CircuitPython Stage game library

Created by Radomir Dopieralski

https://learn.adafruit.com/circuitpython-stage-game-library

Last updated on 2024-03-08 03:30:25 PM EST

Table of Contents

Overview 3

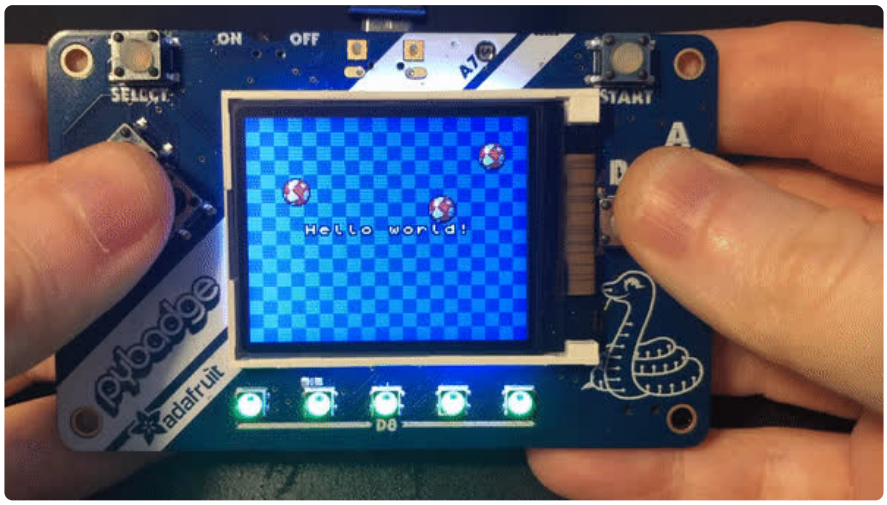
Bouncing Balls 5

- Banks
- Grids
- Stage
- Sprite
- Animations
- Movement
- Multiple Balls
- Text
- Conclusion

Example Games 12

Documentation 13

Overview



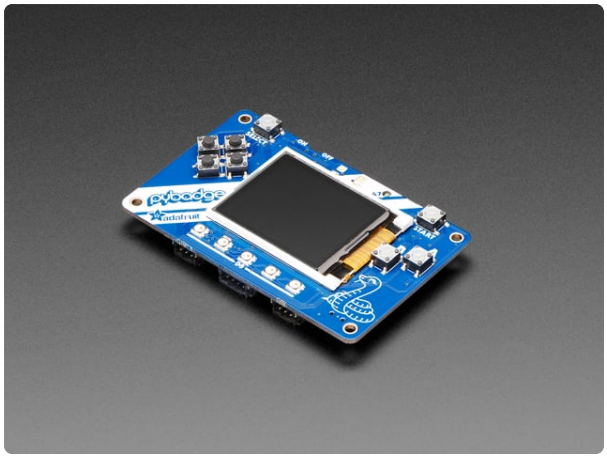
"Stage" is a small library for making simple games with CircuitPython. It lets you display graphics in three ways: as a grid of tiles, as sprites, and as text. Because of its simplicity it is limited to 16×16 images in 16 colors, but that is enough for making games. It is included by default on both PyGamer and PyBadge.

The library consists of two modules. The `stage` module contains all the functions and classes you will need for loading and manipulating the graphics on the screen, while the `ugame` module initializes the screen and gives you access to buttons and sound. Different CircuitPython boards will all use the same `stage` module, but will have different `ugame` modules built-in, because of the differences in what buttons and joysticks they have and how they need to be initialized. The very first board for which this library was originally written was called "µGame", hence the name of the module.

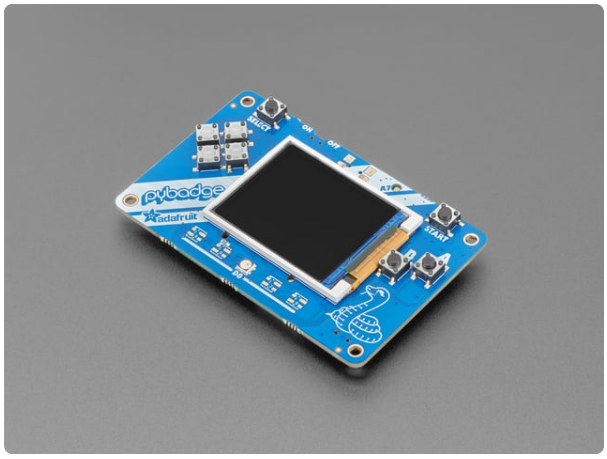
Please note that you can't mix `displayio` graphics with those made using the Stage library — either one or the other can use the display at the same time.

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

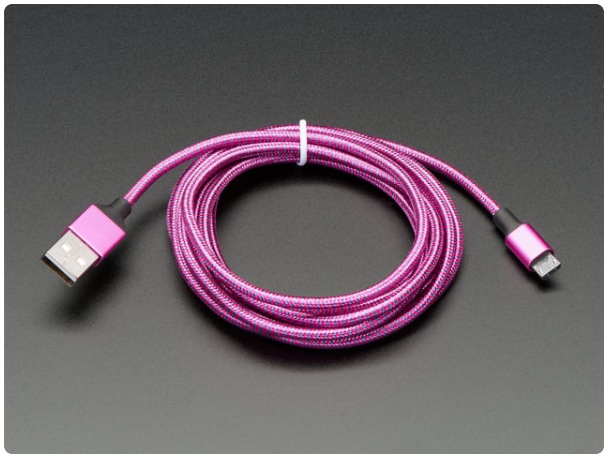
Adafruit PyGamer Starter Kit
Please note: you may get a royal blue or purple case with your starter kit (they're both lovely colors)What fits in your pocket, is fully Open...
https://www.adafruit.com/product/4277



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



Pink and Purple Braided USB A to Micro B Cable - 2 meter long
This cable is super-fashionable with a woven pink and purple Blinka-like pattern! First let's talk about the cover and over-molding. We got these in custom colors,...
https://www.adafruit.com/product/4148

Bouncing Balls

In order to learn how to use this library, we will write a simple program that displays three rotating balls bouncing around the screen.

We will put our program into a file called **code.py**, so that it is automatically run on power-on or restart. Please create this file, open it, and put the following code in it:

```
import ugame
import stage
```

Once you hit “save”, you should see in the console that the board restarts and runs your program. For now there should be no effect of running it, it just imports the two libraries that we are going to use later. If you made a mistake somewhere, however, you will see an error printed on the console, telling you the line number where the error is. Correct it and save again, and the board will restart again and run your code.

Banks

All the graphics used in Stage are organized into so-called `Banks`: sets of 16 images, 16×16 pixels each, with a 16-color palette. You probably noticed that we like the number 16 — it really simplifies a lot of things and makes them fast.

For our bouncing ball demo, we are only going to use a single bank, because we only need 5 images: one for the background, and four for the animated ball. The bank we are going to use looks like this:

In order for the Stage library to be able to load that image, it needs to be saved as a 16-color BMP file onto the CIRCUITPY disk. You can download it this format below:

ball.bmp

https://adafru.it/FaU

If you look at the image, you will notice that it has all the 16 square images stacked together one on top of the other, and that it uses a bright magenta color for the background. That magenta color (100% red, 0% green, 100% blue) is going to always be considered a “transparent” color. This is useful when we want our image of the ball to be round, and not square.

Now let’s add to our program some code for loading that graphics into memory:

```
import ugame
import stage

bank = stage.Bank.from_bmp16("ball.bmp")
```

When you save this, nothing will be displayed, but the file will be read and prepared to be used by our program. If there is any problem with the file, you will get an error on the console.

Grids

Now that we have our graphics loaded into memory, we can display it. There are several ways we can do this, but in this case we are going to use a `Grid`. What is a grid? It’s a map of square tiles, each of them 16×16, all coming from the same bank. Each square of the map can display a different tile from the bank, so you can use them to display levels. Let’s create a grid using our bank:

```
import ugame
import stage

bank = stage.Bank.from_bmp16("ball.bmp")
background = stage.Grid(bank, 10, 8)
```

This creates a grid that is 10 tiles wide and 8 tiles high — with 16×16 tiles, that gives us 160×128, exactly the same size as the screen of PyGamer or PyBadge, so our grid will fill the whole screen. It is also possible to have smaller and larger grids, as needed.

Stage

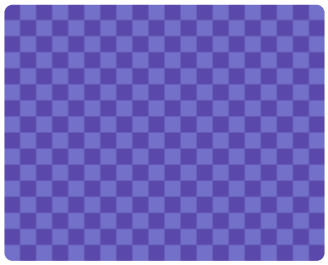
To actually display anything, we will need a `Stage` object — that represents the whole screen of your game, with all the things that need to be displayed. Those things are organized into layers — starting from the ones closest to you, and going deeper. The order of layers is important, because they will cover each other. For now, we only have one thing to be displayed, our grid, so we only need one layer:

```
    import ugame
import stage

bank = stage.Bank.from_bmp16("ball.bmp")
background = stage.Grid(bank, 10, 8)
game = stage.Stage(ugame.display, 12)
game.layers = [background]
game.render_block()

while True:
  pass
```

When you save this code, you will finally see something on the screen: a blue checkered background. That is because a new grid is by default displaying the first tile in the bank, and that is the blue tile, repeated 80 times.



The command that actually displayed it on the screen is `render_block`. You are going to call it in your program at least once, at the beginning. Without any parameters it will simply redraw the whole screen. You can also use this and other methods to only redraw parts of the screen — and we are going to do that later on, as it is much faster than redrawing the whole screen every time. But you need to draw it all at the beginning, so there we go.

You are probably wondering what that number 12 is doing there. We will need it later, when we are actually doing any animations: this is the number of frames per second (FPS) that our game is going to run at. You might be used to playing games where you have 300 fps or more, but on this kind of hardware, 12 or 24 fps is pretty standard.

Sprite

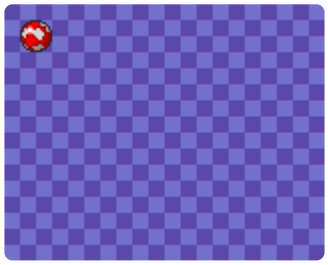
Let's display our ball now. We could create another grid, but instead let's try something new: a `Sprite`. Sprites are 16x16 images representing things in your game such as the player character, the monsters, the items, the bullets, the explosions, etc. Unlike grids, they can only display one image at a time, but you can change that image and move it around the screen easily. So let's modify our code to include a sprite:

```
import ugame
import stage

bank = stage.Bank.from_bmp16("ball.bmp")
background = stage.Grid(bank, 10, 8)
ball = stage.Sprite(bank, 1, 8, 8)
game = stage.Stage(ugame.display, 12)
game.layers = [ball, background]
game.render_block()

while True:
  pass
```

The parameters you have to pass are the bank, the image position from that bank, and the x and y coordinates of the sprite. You can also see that we added our sprite to the layers. This is important, otherwise it wouldn't be displayed. It also has to be in the list before the background, otherwise it wouldn't be visible as it would be rendered behind it. When you save this code, you should see our ball on the screen.



Animations

Now let's make that ball animated. The simplest way to do it is by spinning it — that is, making the sprite display a different image every frame. We can do that by adding a loop to our program:

```
import ugame
import stage

bank = stage.Bank.from_bmp16("ball.bmp")
background = stage.Grid(bank, 10, 8)
ball = stage.Sprite(bank, 1, 8, 8)
game = stage.Stage(ugame.display, 12)
game.layers = [ball, background]
game.render_block()

while True:
    ball.set_frame(ball.frame % 4 + 1)
    game.render_sprites([ball])
    game.tick()
```

If you are familiar with Python, you will know that `while True:` makes things be repeated over and over again infinitely. Now, `set_frame` will change the frame displayed by our sprite — we want it to go 1, 2, 3, 4, 1, 2, 3, 4, ... because we only have 4 frames of animation of the ball. The modulo operator `%` takes care of that. Next, we call `render_sprites` to re-draw our sprite on the screen (as opposed to redrawing the whole screen, as `render_block` does), and then `tick` will wait for the next frame, making sure there are exactly 12 of them per second, as we specified when we created the stage.

Movement

Now let's try to move the ball from where it spins, and make it travel across the screen. To do that, we can use the move method of the sprite, just like we changed its frame:

```
import ugame
import stage

bank = stage.Bank.from_bmp16("ball.bmp")
background = stage.Grid(bank, 10, 8)
ball = stage.Sprite(bank, 1, 8, 8)
game = stage.Stage(ugame.display, 12)
game.layers = [ball, background]
game.render_block()

dx = 2
while True:
    ball.update()
    ball.set_frame(ball.frame % 4 + 1)
    ball.move(ball.x + dx, ball.y)
    if not 0 &lt; ball.x &lt; 144:
        dx = -dx
    game.render_sprites([ball])
    game.tick()
```

We had to add one more complication. The update method of the sprite saves its old position in a temporary memory, so that when we call `render_sprites`, both the old position of the sprite and the new one can be updated. If we didn't call it, we would

have leftovers of the previous ball drawn on the screen. You can try it by commenting out that line.

Vertical movement is added the same way.

Multiple Balls

Now, suppose we wanted to have more moving objects in our game. Obviously we need more sprites, and the code to move all those sprites. Putting it all in the main loop like we did so far may be a little bit too messy. So we can subclass the `Sprite` class, and create our dedicated sprites, with behavior included:

```
import ugame
import stage


class Ball(stage.Sprite):
    def __init__(self, x, y):
        super().__init__(bank, 1, x, y)
        self.dx = 2
        self.dy = 1

    def update(self):
        super().update()
        self.set_frame(self.frame % 4 + 1)
        self.move(self.x + self.dx, self.y + self.dy)
        if not 0 &lt; self.x &lt; 144:
            self.dx = -self.dx
        if not 0 &lt; self.y &lt; 112:
            self.dy = -self.dy


bank = stage.Bank.from_bmp16("ball.bmp")
background = stage.Grid(bank, 10, 8)
ball1 = Ball(64, 0)
ball2 = Ball(0, 76)
ball3 = Ball(111, 64)
game = stage.Stage(ugame.display, 12)
sprites = [ball1, ball2, ball3]
game.layers = [ball1, ball2, ball3, background]
game.render_block()

while True:
    for sprite in sprites:
        sprite.update()
    game.render_sprites(sprites)
    game.tick()
```

Now, the `__init__` method of our new class handles creating a new sprite and setting its initial parameters, and the extended update method handles the behavior. Of course you can have many different classes if you want to have different behaviors. The `super()` call is a way to call the original method of the `Sprite` class, to make sure it is also properly initialized.

Text

Often you will have to display some messages for the player. Whether it is the current score count, the character's dialogue or the traditional "game over". You can do it by using yet another kind of layer, the `Text` layer:

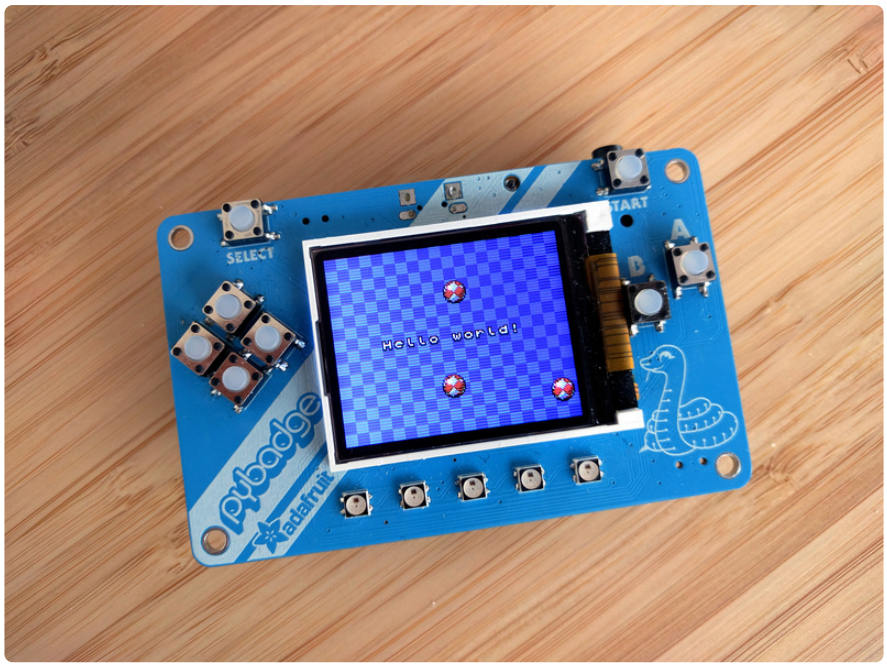
```
import ugame
import stage


class Ball(stage.Sprite):
    def __init__(self, x, y):
        super().__init__(bank, 1, x, y)
        self.dx = 2
        self.dy = 1

    def update(self):
        super().update()
        self.set_frame(self.frame % 4 + 1)
        self.move(self.x + self.dx, self.y + self.dy)
        if not 0 &lt; self.x &lt; 112:
            self.dx = -self.dx
        if not 0 &lt; self.y &lt; 112:
            self.dy = -self.dy


bank = stage.Bank.from_bmp16("ball.bmp")
background = stage.Grid(bank, 10, 8)
text = stage.Text(12, 1)
text.move(32, 60)
text.text("Hello world!")
ball1 = Ball(64, 0)
ball2 = Ball(0, 76)
ball3 = Ball(111, 64)
game = stage.Stage(ugame.display, 12)
sprites = [ball1, ball2, ball3]
game.layers = [text, ball1, ball2, ball3, background]
game.render_block()

while True:
    for sprite in sprites:
        sprite.update()
    game.render_sprites(sprites)
    game.tick()
```



Conclusion

This is as far as we are going to go with this simple demo. There is of course much more to game development than animated bouncing balls. Next we will look at some example games.

Example Games

There are currently two games using the "Stage" library that you can play on PyGamer or PyBadge. There is a simple platformer game called Jumper Wire (https://adafru.it/FaV), and a space shooter called Vacuum Invaders (https://adafru.it/FaW).

The games were originally written for the µGame board, with the screen smaller than the one on PyGamer or PyBadge, so you will see some random garbage drawn at the right hand side of the screen — don't worry about it. When you make your own games, you can use the whole screen.

Jumper Wire Game Repository

https://adafru.it/FaV

Vacuum Invaders Game Repository

https://adafru.it/FaW

Documentation

The full documentation for the `stage` and `ugame` modules is available at ReadTheDocs.

Stage library documentation

https://adafru.it/Fb3

You can also peek under the hood and see how it all is programmed by looking at the GitHub repository for the project.

Stage library repository

https://adafru.it/Fb4