µGame Documentation

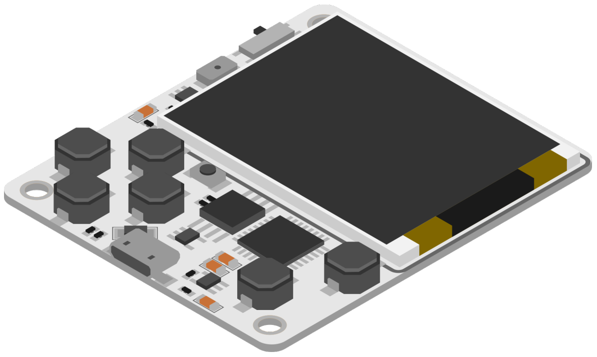
Release 10

Radomir Dopieralski

Mar 19, 2018

Contents

A small handheld game console programmable with Python.



CHAPTER 1

About the Project

The µGame project aims to make it easy and fun to make arcade-style video games in Python. It does that by developing a hardware platform, in form of a handheld game console, and a software platform, in the form of the firmware for that console.

The project started in 2017 with a series of ten prototypes using commonly available and cheap parts. The final version, µGame 10, has been released and is available both in the form of a kit, and as a repository of design files. The software is based on Adafruit's CircuitPython firmware, with a number of helper Python libraries included to make game development easier.

There is also a number of games and demos available, including a tutorial contained in this documentation. Everything is kept in the project's GitHub organization.

1.1 Licensing

All parts of this project are available under open licenses. The hardware is released under a Creative Commons SA-BY license, and all the code is released under a permissive MIT license. You can fork, develop and produce your own versions of whatever this project contains.

1.2 Contributing

Contributions of all kinds are very welcome. You don't have to be a programmer or a hardware engineer to help! There is always something to do.

The most important way you can contribute is by helping everyone else — whether by answering their questions, writing about what you are doing with the device, making available anything you have made, or improving the documentation.

The Stage library is in very early stages of development right now, and it is supposed to stay rather lean and simple, so contributing there may be hard, but you can always develop additional libraries and game frameworks that better fit some of the tasks.

Creating and sharing assets (graphics and sound) is also a great way to contribute – just make sure you make it clear how they can be used by adding an appropriate license.

1.3 Community and Support

For learning and help with CircuitPython itself, it's best to head to Adafruit's Discord server and ask on the `circuitpython` channel.

If you prefer a bit more old-school means of communication, there is also a `##ugame` IRC channel on Freenode.

Please use the GitHub's issue tracker to report any bugs you find. Also, the wiki is available, so you can share any information you think would be useful.

CHAPTER 2

Kit Assembly

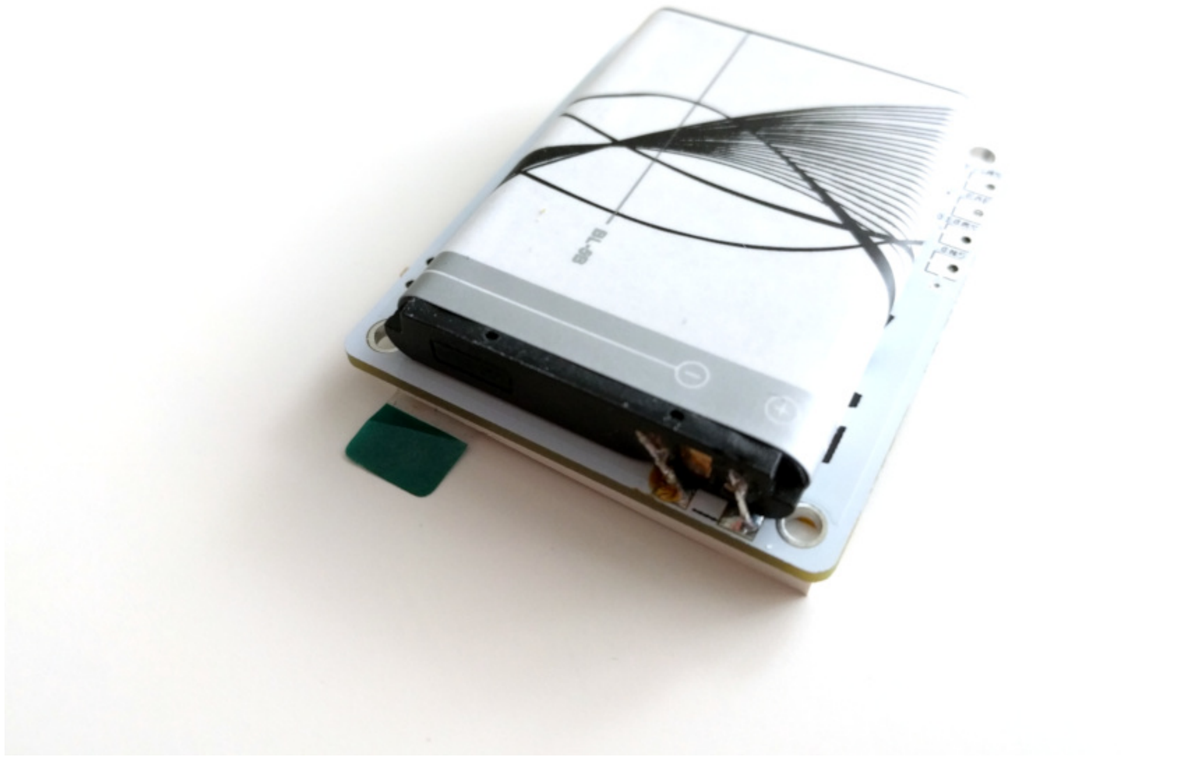
Because of logistics and regulations, µGame currently only comes as a kit without the battery included, and with the display screen not attached. It requires some assembly.

2.1 Battery

You can use any phone battery of at least 400mAh capacity. I have found that the Nokia BL-5B fits on the back of the device nicely, and that you can stick it there with a two-sided tape. Then you will have to solder the battery leads.

Make sure to get the polarity (plus and minus) right, because there is no reverse polarity protection, and you will fry the device and possibly damage the battery in the process if you connect it in reverse.

Also make sure to move the display screen away from the soldering points, so that you don't melt its plastic while working. Solder two short wires to the pads on the back of the device, and to the battery terminals. Refer to this photo.



Note that different batteries have their terminals arranged differently, and that there is usually a third terminal with just a resistor connected to it, for battery identification — you don't need to connect that third terminal to anything. It's best to use a multimeter to make sure you got the connections right.

2.2 Screen

Once you have the battery soldered, you can attach the screen in place. There are two thin brown strips on the back of the display — they are the two-sided tape. Just remove the brown paper from them, and stick them in place.

In practical testing we saw that those strips are rather weak until you let them settle for a while. You might want to add your own two-sided tape there to get a stronger binding. Be sure to stick the screen straight – once the glue settles, it's very hard to unstick it without damaging the display.

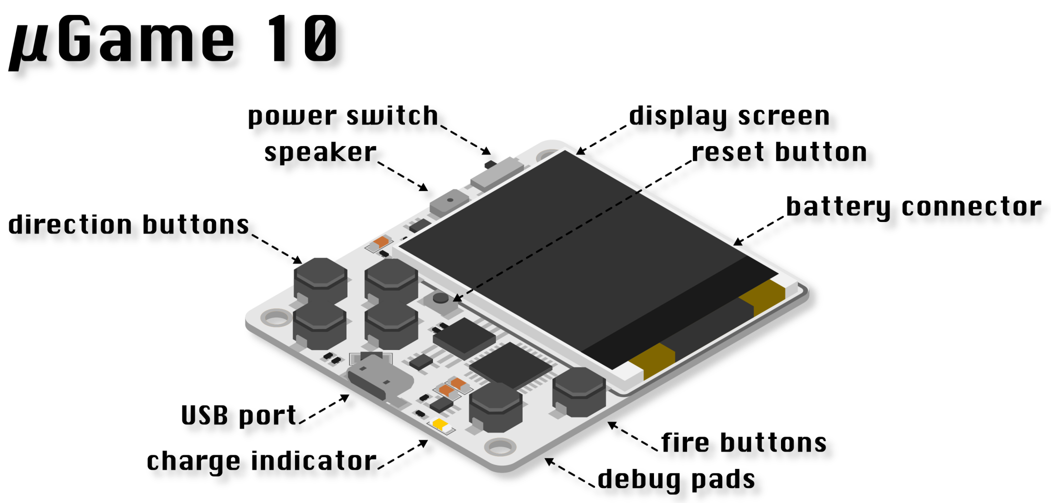
CHAPTER 3

Hardware

The most recent version of hardware for this project is µGame 10.

3.1 µGame 10

3.1.1 Feature Placement



3.1.2 Hardware Specification

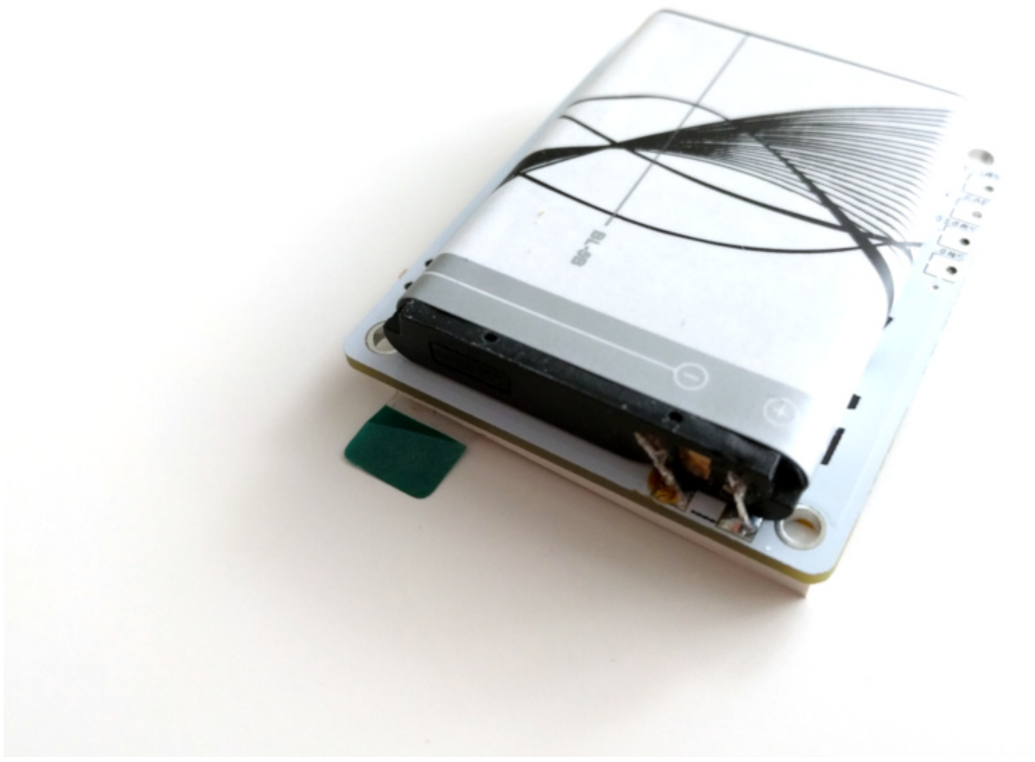
Size	42×50×5mm
Processor	Atmel SAMD21
Clock	Internal 48MHz
Memory	32kB
Storage	2MB External
Display	ST7735R 1.44"
Resolution	128×128
Color	16-bit
Audio	Mono 8-bit
Buttons	6

3.1.3 Battery Notes

For logistic reasons the µGame 10 is shipped without a battery. The user has to attach and solder the battery themselves (before attaching the display). Pretty much any lithium battery will work with the built-in battery charger, but as the charger works at 400mA, it is important that the battery can be charged with that current. Usually this means that the battery has to be at least 400mAh in size. The Nokia BL-5B battery is recommended, as it has the right size and shape, stores around 800mAh, and is contained in a safe package that fits well on the back of the device. Any other phone battery of similar size should also be suitable.

Note that to charge the battery, the power switch has to be on while the device is connected to the USB port.

An example of a battery attached to the back of the device with a two-sided tape and soldered to the battery pads:



CHAPTER 4

Usage Manual

4.1 Filesystem

In order to start programming your µGame, connect it to any USB port of your computer using a micro-USB cable. Unless you are running Windows 7, you won't need any special drivers (the drivers for Windows 7 are available at Adafruit). As soon as you connect your device, you should see a new disk driver labeled "CIRCUITPY" appear. If you browse it, you will either see some demo program already there, or just an empty disk.

You can put any files on that disk, but a few of them have a special meaning. A file called `main.py` will be executed every time the device is powered on or restarted. Any other `.py` and `.mpy` files that you save there will become available to be imported from `main.py` and used as libraries or programs.

You will also want to save some `.bmp` and `.wav` files to be used by your game, and possibly also some text files with level definitions and the like.

Finally, you will probably find a `boot_out.txt` file appearing on that disk every time the device is restarted — it contains diagnostic information about the device and the firmware on it.

Note: Please remember to always "remove" or "unmount" the filesystem after copying or editing any files before unpluggin or hard-resetting the device (the automatic soft-reset is fine). If you don't, and a writing operation gets interrupted, you might accidentally corrupt your filesystem and have problems with the files afterwards. If that happens, see further down this page for troubleshooting options.

4.1.1 Writing Files

By default, the filesystem can be written to from your computer through the USB port, but is read-only for your program. In order to be able to write to the files, or create new ones, you first have to re-mount the filesystem as read-write — this has to happen before the USB connection is made, so the code that does that has to go into the special *boot.py* file. But since only one thing can have write access to the filesystem, that means you will no longer be able to write files through USB from your computer. Detailed explanation is available from Adafruit tutorials.

If you want to be able to save the player's progress, you can also use a small area of non-volatile memory available for that purpose through the *nvm* module.

4.2 Console

You also have access to an interactive Python console (also called REPL) over the USB connection. You can use it to experiment with Python commands and explore the device, but it's also very useful for debugging, as you can see everything your program prints in there, and also the text of any exceptions raised. If your program doesn't work and you don't know why, it's best to first check the console for any messages. This Adafruit tutorial explains how to access the console.

4.2.1 Other Considerations

Generally speaking, µGame is a CircuitPython device, compatible with the Adafruit M0 boards, and behaves exactly the same as those boards.

4.3 Troubleshooting

4.3.1 Errors

If you have uploaded your program but it's not working, or works for some time then abruptly stops, you have probably hit an error. To see what the error is exactly, you need to access the serial console, as explained above, and then you will see what the error is and on which line of your program.

4.3.2 Corrupted Filesystem

It you unplug µGame from your computer while files are still being copied, it can happen that the filesystem gets corrupted. When that happens, the surest way to recover is to copy all the important files to a safe place on your computer and format the filesystem on the µGame. To make that easy, there is a special firmware that will do it for you.

First, you need to download the format.uf2 and firmware.uf2 files. Once you have them, connect your µGame to your computer, and press the reset button twice, so that it switches into the bootloader mode. A disk called `TRINKETBOOT` should appear, with some files on it. When it does, copy the `format.uf2` file on it, and wait for the device to reset. Your flash is now formatted. Now press reset twice again, and this time copy the `firmware.uf2` file onto the disk, to get back to the CircuitPython firmware. When the device resets, you should see a brand new empty `CIRCUITPY` disk.

CHAPTER 5

Bouncing Ball Tutorial

To learn how to write games for µGame, it’s best to start with a practical example. We will try to make a simple demo, showing on the screen a moving ball, bouncing off the screen edges. It’s very simple, but demonstrates some of the most important concepts that you will need to know.

5.1 Clean Slate

We don’t want anything to interfere in our tutorial, so let’s start with making sure that everybody is starting with the same state. There are probably some files on your µGame right now — perhaps a demo program it came with, perhaps the remnants of earlier experimentation. Please copy those files somewhere safe on your computer, and then delete all the files from the *CIRCUITPY* drive.

5.2 The Main File

Now, create a file called *main.py*, and open it in a text editor. This is going to be where we put all the code. Later on, with more complex projects, you will be probably using multiple Python files, but for now everything will go into *main.py*.

Please also open a serial console, as described in the usage manual, so that you can see all the messages and errors.

Now type the first few lines of the program, and save 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.

You won’t find the Python files for those two libraries on your device, because they are included in the firmware directly. The first one, *ugame*, gives you access to all the hardware of your device: the display, the buttons, the audio output, the battery voltage, etc. The second one, *stage*, is a very basic game library, giving you an easy way to draw tiled maps and sprites on the screen. There is complete documentation for that library, if you are curious..

5.3 Banks

All the graphics used in µGame is 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.

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:

Now please right-click on that image, select "save as", and save it onto your *CIRCUITPY* disk as *ball.bmp*, next to the *main.py* file. This way we will be able to load it into memory and use in our demo.

Note: The code reading the graphics files on µGame is very simple, and so the files need to be in a very specific format. It has to be a 16-color BMP file, 16 pixels wide and 256 pixels high. Make sure that your graphics program can save such files.

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.

5.4 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)
```

By default a grid is the same size as the screen, so 16×16 squares. You can also specify width and height, to create a grid of a different size, but for now this is enough for us. Now, how do you display it on the screen?

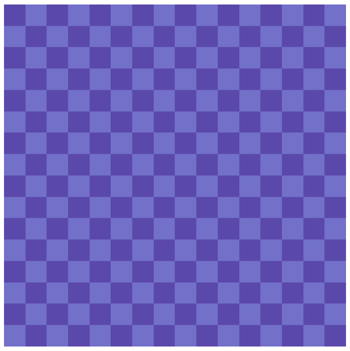
5.5 Stage

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)
game = stage.Stage(ugame.display, 12)
game.layers = [background]
game.render_block()
```

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 128 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.

Note: 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.

5.6 Sprite

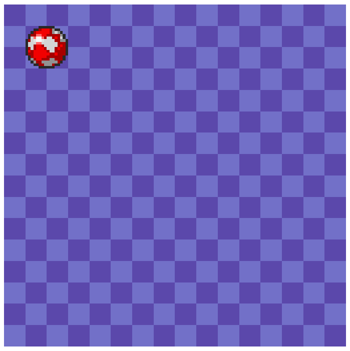
Let’s display our ball now. We could create another grid, but instead let’s try something new: a *Sprite*. Sprites are 16x16 images representing things in your game such as the player character, the monsters, the items, the bullets, the explosions, etc. Unlike grids, they can only display one image at 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)
```

```
ball = stage.Sprite(bank, 1, 8, 8)
game = stage.Stage(ugame.display, 12)
game.layers = [ball, background]
game.render_block()
```

The parameters you have to pass are the bank, the image 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. When you save this code, you should see our ball on the screen.



5.7 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)
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, 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.

5.8 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)
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 < ball.x < 112:
        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.

5.9 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 < self.x < 112:
            self.dx = -self.dx
        if not 0 < self.y < 112:
            self.dy = -self.dy
```

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

5.10 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 < self.x < 112:
            self.dx = -self.dx
        if not 0 < self.y < 112:
            self.dy = -self.dy


bank = stage.Bank.from_bmp16("ball.bmp")
background = stage.Grid(bank)
text = stage.Text(12, 1)
text.move(16, 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()
```

5.11 Conclusion

This is as far as we are going to go with this simple demo. Hopefully it will help you on your journey to becoming an experienced game developer!

CHAPTER 6

Limitations and Workarounds

The µGame is a very simple platform, and as such has certain limitations. The one you are most likely to hit is the limited memory of the device. With 32kB of RAM, and most of it taken by the interpreter, there is a limit of how large your game can be. The limit is especially painful when it comes to graphics, which tend to take up substantial amounts of memory. For example, you will only be able to keep about two or three banks in memory at a time.

The second limitation is the speed of display refresh. With only 24MHz SPI communication between the microcontroller and the display, there is a limit on how many pixels you can update in a single frame — roughly ¼ of the screen. If you try to update more, your game will start skipping frames and you will see a tearing effect. This is fine if it happens from time to time, but it makes it impossible to do things like full-screen scrolling.

Finally, you only have one sound channel, and the software can only play one WAV file at a time. There is no mixing of multiple sounds and no way to have a background music.

There are of course ways to cheat and overcome those limitations, but they are a bit tricky, and don't work in all situations.

6.1 Save Banks

Try to use as few banks of graphics as possible. You can put both grid tiles and sprites in the same bank, if you have the room. Or tiles for several different grids.

6.2 Rotate and Flip

Sprites have a "rotation" parameter that you can change to rotate and flip the image – this way you don't have to have a separate graphic for every possible orientation. It's for example common to do walking animation by just mirroring the graphic of the walking person repeatedly, to make it seem like first one and then the other foot is forward.

6.3 Palettes

Both grids and sprites let you use a different palette than the default one assigned to a given bank. And since palettes are much smaller than banks, you can have much more of them. Re-coloring the graphics lets you change the feel of a level, distinguish between the player character and the enemies, distinguish between different item types, etc. Since the magenta color is always transparent, no matter where it appears in the palette, you can even remove parts of the graphics by using a palette in which they are transparent.

6.4 Compiling

The source code of your program is also taking up space, and as it grows, at some point it will exceed the limits of the device. You can postpone that moment a little by using a pre-compiled bytecode instead of plain-text source. You can download the `mpy-cross` utility from CircuitPython releases page and use it to create a `.mpy` file that you can copy instead of the source, and that will use less memory.

6.5 Unloading

You can also save memory by explicitly removing the things you no longer need from memory using the `del` statement. For example, you only need the data and graphics for the current level, and as soon as the player goes to the next one, you can replace that with the data for the new level. However, this won't always work perfectly, because of memory fragmentation.