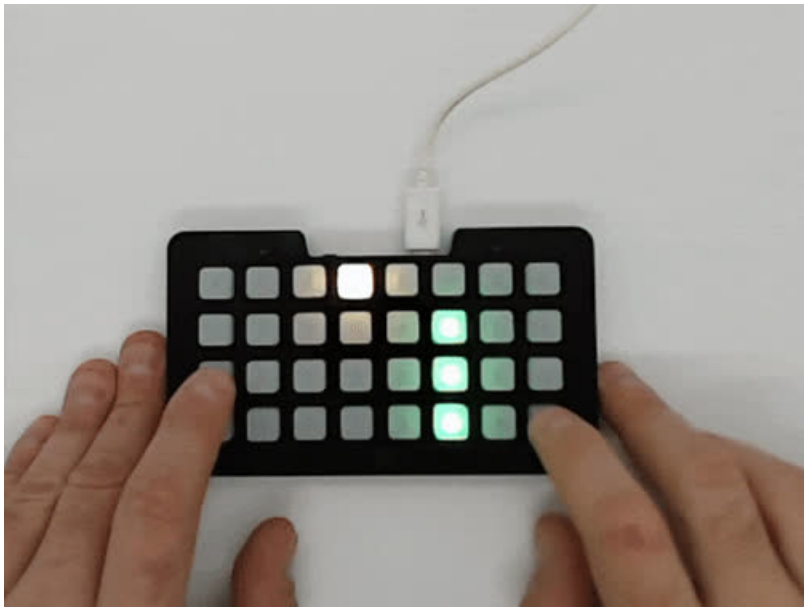
adafruit learning system

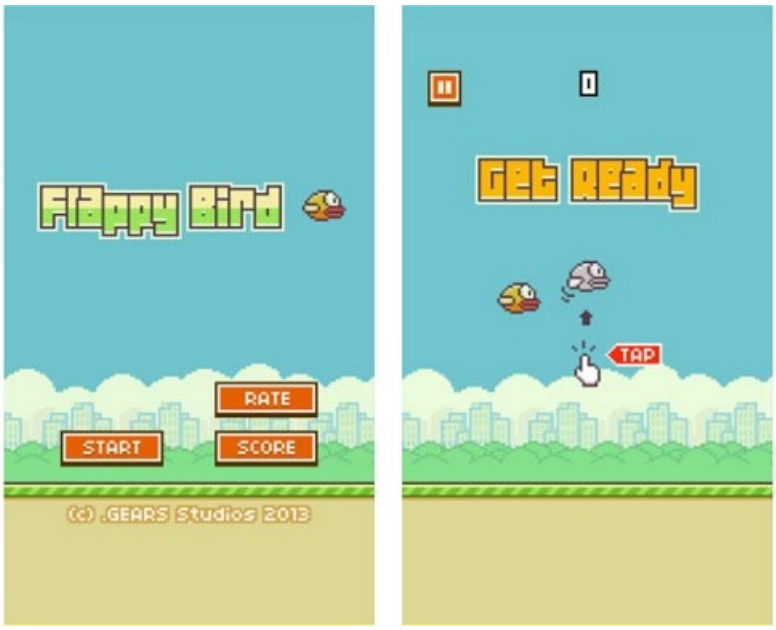
FlappyBird Game for NeoTrellis M4 in CircuitPython

Created by Dave Astels



Last updated on 2019-03-27 02:54:37 PM UTC

Overview



FlappyBird is more than a game. It's a phenomenon. It's a legend. In some ways it's the new Hello World: one of the first programs someone writes as they start exploring programming.

This guide will walk you through a FlappyBird style game implemented in CircuitPython on the NeoTrellis M4 Express. That's right: a video game on the NeoTrellis. Why not? It has lights and buttons. That means you can write games for it. Furthermore, those lights are in a matrix (albeit a scant 4x8) which means they can be used to display more than a single bit at a time.

Parts

Since this board has a SAMD51 at it's core, there's plenty of room to write well structured object-oriented code, so this project will use a few software engineering *best practices* while we're at it.

Your browser does not support the video tag. Adafruit NeoTrellis M4 with Enclosure and Buttons Kit Pack

$59.95
IN STOCK

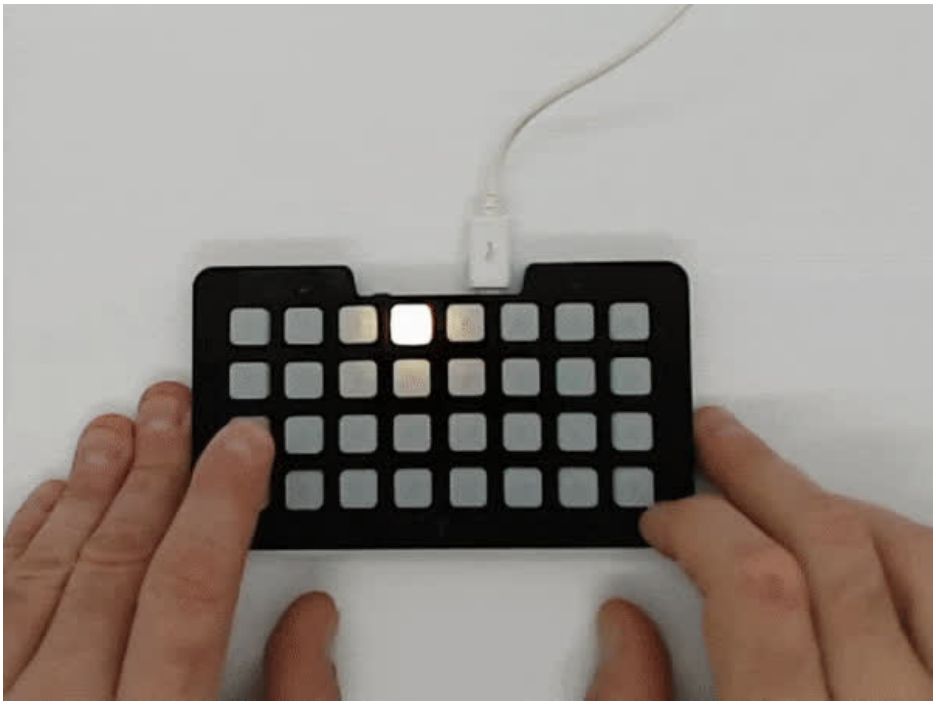
ADD TO CART

USB cable - USB A to Micro-B

$2.95
OUT OF STOCK

OUT OF STOCK

Objectives



Goal

First, what are we trying to do. That's simple: make a FlappyBird clone.

Ok, so what is FlappyBird? It's a simple game of the infinite level type. In this case, you control a bird: pressing a button makes it flap its wings which increases its height. Gravity is also acting on the bird, causing it to fall over time. To complicate things, an unending series of random obstacles enter from the left side of the screen, which must be avoided by flying over, under, or through holes in them. If the bird hits an obstacle, it's "game over".

A simple game: one input (press any button or shake), one character to move up and down, and a series of obstacles that enter from the right and move across the screen and leave on the left.

Some complications can be thrown in: the obstacles are random in size and structure, and the speed at which they move can increase over time.

Design

Let's write this as an object-oriented system. So, the first question is: "What are the things?"

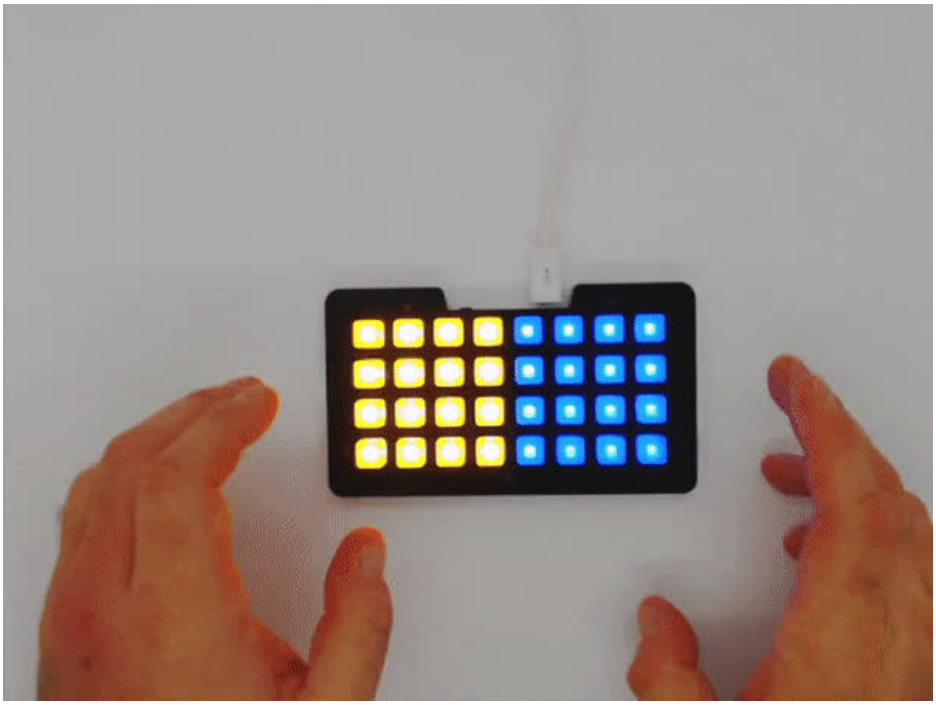
Let's start by having a `Game` class that implements the rules of play, initial state, win conditions (or in this case, loose conditions).

Next we have a class to implement the player character: the `Bird` class. That's the player character in the game. It's what the player has (some) control over.

Finally we need a class to implement the obstacles, which is the `Post` class, because they look sort of like posts.

The `Game` object is the central thing, it has a single `Bird` instance and zero or more `Post` objects that are on the screen. `Game` has a run function that loops until the gamer is over, i.e. until the bird collides with a post.

Each time though the run function's loop, it will look at input and make adjustments to the bird's altitude based on gravity and user activity. Occasionally the scene will be advanced, moving posts to the right. It's at this point that a collision between the bird and a post is checked for and dealt with. Even more infrequently, a new post will be added to the right side of the screen.



Use

When the system starts, it presents the user with a choice of using the accelerometer (shake) to flap, or pressing buttons. Choice is shown by filling the left half of the Trellis with yellow, and the right half with blue.

Pressing a yellow button selects the accelerometer while blue selects using buttons.

The game then starts. The bird (yellow pixel) will gradually fall under the effects of the game's gravity. Flapping (by shaking or pressing any button) will cause the bird to move up. Over time green posts will entry from the right and move across the display, disappearing off the left side. The player's job is to maneuver the bird to avoid hitting the posts. As the game progresses, the speed of the posts increases. Initially posts extend from the bottom of the screen. In time they can extend from the top as well. Eventually there will be posts that extend from both top and bottom and the player has to guide the bird between the two pieces of the post.

Code Walkthrough

We'll be using CircuitPython for this project. Are you new to using CircuitPython? No worries, there is a full getting started guide here (https://adafru.it/cpy-welcome).

Adafruit suggests using the Mu editor to edit your code and have an interactive REPL in CircuitPython. You can learn about Mu and its installation in this tutorial (https://adafru.it/ANO).

There's a guide to get you up and running with CircuitPython specifically for the NeoTrellisM4 (https://adafru.it/C-O). You should read it before starting to get the most recent CircuitPython build for the NeoTrellisM4 installed and running along with the required libraries.

Full code with links to github is on the Downloads page.

Navigating the NeoTrellis

To get your NeoTrellis M4 set up to run this project's code, first follow these steps:

1) Update the bootloader for NeoTrellis (https://adafru.it/C-N)from the NeoTrellis M4 guide

2) Install the latest CircuitPython for NeoTrellis (https://adafru.it/C-O)from the NeoTrellis M4 guide

3) Get the latest library pack (https://adafru.it/zB-), matched for the version of CircuitPython you are running, unzip it, and drag the libraries you need over into the **/lib** folder on **CIRCUITPY**. The latest library package includes support for NeoTrellis.
https://github.com/adafruit/Adafruit_CircuitPython_Bundle/releases/ (https://adafru.it/zB-)

For this project you will need the following libraries:

- **adafruit_trellism4.mpy**
- **adafruit_adxl34x**
- **adafruit_bus_device** directory
- **neopixel.mpy**
- **adafruit_matrixkeypad.mpy**

Code Walkthrough

main.py

This is the code that gets run when the system comes up. It does three things.

First it sets things up: the trellis, accelerometer, and game object.

```
trellis = adafruit_trellism4.TrellisM4Express()
trellis.pixels.auto_write = False

i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
accelerometer = adafruit_adxl34x.ADXL345(i2c)

the_game = game.Game(trellis, accelerometer)
```

Next, it colors the buttons (making, in effect, two big buttons) and let's the user choose whether to use buttons or motion to flap.

```
for x in range(8):
    for y in range(4):
        if x > 3:
            trellis.pixels[x, y] = BLUE
        else:
            trellis.pixels[x, y] = YELLOW
trellis.pixels.show()

keys = []
while not keys:
    keys = trellis.pressed_keys
```

Finally it runs the game, starting a new one when the previous one finishes.

```
while True:
    the_game.play(keys[0][0] < 4)        # False = key, True = accel
```

game.py

This class contains the rules of the game. It orchestrates game play, player interaction, and display. There's a fair bit of code here, but we'll walk through it section by section.

First there's the initialization code. There's the usual `__init__` method that initializes the new instance, but there's also a `_restart` method that is run to start each new game.

```
def __init__(self, trellis, accel, ramp=20, challenge_ramp=30):
       """initialize a Game instance.
       trellis        -- the TrellisM4Express instance to use as input and screen.
       accel          -- the accelerometer interface object to use as input
       ramp           -- how often (in steps) to increase the speed (default 20)
       challenge_ramp -- how often (in steps) to increase the challenge of the posts
       """
       self._trellis = trellis
       self._accel = accel
       self._delay_ramp = ramp
       self._challenge_ramp = challenge_ramp
       self._bird = Bird()
       self._posts = []
       self._interstitial_delay = 1.0
       self._challenge = 10
       self._currently_pressed = set([])
       self._previous_accel_reading = (None, None, None)
       self._previous_shake_result = False

   def _restart(self):
       """Restart the game."""
       self._bird = Bird()
       self._posts = []
       self._interstitial_delay = 1.0
       self._challenge = 10
```

Next, the play method contains the main game loop that controls everything.

Once the game is restarted, the loop begins. It keeps looping until the bird collides with a post.

The first thing that happens in the loop is to grab the current time. Each time through the loop `_update_bird` is called to apply user action and gravity to the bird and update it's position and the display.

The rest of the loop (other than a small delay at the end) is executed only when it's time to move the posts. This interval starts out at a second and gradually gets faster as the game progresses.

Posts are updated and collisions between the bird and the posts is checked for. Nothing is done about a collision at this point, but the information is stored for later. What does happen is that a post is possibly added. This happens occasionally with greater frequency as the game goes on; there's a little randomness added so that it's not completely predictable. Whether or not a post was added, the display is updated.

Next, handling a collision. What happens is that we simply have the bird flash. If there was no collision, we speed up the game and/or increase the challenge level of the posts if it's time to.

The run function is below. notice how most of the details are deferred to other methods in `Game`, or to the `Bird` or `Post` objects.

```
    def play(self, mode=False):
        """Play the game.
        mode -- input mode: False is key, True is accel
        """
        self._restart()
        collided = False
        count = 0
        last_tick = 0
        while not collided:
            now = time.monotonic()
            self._update_bird(mode)
            if now >= last_tick + self._interstitial_delay:
                last_tick = now
                count += 1
                self._update()
                collided = self._check_for_collision()
                if count % (self._challenge - random.randint(0, 4) + 2) == 0:
                    self._add_post()
                self._update_display()
                # handle collision or wait and repeat
                if collided:
                    self._bird.flash(self._trellis)
                else:
                    # time to speed up?
                    if count % self._delay_ramp == 0:
                        self._interstitial_delay -= 0.1
                    # time to increase challenge of the posts?
                    if self._challenge > 0 and count % self._challenge_ramp == 0:
                        self._challenge -= 1
            time.sleep(0.05)
```

Now to look at some details. First, updating the player's bird object.

Erase the bird pixel, if the user has requested a flap, have the bird flap. Otherwise have the bird fall due to gravity. Finally redraw the bird in it's (possibly) new location.

```
    def _update_bird(self, mode):
        """Update the vertical position of the bird based on user activity and gravity.
        mode -- input mode: False is key, True is accel
        """
        self._bird.draw_on(self._trellis, BLACK)
        if self._should_flap(mode):
            self._bird.flap()
        else:
            self._bird.update()
        self._bird.draw_on(self._trellis)
        self._trellis.pixels.show()
```

What does `_should_flap` do?

It polls user input depending on the choice the user made initially.

```
    def _shaken(self):
        """Return whether the Trellis is shaken."""
        last_result = self._previous_shake_result
        result = False
        x, y, z = self._accel.acceleration
        if self._previous_accel_reading[0] is not None:
            result = math.fabs(self._previous_accel_reading[2] - z) > 4.0
        self._previous_accel_reading = (x, y, z)
        self._previous_shake_result = result
        return result and not last_result

    def _key_pressed(self):
        """Return whether a key was pressed since last time."""
        pressed = set(self._trellis.pressed_keys)
        key_just_pressed = len(pressed - self._currently_pressed) > 0
        self._currently_pressed = pressed
        return key_just_pressed

    def _should_flap(self, mode):
        """Return whether the user wants the bird to flap.
        mode -- input mode: False is key, True is accel
        """
        if mode:
            return self._shaken()
        return self._key_pressed()
```

The `_update` method handles the posts: moving them to the left and deleting them as they move off the left side of the screen.

```
    def _update(self):
        """Perform a periodic update: move the posts and remove any that go off the screen."""
        for post in self._posts:
            post.update()
        if self._posts and self._posts[0].off_screen:
            self._posts.pop(0)
```

Like much of the code, checking for a collision defers to something else for a significant part of the job. In this case, most of the work is done my asking the bird if it has collided with each post in turn.

```
    def _check_for_collision(self):
        """Return whether this bird has collided with a post."""
        collided = False
        for post in self._posts:
            collided |= self._bird.is_colliding_with(post)
        return collided
```

Next up is `_update_display` which simply clears the screen, has each post draw itself, has the bird draw itself, and refreshes the screen.

```
    def _update_display(self):
        """Update the screen."""
        self._trellis.pixels.fill(BLACK)
        for post in self._posts:
            post.draw_on(self._trellis)
        self._bird.draw_on(self._trellis)
        self._trellis.pixels.show()
```

The last piece of Game is adding a post. Adding a post simply creates a new post and appends it to the list of posts.

Creating a new post is a bit more complex. Depending on the current challenge level (which starts at 10 and decreases) the form of the post can vary. If the value is high (meaning a low challenge level) only posts that extend from the bottom are created. If the challenge level is moderate, there's an equal chance of posts extending from the top as well as ones that extend from the bottom. At higher challenge levels, there can also be posts that extend from both the top and bottom.

```
    def _new_post(self):
        """Return a new post based on the current challenge level"""
        blocks = random.randint(1, 3)
        # bottom post
        if self._challenge > 6:
            return Post(from_bottom=blocks)
        # top possible as well
        if self._challenge > 3:
            if random.randint(1, 2) == 1:
                return Post(from_bottom=blocks)
            return Post(from_top=blocks)
        # top, bottom, and both possible
        r = random.randint(1, 3)
        if r == 1:
            return Post(from_bottom=blocks)
        if r == 2:
            return Post(from_top=blocks)
        return Post(from_bottom=blocks, from_top=random.randint(1, 4 - blocks))

    def _add_post(self):
        """Add a post."""
        self._posts.append(self._new_post())
```

bird.py

The `Bird` class is much simpler. It just manages the bird. Initialization is simple: position and weight are set. The weight combines with the game's gravity to determine how quickly the bird falls toward the bottom of the screen.

```
    def __init__(self, weight=0.5):
        """Initialize a Bird instance.
        weight -- the weight of the bird (default 0.5)
        """
        self._position = 0.75
        self._weight = weight
```

The vertical position of the bird is maintained as a float between 0.0 and 1.0. We have the `_y_position` method to convert this fractional position to a pixel coordinate that can be used for display and collision detection.

```
    def _y_position(self):
        """Get the verical pixel position."""
        if self._position >= 0.75:
            return 0
        elif self._position >= 0.5:
            return 1
        elif self._position >= 0.25:
            return 2
        return 3
```

Flapping and updating due to gravity move the bird up or down, clipping at 1.0 and 0.0.

```
    def _move_up(self, amount):
        """Move the bird up.
        amount -- how much to move up, 0.0-1.0
        """
        self._position = min(1.0, self._position + amount)

    def _move_down(self, amount):
        """Move the bird down.
        amount -- how much to move down, 0.0-1.0
        """
        self._position = max(0.0, self._position - amount)

    def flap(self):
        """Flap. This moves the bird up by a fixed amount."""
        self._move_up(0.25)

    def update(self):
        """Periodic update: add the effect of gravity."""
        self._move_down(0.05 * self._weight)
```

Collision detection is a classic case of what's called double dispatch and serves to maintain encapsulation. Encapsulation is a fundamental tenet of object-oriented design. The idea is that only the bird knows where it is, and only a post knows where it is.

So how does anything happen? The game knows the bird and all the posts, so it asks the bird if it's colliding with each post. The bird then asks the post if there's a collision at the bird's location. It's a little convoluted, but it means that only the bird knows it's location, or how it's computed, and only the post has to worry about how to determine if a collision occurred.

Why bother? Well, it means that you can change the internal details of any of the pieces, and everything will still work if the public API is kept the same. It also means that each object is in control of its information.

```
    def is_colliding_with(self, post):
        """Check for a collision.
        post -- the Post instance to check for a collicion with
        """
        return post.is_collision_at(3, self._y_position())
```

The last bit of the `Bird` class deals with the display. The `draw_on` method simply sets the appropriate pixel to the specified color, while `flash` toggles it between yellow and red 5 times.

```
    def draw_on(self, trellis, color=YELLOW):
        """Draw the bird.
        trellis -- the TrellisM4Express instance to use as a screen
        color   -- the color to display as (default YELLOW)
        """
        trellis.pixels[3, self._y_position()] = color

    def flash(self, trellis):
        """Flash between RED and YELLOW to indicate a collision.
        trellis -- the TrellisM4Express instance to use as a screen """
        for _ in range(5):
            time.sleep(0.1)
            self.draw_on(trellis, RED)
            trellis.pixels.show()
            time.sleep(0.1)
            self.draw_on(trellis, YELLOW)
            trellis.pixels.show()
```

post.py

The `Post` class implements the behaviors of posts, the obstacles in the game. It starts with the initializer that sets the new post to start at the far right side of the screen and extending from the top and bottom as specified.

```
    def __init__(self, from_bottom=0, from_top=0):
        """Initialize a Post instance.
        from_bottom -- how far the post extends from the bottom of the screen (default 0)
        from_top    -- how far the post extends from the top of the screen (default 0)
        """
        self._x = 7
        self._top = from_top
        self._bottom = from_bottom
```

The `update` method moves the post one pixel to the left.

```
    def update(self):
        """Periodic update: move one step to the left."""
        self._x -= 1
```

Checking whether a given pixel lies on a post is the job of the `_on_post` method. The post knows what pixels it covers, so where better to put the code that answers questions relating to it.

```
    def _on_post(self, x, y):
        """Determine whether the supplied coordinate is occupied by part of this post.
        x -- the horizontal pixel coordinate
        y -- the vertical pixel coordinate
        """
        return x == self._x and (y < self._top or y > (3 - self._bottom))
```

Note that this is done as a private method and not part of the public API of the `Post` class. It's not used by code outside of `Post` objects. Specifically, it's used by the next two methods: `draw_on` sets the pixels where the post is, and `is_collision_at` which returns whether the given location is occupied by the post.

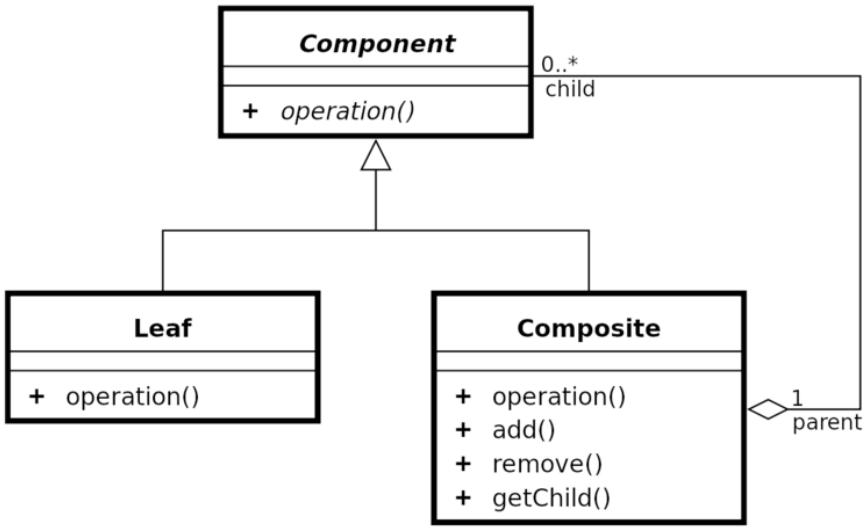
```
    def draw_on(self, trellis):
        """Draw this post on the screen.
        trellis -- the TrellisM4Express instance to use as a screen
        """
        for i in range(4):
            if self._on_post(self._x, i):
                trellis.pixels[self._x, i] = GREEN

    def is_collision_at(self, x, y):
        """Determine whether something at the supplied coordinate is colliding with this post.
        x -- the horizontal pixel coordinate
        y -- the vertical pixel coordinate
        """
        return self._on_post(x, y)
```

Finally there's a property that indicates whether the post has gone off the left edge of the screen.

```
    @property
    def off_screen(self):
        """Return whether this post has moved off the left edge of the screen."""
        return self._x < 0
```

Design Discussion



One of the goals of this project is to take advantage of the memory and performance of the SAMD51 on the NeoTrellis M4 Express to use some more advanced (some might say proper) object-oriented techniques.

Composition

We started by breaking down the game into it's parts: a bird flying to avoid running into obstacles (which we called posts), and a game entity that managed the bird and posts and oversaw the rules and mechanics of play. The game maintains an instance of the `Bird` class and a list of `Post` instances.

Encapsulation

This is one of the most important, and sadly often ignored, principles of object-oriented design. Put simply, nothing outside of an object has any business knowing anything about the object other than the public API is exposes. The espousal of encapsulation can be seen in the generous use of *private* instance variables and methods (those that begin with an underscore).

Python's property mechanism is perfect for enforcing encapsulation. It's not always bad to expose internal details, but the object should generally be in control of it. As the system gets more complex it becomes more important to maintain control over who gets access to what so as to avoid unexpected consequences.

Double Dispatch

In this code, the game has a bird and several posts. It wants to know if the bird collided with a post.

The brute force approach is to get the coordinates from the bird and from each post and do the math to figure out if there is a collision. However, doing so throws out much of the advantage of using an object-oriented system.

The object-oriented way is to put the computation where the data is. In this case the data is in two places: the bird

knows where it is, and each post knows where it is. Look again at the Game code for detecting a collision.

```
    def _check_for_collision(self):
        """Return whether this bird has collided with a post."""
        collided = False
        for post in self._posts:
            collided |= self._bird.is_colliding_with(post)
        return collided
```

For each post, the bird is asked if it is colliding with that post. Next let's look at the `is_colliding_with` method in `Bird`.

```
    def is_colliding_with(self, post):
        """Check for a collision.
        post -- the Post instance to check for a collicion with
        """
        return post.is_collision_at(3, self._y_position())
```

The bird knows where it is, and simply asks the post if it is occupying that location. The post knows where it is and has been given a location to check against. Now all the information needed is at hand and the computation can be done by the `is_collision_at` method.

```
    def _on_post(self, x, y):
        """Determine whether the supplied coordinate is occupied by part of this post.
        x -- the horizontal pixel coordinate
        y -- the vertical pixel coordinate
        """
        return x == self._x and (y < self._top or y > (3 - self._bottom))

    def is_collision_at(self, x, y):
        """Determine whether something at the supplied coordinate is colliding with this post.
        x -- the horizontal pixel coordinate
        y -- the vertical pixel coordinate
        """
        return self._on_post(x, y)
```

Note that the actual comparison is in the `_on_post` method, as this functionality is used in multiple places in the `Post` class (here and `draw_on`). Another fundamental best practice is avoiding the duplication of functionality.

As indicated by the title of this section, this technique is known as *Double Dispatch*. A Post is passed to a method in the Bird, which then calls a method in that Post with information that only the bird knows (the bird's location). This way each object is in control of where its information goes.

Next Steps

So that's the basic game. There's plenty of space on the SAMD51 to add features. Some could be:

- scoring
- different colored posts
- make gravity increase over time (making it harder to stay aloft)
- add starting difficulty selection
- progress tracking, i.e. how many screen advances the player gets before colliding with a post
- dynamic posts that change height as they move

As you can see there are all sorts of additions that could be made. Have fun exploring the possibilities.

Downloads

```
"""
FlappyBird type game for the NeoTrellisM4

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Written by Dave Astels for Adafruit Industries
Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license.

All text above must be included in any redistribution.
"""

# pylint: disable=wildcard-import,unused-wildcard-import,eval-used

import game
import board
import adafruit_trellism4
import adafruit_adxl34x
import busio
from color_names import *


trellis = adafruit_trellism4.TrellisM4Express()
trellis.pixels.auto_write = False

i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
accelerometer = adafruit_adxl34x.ADXL345(i2c)

the_game = game.Game(trellis, accelerometer)

for x in range(8):
    for y in range(4):
        if x > 3:
            trellis.pixels[x, y] = BLUE
        else:
            trellis.pixels[x, y] = YELLOW
trellis.pixels.show()

keys = []
while not keys:
    keys = trellis.pressed_keys

while True:
    the_game.play(keys[0][0] < 4)        # False = key, True = accel
```

```
"""
RGB Color Names

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license.

All text above must be included in any redistribution.
"""

RED = 0xFF0000
MAROON = 0x800000
ORANGE = 0xFF8000
YELLOW = 0xFFFF00
OLIVE = 0x808000
GREEN = 0x008000
AQUA = 0x00FFFF
TEAL = 0x008080
BLUE = 0x0000FF
NAVY = 0x000080
PURPLE = 0x800080
PINK = 0xFF0080
WHITE = 0xFFFFFF
BLACK = 0x000000
```

```
"""
FlappyBird type game for the NeoTrellisM4

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Written by Dave Astels for Adafruit Industries
Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license.

All text above must be included in any redistribution.
"""

# pylint: disable=wildcard-import,unused-wildcard-import,eval-used

import time
import random
import math
from bird import Bird
from post import Post
from color_names import *

BLACK = 0x000000

class Game(object):
    """Overall game control."""

    def __init__(self, trellis, accel, ramp=20, challenge_ramp=30):
        """initialize a Game instance.
```

```
        trellis        -- the TrellisM4Express instance to use as input and screen.
        accel          -- the accelerometer interface object to use as input
        ramp           -- how often (in steps) to increase the speed (default 20)
        challenge_ramp -- how often (in steps) to increase the challenge of the posts
        """
        self._trellis = trellis
        self._accel = accel
        self._delay_ramp = ramp
        self._challenge_ramp = challenge_ramp
        self._bird = Bird()
        self._posts = []
        self._interstitial_delay = 1.0
        self._challenge = 10
        self._currently_pressed = set([])
        self._previous_accel_reading = (None, None, None)
        self._previous_shake_result = False

    def _restart(self):
        """Restart the game."""
        self._bird = Bird()
        self._posts = []
        self._interstitial_delay = 0.5
        self._challenge = 10

    def _update(self):
        """Perform a periodic update: move the posts and remove any that go off the screen."""
        for post in self._posts:
            post.update()
        if self._posts and self._posts[0].off_screen:
            self._posts.pop(0)

    def _shaken(self):
        """Return whether the Trellis is shaken."""
        last_result = self._previous_shake_result
        result = False
        x, y, z = self._accel.acceleration
        if self._previous_accel_reading[0] is not None:
            result = math.fabs(self._previous_accel_reading[2] - z) > 4.0
        self._previous_accel_reading = (x, y, z)
        self._previous_shake_result = result
        return result and not last_result

    def _key_pressed(self):
        """Return whether a key was pressed since last time."""
        pressed = set(self._trellis.pressed_keys)
        key_just_pressed = len(pressed - self._currently_pressed) > 0
        self._currently_pressed = pressed
        return key_just_pressed

    def _should_flap(self, mode):
        """Return whether the user wants the bird to flap.
        mode -- input mode: False is key, True is accel
        """
        if mode:
            return self._shaken()
        return self._key_pressed()

    def _update_bird(self, mode):
        """Update the vertical position of the bird based on user activity and gravity.
```

```
        mode -- input mode: False is key, True is accel
        """
        self._bird.draw_on(self._trellis, BLACK)
        if self._should_flap(mode):
            self._bird.flap()
        else:
            self._bird.update()
        self._bird.draw_on(self._trellis)
        self._trellis.pixels.show()

    def _check_for_collision(self):
        """Return whether this bird has collided with a post."""
        collided = self._bird.did_hit_ground()
        for post in self._posts:
            collided |= self._bird.is_colliding_with(post)
        return collided

    def _update_display(self):
        """Update the screen."""
        self._trellis.pixels.fill(BLACK)
        for post in self._posts:
            post.draw_on(self._trellis)
        self._bird.draw_on(self._trellis)
        self._trellis.pixels.show()

    def _new_post(self):
        """Return a new post based on the current challenge level"""
        bottom_blocks = random.randint(1, 3)
        top_blocks = random.randint(1, 2)
        # bottom post
        if self._challenge > 6:
            return Post(from_bottom=bottom_blocks)
        # top possible as well
        if self._challenge > 3:
            if random.randint(1, 2) == 1:
                return Post(from_bottom=bottom_blocks)
            return Post(from_top=top_blocks)
        # top, bottom, and both possible
        r = random.randint(1, 3)
        if r == 1:
            return Post(from_bottom=bottom_blocks)
        if r == 2:
            return Post(from_top=top_blocks)
        return Post(from_bottom=bottom_blocks, from_top=random.randint(1, 4 - bottom_blocks))

    def _add_post(self):
        """Add a post."""
        self._posts.append(self._new_post())

    def play(self, mode=False):
        """Play the game.
        mode -- input mode: False is key, True is accel
        """
        self._restart()
        collided = False
        count = 0
        last_tick = 0
        while not collided:
            now = time.monotonic()
            self._update_bird(mode)
```

```
            if now >= last_tick + self._interstitial_delay:
                last_tick = now
                count += 1
                self._update()
                collided = self._check_for_collision()
                if count % max(1, (self._challenge - random.randint(0, 4))) == 0:
                    self._add_post()
                self._update_display()
                # handle collision or wait and repeat
                if collided:
                    self._bird.flash(self._trellis)
                else:
                    # time to speed up?
                    if count % self._delay_ramp == 0:
                        self._interstitial_delay -= 0.01
                    # time to increase challenge of the posts?
                    if self._challenge > 0 and count % self._challenge_ramp == 0:
                        self._challenge -= 1
            time.sleep(0.05)
```

```
"""
FlappyBird type game for the NeoTrellisM4

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Written by Dave Astels for Adafruit Industries
Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license.

All text above must be included in any redistribution.
"""

# pylint: disable=wildcard-import,unused-wildcard-import,eval-used

import time
from color_names import *

class Bird(object):
    """The 'bird': the user's piece."""

    def __init__(self, weight=0.5):
        """Initialize a Bird instance.
        weight -- the weight of the bird (default 0.5)
        """
        self._position = 0.75
        self._weight = weight

    def _y_position(self):
        """Get the verical pixel position."""
        if self._position >= 0.75:
            return 0
        elif self._position >= 0.5:
            return 1
        elif self._position >= 0.25:
            return 2
        return 3
```

```
    def _move_up(self, amount):
        """Move the bird up.
        amount -- how much to move up, 0.0-1.0
        """
        self._position = min(1.0, self._position + amount)

    def _move_down(self, amount):
        """Move the bird down.
        amount -- how much to move down, 0.0-1.0
        """
        self._position = max(0.0, self._position - amount)

    def flap(self):
        """Flap. This moves the bird up by a fixed amount."""
        self._move_up(0.25)

    def update(self):
        """Periodic update: add the effect of gravity."""
        self._move_down(0.05 * self._weight)

    def did_hit_ground(self):
        """Return whether this bird hit the ground."""
        return self._position == 0.0

    def is_colliding_with(self, post):
        """Check for a collision.
        post -- the Post instance to check for a collicion with
        """
        return post.is_collision_at(3, self._y_position())

    def draw_on(self, trellis, color=YELLOW):
        """Draw the bird.
        trellis -- the TrellisM4Express instance to use as a screen
        color   -- the color to display as (default YELLOW)
        """
        trellis.pixels[3, self._y_position()] = color

    def flash(self, trellis):
        """Flash between RED and YELLOW to indicate a collision.
        trellis -- the TrellisM4Express instance to use as a screen """
        for _ in range(5):
            time.sleep(0.1)
            self.draw_on(trellis, RED)
            trellis.pixels.show()
            time.sleep(0.1)
            self.draw_on(trellis, YELLOW)
            trellis.pixels.show()
```

```
"""
FlappyBird type game for the NeoTrellisM4

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Written by Dave Astels for Adafruit Industries
Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license
```

```
Licensed under the MIT license.

All text above must be included in any redistribution.
"""

# pylint: disable=wildcard-import,unused-wildcard-import,eval-used

from color_names import *


class Post(object):
    """Obstacles the user must avoice colliding with."""

    def __init__(self, from_bottom=0, from_top=0):
        """Initialize a Post instance.
        from_bottom -- how far the post extends from the bottom of the screen (default 0)
        from_top    -- how far the post extends from the top of the screen (default 0)
        """
        self._x = 7
        self._top = from_top
        self._bottom = from_bottom

    def update(self):
        """Periodic update: move one step to the left."""
        self._x -= 1

    def _on_post(self, x, y):
        """Determine whether the supplied coordinate is occupied by part of this post.
        x -- the horizontal pixel coordinate
        y -- the vertical pixel coordinate
        """
        return x == self._x and (y < self._top or y > (3 - self._bottom))

    def draw_on(self, trellis):
        """Draw this post on the screen.
        trellis -- the TrellisM4Express instance to use as a screen
        """
        for i in range(4):
            if self._on_post(self._x, i):
                trellis.pixels[self._x, i] = GREEN

    def is_collision_at(self, x, y):
        """Determine whether something at the supplied coordinate is colliding with this post.
        x -- the horizontal pixel coordinate
        y -- the vertical pixel coordinate
        """
        return self._on_post(x, y)

    @property
    def off_screen(self):
        """Return whether this post has moved off the left edge of the screen."""
        return self._x < 0
```