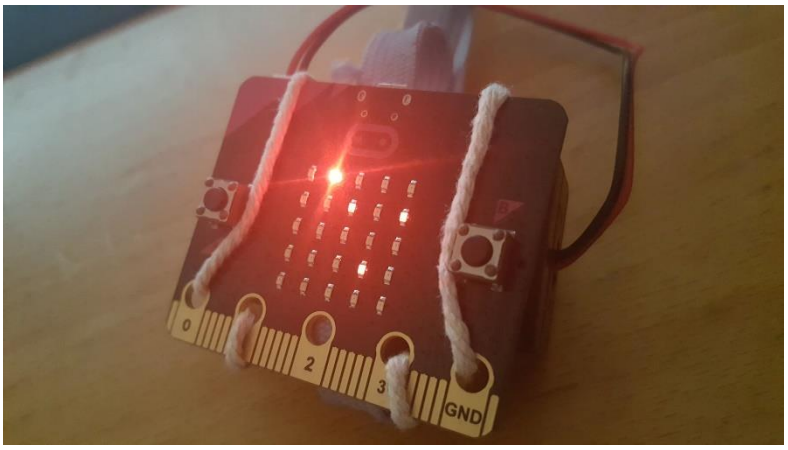
Computing in Modern Culture – Project Application Report.



Introduction

The aim of this report is to give an overview on the design and development of a wearable video game technology with the use of a Micro:bit, a microcontroller developed by the BBC for use in the classroom as an educational learning tool.

Most Micro:bit applications are developed using the "make code" web package which involves the manipulation of coding blocks akin to those found in Scratch. This application, however, was designed, developed, implemented and tested in the Python programming language. Python is a beginner friendly programming language that is easy to pick up and use and is suited for use with the Micro:bit. As Python is beginner friendly it could be suited to teenagers or younger students looking to get their first experience with programming in a classroom setting as well.

This report will begin with a brief overview of the functionality offered by the Micro:bit, giving an look into what it is capable of and how it can be used.

Following this, the Python application design and implementation is discussed in detail, including discussion regarding various limitations that were involved and discovered.

From here, an analysis of how the application can be used in a teaching setting is given, discussing the use of the compass and accelerometer and programming in general.

Lastly, a conclusion on the project is given which will include the learning outcomes of the project.

Micro:bit

The Micro:bit can be most easily seen as a "mini" Arduino board as it contains much of the design philosophies and functionalities present in that product, albeit at a much more scaled down delivery. The unit cost is around £20 or €25.

The Micro:bit brings with it:

- 25 individually programmed LEDs.
- 5 physical connection pins that can be used to send analogue and digital signals into the device. These form part of a larger 23-pin edge connector.
- A built-in compass and accelerometer.
- 2 input buttons. (Which can be pressed simultaneously to give a third option.)
- Wireless communication via Radio and Bluetooth.
- Light and temperature sensors.
- A reset button.
- A battery socket.
- A micro USB interface for flashing and firmware updates.

With these features a huge variety of applications and interactions can be designed, from simple timer and text displays and to fully functioning video games and step counters.

Other perhaps more advanced applications include the use of the radio or Bluetooth functionality to implement two-way communications and interactions over a distance, the use of the accelerometer and a compass to implement motion sensing and directional perception or even novel implementations of blockchain technology.

As it is designed for use in IT education it is very easy to use and the basics are extremely well delivered through many supporting tutorials, guidance and the Make Code interface.

This web service provides a Turing complete (can be used to simulate a Turing machine) programming interface that abstracts many of the fundamentals found in programming. It can work with repetition structures, selection structures, data types and variables, data structures, functions, input and output, logical conditionals and comparisons and so on.

As it abstracts and provides ease of implementation it is well suited to its job of being a learning tool. Within seconds the program can be downloaded and flashed onto the Micro:bit and be physically seen in operation.

Application Design.

Game overview.

The application provided as part of this project is a video game where the objective is to “pick-up” items on the “ground” by moving the player around the Micro:bit display area and guiding her to the items. An additional goal is provided in the form of a time incentive where at the end of each playthrough an elapsed time is shown, and the player can strive to beat this time in the future.

The player is represented as a bright LED that begins in the centre of the display and the items are represented as less bright LEDs dotted throughout the display area. The

player moves around the display space, which is done through the use of the accelerometer, to detect movement, and the compass, to detect direction.

The accelerometer reading works by measuring movement along three axes:

X – Tiling the Micro:bit from left to right.

Y – Tilting forwards and backwards.

Z - Moving the Micro:bit up and down in the air.

With these three axes, the movement detection works by finding the acceleration of the device (the combined force in all directions). This is done by adding the square of each axis together and then finding the square root of the summed value. For usability, it was important to have this implementation and not use the built-in "shake" operation that is popular in detecting the type of movement that a step-counter might use as that operation requires a decent amount of force to be recognized. This shaking about could change the direction of the device without the user realising and could make them think they are doing something wrong or the application doesn't work as intended.

Additionally, as part of the built in compass a novel calibration game is shown to the player in which they must try get a blinking light to visit each LED which is done by rotating, shifting and shaking the Micro:bit around. This is required at least once in the lifetime of the application to calibrate the compass to avoid garbage values being returned. The players are given the option to repeat this at any time by pressing the left button(A). Some repeat calibration is required if the Micro:bit is exposed to strong magnetic forces such as those found in stereos, earphones, televisions and hard drives etc or if the Micro:bit is near any large metallic objects during any attempted calibration.

The calibration is stored in non-volatile memory and thus is saved on power-down or reset but must be re-calibrated after every time it is externally flashed, as all memory is wiped as a result of this operation. First time calibration cannot be skipped and is

presented to the user automatically upon start-up if the application uses the compass. A short instructional message is shown before the calibration and upon success, a smiley face is drawn on the display. The embedded application is then executed.

System Design Approach:

The Python application is designed around the object orientated paradigm and thus contains classes who each have their own responsibility and fit within a hierarchy. These classes contain functions that define the possible behaviour of each class and data variables that store information related to the class. A structured programming approach could have been taken, but that type of implementation is often harder to understand and follow and wouldn't be suited to this application, as the code is designed to be easier to read and easier to alter and maintain.

A small amount of inheritance is used to simplify the objects at play, namely the pick-up and player object as both of these have a position and brightness and so can share the definition of those aspects.

There is a total of 6 library imports that the application uses to function:

- MicroPython API – The main application programming interface, an efficient implementation of Python 3 that comes with it a minimal subset of the Python standard library. This API is optimised to run in constrained environments often found in microprocessors and microcontrollers. This is extremely important to have, as discussed in the "Limitations" section of this document.
- Math – To perform the square root operation on the accelerometer axes.
- Random – To pick a random position for the pick-ups to spawn at,
- Compass – To access the compass heading functionality.
- Accelerometer – To access the accelerometer axes functionality.
- Sleep – To impose a forced wait time on program execution for a smoother and more predictable user experience. This implementation is important as without

it the application would simply execute as fast as it can, causing unintended behaviour in relation to the game mechanics.

Main Function.

The main function, *playGame*() uses a simplified traditional game loop pattern approach. Essentially, in a pseudo-infinite loop three steps are taken:

1. Retrieve input from the user.
2. Update the state of the game.
3. Render or "draw" the game elements to the screen.
4. Pause execution for some amount of time, to maintain game time consistency.

This approach allows for well-defined forward steps in execution. Each step can be deterministically perceived, in that it is abstractly clear what takes place when.

The game over state is defined as being active the moment the player has "picked up" all the objects on the screen, by colliding the player with them. When this happens, the player is shown the time that it took them to complete that level and when that output is finished, a new level is begun. Each time the player makes it past a level the number of pick-ups that spawn is incremented by 1.

```
while not self.isGameOver():
    self.getInput()
    self.update()
    self.draw()

    if self.isGameOver():
        current_time = microbit.running_time()
        seconds_elapsed = (current_time - level_start_time) / 1000

        microbit.display.scroll("TIME:" + str(int(seconds_elapsed)) + " SECONDS.")

        pickup_amount += 1

    sleep(100)
```

Figure 1: The game loop.

At the start of each level the starting time is stored, the pickups are spawned at random positions and the player is placed in the centre of the screen. This centre is defined as being at position 2 of both the X and Y axis. This is because even though the grid is 5x5, the LED on the top left is defined as being (0, 0) while the LED on the bottom right is defined as being (4, 4), a zero-based index.

getInput()

Input is defined as:

- The cardinal direction that the user is facing.
- The event that occurs when the user moves forward fast enough to trigger the acceleration check.
- Either button being pressed.

```
def getInput(self):
    direction = self.findApproxFacingDirection()

    if self.getAcceleration() > self.acceleration_needed_to_move:
        self.player.move(direction)

    if microbit.button_b.was_pressed():
        scroll_delay = 100
        microbit.display.scroll(self.findApproxFacingDirection(), scroll_delay)
    elif microbit.button_a.was_pressed():
        compass.calibrate()
```

Figure 2: Detecting input.

The cardinal direction that the user is facing is an approximation of their heading. For example, if the compass heading is less than 45 but greater than 315 they are perceived as facing north and if the heading is greater than 45 but less than 135 they are perceived as facing east and so on.

Using the direction they are facing, the current acceleration is checked against the acceleration value needed to move, which is 1200 milli-g. This value is 200 milli-g's higher than the resting acceleration of 1000 milli-g, or 1g (the acceleration due to the force of gravity).

As previously discussed, the user can recalibrate the compass at any time, and they do this by pressing the left button. The right button, b, shows them which direction they are facing by scrolling the first letter of the direction across the screen e.g. "N" for north.

update().

The main gameplay mechanic is picking up the items off the ground and this is resolved in update. Here, the position of the player is checked against the position of each pickup. If any of those positions matches, the pick-up is removed from the game.

```
def update(self):
    for pickup in self.pickups:
        if pickup.position.x == self.player.position.x:
            if pickup.position.y == self.player.position.y:
                self.pickups.remove(pickup)
```

Figure 3: Updating game state, checking collisions.

As an aside, Python uses the *enhanced for loop* as standard and it is amazingly simplistic, "For each object in a container of objects, do something with that object." As it will only ever deal with any objects that are in the container, no explicit bounds checking is needed.

draw().

Draw is the function used to display the game elements on the screen. More specifically, it turns off and on the LED's on the Micro:bit display and uses a brightness value to do so. The LED that represents the player is purposefully much brighter than the pickup objects because it helps the user more easily identify themselves.

Before the drawing of the pickups or player takes place, the display is cleared of any activation i.e. all the LED's are turned off. There are more elegant solutions to solving this problem of having stale pixels on the screen such as turning off the LED at the

position that the objects were if they moved or not turning them off it they didn't move but this approach was chosen for two reasons, simplicity and more importantly, limitations, as discussed below. As an aside, an intended but welcomed result of this implementation meant that the LEDs flicker and pulse somewhat, which can be more appealing than that of static displays.

Development Environment.

The IDE chose for this application was PyCharm, a JetBrains developed editor for Python that comes in free community and paid editions. A external tool, *uflash*, was configured in this environment to flash the Python file onto the Micro:bit

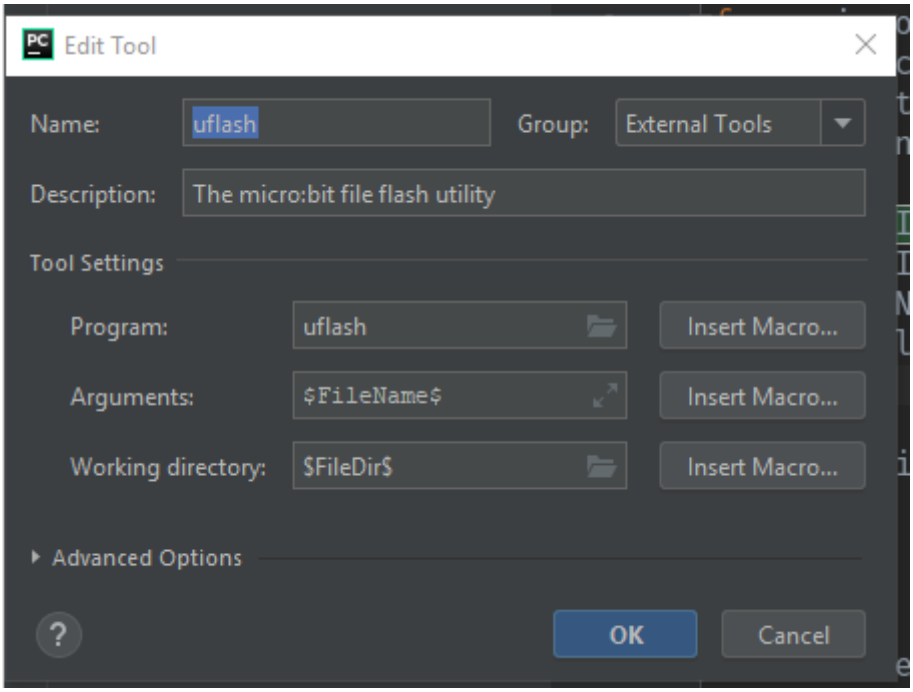


Figure 4: Part of creating an external tool to flash to program onto the micro:bit.

More information on this can be found here: https://mryslab.github.io/pseudo-microbit/install/ (Note: At the time of writing the minifier instructions did not function)

Additional editors can be found at https://codewith.mu/ and https://python.microbit.org/v/1.1 but may not contain the more useful features found in Pycharm, such as auto completion, varying syntax highlighting and source control.

Limitations.

The Micro:bit board holds only 16KB of static RAM, which is approximately 200 thousand time less than a modern smartphone or approximately 500 thousand times less than a modern laptop or desktop computer. This means that the general raw amount of code that can be within execution is limited and going over this amount will result in a memory error. The Python file intended to be flashed onto a Micro:bit that is included with this project comes extremely close to hitting this limit.

There are some optimisations that can be done to squeeze in the most code possible and they are, in order of ease:

1. Remove comments.
2. Remove unnecessary whitespace.
3. Shorten variable names as much as possible.
4. Redesign the system, taking strict note of the "Don't Repeat Yourself" principle.
5. Redefine the scope of the application.

The first three workarounds come with a cost in the form of code readability. Removing comments causes loss of communicated implementations, removing whitespace causes formatting to become unwieldy and shortening variable names can completely remove the semantic meaning of those variables often making it impossible to understand what is going on. Note that some whitespace is used in Python to denote blocks of code (commonly seen as curly brackets in other languages) and cannot be altered while remaining functional.

The last two workarounds require significantly more effort on the part of the developer but will prove to sometimes be the only way to have a well-defined system and a functioning application at the same time.

One observed outcome of this memory limitation was that it wasn't feasible to program in functions that could:

1. Check if a pick-up was going to spawn underneath the player at the start of the game or going to spawn at the same location as another pickup.
2. More elegantly deal with stale LED state.

It also almost completely halts any additional functionality of the application, which can be a very bad thing if not known ahead of time and designed for beforehand.

Another limitation is that the Micro:bit code flashing only works with a single file so all code must be in that one file, leaving out the possibility of defining classes separate to each other. This also means that class definitions must be in proper order and cannot interpret the definition of a class below itself if itself uses that class. E.g. *Position* must be defined before *GameObject*, as *GameObject* contains the use of the *Position* class.

A more abstract limitation is the fact that without a console to report errors or show debug statements it can be extremely slow to develop at times, often having to wait for slow scrolling text to go by, one character at a time. There may be a tool for this, but none was found at the time of writing.

Teaching Applications.

The Micro:bit has already established itself as a brilliant educational tool for use in the classroom as it holds the possibility for many fun and exciting developments while still being able to incorporate the fundamentals of programming and electronics to students.

The application that this project includes contains the necessary components to help students understand cardinal directions and the inclusion of the compass reading display could hold with it the possibility for them better be able to understand how to navigate maps or understand human-defined orientations in physical space.

The application also shows how basic game design can be implemented in Python as most of the functions present are mirrored in some way for many games.

The accelerometer and display could be used to show, in a novel way, how objects react when forces are applied to them and the in-game representation can help them along with this as it stores and shows the result of their actions.

Conclusion.

This project was very fun to work on yet did provide some challenges and learning opportunities along the way, resulting in a better understanding of how compasses and accelerometers work and how code can be designed to be readable and still fit in a very small amount of memory.

The developed application also proved useful as a refresher on the use of Python and proved helpful to gaining an understanding on how to implement single file object-oriented design in an orderly way.