sino:bit MicroPython
Created by Tony DiCola

https://learn.adafruit.com/sino-bit-micropython

Last updated on 2023-08-29 03:40:31 PM EDT
# Table of Contents

**Overview** 3

**What is the sino:bit?** 4

**What is MicroPython?** 6

**Running code with the web editor** 7

**Running code with ampy** 9
- Load sino:bit MicroPython Firmware
- Install ampy
- Find Board Serial Port
- Run Code With ampy
- Run Code at Startup

**Interactive REPL** 14

**Examples** 16
- Drawing on the 12x12 LED Display
- Printing Text on the 12x12 LED Display
- Digital Sand

**Learn more** 21
Overview

The sino:bit is a small board computer for learning about programming and science like the BBC micro:bit. What makes the sino:bit special is that it's designed for a world-wide audience. The inspiration for the sino:bit came from Naomi Wu who realized boards like the micro:bit could only be used by Western, English-speaking audiences because their small grid of LEDs couldn't display large characters from Chinese and other languages. Naomi worked with Elecrow to design the sino:bit as a world learning computer, one which has a large enough 12x12 LED matrix to display almost any character. The power of the sino:bit is that it can open the doorway to learning about computing to a world-wide audience of makers!

MicroPython is a port of the Python programming language that supports small computers like the BBC micro:bit. Since the sino:bit shares so much hardware with the micro:bit it's possible to use a sino:bit MicroPython port to program the sino:bit with Python code! This guide explores how to use the sino:bit MicroPython port.

Be aware the sino:bit MicroPython port is very early in development (as of this guide writing just a few weeks old!). Features, documentation, bugs, and other changes will be added and fixed over time so expect a few issues as you explore and learn from the project today!

Before you get started it might be handy to review these guides for more background on the sino:bit and MicroPython:

- sino:bit With Arduino
- What is MicroPython?
What is the sino:bit?

The sino:bit is a small board learning computer designed by Naomi Wu and Elecrow in Shenzhen, China. Naomi took inspiration from the BBC micro:bit, a small computer designed for education in the United Kingdom. The micro:bit was hugely successful at introducing students to programming by providing a tiny computer that was ready to use out of the box without any soldering or complex software setup. However the micro:bit was targeted at the United Kingdom and an audience of English speaking students as its tiny 5x5 matrix of LEDs can only display English language characters. Naomi saw the limitations of the micro:bit and worked with Elecrow to build a variant with a larger 12x12 matrix of LEDs that can display characters from Chinese and other languages. The brilliance of the sino:bit is that it opens the door for computer education to a worldwide audience!

Since the sino:bit is based on the design of the BBC micro:bit (specifically a variant of the micro:bit called the Calliope Mini) it is almost identical in functionality to the micro:bit. Both boards have the exact same processor, sensors, and input hardware. The big difference with the sino:bit is that it has a larger grid of LEDs. At a lower level though there are a few other minor differences:

- The sino:bit LED matrix is driven by a dedicated LED matrix driver chip instead of by the board's CPU with the micro:bit. This has an advantage of freeing the processor from complex display update logic and enables a larger grid of LEDs. However one important difference between the boards is that the micro:bit supports 10 levels of brightness for each individual pixel, whereas the sino:bit only supports a global brightness of 16 levels that applies to all pixels at once.
• The sino:bit buttons are reversed relative to the micro:bit. If you hold the boards in the same orientation the A and B buttons are swapped in position.

• The sino:bit exposes access to its serial UART and I2C bus through grove style connectors, making it easy to interface with many third-party sensors. In addition the sino:bit exposes all of the processor's GPIO pins through a 13x2 pin header near the bottom of the board. This header is exactly the same type as used on a Raspberry Pi so many Pi accessories can potentially be interfaced with the board, however note the pin functionality and pinout is not the same as the Raspberry Pi!

• In addition to the interfaces mentioned above the sino:bit includes a few more large GPIO ports on the corners of the board. The micro:bit exposes pins 0, 1, 2 whereas the sino:bit exposes pins 0, 1, 2, 3, 4, and 5. Note that the holes on the sino:bit are slightly larger than on the micro:bit and won't grip a banana plug as well as on the micro:bit.

All other functions of the sino:bit, like its accelerometer and magnetometer, are exactly the same as the micro:bit!

With the micro:bit there are primarily three ways to program or run code on it:

• Using [Microsoft MakeCode](https://makecode.microbit.org/) and its graphical or Javascript-based language.

• Using the [Arduino IDE and the Arduino language](https://www.arduino.cc/).

• Using [MicroPython and the Python programming language](https://micro-python.org/).

Since the sino:bit is so similar to the micro:bit it's possible to use with almost all the above tools too! In fact there's already a handy guide on [using the sino:bit with Arduino](https://learn.adafruit.com/using-the-sino-bit-with-arduino) that you can explore now. This guide will explore how to use a new port of [MicroPython for the sino:bit](https://learn.adafruit.com/micropython-for-the-sino-bit) and program the board with Python code just like a micro:bit! Note that Microsoft's MakeCode doesn't yet unfortunately support the full capabilities of the sino:bit (like drawing to its display).

More information about the sino:bit can be found here:

• [sino:bit GitHub home (schematics and more)](https://github.com/sino-bit) 

• [sino:bit Product Page (where to buy one)](https://sino-bit.com) 

• [sino:bit with Arduino Guide](https://learn.adafruit.com/sino-bit-with-arduino) 


• [sino:bit Open Source coding projects for children by Keith Parkins](https://github.com/sino-bit)
What is MicroPython?

**MicroPython** is a version of the [Python programming language](https://www.python.org) that can run on small, resource-constrained computers. MicroPython was created by Damien George to power robotics and other hardware projects, but a community of users sprung up around MicroPython as they saw the potential for it to be used in education, physical computing, and more. Damien and others were even able to [port MicroPython to run on the BBC micro:bit](https://microbit.org/) which opened the doorway to computer education with simple hardware and software. Like the Python programming language, MicroPython is interpreted by the processor and can easily be modified and run interactively without any complex compile, upload, or other software steps. Learning about hardware and programming with MicroPython is as easy as connecting to a board and typing in Python code!

Just as the sino:bit was built on top of open source designs for the micro:bit, the [sino:bit MicroPython project](https://github.com/adafruit/sino-bit) is built on top of the [BBC micro:bit MicroPython project](https://github.com/adafruit/microbit-micropython). The sino:bit port of MicroPython aims to expose all the functions of the sino:bit, like drawing and writing on its large 12x12 LED matrix, to users with a simple Python-based programming language. In addition the sino:bit MicroPython port provides compatibility with existing micro:bit MicroPython code and adapts it to run on the larger sino:bit display automatically. The combination of the easy to use sino:bit hardware and MicroPython programming language opens the doors to computer education for a worldwide audience!

Adafruit has quite a few resources available to learn more about MicroPython and variants of it like CircuitPython (a port of MicroPython for Adafruit hardware):

- Adafruit's [What is MicroPython? guide](https://learn.adafruit.com/micro-python)
- Adafruit's [MicroPython guides](https://learn.adafruit.com/micropython)
- [BBC micro:bit MicroPython documentation](https://github.com/adafruit/microbit-micropython)
- [BBC micro:bit MicroPython guide](https://learn.adafruit.com/micro-bit-micropython)
- [MicroPython project](https://micropython.org)

In addition if you’re new to the Python programming language take a moment to review the guides and resources mentioned in the [What is MicroPython?](https://learn.adafruit.com/micro-python) guide:

- [Learning Python resources from the Hitchhiker's Guide to Python](https://www.hitchhikersguidepython.com)
- [Learn Python The Hard Way](https://learnpythonthehardway.org)
Running code with the web editor

There are two ways to run MicroPython code on the sino:bit. The first method is using a web-based editor to enter code and download a firmware file with both MicroPython and the desired code to run. The second method is by loading MicroPython firmware onto the sino:bit and then separately loading code onto it using a tool called ampy. This page explores using the web editor to write and run MicroPython code for the sino:bit. The next page explores using ampy to load and run code on the sino:bit.

The web editor for the sino:bit MicroPython port is available at the following address: https://tdicola.github.io/sinobit-micropython/editor/editor.html

If you navigate to the editor you'll see it's very similar to the BBC micro:bit web editor. In fact the sino:bit MicroPython web editor is based on the code for the BBC micro:bit web editor!

At the top you'll find a toolbar of buttons with basic commands for downloading the firmware & code, saving, loading, and other code operations. Below the toolbar you'll see a text editor where you can enter your MicroPython code that will run immediately on power-up. That's all there is to using the web editor—just write code, download the firmware, and drag to the sino:bit drive to program and run the code. No installation of any other tool or plugin is necessary!

Notice the example code the editor loads with to start:

```python
# Add your Python code here. E.g.
from microbit import *

while True:
    display.scroll('Hello, World!')
    display.show(Image.HEART)
    sleep(2000)
```

©Adafruit Industries
This code is compatible with the micro:bit MicroPython port and will also run on the sino:bit. You can see it calls a scroll function to scroll the message "Hello, World!" across the display. Let's try running the code as-is on the sino:bit!

First plug in the sino:bit to your computer using its USB port. Be sure you're using a USB data cable and NOT a USB power-only cable. You should see a USB drive appear called MICROBIT (remember the sino:bit inherits much of the software and design from the micro:bit, including how it exposes its USB drive). Just like with the micro:bit you can drag a special .hex file to the MICROBIT drive and it will program the board with the firmware inside the file.

Click the download button in the upper left corner of the web editor. After a moment a sino_bit.hex file should be downloaded to your computer (note on the Safari web browser it will download as an 'Unnamed' file and you must change the name to end in .hex--the editor will warn you when this happens). In your file explorer find the sino_bit.hex file and drag it onto the MICROBIT drive. You should see the blue light on the sino:bit flash as it programs the firmware!

After a few moments the blue light will stop flashing and your code will be running on the sino:bit! You should see the message scroll across the display.

Try changing the code, like modifying the message to say your name. Download the hex file again and drag to the sino:bit to see it run.

That's all there is to the basic web editor usage of sino:bit MicroPython! The web editor makes it very easy to get started programming the sino:bit without any tools or complex setup. You can run almost any Python program from the web editor, including any of the tutorials and examples of micro:bit MicroPython code () (notice the sino:bit display will increase the size of the displayed pixels to fill the entire matrix).
The one limitation to be aware of with the web editor is that it only supports downloading 8 kilobytes of program code. To run slightly larger or more complex programs you can use a tool called ampy to copy the code manually to the board--continue to the next page to learn more!

Running code with ampy

Another way to run code on your sino:bit (or any other MicroPython board) is by copying it to the board's file system. This is slightly different than using the web editor where you download an entire hex file to reprogram your board with both the MicroPython firmware and your program code at the same time. Instead with this process you program your board once with a blank MicroPython firmware, then use tools to copy Python code to the board separately. By copying files to the board you actually get a little bit more room for your program as the board's file system takes up as much free space as is available (typically about 14 kilobytes) vs. the fixed 8 kilobytes you get with web editor programs.

To copy programs to a sino:bit or MicroPython board you can use a tool called ampy. There's actually a handy guide already on installing and using ampy ( ), but we'll run through the basics with the sino:bit below.

Load sino:bit MicroPython Firmware

The first step is to load a general or 'blank' version of the sino:bit MicroPython firmware on your board. If you try to copy files to your board using a firmware loaded from the web editor you won't actually see your files run as the web editor code takes over and is always run first. Instead go to the releases page for sino:bit MicroPython ( ) and download the .hex file for the latest release.
Then just like programing from the web editor drag that .hex file (like sinobit-micropython-0.0.4.hex) to your board's MICROBIT drive. You should see the blue light flash for a bit as the firmware is programmed.

Install ampy

After your board is programmed you're ready to copy code to it using ampy. However on some platforms like Windows you might need to install a driver to access the board's serial port. Check out this page on installing the mBED driver for the micro:bit --this is the same driver you need for the sino:bit too. On Mac OSX and Linux you don't need to install any driver to access the board serial port.

Next make sure you've followed the guide on using ampy to install Python and the ampy tool () with the pip package manager. Once you have Python installed and in your path it's just a matter of running:

```
pip install adafruit-ampy
```

Or if you're using Python 3.x:

```
pip3 install adafruit-ampy
```

Then the ampy command should be available from your command terminal, try running it with --help to confirm:

```
ampy --help
```
Find Board Serial Port

Before copying code to your board you’ll need to figure out which serial port the board appears as on your computer. Again be sure you installed the driver on Windows () first!

On Windows, open the device manager and scroll down to the list of serial ports (typically named COM1, COM2, etc.). If you unplug the sino:bit and then plug it back in you should see a new serial port appear—that's the port name you want to use. If you haven't used a lot of other devices this is typically assigned COM4.

On Mac OSX, run in a terminal the `ls -l /dev/tty.*` command to see a list of all the serial devices. Again try the command with and without the board attached to see which device appears after the board is plugged in (you might need to wait a few moments for the board’s blue light to flash indicating it has finished talking to the computer to create the serial port). Typically on Mac OSX you’ll see a device like /dev/tty.usbmodem1411 or another number at the end.

On Linux, run in a terminal the `ls -l /dev/tty*` command to see a list of all serial devices. Look for a new device to appear after running the command with the board disconnected, then connected. Although it varies by distribution you typically will see the board under a /dev/ttyACM0 path.

If you’re still having trouble finding the board serial port another way is using a tool in the PySerial package () that is a cross-platform serial port list (i.e. it works on
Windows, Mac OSX, and Linux). First install PySerial with the pip tool just like you did with ampy:

```
pip install pyserial
```

(remember if you're using Python 3.x use the pip3 command instead)

Then run the `tools.serial.list_ports()` command provided by PySerial:

```
python -m serial.tools.list_ports
```

(again be sure to instead run with python3 if you've installed into Python 3.x with pip3)

You can see all the serial devices are listed in a simple to read list. On my Mac device the sino:bit appears as `/dev/cu.usbmodem1412`.

And if you're really unsure about the right device you can run list_ports with a `-v` option that prints more information about each device. The sino:bit will appear as a device with description MBED CMSIS_DAP:

```
python -m serial.tools.list_ports -v
```

### Run Code With ampy

It's easy to run a python script on the sino:bit with ampy. First create a simple Python script to run, for example this is a hello world that will count to 10. Save this as a file called `helloworld.py`:

```
print('Hello world! I can count to 10:
for i in range(1, 11):
    print(i)
```

Now in a terminal navigate to the folder with the `helloworld.py` file and run this ampy command to send it to the sino:bit to run. Be sure you've looked up your board serial port name like shown above!
ampy --port <serial port name> run helloworld.py

(replace < serial port name > with your board's serial port like COM4, /dev/cu.usbmodem1411, etc.)

You should see the sino:bit blue light flash for a moment as the script is transferred, then output from it printed in the terminal. What's happening is the Python code from helloworld.py is being sent from your computer to the sino:bit and run on it. Anything the code prints out from the sino:bit is returned and printed by the ampy tool. Think of it like telling the MicroPython interpreter on the sino:bit that you want it to run a Python script and grab the output!

Run Code at Startup

Running a single script is simple with ampy, but what about saving a script to the board and running it automatically when it powers up? This is easy to do using a special file called main.py that's saved on the board's file system. The way MicroPython works is that it looks for a main.py file at startup and if it finds one it will start running it immediately. With ampy you can copy a file to the board's file system and name it main.py so it runs when the sino:bit powers up or resets. This is typically how you develop and run code for a MicroPython board--put all your logic in a main.py and save it on your board to run.

To demonstrate try copying this code to a scroll.py file that will scroll a message on the display like on a micro:bit:

```python
from microbit import *

while True:
    display.scroll('Hello, World!')
    display.show(Image.HEART)
    sleep(2000)
```

Then use this ampy command to copy scroll.py to your board and save it as a file called main.py on the board's file system:
You won't see any output from the ampy command, but you should see the sino:bit's blue light flash for a moment and then the display spring to life scrolling the message! Try changing the message in scroll.py, saving it, and running the ampy command again to push the updated file to the board as main.py again.

That's all there is to using ampy to copy files and run them on the sino:bit! Be sure to read the full ampy guide for more information on using it with MicroPython boards. Remember the sino:bit MicroPython port is still early in development so not all the features of ampy might be supported yet (like listing files or retrieving them from the board).

---

Interactive REPL

One of the most interesting features of MicroPython boards is the ability to connect to a REPL, or read-evaluate-print loop, and interactively run Python code. Think of this like a Python 'command line' where you type in Python code and instantly see the results of running it. This is perfect for exploring hardware and learning about Python code. With the sino:bit MicroPython port you can connect to its REPL just like any other MicroPython board, including the micro:bit.

To demonstrate using the REPL you can install a simple Python serial terminal program. This program is cross-platform and works on Windows, Mac OSX, and Linux so it's good for demonstrating here. However check out this MicroPython guide for more information about other REPL tools.

First follow the steps from the ampy page to locate your board's serial port, and to install the PySerial tool (and again note on Windows you need to install a driver to access the serial port). Inside PySerial it includes both the list_port tool and a miniterm tool that's a handy cross-platform serial terminal.

Once PySerial is installed and you've located the board's serial port, run the following command to open the REPL through the serial port:

```
python -m serial.tools.miniterm --raw &lt;serial port name&gt; 115200
```
(remember to run with python3 if you pip3 installed PySerial into your Python 3.x installation)

Be sure to specify your board's serial port name in < serial port name >, like COM4, /dev/cu.usbmodem1411, etc.

You should see miniterm start and might not see anything else after that (particularly if your sino:bit is already running code. Press Ctrl-C to tell the sino:bit to stop running code and get ready for your commands:

```
try: import tony5 python # serial tools_miniterm --raw /dev/cu.usbmodem1411 115200
... Miniterm on /dev/cu.usbmodem1411 115200 B.N.1 ... "` Ctrl+b Q: Quit [C] Ctrl+f Menu [M] Ctrl+h Help: Ctrl+t followed by Ctrl+H ... Traceback (most recent call last):
  File "<main>" line 4, in <module>
KeyboardInterrupt
MicroPython v1.7.9-gbe028eb on 2016-04-18: sino:bit with nRF51822
>>> "help()" for more information.
```  

Notice a KeyboardInterrupt error occurs (this is how the running program is stopped) and then a Python prompt >>> appears!

Try typing in Python code and pressing enter to see it immediately run, like to print hello world:

```
print('Hello world!')
```

```
>>> print('Hello world')
Hello world
```  

You can run any Python code from the REPL! Try importing the microbit module and using its display scroll function to scroll a message:

```
import microbit
microbit.display.scroll('Hello world!')
```

```
>>> import microbit
>>> microbit.display.scroll('Hello world!')
```  

You should see the message scroll across the sino:bit's LED display. Notice that as the message scrolls you can't type anything else in the REPL. This is because the sino:bit can only do one thing at a time, and running code like scrolling a message takes over anything else, like running code from the REPL. Once the message finishes scrolling you'll see the REPL prompt >>> return and you can enter more code. Remember using the REPL is just like running a Python script line by line--if you run a loop or other complex processing you might have to wait for it to finish!

The REPL is a great way to test out and explore hardware or Python code. You can do anything from a Python script in the REPL--don't be afraid to experiment and explore!
To quit the miniterm program press Ctrl-] like it mentions at the start. That's all there is to the basics of using the REPL with the sino:bit!

Examples

Development of the sino:bit MicroPython port is still very early (as of this writing just a few weeks old!). However here are a few example programs that demonstrate some of the current capabilities of the board. Remember the board is fully compatible with micro:bit MicroPython code and any of its examples will run on the sino:bit. Check out the sino:bit MicroPython GitHub examples folder for more examples as they are created too!

Drawing on the 12x12 LED Display

To access the full 12x12 LED matrix you'll want to use the sinobit.display module. This is very similar to the microbit.display module but gives you pixel access to the 12x12 LED grid instead of a 5x5 grid like on the micro:bit. Here's a demo of an 'arc reactor' animation that animates some squares moving into the board. Load this as a main.py on your board (or load it from the web editor):

```python
# sinobit 'Arc Reactor' Animation
# This will animate a 2 pixel wide rectangle that shrinks down across the # entire 12x12 display. The brightness level fades away over time at each step # for a nicer effect too.
# Author: Tony DiCola

# Import the microbit module to access its sleep delay function.
# You can use all the other parts of the microbit module too except for its
# display module (for now!). Instead see the sinobit module below for access
# to the display.
import microbit

# Import the display submodule from the sinobit module to write to the display.
from sinobit import display

# Define some functions in Python code to draw straight lines and boxes.
def hline(x, y, width, color):
    # Horizontal line at x, y position and width pixels to the right.
    # Color is a boolean true/false for on or off pixels.
    for i in range(x, x+width):
        # The set_pixel function will set a pixel at the provided x, y position
        # to on or off with the provided true/false color value.
        # Remember you won't see the pixel LED turn on/off until write is
        # called!
        display.set_pixel(i, y, color)

def vline(x, y, height, color):
    # Vertical line at x, y position and height pixels down.
    # Color is a boolean true/false for on or off pixels.
    for i in range(y, y+height):
        # The set_pixel function will set a pixel at the provided x, y position
        # to on or off with the provided true/false color value.
        # Remember you won't see the pixel LED turn on/off until write is
        # called!
        display.set_pixel(x, i, color)
```
def box(x, y, width, height, color):
    # Draw a 1 pixel wide box with upper left corner at x, y and of the
    # specified width and height of pixels. Color is a boolean true/false
    # for turning the pixels on or off.
    hline(x, y, width, color)
    hline(x, y+height-1, width, color)
    vline(x, y, height, color)
    vline(x+width-1, y, height, color)

    # Clear the display.  Note that all the display commands just update the
    # internal memory and have to be followed by a write call to update the LEDs
    # with the new memory value.  This way you can make a lot of pixel changes at
    # once and then write them all in one call (as opposed to seeing all the pixel
    # updates as they occur).
    display.clear()
    display.write()

# Main loop will run the code inside it forever:
while True:
    # We'll count from 0 to 5 to move the rectangle from a starting x, y
    # of 0, 0 down to 5, 5.
    for i in range(6):
        # First clear the display.  This is another way to do it by calling
        # fill and passing a color boolean to turn off the pixels (but you
        # could instead pass true to turn them all on!).
        display.fill(False)
        # Calculate the size of the square so that it shrinks down with
        # each step.
        size = 12-2*i
        # Draw the square.
        box(i, i, size, size, True)
        # If we're past the first iteration draw a second square behind this
        # one to double the size of the box.
        if i &gt; 0:
            box(i-1, i-1, size+2, size+2, True)
        # Finally make sure to call write on the display to push out all the
        # pixels that were set with the box drawing commands above.  This will
        # turn on and off the appropriate LEDs to draw this frame of the
        # animation.
        display.write()
        # Once the pixels are lit you can change their brightness (you don't
        # have to call write after changing brightness, it will instantly
        # update!).  There are 16 brightness levels from 0 (lowest) to 15
        # (maximum, the default).  We'll loop down from 15 to 0 to dim the
        # square that was just drawn and make it appear to fade away over time.
        for b in range(15, -1, -1):
            display.brightness(b)
            # Delay at each brightness for a longer and longer period of time.
            # This means the lower brightness values will linger and give a
            # nicer exponential decay to brightness.  The sleep function will
            # pause for a number of milliseconds, like the delay function in
            # Arduino.
            microbit.sleep(17-b)

Printing Text on the 12x12 LED Display

Another example of using the entire LED display is with the text and text_width
functions of the sinobit.display module. These allow you to print a string of text
anywhere on the display. For example this code will scroll a message across the
display--even displaying accented Latin unicode characters! Again save this as
main.py on your board (note you must currently use ampy to load this script as the web editor has a bug with saving files that contain advanced Unicode characters):

```python
# Simple message scrolling demo. Will scroll the specified message across
# the display from right edge to left edge. Try changing the message to
# to text with Unicode Latin & Latin-1 supplement characters!
# Author: Tony DiCola
import microbit
import sinobit
MESSAGE = '¿Hablas español? Parlez-vous Français?'
x = 11
width = sinobit.display.text_width(MESSAGE)
while True:
    sinobit.display.clear()
    sinobit.display.text(x, 0, MESSAGE)
    sinobit.display.write()
    microbit.sleep(100)
    x -= 1
    if x < -width:
        x = 11
```

Digital Sand

Another fun demo is a port of the 'digital sand' / 'pixeldust' animation () to the sino:bit. This is an advanced demo that shows using the accelerometer (again using the microbit module just like on a BBC micro:bit) and the full 12x12 LED matrix from the sinobit.display module. Save this as main.py on your board and load it with ampy (again the web editor currently doesn't support loading this large of an example).

```
# Digital sand demo uses the accelerometer to move sand particles in a
# realistic way. Tilt the board to see the sand grains tumble around and light
# up LEDs. Based on the code created by Phil Burgess and Dave Astels, see:
#   https://learn.adafruit.com/animated-led-sand
# Ported to sino:bit by Tony DiCola
#
# The MIT License (MIT)
#
# Copyright (c) 2018 Tony DiCola
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
```
import math
import random
import microbit
import sinobit

# Configuration:
GRAINS   = 20    # Number of grains of sand
WIDTH    = 12    # Display width in pixels
HEIGHT   = 12    # Display height in pixels

# Class to represent the position of each grain.
class Grain:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.vx = 0
        self.vy = 0

    # Helper to find a grain at x, y within the occupied_bits list.
def index_of_xy(x, y):
        return (y >> 8)  * WIDTH + (x >> 8)

# Global state
max_x = WIDTH * 256 - 1    # Grain coordinates are 256 times the pixel
max_y = HEIGHT * 256 - 1   # coordinates to allow finer sub-pixel movements.
grains = [Grain() for _ in range(GRAINS)]
occupied_bits = [False for _ in range(WIDTH * HEIGHT)]
oldidx = 0
newidx = 0
delta = 0
newx = 0
newy = 0

# Randomly place grains to start. Go through each grain and pick random
# positions until one is found. Start with no initial velocity too.
for g in grains:
    placed = False
    while not placed:
        g.x = random.randint(0, max_x)
        g.y = random.randint(0, max_y)
        placed = not occupied_bits[index_of_xy(g.x, g.y)]
        occupied_bits[index_of_xy(g.x, g.y)] = True

# Main loop.
while True:
    # Draw each grain.
    sinobit.display.clear()
    for g in grains:
        x = g.x >> 8  # Convert from grain coordinates to pixel coordinates by
        y = g.y >> 8  # dividing by 256.
        sinobit.display.set_pixel(x, y, True)
        sinobit.display.write()

    # Read accelerometer...
    f_x, f_y, f_z = microbit.accelerometer.get_values()
    # sinobit accelerometer returns values in signed -1024 to 1024 values
    # that are millig's. We'll divide by 8 to get a value in the -127 to 127
    # range for the sand coordinates. We invert the y axis to match the
    # current display orientation too.
    f_y *= -1
    ax = f_x &gt;&gt; 3
    ay = f_y &gt;&gt; 3
    az = abs(f_z) &gt;&gt; 6
    # Random motion factor grabs a few top
    # bits from Z axis.
az = 1 if (az &gt;= 3) else (4 - az) # Clip &amp; invert
ax -= az                          # Subtract motion factor from X, Y
ay -= az
az2 = (az &amp;&amp; 1) + 1         # Range of random motion to add back in

# ...and apply 2D accel vector to grain velocities...

for g in grains:
    g.vx += ax + random.randint(0, az2) # A little randomness makes tall stacks topple better!
    g.vy += ay + random.randint(0, az2)

    # Terminal velocity (in any direction) is 256 units -- equal to 1 pixel -- which keeps moving grains from passing through each other and other such mayhem. Though it takes some extra math, velocity is clipped as a 2D vector (not separately-limited X &amp; Y) so that diagonal movement isn't faster
    v2 = 0                      # Velocity squared
    v = 0.0                     # Absolute velocity
    for g in grains:
        v2 = g.vx * g.vx + g.vy * g.vy
        if v2 &gt; 65536:                    # If v^2 &gt; 65536, then v &gt;= 256
            v = math.floor(math.sqrt(v2)) # Velocity vector magnitude
            g.vx = (g.vx // v) &amp;&amp; 8       # Maintain heading
            g.vy = (g.vy // v) &amp;&amp; 8       # Limit magnitude

            # ...then update position of each grain, one at a time, checking for collisions and having them react. This really seems like it shouldn't work, as only one grain is considered at a time while the rest are regarded as stationary. Yet this naive algorithm, taking many not-technically-quite-correct steps, and repeated quickly enough, visually integrates into something that somewhat resembles physics. (I'd initially tried implementing this as a bunch of concurrent and "realistic" elastic collisions among circular grains, but the calculations and volume of code quickly got out of hand for both the tiny 8-bit AVR microcontroller and my tiny dinosaur brain.)

        for g in grains:
            newx = g.x + g.vx       # New position in grain space
            newy = g.y + g.vy
            if newx &amp;&amp; max_x:
                newx = max_x        # If grain would go out of bounds
                g.vx //= -2         # keep it inside, and
g.vy //=-2          # give a slight bounce off the wall
            elif newx &amp;&amp; 0:
                newx = 0
                g.vx //= -2
            if newy &amp;&amp; max_y:
                newy = max_y        # If grain would go out of bounds
                g.vy //= -2
            elif newy &amp;&amp; 0:
                newy = 0
                g.vy //= -2

            oldidx = index_of_xy(g.x, g.y)            # prior pixel
            newidx = index_of_xy(newx, newy)          # new pixel
            if oldidx != newidx and occupied_bits[newidx]: # If grain is moving to a new pixel...
                delta = abs(newidx - oldidx) # What direction when blocked?
                if delta == 1: # 1 pixel left or right
                    g.vx //= -2 # cancel x motion
                    g.vy //= -2 # and bounce X velocity (Y is ok)
                    if delta == WIDTH:
                        # 1 pixel up or down
                        g.vy //= -2
                        g.vx //= -2 # cancel Y motion
                    newidx = oldidx # no pixel change
                elif delta == WIDTH:
                    # Diagonal intersection is more tricky...
                    # Try skidding along just one axis of motion if possible (start w/

©Adafruit Industries
# faster axis). Because we've already established that diagonal
# (both-axis) motion is occurring, moving on either axis alone WILL
# change the pixel index, no need to check that again.
if abs(g.vx) &gt; abs(g.vy): # x axis is faster
    newidx = index_of_xy(newx, g.y)
    if not occupied_bits[newidx]: # that pixel is free, take it!
        But...
        newy = g.y                   # cancel Y motion
        g.vy //= -2                  # and bounce Y velocity
        else:
            # X pixel is taken, so try Y...
            newidx = index_of_xy(g.x, newy)
            if not occupied_bits[newidx]: # Pixel is free, take it, but
                first...
                newx = g.x                   # Cancel X motion
                g.vx //= -2                  # Bounce X velocity
                else:
                    # both spots are occupied
                    newx = g.x                   # Cancel X &amp; Y motion
                    newy = g.y
                    g.vx //= -2                 # Bounce X &amp; Y velocity
                    g.vy //= -2
                    newidx = oldidx # Not moving
                else: # y axis is faster. start there
                    newidx = index_of_xy(g.x, newy)
                    if not occupied_bits[newidx]: # Pixel's free! Take it! But...
                        newx = g.x                   # Cancel X motion
                        g.vx //= -2                  # Bounce X velocity
                        else:
                            # Y pixel is taken, so try X...
                            newidx = index_of_xy(newx, g.y)
                            if not occupied_bits[newidx]: # Pixel is free, take it, but
                                first...
                                newy = g.y                   # cancel Y motion
                                g.vy //= -2                  # and bounce Y velocity
                                else:
                                    # both spots are occupied
                                    newx = g.x                   # Cancel X &amp; Y motion
                                    newy = g.y
                                    g.vx //= -2                 # Bounce X &amp; Y velocity
                                    g.vy //= -2
                                    newidx = oldidx # Not moving
                                    occupied_bits[oldidx] = False
                                    occupied_bits[newidx] = True
                                    g.x = newx
                                    g.y = newy

Learn more

That's all there is to using the sino:bit MicroPython port! The port is currently very early in development and will likely change and improve over time. To keep an eye on updates, new releases, and more information be sure to check the sino:bit MicroPython homepage. This page will be updated with news and other information as the port evolves. In addition this is a landing page for all the info you might need to use sino:bit MicroPython, like links to the web code editor, firmware releases, examples and more.

Happy hacking with the sino:bit MicroPython port!