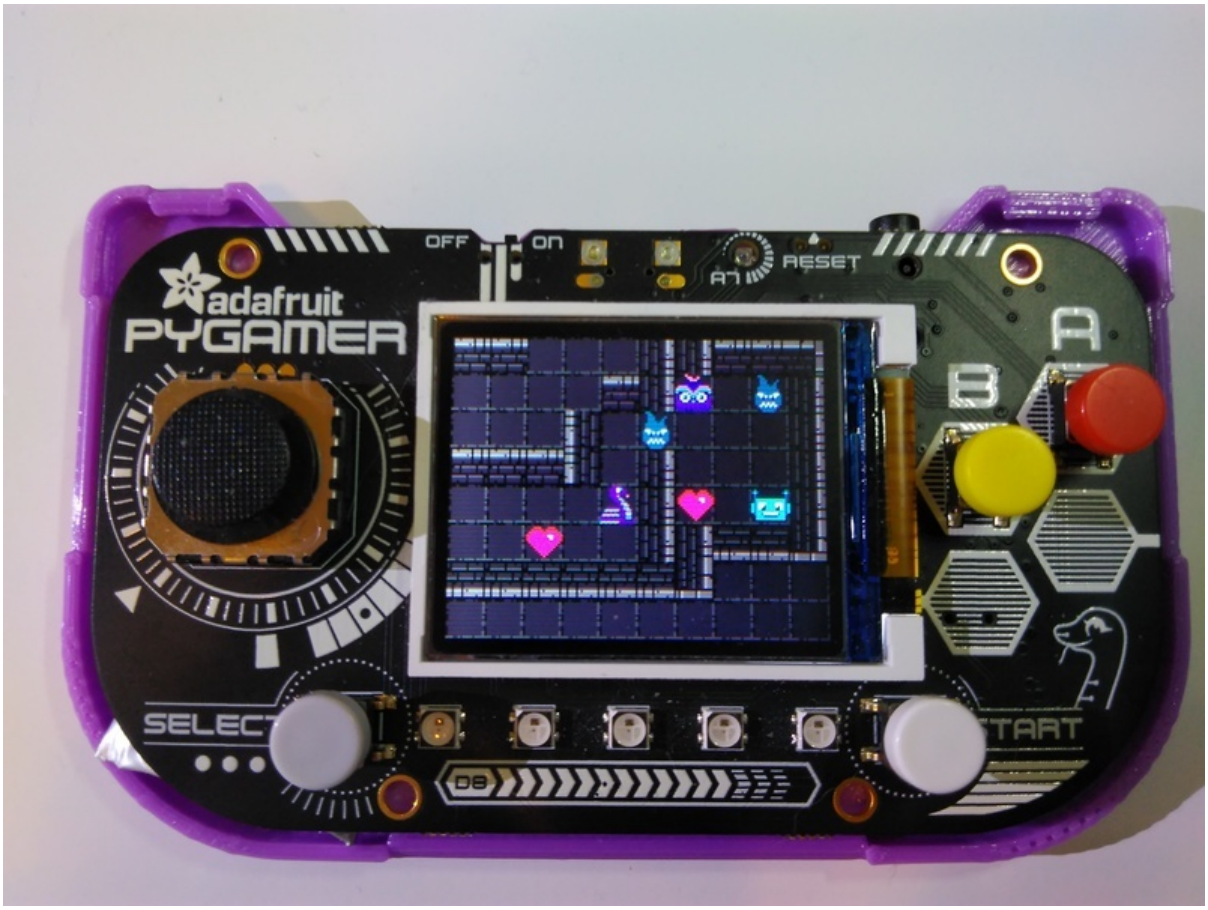
Creating Your First Tilemap Game with CircuitPython

Created by Tim C



https://learn.adafruit.com/creating-your-first-tilemap-game-with-circuitpython

Last updated on 2024-06-03 03:05:25 PM EDT

Table of Contents

Overview

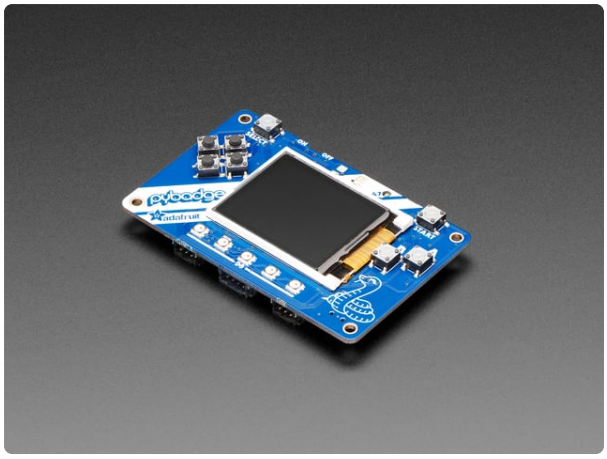
This project will show you how to create your first tile map game with CircuitPython. We will learn how to make and modify indexed bmp graphics for the game. `ugame` is used for device agnostic control handling. See how to use spreadsheet applications to create and edit game maps.

Everything will come together in a sample game with two levels. Once you reach the end of the guide you'll have all the tools you need to customize the sample game or create one of your very own!

The guide is meant to be used with any of these devices: PyGamer, PyBadge, PyBadge LC, Edge Badge, or Pew Pew M4.

Parts

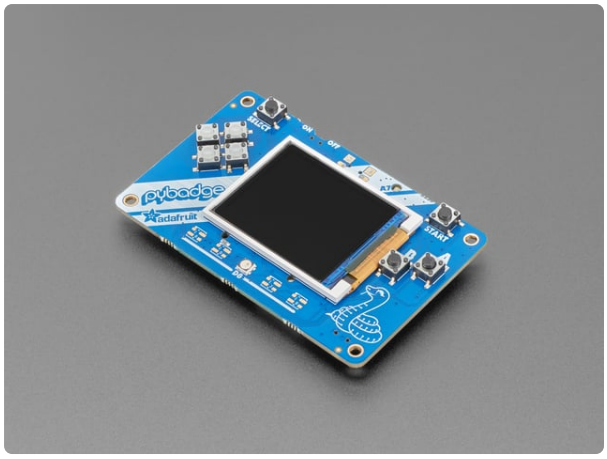
Adafruit PyGamer for MakeCode Arcade, CircuitPython or Arduino
What fits in your pocket, is fully Open Source, and can run CircuitPython, MakeCode Arcade or Arduino games you write yourself? That's right, it's the Adafruit...
https://www.adafruit.com/product/4242



[Adafruit PyBadge for MakeCode Arcade, CircuitPython, or Arduino]

What's the size of a credit card and can run CircuitPython, MakeCode Arcade or Arduino? That's right, its the Adafruit PyBadge! We wanted to see how much we...

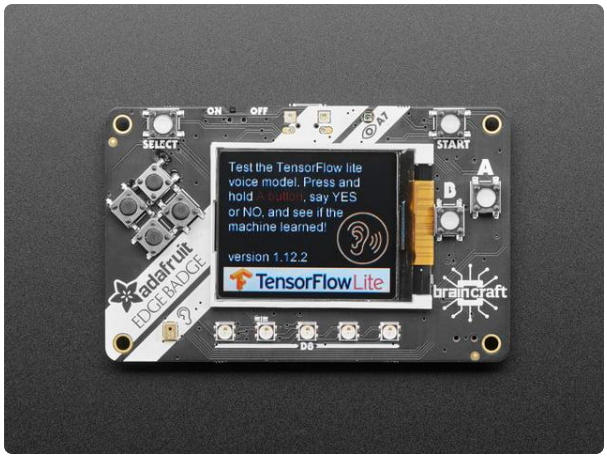
https://www.adafruit.com/product/4200



[Adafruit PyBadge LC - MakeCode Arcade, CircuitPython, or Arduino]

What's the size of a credit card and can run CircuitPython, MakeCode Arcade or Arduino even when you're on a budget? That's right, it's the Adafruit...

https://www.adafruit.com/product/3939



[Adafruit EdgeBadge - TensorFlow Lite for Microcontrollers]

Machine learning has come to the 'edge' - small microcontrollers that can run a very miniature version of TensorFlow Lite to do ML computations. But you don't...

https://www.adafruit.com/product/4400

[Adafruit PyGamer Starter Kit]

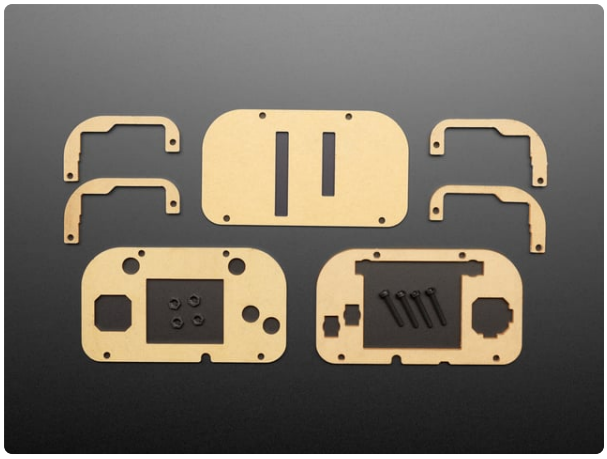
Please note: you may get a royal blue or purple case with your starter kit (they're both lovely colors)What fits in your pocket, is fully Open...

https://www.adafruit.com/product/4277

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



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

CircuitPython Setup

CircuitPython Setup for the PyBadge and PyGamer

First, to make sure you're running the most recent version of **CircuitPython** for the **PyBadge** or **PyGamer:**

Click to read on installing CircuitPython on the PyBadge devices

https://adafru.it/LxE

Or

Click to read installing CircuitPython on the PyGamer

https://adafru.it/FoA

If you have an **Edge Badge** the process is the same as **PyBadge**, but use the button below to download the firmware file.

If you have a **Pew Pew M4** the process is very similar as the others, but there is no button for the reset pin. Short the `R` (reset) and `-` (GND) pins with a wire twice in quick succession to access the BOOT drive. Use the button below to download the firmware file.

Download Edge Badge firmware

https://adafru.it/LxF

Or

Download Pew Pew M4 firmware

https://adafru.it/Lya

CircuitPython Libraries

You'll need a few CircuitPython libraries in the **lib** folder on the **CIRCUITPY** drive of your device for the code to work. Head to https://circuitpython.org/libraries (https://adafru.it/ENC) to download the latest library bundle matching the major version of CircuitPython now on your board (5 for CircuitPython 5.x, etc.). The procedure is available in the Grand Central M4 Express guide (https://adafru.it/JF9).

Once you've downloaded the libraries bundle, add these library directories to the **lib** folder on the **CIRCUITPY** drive:

- **adafruit_display_text**
- **adafruit_imageload**

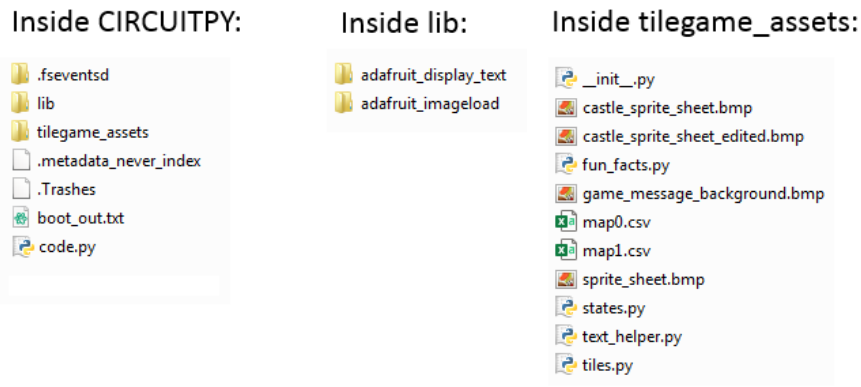
Download Circuit Python Library bundle

https://adafru.it/ENC

Tilegame Assets

Included with this project is a directory called **tilegame_assets**. You must copy this directory onto the **CIRCUITPY** drive. This contains the artwork, maps, and other modules needed for the game.

With everything in place you drive should have these files on it:



It's okay if your device has other files as well, but it must have these ones at a minimum.

Project Files

To get all the code and files for this project, click Download: Project Zip in the code box below. Unzip them and place a copy of them on the **CIRCUITPY** drive of your device in the directories noted above.

```
# SPDX-FileCopyrightText: 2020 FoamyGuy for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time
import random
import gc
import board
import displayio
import adafruit_imageload
import ugame
import terminalio
from adafruit_display_text import label
from tilegame_assets.tiles import TILES
from tilegame_assets.states import (
    STATE_PLAYING,
    STATE_MAPWIN,
    STATE_WAITING,
    STATE_LOST_SPARKY,
    STATE_MINERVA,
)
from tilegame_assets.fun_facts import FACTS
from tilegame_assets.text_helper import wrap_nicely

# pylint: disable=bad-continuation

# Direction constants for comparison
UP = 0
DOWN = 1
RIGHT = 2
LEFT = 3

# how long to wait between rendering frames
```

```
FPS_DELAY = 1 / 60

# how many tiles can fit on thes screen. Tiles are 16x16 pixels
SCREEN_HEIGHT_TILES = 8
SCREEN_WIDTH_TILES = 10

# list of maps in order they should be played
MAPS = ["map0.csv", "map1.csv"]

GAME_STATE = {
    # hold the map state as it came out of the csv. Only holds non-entities.
    "ORIGINAL_MAP": {},
    # hold the current map state as it changes. Only holds non-entities.
    "CURRENT_MAP": {},
    # Dictionary with touple keys that map to lists of entity objects.
    # Each one has the index of the sprite in the ENTITY_SPRITES list
    # and the tile type string
    "ENTITY_SPRITES_DICT": {},
    # hold the location of the player in tile coordinates
    "PLAYER_LOC": (0, 0),
    # list of items the player has in inventory
    "INVENTORY": [],
    # how many hearts there are in this map level
    "TOTAL_HEARTS": 0,
    # sprite object to draw for the player
    "PLAYER_SPRITE": None,
    # size of the map
    "MAP_WIDTH": 0,
    "MAP_HEIGHT": 0,
    # which map level within MAPS we are currently playing
    "MAP_INDEX": 0,
    # current state of the state machine
    "STATE": STATE_PLAYING,
}

# dictionary with tuple keys that map to tile type values
# e.x. {(0,0): "left_wall", (1,1): "floor"}
CAMERA_VIEW = {}

# how far offset the camera is from the GAME_STATE['CURRENT_MAP']
# used to determine where things are at in the camera view vs. the MAP
CAMERA_OFFSET_X = 0
CAMERA_OFFSET_Y = 0

# list of sprite objects, one for each entity
ENTITY_SPRITES = []

# list of entities that need to be on the screen currently based on the camera view
NEED_TO_DRAW_ENTITIES = []


def get_tile(coords):
    """
    :param coords: (x, y) tuple
    :return: tile name of the tile at the given coords from
GAME_STATE['CURRENT_MAP']
    """
    return GAME_STATE["CURRENT_MAP"][coords[0], coords[1]]


def get_tile_obj(coords):
    """
    :param coords: (x, y) tuple
    :return: tile object with stats and behavior for the tile at the given coords.
    """
    return TILES[GAME_STATE["CURRENT_MAP"][coords[0], coords[1]]]


#
```

```
def is_tile_moveable(tile_coords):
    """
    Check the can_walk property of the tile at the given coordinates
    :param tile_coords: (x, y) tuple
    :return: True if the player can walk on this tile. False otherwise.
    """
    return TILES[GAME_STATE["CURRENT_MAP"][tile_coords[0], tile_coords[1]]]
["can_walk"]


print("after funcs {}".format(gc.mem_free()))

# display object variable
display = board.DISPLAY

# Load the sprite sheet (bitmap)
sprite_sheet, palette = adafruit_imageload.load(
    "tilegame_assets/sprite_sheet.bmp",
    bitmap=displayio.Bitmap,
    palette=displayio.Palette,
)

# make green be transparent so entities can be drawn on top of map tiles
palette.make_transparent(0)

# Create the castle TileGrid
castle = displayio.TileGrid(
    sprite_sheet,
    pixel_shader=palette,
    width=10,
    height=8,
    tile_width=16,
    tile_height=16,
)

# Create a Group to hold the sprite and castle
group = displayio.Group()

# Add castle to the group
group.append(castle)


def load_map(file_name):
    # pylint: disable=global-statement,too-many-statements,too-many-nested-
blocks,too-many-branches
    global ENTITY_SPRITES, CAMERA_VIEW

    # empty the sprites from the group
    for cur_s in ENTITY_SPRITES:
        group.remove(cur_s)
    # remove player sprite
    try:
        group.remove(GAME_STATE["PLAYER_SPRITE"])
    except ValueError:
        pass

    # reset map and other game state objects
    GAME_STATE["ORIGINAL_MAP"] = {}
    GAME_STATE["CURRENT_MAP"] = {}
    ENTITY_SPRITES = []
    GAME_STATE["ENTITY_SPRITES_DICT"] = {}
    CAMERA_VIEW = {}
    GAME_STATE["INVENTORY"] = []
    GAME_STATE["TOTAL_HEARTS"] = 0

    # Open and read raw string from the map csv file
    f = open("tilegame_assets/{}".format(file_name), "r")
    map_csv_str = f.read()
    f.close()
```

```
    # split the raw string into lines
    map_csv_lines = map_csv_str.replace("\r", "").split("\n")

    # set the WIDTH and HEIGHT variables.
    # this assumes the map is rectangular.
    GAME_STATE["MAP_HEIGHT"] = len(map_csv_lines)
    GAME_STATE["MAP_WIDTH"] = len(map_csv_lines[0].split(","))

    # loop over each line storing index in y variable
    for y, line in enumerate(map_csv_lines):
        # ignore empty line
        if line != "":
            # loop over each tile type separated by commas, storing index in x
variable
            for x, tile_name in enumerate(line.split(",")):
                print("%s '%s'" % (len(tile_name), str(tile_name)))

                # if the tile exists in our main dictionary
                if tile_name in TILES.keys():

                    # if the tile is an entity
                    if (
                        "entity" in TILES[tile_name].keys()
                        and TILES[tile_name]["entity"]
                    ):
                        # set the map tiles to floor
                        GAME_STATE["ORIGINAL_MAP"][x, y] = "floor"
                        GAME_STATE["CURRENT_MAP"][x, y] = "floor"

                        if tile_name == "heart":
                            GAME_STATE["TOTAL_HEARTS"] += 1

                        # if it's the player
                        if tile_name == "player":
                            # Create the sprite TileGrid
                            GAME_STATE["PLAYER_SPRITE"] = displayio.TileGrid(
                                sprite_sheet,
                                pixel_shader=palette,
                                width=1,
                                height=1,
                                tile_width=16,
                                tile_height=16,
                                default_tile=TILES[tile_name]["sprite_index"],
                            )

                            # set the position of sprite on screen
                            GAME_STATE["PLAYER_SPRITE"].x = x * 16
                            GAME_STATE["PLAYER_SPRITE"].y = y * 16

                            # set position in x,y tile coords for reference later
                            GAME_STATE["PLAYER_LOC"] = (x, y)

                            # add sprite to the group
                            group.append(GAME_STATE["PLAYER_SPRITE"])
                        else:  # not the player
                            # Create the sprite TileGrid
                            entity_srite = displayio.TileGrid(
                                sprite_sheet,
                                pixel_shader=palette,
                                width=1,
                                height=1,
                                tile_width=16,
                                tile_height=16,
                                default_tile=TILES[tile_name]["sprite_index"],
                            )
                            # set the position of sprite on screen
                            # default to off the edge
                            entity_srite.x = -16
```

```
                            entity_srite.y = -16

                            # add the sprite object to ENTITY_SPRITES list
                            ENTITY_SPRITES.append(entity_srite)
                            # print("setting GAME_STATE['ENTITY_SPRITES_DICT'][%s,
%s]" % (x,y))

                            # create an entity obj
                            _entity_obj = {
                                "entity_sprite_index": len(ENTITY_SPRITES) - 1,
                                "map_tile_name": tile_name,
                            }

                            # if there are no entities at this location yet
                            if (x, y) not in GAME_STATE["ENTITY_SPRITES_DICT"]:
                                # create a list and add it to the dictionary at the
x,y location
                                GAME_STATE["ENTITY_SPRITES_DICT"][x, y] =
[_entity_obj]
                            else:
                                # append the entity to the existing list in the
dictionary
                                GAME_STATE["ENTITY_SPRITES_DICT"][x, y].append(
                                    _entity_obj
                                )

                    else:  # tile is not entity
                        # set the tile_name into MAP dictionaries
                        GAME_STATE["ORIGINAL_MAP"][x, y] = tile_name
                        GAME_STATE["CURRENT_MAP"][x, y] = tile_name

                else:  # tile type wasn't found in dict
                    print("tile: %s not found in TILES dict" % tile_name)
    # add all entity sprites to the group
    print("appending {} sprites".format(len(ENTITY_SPRITES)))
    for entity in ENTITY_SPRITES:
        group.append(entity)


print("loading map")
load_map(MAPS[GAME_STATE["MAP_INDEX"]])

# Add the Group to the Display
display.root_group = group

# variables to store previous value of button state
prev_up = False
prev_down = False
prev_left = False
prev_right = False


# helper function returns true if player is allowed to move given direction
# based on can_walk property of the tiles next to the player
def can_player_move(direction):
    try:
        if direction == UP:
            tile_above_coords = (
                GAME_STATE["PLAYER_LOC"][0],
                GAME_STATE["PLAYER_LOC"][1] - 1,
            )

            return TILES[
                GAME_STATE["CURRENT_MAP"][tile_above_coords[0],
tile_above_coords[1]]
            ]["can_walk"]

        if direction == DOWN:
            tile_below_coords = (
```

```
                GAME_STATE["PLAYER_LOC"][0],
                GAME_STATE["PLAYER_LOC"][1] + 1,
            )
            return TILES[
                GAME_STATE["CURRENT_MAP"][tile_below_coords[0],
tile_below_coords[1]]
            ]["can_walk"]

        if direction == LEFT:
            tile_left_of_coords = (
                GAME_STATE["PLAYER_LOC"][0] - 1,
                GAME_STATE["PLAYER_LOC"][1],
            )
            return TILES[
                GAME_STATE["CURRENT_MAP"][
                    tile_left_of_coords[0], tile_left_of_coords[1]
                ]
            ]["can_walk"]

        if direction == RIGHT:
            tile_right_of_coords = (
                GAME_STATE["PLAYER_LOC"][0] + 1,
                GAME_STATE["PLAYER_LOC"][1],
            )
            return TILES[
                GAME_STATE["CURRENT_MAP"][
                    tile_right_of_coords[0], tile_right_of_coords[1]
                ]
            ]["can_walk"]
    except KeyError:
        return False

    return None


# set the appropriate tiles into the CAMERA_VIEW dictionary
# based on given starting coords and size
def set_camera_view(startX, startY, width, height):
    # pylint: disable=global-statement
    global CAMERA_OFFSET_X
    global CAMERA_OFFSET_Y
    # set the offset variables for use in other parts of the code
    CAMERA_OFFSET_X = startX
    CAMERA_OFFSET_Y = startY

    # loop over the rows and indexes in the desired size section
    for y_index, y in enumerate(range(startY, startY + height)):
        # loop over columns and indexes in the desired size section
        for x_index, x in enumerate(range(startX, startX + width)):
            # print("setting camera_view[%s,%s]" % (x_index,y_index))
            try:
                # set the tile at the current coordinate of the MAP into the
CAMERA_VIEW
                CAMERA_VIEW[x_index, y_index] = GAME_STATE["CURRENT_MAP"][x, y]
            except KeyError:
                # if coordinate is out of bounds set it to floor by default
                CAMERA_VIEW[x_index, y_index] = "floor"


# draw the current CAMERA_VIEW dictionary and the GAME_STATE['ENTITY_SPRITES_DICT']
def draw_camera_view():
    # list that will hold all entities that have been drawn based on their MAP
location
    # any entities not in this list should get moved off the screen
    drew_entities = []
    # print(CAMERA_VIEW)
    # pylint: disable=too-many-nested-blocks

    # loop over y tile coordinates
```

```
    for y in range(0, SCREEN_HEIGHT_TILES):
        # loop over x tile coordinates
        for x in range(0, SCREEN_WIDTH_TILES):
            # tile name at this location
            tile_name = CAMERA_VIEW[x, y]

            # if tile exists in the main dictionary
            if tile_name in TILES.keys():
                # if there are entity(s) at this location
                if (x + CAMERA_OFFSET_X, y + CAMERA_OFFSET_Y) in GAME_STATE[
                    "ENTITY_SPRITES_DICT"
                ]:
                    # default background for entities is floor
                    castle[x, y] = TILES["floor"]["sprite_index"]

                    # if it's not the player
                    if tile_name != "player":
                        # loop over all entities at this location
                        for entity_obj_at_tile in GAME_STATE["ENTITY_SPRITES_DICT"][
                            x + CAMERA_OFFSET_X, y + CAMERA_OFFSET_Y
                        ]:
                            # set appropriate x,y screen coordinates
                            # based on tile coordinates
                            ENTITY_SPRITES[
                                int(entity_obj_at_tile["entity_sprite_index"])
                            ].x = (x * 16)
                            ENTITY_SPRITES[
                                int(entity_obj_at_tile["entity_sprite_index"])
                            ].y = (y * 16)

                            # add the index of the entity sprite to the
draw_entities
                            # list so we know not to hide it later.
                            drew_entities.append(
                                entity_obj_at_tile["entity_sprite_index"]
                            )

                else:  # no entities at this location
                    # set the sprite index of this tile into the CASTLE dictionary
                    castle[x, y] = TILES[tile_name]["sprite_index"]

            else:  # tile type not found in main dictionary
                # default to floor tile
                castle[x, y] = TILES["floor"]["sprite_index"]

            # if the player is at this x,y tile coordinate accounting for camera
offset
            if GAME_STATE["PLAYER_LOC"] == ((x + CAMERA_OFFSET_X, y +
CAMERA_OFFSET_Y)):
                # set player sprite screen coordinates
                GAME_STATE["PLAYER_SPRITE"].x = x * 16
                GAME_STATE["PLAYER_SPRITE"].y = y * 16

    # loop over all entity sprites
    for index in range(0, len(ENTITY_SPRITES)):
        # if the sprite wasn't drawn then it's outside the camera view
        if index not in drew_entities:
            # hide the sprite by moving it off screen
            ENTITY_SPRITES[index].x = int(-16)
            ENTITY_SPRITES[index].y = int(-16)


# variable to store timestamp of last drawn frame
last_update_time = 0

# variables to store movement offset values
x_offset = 0
y_offset = 0
```

```
def show_splash(new_text, color, vertical_offset=18):
    text_area.text = ""
    text_area.text = new_text
    text_area.anchor_point = (0, 0)
    text_area.anchored_position = (0, vertical_offset)
    text_area.color = color
    group.append(splash)


# Make the splash context
splash = displayio.Group()

# CircuitPython 6 & 7 compatible

# game message background bmp file
game_message_background = open("tilegame_assets/game_message_background.bmp", "rb")
odb = displayio.OnDiskBitmap(game_message_background)
bg_grid = displayio.TileGrid(odb, pixel_shader=getattr(odb, 'pixel_shader',
displayio.ColorConverter()))

# # CircuitPython 7+ compatible
# game message background bmp file
# odb = displayio.OnDiskBitmap("tilegame_assets/game_message_background.bmp")
# bg_grid = displayio.TileGrid(odb, pixel_shader=odb.pixel_shader)

splash.append(bg_grid)

# Text for the message
text_group = displayio.Group(x=14, y=8)
text_area = label.Label(terminalio.FONT, text=" " * 180, color=0xD39AE5)
text_group.append(text_area)
splash.append(text_group)

# main loop
while True:
    # set the current button values into variables
    cur_btn_vals = ugame.buttons.get_pressed()
    cur_up = cur_btn_vals & ugame.K_UP
    cur_down = cur_btn_vals & ugame.K_DOWN
    cur_right = cur_btn_vals & ugame.K_RIGHT
    cur_left = cur_btn_vals & ugame.K_LEFT
    cur_a = cur_btn_vals & ugame.K_O or cur_btn_vals & ugame.K_X

    if GAME_STATE["STATE"] == STATE_WAITING:
        print(cur_a)
        if cur_a:
            GAME_STATE["STATE"] = STATE_PLAYING
            group.remove(splash)

    if GAME_STATE["STATE"] == STATE_PLAYING:
        # check for up button press / release
        if not cur_up and prev_up:
            if can_player_move(UP):
                x_offset = 0
                y_offset = -1

        # check for down button press / release
        if not cur_down and prev_down:
            if can_player_move(DOWN):
                x_offset = 0
                y_offset = 1

        # check for right button press / release
        if not cur_right and prev_right:
            if can_player_move(RIGHT):
                x_offset = 1
                y_offset = 0
```

```
        # check for left button press / release
        if not cur_left and prev_left:
            if can_player_move(LEFT):
                x_offset = -1
                y_offset = 0

        # if any offset is not zero then we need to process player movement
        if x_offset != 0 or y_offset != 0:
            # variable to store if player is allowed to move
            can_move = False

            # coordinates the player is moving to
            moving_to_coords = (
                GAME_STATE["PLAYER_LOC"][0] + x_offset,
                GAME_STATE["PLAYER_LOC"][1] + y_offset,
            )

            # tile name of the spot player is moving to
            moving_to_tile_name = GAME_STATE["CURRENT_MAP"][
                moving_to_coords[0], moving_to_coords[1]
            ]

            # if there are entity(s) at spot the player is moving to
            if moving_to_coords in GAME_STATE["ENTITY_SPRITES_DICT"]:
                print("found entity(s) where we are moving to")

                # loop over all entities at the location player is moving to
                for entity_obj in GAME_STATE["ENTITY_SPRITES_DICT"][
                    moving_to_coords
                ]:
                    print("checking entity %s" % entity_obj["map_tile_name"])
                    # if the entity has a before_move behavior function
                    if "before_move" in TILES[entity_obj["map_tile_name"]].keys():
                        print(
                            "calling before_move %s, %s, %s"
                            % (
                                moving_to_coords,
                                GAME_STATE["PLAYER_LOC"],
                                entity_obj,
                            )
                        )
                        # call the before_move behavior function act upon it's
result
                        if TILES[entity_obj["map_tile_name"]]["before_move"](
                            moving_to_coords,
                            GAME_STATE["PLAYER_LOC"],
                            entity_obj,
                            GAME_STATE,
                        ):
                            # all the movement if it returned true
                            can_move = True
                        else:
                            # break and don't allow movement if it returned false
                            break
                    else:  # entity does not have a before_move function
                        # allow movement
                        can_move = True
                if can_move:
                    # set the player loc variable to the new coords
                    GAME_STATE["PLAYER_LOC"] = moving_to_coords

            else:  # no entities at the location player is moving to
                # set player loc variable to new coords
                GAME_STATE["PLAYER_LOC"] = moving_to_coords

        # reset movement offset variables
        y_offset = 0
        x_offset = 0
```

```
        # set previous button values for next iteration
        prev_up = cur_up
        prev_down = cur_down
        prev_right = cur_right
        prev_left = cur_left

        # current time
        now = time.monotonic()

        # if it has been long enough based on FPS delay
        if now > last_update_time + FPS_DELAY:
            # Set camera to 10x8 centered on the player
            # Clamped to (0, MAP_WIDTH) and (0, MAP_HEIGHT)
            set_camera_view(
                max(
                    min(
                        GAME_STATE["PLAYER_LOC"][0] - 4,
                        GAME_STATE["MAP_WIDTH"] - SCREEN_WIDTH_TILES,
                    ),
                    0,
                ),
                max(
                    min(
                        GAME_STATE["PLAYER_LOC"][1] - 3,
                        GAME_STATE["MAP_HEIGHT"] - SCREEN_HEIGHT_TILES,
                    ),
                    0,
                ),
                10,
                8,
            )
            # draw the camera
            draw_camera_view()
        # if player beat this map
        if GAME_STATE["STATE"] == STATE_MAPWIN:
            GAME_STATE["MAP_INDEX"] += 1
            # if player has beaten all maps
            if GAME_STATE["MAP_INDEX"] >= len(MAPS):
                GAME_STATE["MAP_INDEX"] = 0
                GAME_STATE["STATE"] = STATE_WAITING
                load_map(MAPS[GAME_STATE["MAP_INDEX"]])
                show_splash(
                    "You Win \n =D \nCongratulations. \nStart Over?", 0x29C1CF
                )
            else:
                # prompt to start next
                GAME_STATE["STATE"] = STATE_WAITING
                load_map(MAPS[GAME_STATE["MAP_INDEX"]])
                show_splash(
                    "You beat this level\n =D \nCongratulations. \nStart Next?",
                    0x29C1CF,
                )
        # game over from sparky
        elif GAME_STATE["STATE"] == STATE_LOST_SPARKY:
            GAME_STATE["MAP_INDEX"] = 0
            GAME_STATE["STATE"] = STATE_WAITING
            game_over_text = (
                "Be careful not to \ntouch Sparky unless \n"
                "you've collected \nenough Mho's.\nStarting Over"
            )
            load_map(MAPS[GAME_STATE["MAP_INDEX"]])
            show_splash(game_over_text, 0x25AFBB)

        # talking to minerva
        elif GAME_STATE["STATE"] == STATE_MINERVA:
            GAME_STATE["STATE"] = STATE_WAITING
            random_fact = random.choice(FACTS)
            minerva_txt = wrap_nicely("Minerva: {}".format(random_fact), 23)
            show_splash(minerva_txt, 0xD39AE5, 0)
```

```
        # store the last update time
        last_update_time = now
```

Rendering and Movement

Basic Rendering Code

If you are brand new to CircuitPython or `displayio` based programs, or even if it's just been a while, you should start by going over the Multiple Tilegrids page in the displayio guide (https://adafru.it/Lyc). After you've done that then come back here.

To start, lets look at this example:

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
# SPDX-FileCopyrightText: 2020 FoamyGuy for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import board
import displayio
import adafruit_imageload

display = board.DISPLAY

# Load the sprite sheet (bitmap)
sprite_sheet, palette = adafruit_imageload.load(
    "tilegame_assets/castle_sprite_sheet.bmp",
    bitmap=displayio.Bitmap,
    palette=displayio.Palette,
)

# Create the sprite TileGrid
sprite = displayio.TileGrid(
    sprite_sheet,
    pixel_shader=palette,
    width=1,
    height=1,
    tile_width=16,
    tile_height=16,
    default_tile=0,
)

# Create the castle TileGrid
castle = displayio.TileGrid(
    sprite_sheet,
    pixel_shader=palette,
    width=10,
    height=8,
    tile_width=16,
    tile_height=16,
)

# Create a Group to hold the sprite and add it
sprite_group = displayio.Group()
sprite_group.append(sprite)

# Create a Group to hold the castle and add it
castle_group = displayio.Group(scale=1)
```

```
castle_group.append(castle)

# Create a Group to hold the sprite and castle
group = displayio.Group()

# Add the sprite and castle to the group
group.append(castle_group)
group.append(sprite_group)

# Castle tile assignments
# corners
castle[0, 0] = 3  # upper left
castle[9, 0] = 5  # upper right
castle[0, 7] = 9  # lower left
castle[9, 7] = 11  # lower right
# top / bottom walls
for x in range(1, 9):
    castle[x, 0] = 4  # top
    castle[x, 7] = 10  # bottom
# left/ right walls
for y in range(1, 7):
    castle[0, y] = 6  # left
    castle[9, y] = 8  # right
# floor
for x in range(1, 9):
    for y in range(1, 7):
        castle[x, y] = 7  # floor

# put the sprite somewhere in the castle
sprite.x = 16 * 4
sprite.y = 16 * 3

# Add the Group to the Display
display.root_group = group
while True:
    pass
```

In this code we have two Groups, one for the castle and one for the sprite or player icon, Blinka. These groups each get a TileGrid created and added to them. The `castle` TileGrid is sized 10x8 and divides evenly into the 160x128 pixel screen with each tile being 16x16 pixels. It holds the tiles that make up the portion of the map visible currently. The `sprite` TileGrid is only 1x1 but does still use the same 16x16 pixel size. It holds the Blinka image for the player.

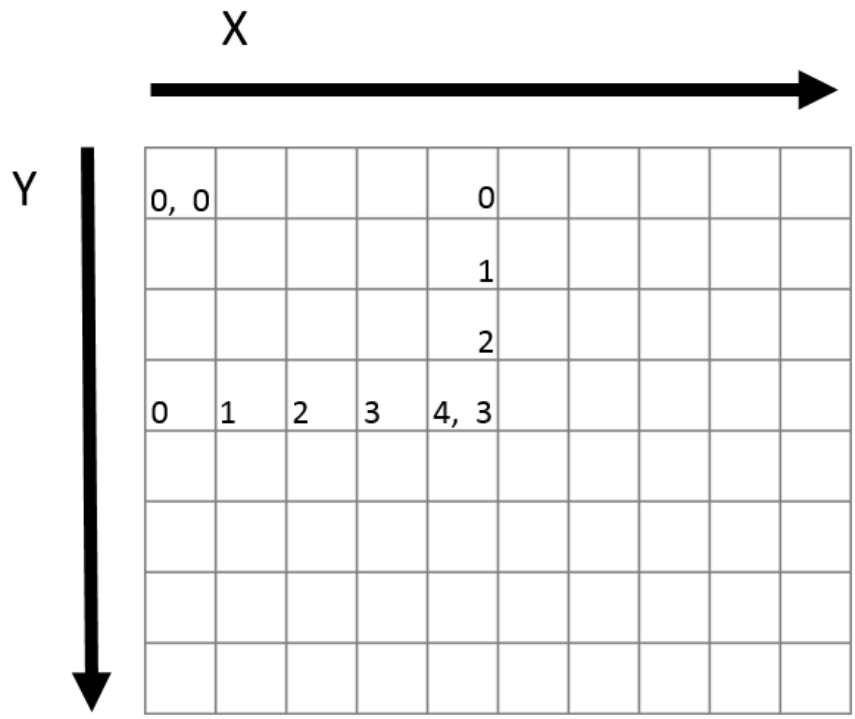
If you do you do not understand the basics of how this layout works take a look back at the Multiple Tilegrids page in the displayio guide (https://adafru.it/Lyc). We will build on these concepts to create the rest of the game map. The first thing we'll do is add a movement system so we can make Blinka move around using the d-pad or joystick.

Basic Movement Code

```
# SPDX-FileCopyrightText: 2020 FoamyGuy for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import board
import displayio
import adafruit_imageload

# from tilegame_assets.controls_helper import controls
import ugame

display = board.DISPLAY
player_loc = {"x": 4, "y": 3}

# Load the sprite sheet (bitmap)
sprite_sheet, palette = adafruit_imageload.load(
    "tilegame_assets/castle_sprite_sheet.bmp",
    bitmap=displayio.Bitmap,
    palette=displayio.Palette,
)

# Create the sprite TileGrid
sprite = displayio.TileGrid(
    sprite_sheet,
    pixel_shader=palette,
    width=1,
    height=1,
    tile_width=16,
    tile_height=16,
    default_tile=0,
)

# Create the castle TileGrid
castle = displayio.TileGrid(
    sprite_sheet,
    pixel_shader=palette,
    width=10,
    height=8,
    tile_width=16,
    tile_height=16,
)

# Create a Group to hold the sprite and add it
sprite_group = displayio.Group()
sprite_group.append(sprite)

# Create a Group to hold the castle and add it
castle_group = displayio.Group(scale=1)
castle_group.append(castle)

# Create a Group to hold the sprite and castle
group = displayio.Group()

# Add the sprite and castle to the group
group.append(castle_group)
group.append(sprite_group)

# Castle tile assignments
# corners
castle[0, 0] = 3  # upper left
castle[9, 0] = 5  # upper right
castle[0, 7] = 9  # lower left
castle[9, 7] = 11  # lower right
# top / bottom walls
```

```
for x in range(1, 9):
    castle[x, 0] = 4  # top
    castle[x, 7] = 10  # bottom
# left/ right walls
for y in range(1, 7):
    castle[0, y] = 6  # left
    castle[9, y] = 8  # right
# floor
for x in range(1, 9):
    for y in range(1, 7):
        castle[x, y] = 7  # floor

# put the sprite somewhere in the castle
sprite.x = 16 * player_loc["x"]
sprite.y = 16 * player_loc["y"]

# Add the Group to the Display
display.root_group = group

prev_btn_vals = ugame.buttons.get_pressed()

while True:
    cur_btn_vals = ugame.buttons.get_pressed()
    if not prev_btn_vals & ugame.K_UP and cur_btn_vals & ugame.K_UP:
        player_loc["y"] = max(1, player_loc["y"] - 1)
    if not prev_btn_vals & ugame.K_DOWN and cur_btn_vals & ugame.K_DOWN:
        player_loc["y"] = min(6, player_loc["y"] + 1)

    if not prev_btn_vals & ugame.K_RIGHT and cur_btn_vals & ugame.K_RIGHT:
        player_loc["x"] = min(8, player_loc["x"] + 1)
    if not prev_btn_vals & ugame.K_LEFT and cur_btn_vals & ugame.K_LEFT:
        player_loc["x"] = max(1, player_loc["x"] - 1)

    # update the the player sprite position
    sprite.x = 16 * player_loc["x"]
    sprite.y = 16 * player_loc["y"]

    # update the previous values
    prev_btn_vals = cur_btn_vals
```

In the code above we have a variable `player_loc` that stores the coordinates of the player in tile coordinates. It's set to `{"x":4, "y":3}` by default. These correspond to the indexes of the tiles starting from the top left corner at `{"x":0, "y":0}`.



To handle player movement we will use a library called **ugame.** This helper module abstracts away the differences between different boards and leaves us with common signals to use for D-pad and other buttons.

The code above already contains the import for it, here is what it looks like:

```
import ugame
```

After it's been imported we can access the current button states with `ugame.buttons.get_pressed()` it will return a number which holds each button state accessible by anding with the button constants like this:

```
ugame.buttons.get_pressed() &amp;amp; ugame.K_UP # True if up btn is pressed.
ugame.buttons.get_pressed() &amp;amp; ugame.K_DOWN # True if down btn is pressed.
ugame.buttons.get_pressed() &amp;amp; ugame.K_RIGHT # True if right btn is pressed.
ugame.buttons.get_pressed() &amp;amp; ugame.K_LEFT # True if left btn is pressed.
```

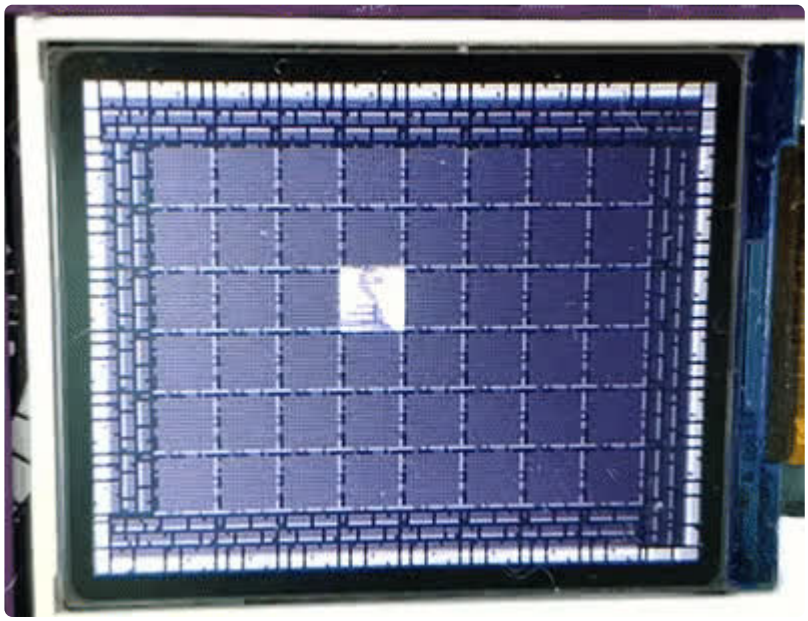
We'll use these values in the main loop along with the `prev_btn_vals` to decide when we need to change the `player_loc`. Inside the main loop we update the `sprite.x` and `sprite.y` values to match the `player_loc`. This will make Blinka move around when the player presses D-pad buttons or uses the joystick.

Update the code from above to use this main loop.

```
# Variable to old previous button state
prev_btn_vals = ugame.buttons.get_pressed()
while True:
    cur_btn_vals = ugame.buttons.get_pressed() # update button sate
```

```
    # if up button was pressed
    if not prev_btn_vals &amp;amp; ugame.K_UP and cur_btn_vals &amp;amp; ugame.K_UP:
        player_loc["y"] = max(1, player_loc["y"]-1)
    # if down button was pressed
    if not prev_btn_vals &amp;amp; ugame.K_DOWN and cur_btn_vals &amp;amp; ugame.K_DOWN:
        player_loc["y"] = min(6, player_loc["y"]+1)
    # if right button was pressed
    if not prev_btn_vals &amp;amp; ugame.K_RIGHT and cur_btn_vals &amp;amp; ugame.K_RIGHT:
        player_loc["x"] = min(8, player_loc["x"]+1)
    # if left button was pressed
    if not prev_btn_vals &amp;amp; ugame.K_LEFT and cur_btn_vals &amp;amp; ugame.K_LEFT:
        player_loc["x"] = max(1, player_loc["x"]-1)

    # update the the player sprite position
    sprite.x = 16 * player_loc["x"]
    sprite.y = 16 * player_loc["y"]

    # update the previous values
    prev_btn_vals = cur_btn_vals
```

Now we can use the D-pad or joystick to move Blinka around!



Next we will change the graphics to get rid of the white square around Blinka.

Indexed BMP Graphics

There are many image editing programs that can be used to create and modify bmp assets for use with CircuitPython. If you have access to and are familiar with Adobe Photoshop you can use that. If you are into making games and/or pixel art specifically, there is Aseprite (https://adafru.it/ECC) which offers many great features that make it very easy to work with pixel art indexed bmp files. It's not free, but it is a nice utility for making and editing graphics.

For this guide, we'll use a free alternative: The GNU Image (https://adafru.it/Lyd)Manipulation Program or GIMP (https://adafru.it/Lyd). This works on Linux, Mac,

and Windows PCs, and it's free to use and open sourced. We will use this to edit the image so that we can make the white background appear as transparent in the game. If you don't already have GIMP installed, use the link above and follow the installation instructions for your OS.

We'll starting from this bmp file:

castle_sprite_sheet.bmp

https://adafru.it/EFO

Open this file in GIMP by using File -> Open or Ctrl+O then navigate to the directory where you've download the bmp file or project zip.

At real size 100% zoom the image will likely appear very small in GIMP. It's very helpful to zoom in with Ctrl+MouseWheel or the plus and minus buttons.

Transparency within Indexed BMPs

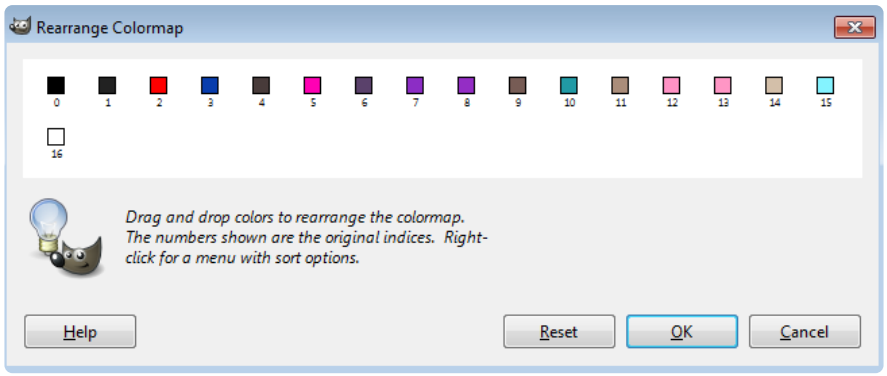
The bmp file format does not directly support transparency. Instead we are able to add code in CircuitPython that declares certain colors to appear as transparent instead of showing normally. You can think of it sort of like a green screen effect.

In the existing image, white is the color in the background, but we don't want to set white as transparent because Blinka and the Robot's eye's contain white so they would look wrong. Sometimes bright pink is used as the background color in sprite sheets for this purpose. But we have the heart graphic already using pink though, so we don't want to use pink either. We'll use green instead to follow with the green screen analogy. But really it could be any color that is not already used in the graphic.

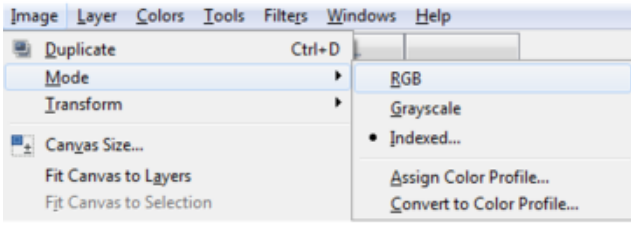
Converting to RGB Mode

Before we can modify the image we need to change it back to RGB color mode. Right now the image is in indexed color mode which means it only has access to a small set of colors.

Here are the currently available colors for this image:



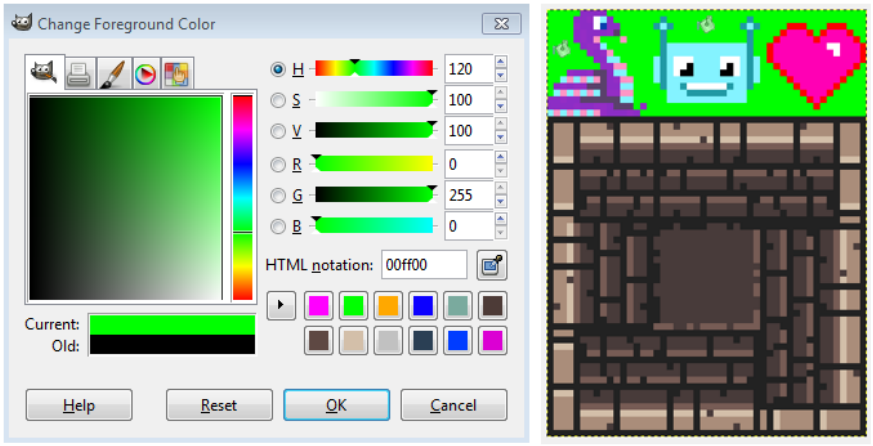
Since the green we want to use is not one of these colors it won't work, by default GIMP would choose the closest available color. But instead we want to add green to the available colors and use it. So we need to switch back to RGB color mode for now.



To do that click Image -> Mode -> RGB

Filling in the Background

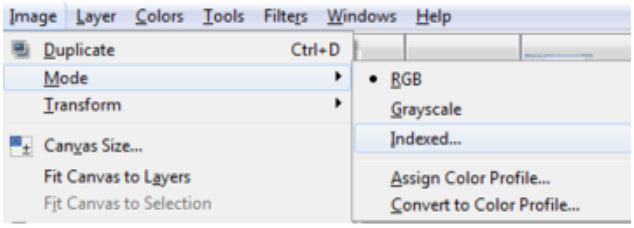
Now you can set your foreground color and use the paint bucket tool to fill in the background parts of the sprite image with pure green `#00FF00`.



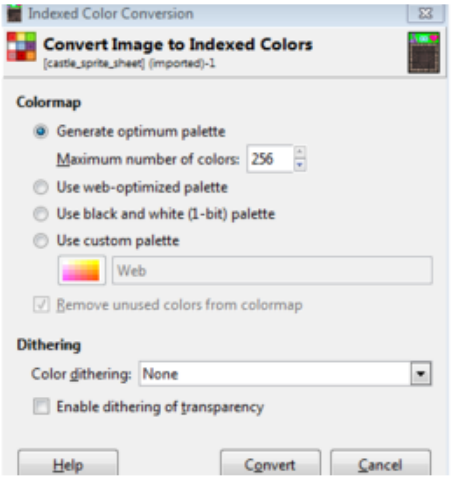
In the image above, I've added small paint bucket icons in the two spots I clicked with the paint bucket tool to fill in the background green.

Converting back to Indexed Color Mode

Once the background is filled in we need to switch the image back to indexed color mode so that CircuitPython can use it.



Click: Image -> Mode -> Indexed



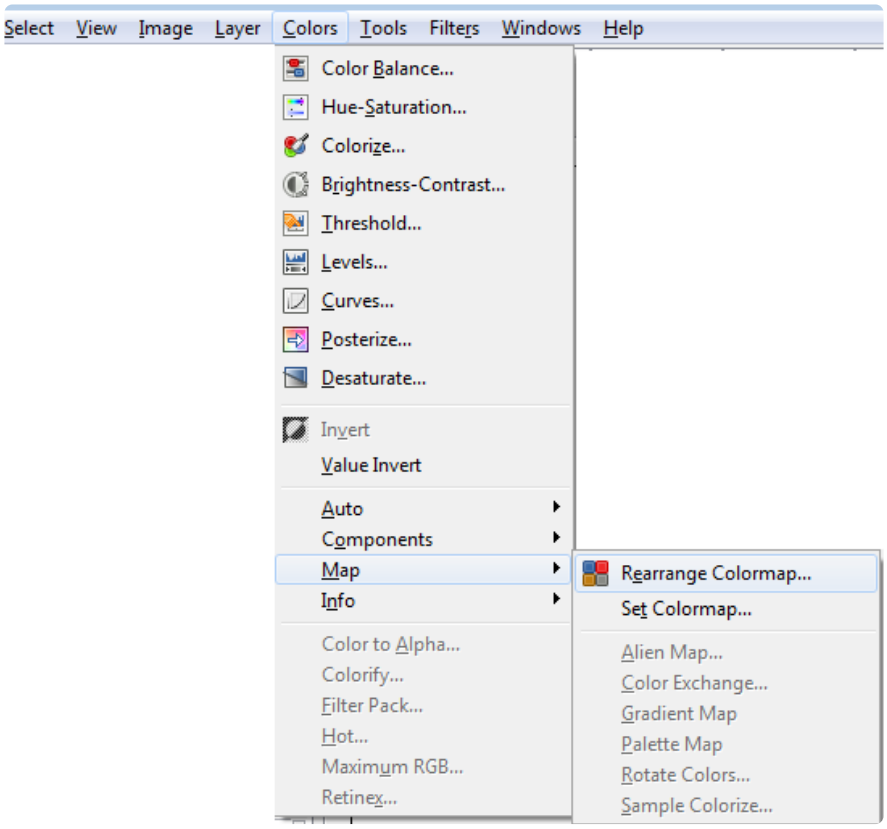
Leave the default settings in the Convert Image to Index Colors configuration dialog. It should look like this.

Click the Convert button.

Re-arrange the Color Index

In the CircuitPython code we will need to specify a color index that is going to be shown as transparent. We can see the colors and their indexes inside of gimp in the Rearrange Colormap window. To access it:

Click: Color -> Map -> Rearrange Colormap

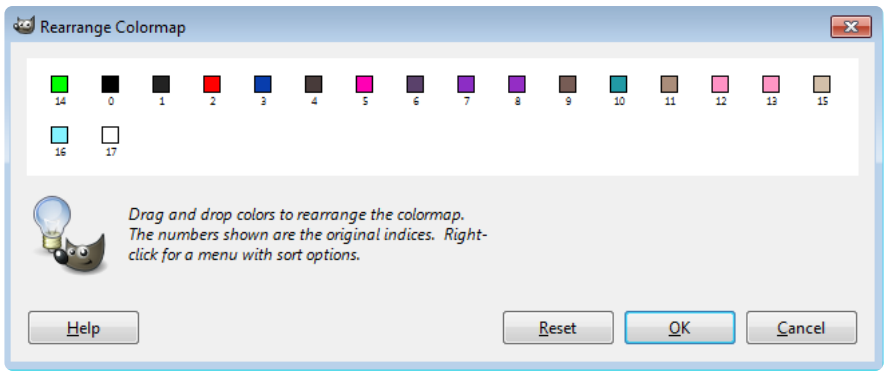


In my case, the newly added pure green transparency placeholder is at index `14`. It may be different for you, if so that's no problem.

We could make note of this index and use it in the CircuitPython code. But then if we ever change the image, we would need to make sure to go change the code if the index changed again due to us adding or removing colors.

Instead let's move the transparency placeholder to index 0. Then we can always make index 0 transparent in the code and as long as we always follow this rule when preparing bmp files, we can use the same code everywhere and it will work.

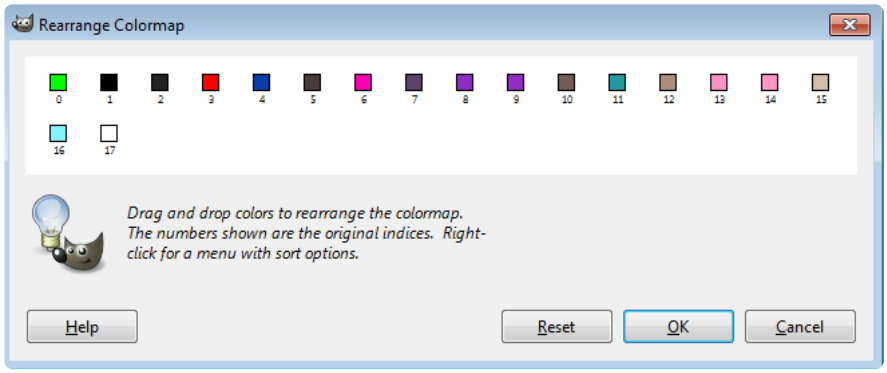
So click and drag your placeholder color over to the very first position in the list. The index still shows the wrong number but that is okay for now.



Once your transparency placeholder is in the first position, click the Ok button.

At this point you are ready to export your image as a bmp file.

But I know it is weird the indexes were wrong on the previous screen... If you like to double check on things you can click Color -> Map -> Rearrange Colormap again to open up a new window and see that the indexes are updated to the proper numbers. It would look like this now:



Saving the Image as a BMP File

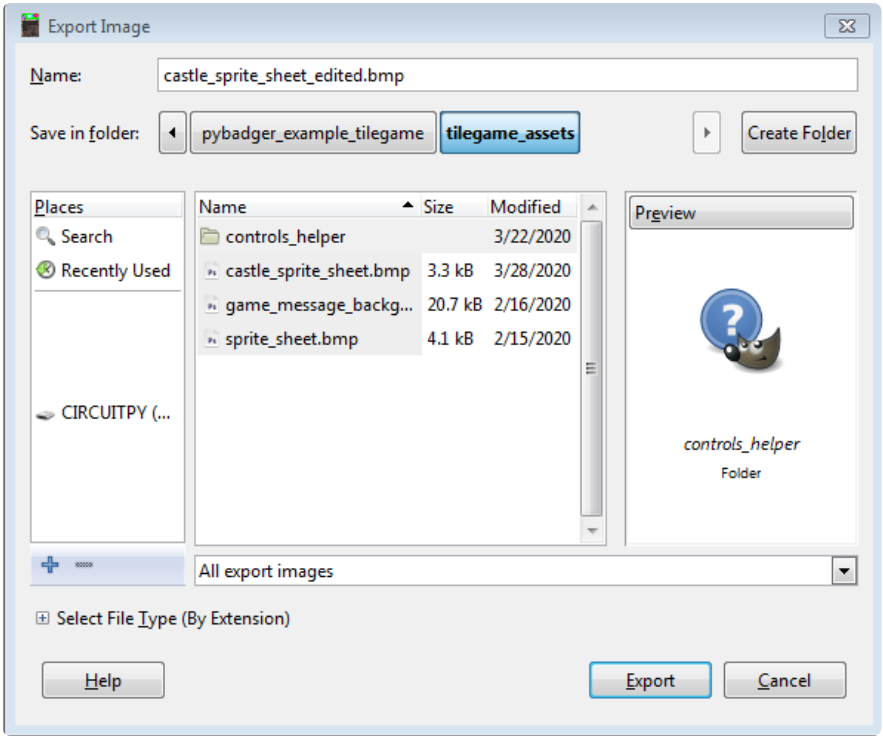
Now the image is ready to be exported out of gimp as a bitmap file that is ready to show with CircuitPython `displayio`.

Click: File -> Export As..

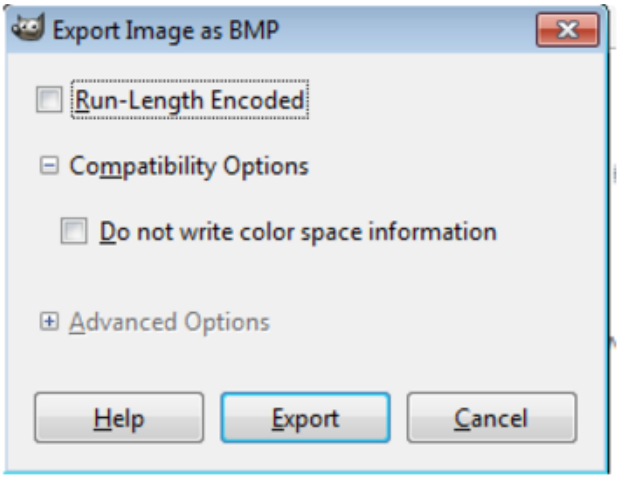
or Press: Ctrl+Shift+E

You can choose any name for your file, but it must end with `.bmp` because this will be the format that gimp saves the file with. I am going to use the name `castle_sprite_sheet_edited.bmp`. You can choose a different name if you like, if you do make sure to edit the rest of the sample code to use your own bmp filename.

At a minimum, you need to export one copy to your **CIRCUITPY** drive, but you can also save a copy to your local file system if you like. For this example we are going to save it into the `tilegame_assets` directory.



Once you've got the filename and location selected click on the Export button at the bottom.

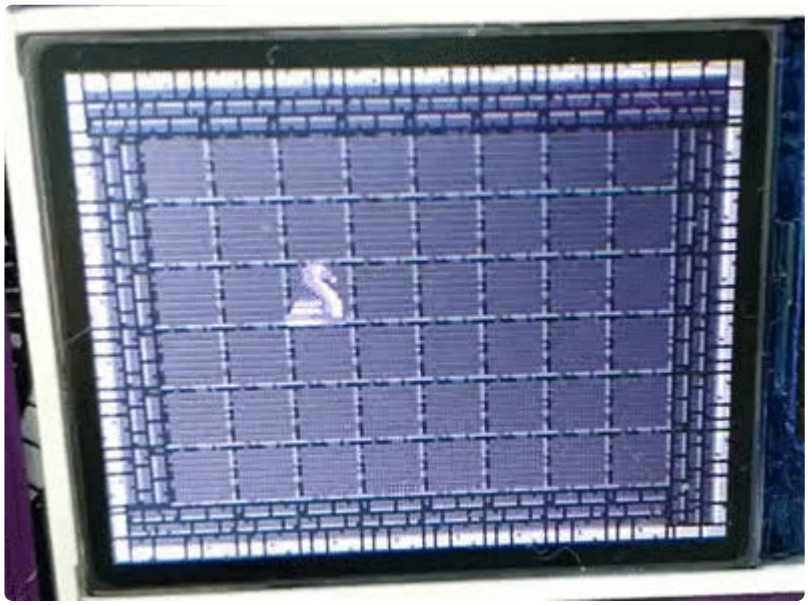


Leave the default values in he export dialog.

Click the Export button.

Now we can change the sprite image loading code to use the new image and set the color at index `0` to be transparent.

```
sprite_sheet, palette = adafruit_imageload.load("tilegame_assets/
castle_sprite_sheet_edited.bmp",
                                                bitmap=displayio.Bitmap,
                                                palette=displayio.Palette)

# Make the color at index 0 show as transparent
palette.make_transparent(0)
```



```
# SPDX-FileCopyrightText: 2020 FoamyGuy for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import board
import displayio
import adafruit_imageload
import ugame

display = board.DISPLAY
player_loc = {"x": 4, "y": 3}

# Load the sprite sheet (bitmap)
sprite_sheet, palette = adafruit_imageload.load(
    "tilegame_assets/castle_sprite_sheet_edited.bmp",
    bitmap=displayio.Bitmap,
    palette=displayio.Palette,
)
# make the color at 0 index transparent.
palette.make_transparent(0)

# Create the sprite TileGrid
sprite = displayio.TileGrid(
    sprite_sheet,
    pixel_shader=palette,
    width=1,
    height=1,
    tile_width=16,
    tile_height=16,
    default_tile=0,
)

# Create the castle TileGrid
castle = displayio.TileGrid(
    sprite_sheet,
    pixel_shader=palette,
```

```
    width=10,
    height=8,
    tile_width=16,
    tile_height=16,
)

# Create a Group to hold the sprite and add it
sprite_group = displayio.Group()
sprite_group.append(sprite)

# Create a Group to hold the castle and add it
castle_group = displayio.Group(scale=1)
castle_group.append(castle)

# Create a Group to hold the sprite and castle
group = displayio.Group()

# Add the sprite and castle to the group
group.append(castle_group)
group.append(sprite_group)

# Castle tile assignments
# corners
castle[0, 0] = 3  # upper left
castle[9, 0] = 5  # upper right
castle[0, 7] = 9  # lower left
castle[9, 7] = 11  # lower right
# top / bottom walls
for x in range(1, 9):
    castle[x, 0] = 4  # top
    castle[x, 7] = 10  # bottom
# left/ right walls
for y in range(1, 7):
    castle[0, y] = 6  # left
    castle[9, y] = 8  # right
# floor
for x in range(1, 9):
    for y in range(1, 7):
        castle[x, y] = 7  # floor

# put the sprite somewhere in the castle
sprite.x = 16 * player_loc["x"]
sprite.y = 16 * player_loc["y"]

# Add the Group to the Display
display.root_group = group

prev_btn_vals = ugame.buttons.get_pressed()

while True:
    cur_btn_vals = ugame.buttons.get_pressed()
    if not prev_btn_vals & ugame.K_UP and cur_btn_vals & ugame.K_UP:
        player_loc["y"] = max(1, player_loc["y"] - 1)
    if not prev_btn_vals & ugame.K_DOWN and cur_btn_vals & ugame.K_DOWN:
        player_loc["y"] = min(6, player_loc["y"] + 1)

    if not prev_btn_vals & ugame.K_RIGHT and cur_btn_vals & ugame.K_RIGHT:
        player_loc["x"] = min(8, player_loc["x"] + 1)
    if not prev_btn_vals & ugame.K_LEFT and cur_btn_vals & ugame.K_LEFT:
        player_loc["x"] = max(1, player_loc["x"] - 1)

    # update the the player sprite position
    sprite.x = 16 * player_loc["x"]
    sprite.y = 16 * player_loc["y"]

    # update the previous values
    prev_btn_vals = cur_btn_vals
```

Game Code Structure

State Machine

At the highest level inside the main loop, the game program uses a state machine to behave the correct way at any given time, based on what the player has done so far. If you don't know what a state machine is, or just want a refresher there is a nice guide here: https://learn.adafruit.com/circuitpython-101-state-machines (https://adafru.it/DtL)

GAME_STATE Object

The state machine in this game uses numbers to track the state and we give each number a descriptive variable name to use in the code.

The current state that the state machine is in gets stored in `GAME_STATE["STATE"]`. All of the possible states are in `tilegame_assets\states.py`:

```
# SPDX-FileCopyrightText: 2020 FoamyGuy for Adafruit Industries
#
# SPDX-License-Identifier: MIT

# State machine constants

# playing the game: draw the map, listen for D-pad buttons to move player
STATE_PLAYING = 0

# player beat this map level
STATE_MAPWIN = 1

# waiting for player to press A button to continue
STATE_WAITING = 2

# player lost by touching sparky
STATE_LOST_SPARKY = 3

# minerva shows a fun fact
STATE_MINERVA = 4
```

The `"STATE"` item within the `GAME_STATE` dictionary is the only one that refers to the state machine logic.

The rest of the items in the `GAME_STATE` dictionary store information about the current game and map.

Here are the starting values:

```
GAME_STATE = {
    # hold the map state as it came out of the csv. Only holds non-entities.
    "ORIGINAL_MAP": {},
```

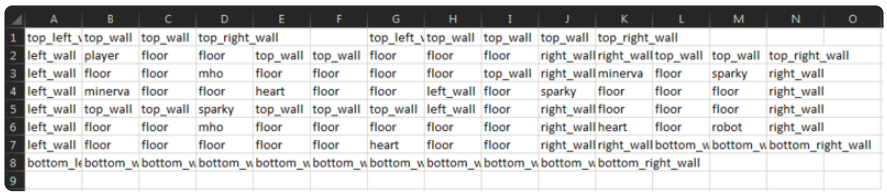
```
    # hold the current map state as it changes. Only holds non-entities.
    "CURRENT_MAP": {},
    # Dictionary with touple keys that map to lists of entity objects.
    # Each one has the index of the sprite in the ENTITY_SPRITES list
    # and the tile type string
    "ENTITY_SPRITES_DICT": {},
    # hold the location of the player in tile coordinates
    "PLAYER_LOC": (0, 0),
    # list of items the player has in inventory
    "INVENTORY": [],
    # how many hearts there are in this map level
    "TOTAL_HEARTS": 0,
    # sprite object to draw for the player
    "PLAYER_SPRITE": None,
    # size of the map
    "MAP_WIDTH": 0,
    "MAP_HEIGHT": 0,
    # which map level within MAPS we are currently playing
    "MAP_INDEX": 0,
    # current state of the state machine
    "STATE": STATE_PLAYING,
}
```

Their values will change as the map gets loaded and the player moves around and does things within the game.

CSV Maps and Tile Types

Map File

The game will load the maps from CSV (Comma Separated Value) files. We can edit these with any spreadsheet program. Here is an example of a map:



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	top_left_v	top_wall	top_wall	top_right_wall			top_left_v	top_wall	top_wall	top_wall	top_right_wall				
2	left_wall	player	floor	floor	top_wall	top_wall	floor	floor	floor	right_wall	right_wall	top_wall	top_wall	top_right_wall	
3	left_wall	floor	floor	mho	floor	floor	floor	floor	top_wall	right_wall	minerva	floor	sparky	right_wall	
4	left_wall	minerva	floor	floor	heart	floor	floor	left_wall	floor	sparky	floor	floor	floor	right_wall	
5	left_wall	top_wall	top_wall	sparky	top_wall	top_wall	top_wall	left_wall	floor	right_wall	floor	floor	floor	right_wall	
6	left_wall	floor	floor	mho	floor	floor	floor	floor	floor	right_wall	heart	floor	robot	right_wall	
7	left_wall	floor	floor	floor	floor	floor	heart	floor	floor	right_wall	right_wall	bottom_w	bottom_w	bottom_right_wall	
8	bottom_le	bottom_w	bottom_w	bottom_w	bottom_w	bottom_w	bottom_w	bottom_w	bottom_w	bottom_w	bottom_right_wall				
9															

Any blank cells will get treated as empty tiles that can't be walked on. Every cell that isn't empty must have an entry in the main `TILES` dictionary which is located inside of `tilegame_assets\tiles.py`.

Loading The Map

To load the CSV map we start by loop over each row. `split(',')` breaks the row down into a list which we will also loop over. On each item within the row, we look it up in the **TILES** dictionary. Non-entities get added to the `ORIGINAL_MAP` and `CURRENT_MAP` game state objects.

The `enumerate()` function is used on the loops to easily track `x` and `y` coordinates so we know where to add tiles to the map objects.

Entities also get added to the `ENTITY_SPRITES_DICT` game state object. Their sprites are loaded and added to the `sprite_group` so they'll be drawn on the screen. In the map objects a `"floor"` tile is placed at the location of the entity. The entity will be drawn on top of the floor.

Here is the code that loads the map:

```
def load_map(file_name):
    # pylint: disable=global-statement,too-many-statements,too-many-nested-
blocks,too-many-branches
    global ENTITY_SPRITES, CAMERA_VIEW

    # empty the sprites from the group
    for cur_s in ENTITY_SPRITES:
        group.remove(cur_s)
    # remove player sprite
    try:
        group.remove(GAME_STATE["PLAYER_SPRITE"])
    except ValueError:
        pass

    # reset map and other game state objects
    GAME_STATE["ORIGINAL_MAP"] = {}
    GAME_STATE["CURRENT_MAP"] = {}
    ENTITY_SPRITES = []
    GAME_STATE["ENTITY_SPRITES_DICT"] = {}
    CAMERA_VIEW = {}
    GAME_STATE["INVENTORY"] = []
    GAME_STATE["TOTAL_HEARTS"] = 0

    # Open and read raw string from the map csv file
    f = open("tilegame_assets/{}".format(file_name), "r")
    map_csv_str = f.read()
    f.close()

    # split the raw string into lines
    map_csv_lines = map_csv_str.replace("\r", "").split("\n")

    # set the WIDTH and HEIGHT variables.
    # this assumes the map is rectangular.
    GAME_STATE["MAP_HEIGHT"] = len(map_csv_lines)
    GAME_STATE["MAP_WIDTH"] = len(map_csv_lines[0].split(","))

    # loop over each line storing index in y variable
    for y, line in enumerate(map_csv_lines):
        # ignore empty line
        if line != "":
            # loop over each tile type separated by commas, storing index in x
variable
            for x, tile_name in enumerate(line.split(",")):
                print("%s '%s'" % (len(tile_name), str(tile_name)))

                # if the tile exists in our main dictionary
                if tile_name in TILES.keys():

                    # if the tile is an entity
                    if (
                        "entity" in TILES[tile_name].keys()
                        and TILES[tile_name]["entity"]
```

```
                    ):
                        # set the map tiles to floor
                        GAME_STATE["ORIGINAL_MAP"][x, y] = "floor"
                        GAME_STATE["CURRENT_MAP"][x, y] = "floor"

                        if tile_name == "heart":
                            GAME_STATE["TOTAL_HEARTS"] += 1

                        # if it's the player
                        if tile_name == "player":
                            # Create the sprite TileGrid
                            GAME_STATE["PLAYER_SPRITE"] = displayio.TileGrid(
                                sprite_sheet,
                                pixel_shader=palette,
                                width=1,
                                height=1,
                                tile_width=16,
                                tile_height=16,
                                default_tile=TILES[tile_name]["sprite_index"],
                            )

                            # set the position of sprite on screen
                            GAME_STATE["PLAYER_SPRITE"].x = x * 16
                            GAME_STATE["PLAYER_SPRITE"].y = y * 16

                            # set position in x,y tile coords for reference later
                            GAME_STATE["PLAYER_LOC"] = (x, y)

                            # add sprite to the group
                            group.append(GAME_STATE["PLAYER_SPRITE"])
                        else:  # not the player
                            # Create the sprite TileGrid
                            entity_srite = displayio.TileGrid(
                                sprite_sheet,
                                pixel_shader=palette,
                                width=1,
                                height=1,
                                tile_width=16,
                                tile_height=16,
                                default_tile=TILES[tile_name]["sprite_index"],
                            )
                            # set the position of sprite on screen
                            # default to off the edge
                            entity_srite.x = -16
                            entity_srite.y = -16

                            # add the sprite object to ENTITY_SPRITES list
                            ENTITY_SPRITES.append(entity_srite)
                            # print("setting GAME_STATE['ENTITY_SPRITES_DICT'][%s,
%s]" % (x,y))

                            # create an entity obj
                            _entity_obj = {
                                "entity_sprite_index": len(ENTITY_SPRITES) - 1,
                                "map_tile_name": tile_name,
                            }

                            # if there are no entities at this location yet
                            if (x, y) not in GAME_STATE["ENTITY_SPRITES_DICT"]:
                                # create a list and add it to the dictionary at the
x,y location
                                GAME_STATE["ENTITY_SPRITES_DICT"][x, y] =
[_entity_obj]
                            else:
                                # append the entity to the existing list in the
dictionary
                                GAME_STATE["ENTITY_SPRITES_DICT"][x, y].append(
                                    _entity_obj
                                )
```

```
                else:  # tile is not entity
                    # set the tile_name into MAP dictionaries
                    GAME_STATE["ORIGINAL_MAP"][x, y] = tile_name
                    GAME_STATE["CURRENT_MAP"][x, y] = tile_name

            else:  # tile type wasn't found in dict
                print("tile: %s not found in TILES dict" % tile_name)
    # add all entity sprites to the group
    print("appending {} sprites".format(len(ENTITY_SPRITES)))
    for entity in ENTITY_SPRITES:
        group.append(entity)
```

Tile Types

These entries define the look and behavior of the tiles. Here is an example of a tile type entry:

```
# ... behavior functions declared above...
TILES = {
  # ... many tile type entries ...
  "mho": {
    "sprite_index": 2,
    "can_walk": True,
    "entity": True,
    "before_move": take_item
  },
  # ... more entries ...
}
```

The keys within the `TILES` dictionary match up with the values from the map CSV file. One of the `mho` tiles is located at cell `D3`. There may be tile keys that aren't present in a particular map, they will have no effect when playing that map level in the game.

These entries define the look and behavior of tiles using the following properties:

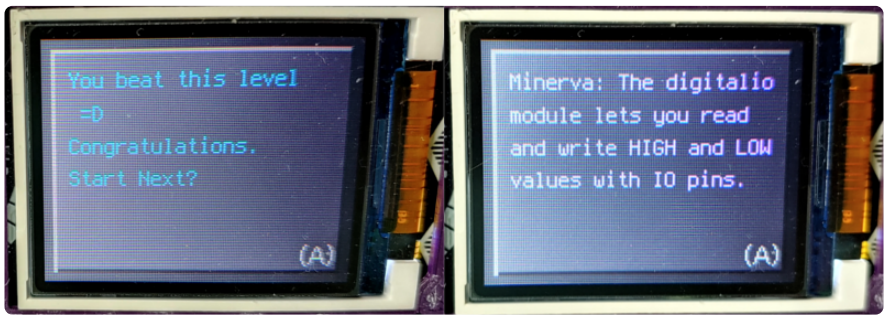
- `sprite_index` - The index within the sprite sheet to show for this tile.
- `can_walk` - Whether the player is allowed to walk on this tile.
- `entity` - Whether this tile represents an entity. If so it will get drawn on top of a floor tile and theoretically it could move around the map. The player is an entity as well as Mho, Heart, Minerva, and Robot. However the player is the only entity in the example game that moves.
- `before_move` - (Optional) If set to a function the function will get called when the player tries to walk on this tile. If the function returns `True` the player will be allowed to walk on the tile, if the function returns `False` the player will not be allowed to walk on it.

Behavior Functions

These functions must be declared before the main **TILES** dictionary because tiles that want special behaviors will get them by setting the `before_move` property in their tile type entry to a behavior function. These are also declared inside the file: **tilegame_assets\tiles.py**. The existing behavior functions are:

- `take_item` - if the item is still here then add it to the player's inventory and remove it from this tile. In the example game both `"mho"` and `"heart"` tile types use this function.
- `sparky_walk` - if the player has a `"Mho"` in their inventory it gets consumed and the Sparky is removed from this tile. If they player has no `"Mho"` they let the smoke out and must restart this level.
- `minerva_walk` - change the game state to `STATE_MINERVA` in this state Minerva will show the player a fun fact and prompt them to press a button to continue. This function always returns `False` so the player is never allowed to actually move on to this tile. The fun facts are stored as a list inside he file: **tilegame_assets\fun_facts.py**. If you add your own facts to this list Minerva will show them to the player sometimes.
- `robot_walk` - if the player has gathered all of the hearts from this map then they are allowed to move to the robot, and they win this level. Game state is changed to `STATE_MAPWIN`. If they have not gathered all available hearts then they are not allowed to move to this tile.

You can declare your own tile types and behavior functions in this file to customize your game!



tiles.py

```
# SPDX-FileCopyrightText: 2020 FoamyGuy for Adafruit Industries
#
# SPDX-License-Identifier: MIT

from tilegame_assets.states import (
```

```
    STATE_MAPWIN,
    STATE_LOST_SPARKY,
    STATE_MINERVA,
)
# pylint: disable=unused-argument


# Minerva before_move. Set game state to STATE_MINERVA
def minerva_walk(to_coords, from_coords, entity_obj, GAME_STATE):
    GAME_STATE["STATE"] = STATE_MINERVA
    return False


# Sparky before_move. If user does not have a Mho in inventory they lose.
# If user does have Mho subtract one from inventory and consume Sparky.
def sparky_walk(to_coords, from_coords, entity_obj, GAME_STATE):
    if GAME_STATE["INVENTORY"].count("mho") > 0:
        GAME_STATE["INVENTORY"].remove("mho")
        GAME_STATE["ENTITY_SPRITES_DICT"][to_coords].remove(entity_obj)
        if len(GAME_STATE["ENTITY_SPRITES_DICT"][to_coords]) == 0:
            del GAME_STATE["ENTITY_SPRITES_DICT"][to_coords]
        if (-1, -1) in GAME_STATE["ENTITY_SPRITES_DICT"]:
            GAME_STATE["ENTITY_SPRITES_DICT"][-1, -1].append(entity_obj)
        else:
            GAME_STATE["ENTITY_SPRITES_DICT"][-1, -1] = [entity_obj]
        return True
    else:
        GAME_STATE["STATE"] = STATE_LOST_SPARKY
        return True


# Robot before_move. If user has all Hearts they win the map.
def robot_walk(to_coords, from_coords, entity_obj, GAME_STATE):
    if GAME_STATE["INVENTORY"].count("heart") == GAME_STATE["TOTAL_HEARTS"]:
        GAME_STATE["STATE"] = STATE_MAPWIN
        return True

    return False


# Remove the item from this location and add it to player inventory.
def take_item(to_coords, from_coords, entity_obj, GAME_STATE):
    print(entity_obj)
    GAME_STATE["INVENTORY"].append(entity_obj["map_tile_name"])
    GAME_STATE["ENTITY_SPRITES_DICT"][to_coords].remove(entity_obj)
    if len(GAME_STATE["ENTITY_SPRITES_DICT"][to_coords]) == 0:
        del GAME_STATE["ENTITY_SPRITES_DICT"][to_coords]

    if (-1, -1) in GAME_STATE["ENTITY_SPRITES_DICT"]:
        GAME_STATE["ENTITY_SPRITES_DICT"][-1, -1].append(entity_obj)
    else:
        GAME_STATE["ENTITY_SPRITES_DICT"][-1, -1] = [entity_obj]

    return True


# main dictionary that maps tile type strings to objects.
# each one stores the sprite_sheet index and any necessary
# behavioral stats like can_walk or before_move
TILES = {
    # empty strings default to floor and no walk.
    "": {"sprite_index": 10, "can_walk": False},
    "floor": {"sprite_index": 10, "can_walk": True},
    "top_wall": {"sprite_index": 7, "can_walk": False},
    "top_right_wall": {"sprite_index": 8, "can_walk": False},
    "top_left_wall": {"sprite_index": 6, "can_walk": False},
    "bottom_right_wall": {"sprite_index": 14, "can_walk": False},
    "bottom_left_wall": {"sprite_index": 12, "can_walk": False},
    "right_wall": {"sprite_index": 11, "can_walk": False},
```

```
    "left_wall": {"sprite_index": 9, "can_walk": False},
    "bottom_wall": {"sprite_index": 13, "can_walk": False},
    "robot": {
        "sprite_index": 1,
        "can_walk": True,
        "entity": True,
        "before_move": robot_walk,
    },
    "heart": {
        "sprite_index": 5,
        "can_walk": True,
        "entity": True,
        "before_move": take_item,
    },
    "mho": {
        "sprite_index": 2,
        "can_walk": True,
        "entity": True,
        "before_move": take_item,
    },
    "sparky": {
        "sprite_index": 4,
        "can_walk": True,
        "entity": True,
        "before_move": sparky_walk,
    },
    "minerva": {
        "sprite_index": 3,
        "can_walk": True,
        "entity": True,
        "before_move": minerva_walk,
    },
    "player": {"sprite_index": 0, "entity": True,},
}
```

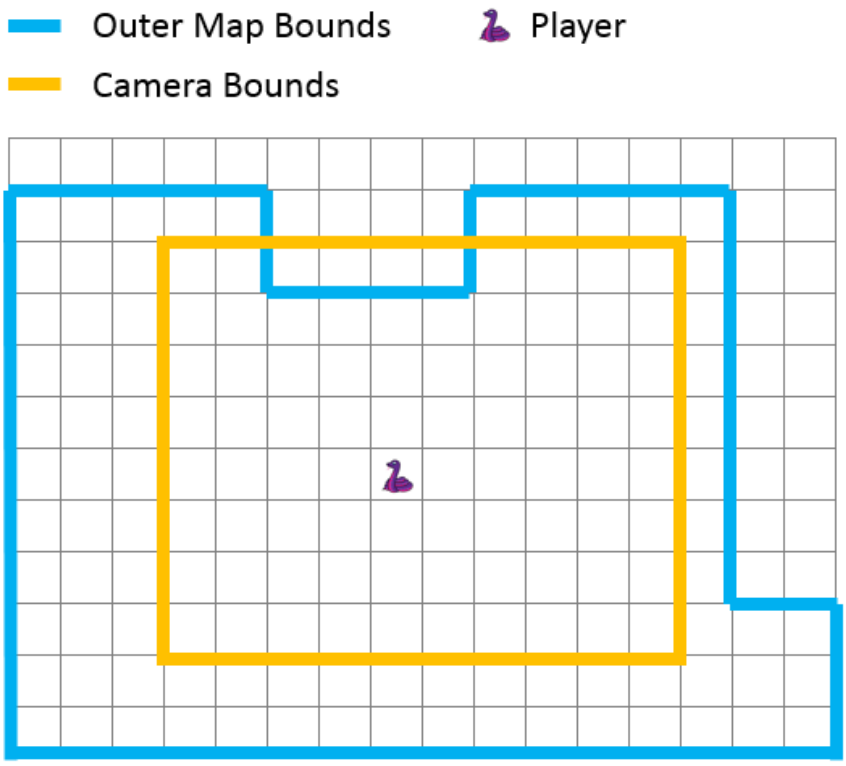
Camera View

In order to allow the map to be larger than the screen, or shaped differently than a plain rectangle we use a `CAMERA_VIEW` dictionary. Just like the full map dictionary, the keys are (x,y) coordinate tuples and the values are tile type strings. But unlike the full map dictionary the `CAMERA_VIEW` is always the exact same size and shape 10x8 tiles in this game, which matches up with the screen size on the PyGamer and PyBadge devices.

There are two functions related to camera management: `set_camera_view()` and `draw_camera_view()`.

`set_camera_view()` will build the `CAMERA_VIEW` dictionary based on the current game state. It accepts parameters for x, y starting point within the map as well as width and height. The example game will always use `10, 8` for the size so it fits perfectly on the `160x128` pixel screen. If you wanted to adapt the game code to work with a different sized screen you could change these values.

`draw_camera_view()` will draw the current contents of the `CAMERA_VIEW` dictionary onto the screen. It will also check for the existence of and draw any entities that are within the coordinates of the `CAMERA_VIEW`.



In this game, the camera will always stay centered on the player because we reference the player location in the x, and y value parameters when `set_camera_view()` is called.

Here are the `set_camera_view()` and `draw_camera_view()` function definitions:

```
# set the appropriate tiles into the CAMERA_VIEW dictionary
# based on given starting coords and size
def set_camera_view(startX, startY, width, height):
    global CAMERA_OFFSET_X
    global CAMERA_OFFSET_Y
    # set the offset variables for use in other parts of the code
    CAMERA_OFFSET_X = startX
    CAMERA_OFFSET_Y = startY

    # loop over the rows and indexes in the desired size section
    for y_index, y in enumerate(range(startY, startY + height)):
        # loop over columns and indexes in the desired size section
        for x_index, x in enumerate(range(startX, startX + width)):
            # print("setting camera_view[%s,%s]" % (x_index,y_index))
            try:
                # set the tile at the current coordinate of the MAP into the
CAMERA_VIEW
                CAMERA_VIEW[x_index, y_index] = GAME_STATE["CURRENT_MAP"][x, y]
            except KeyError:
                # if coordinate is out of bounds set it to floor by default
                CAMERA_VIEW[x_index, y_index] = "floor"
```

```
# draw the current CAMERA_VIEW dictionary and the GAME_STATE['ENTITY_SPRITES_DICT']
def draw_camera_view():
    # list that will hold all entities that have been drawn based on their MAP
location
    # any entities not in this list should get moved off the screen
    drew_entities = []
    # print(CAMERA_VIEW)

    # loop over y tile coordinates
    for y in range(0, SCREEN_HEIGHT_TILES):
        # loop over x tile coordinates
        for x in range(0, SCREEN_WIDTH_TILES):
            # tile name at this location
            tile_name = CAMERA_VIEW[x, y]

            # if tile exists in the main dictionary
            if tile_name in TILES.keys():
                # if there are entity(s) at this location
                if (x + CAMERA_OFFSET_X, y + CAMERA_OFFSET_Y) in GAME_STATE[
                    "ENTITY_SPRITES_DICT"
                ]:
                    # default background for entities is floor
                    castle[x, y] = TILES["floor"]["sprite_index"]

                    # if it's not the player
                    if tile_name != "player":
                        # loop over all entities at this location
                        for entity_obj_at_tile in GAME_STATE["ENTITY_SPRITES_DICT"][
                            x + CAMERA_OFFSET_X, y + CAMERA_OFFSET_Y
                        ]:
                            # set appropriate x,y screen coordinates
                            # based on tile coordinates
                            ENTITY_SPRITES[
                                int(entity_obj_at_tile["entity_sprite_index"])
                            ].x = (x * 16)
                            ENTITY_SPRITES[
                                int(entity_obj_at_tile["entity_sprite_index"])
                            ].y = (y * 16)

                            # add the index of the entity sprite to the
draw_entities
                            # list so we know not to hide it later.
                            drew_entities.append(
                                entity_obj_at_tile["entity_sprite_index"]
                            )

                else:  # no entities at this location
                    # set the sprite index of this tile into the CASTLE dictionary
                    castle[x, y] = TILES[tile_name]["sprite_index"]

            else:  # tile type not found in main dictionary
                # default to floor tile
                castle[x, y] = TILES["floor"]["sprite_index"]

            # if the player is at this x,y tile coordinate accounting for camera
offset
            if GAME_STATE["PLAYER_LOC"] == ((x + CAMERA_OFFSET_X, y +
CAMERA_OFFSET_Y)):
                # set player sprite screen coordinates
                GAME_STATE["PLAYER_SPRITE"].x = x * 16
                GAME_STATE["PLAYER_SPRITE"].y = y * 16

    # loop over all entity sprites
    for index in range(0, len(ENTITY_SPRITES)):
        # if the sprite wasn't drawn then it's outside the camera view
        if index not in drew_entities:
            # hide the sprite by moving it off screen
```

```
            ENTITY_SPRITES[index].x = int(-16)
            ENTITY_SPRITES[index].y = int(-16)
```

In this game, the camera will always stay mostly centered on the player, because we reference the player location in the x, and y value parameters when `set_camera_view()` is called. The max and min functions are used to minimize the amount of "outside the map" area that we show.

Here is the code inside the main loop that calls `set_camera_view()` and `draw_camera_view()`:

```
# inside main game loop:
  set_camera_view(
    max(min(GAME_STATE['PLAYER_LOC'][0]-4,GAME_STATE['MAP_WIDTH']-
SCREEN_WIDTH_TILES),0),
    max(min(GAME_STATE['PLAYER_LOC'][1]-3,GAME_STATE['MAP_HEIGHT']-
SCREEN_HEIGHT_TILES),0),
    10,
    8
  )
# draw the camera
draw_camera_view()
```