

Mario

Assignment 3
CSSE1001/7030
Semester 2, 2019

Version 1.1.0
20 marks

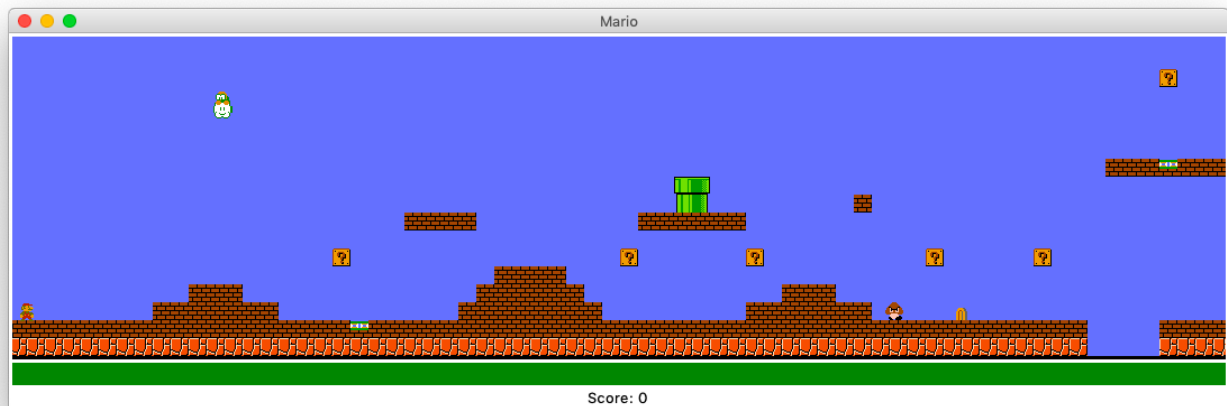
Due Friday 18th October, 2019, 20:30

Introduction

The goal of this assignment is to extend the existing support code of a mario style 2D-platformer written in Python using tkinter.

To be successful in completing this assignment you will have to use the concepts and skills that you have learnt. Specifically, you will need to have a good understanding of GUI programming, inter-class interactions, extending classes and file IO.

This document outlines the tasks that you will need to implement for this assignment.



Basic Running of the Game

Getting Started

The archive file `a3_files.zip` contains all the necessary files to start this assignment. A significant amount of support code has been provided to allow for the basic working game functionality to be implemented relatively easily.

The main assignment file is `app.py`, which contains an incomplete implementation of `MarioApp`, the top-level GUI application class. You should add code to `app.py` and modify `MarioApp` to implement the necessary functionality.

All features and code you write should be placed inside `app.py`, `level.py` or `player.py`. **You are not permitted to modify any files inside the game directory. Your assignment must be able to be launched by running `app.py`**

Because this assignment deals with multiple files, while not required, you may wish to investigate a more sophisticated IDE. One option is [PyCharm](#), which is [free for students](#).

Pymunk Library

Physics is implemented in the game using the [Pymunk library](#). It may be useful in completing the assignment to have an understanding of some of the components of the Pymunk library as you may need to refer to these libraries.

You will need to install this library in order to implement your tasks for this assignment. Pymunk can be installed by running the included `setup.py`.

Overview

The assignment is broken down into three main tasks:

1. The first task involves modifying the `app.py` file to implement basic launching of the game and some additional functionality
2. The second task involves extending the game mechanics to add more functionality to the game
3. The third task is for you to independently design and implement a sufficiently complex feature of your choosing

For postgraduates: There is an additional task for you to complete. You will need to appropriately implement animations using sprite sheets in the `spritesheets` folder.

Task 1 - Basic GUI

1.1 - Working Game

For this task, you will need to write code in `app.py`, which will start the game when the file is run. This will require you to write the `main` function to launch the `MarioApp` GUI.

You should modify `MarioApp` so that the window title is something appropriate (e.g. "Mario")

Once the game runs, you will need to implement keyboard bindings. You should find an appropriate location in the class to make `bind` calls for each of the keyboard presses to the appropriate method. The binds should behave as follows:

Key	Action
W, UP, SPACE	Makes the player jump (Hint: see <code>MarioApp._jump</code>).
A, LEFT	Moves the player to the left (Hint: see <code>MarioApp._move</code>).
S, DOWN	Makes the player duck (Hint: see <code>MarioApp._duck</code>).
D, RIGHT	Moves the player to the right (Hint: see <code>MarioApp._move</code>).

1.2 - File Menu and Dialogs

Implement a menu for the game which has a top level "File" menu. Within the "File" menu, you will need the following buttons:

Button	Purpose
Load Level	Prompts the user with a popup text input dialog. When the player inputs a level filename, load that level replacing the currently loaded level.

Button	Purpose
Reset Level	Reset all player progress (e.g. health, score, etc) in the current level.
Exit	Exits the game.

When a player runs out of health, you should show a dialogue asking whether they want to restart the current level or exit the game.

Note: On Mac OS X, the file menu should appear in the global menu bar (top of the screen).

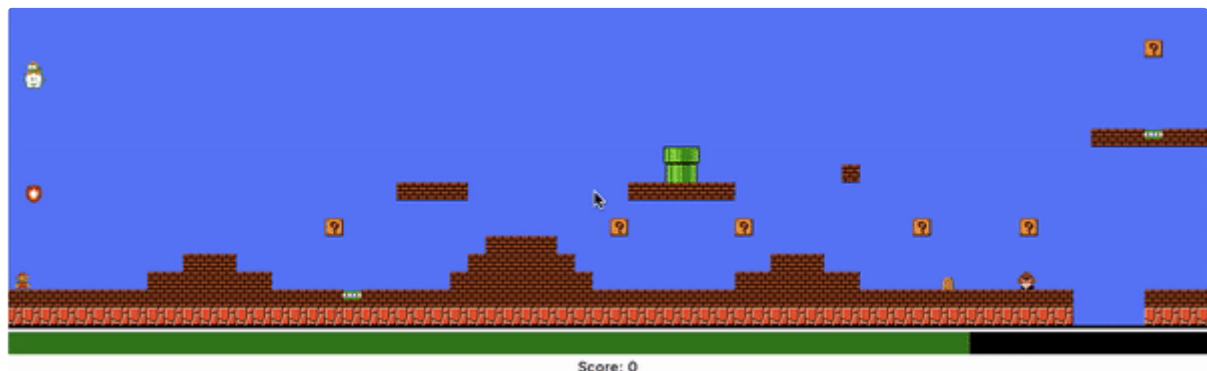
1.3 - Status Display

Implement a custom tkinter widget (i.e. a class which inherits from `tk.Frame`) which displays the score and health of the player at the bottom of the window.

The player's score should be shown as a single number. The health of the player should be displayed as a 'health bar' (similar to the image below). The health bar should be coloured as follows:

- When the player has greater than or equal to 50% of their maximum health, it should be coloured green.
- When the player has between 25% and 50% of their maximum health, it should be coloured orange.
- When the player has less than or equal to 25% of their maximum health, it should be coloured red.

This widget needs to be updated when the score and health of the player updates during gameplay.



Animated Health Bar Example

1.4 - Bounce Block 🧱

Implement a type of block which will propel the player into the air when they walk over or jump on top of the block.

The velocity with which you propel the player should be sensible but noticeable.

Hint: You should implement a bounce block by making a new class which extends `Block`.

You may (and should) utilise the `bounce_block` image files found in `a3_files.zip`

The bounce block is represented by the character, `b` in level files.

1.5 - Mushroom Mob 🍄

Implement a new type of mob which has the following properties:

- The mob should move at a reasonably slow rate
- When the mob collides with a block, player, or other mob it should reverse its direction (HINT: `Mob.set_tempo`)
- When the mob collides with the side of a player, the player should lose 1 health point and be slightly repelled away from the mob
- When a player lands on top of the mob, the player should bounce off the top of the mob and the mob should be destroyed

You may find it useful to look at the existing mob classes and collision handling methods before commencing this task.

You may (and should) utilise the `mushroom` image files found in `a3_files.zip`

The mushroom mob is represented by the character, `@` in level files.

Task 2 - Extending the Game

2.1 - Star Item ★

Implement a type of item that makes the player invincible for 10 seconds (that is, they should not be able to take any damage during this time). Any mobs that the player collides with during this time should be immediately destroyed.



During the time the player is invincible, the player's health bar should be coloured yellow.

You should utilise the `star` image files found in `a3_files.zip`

The star is represented by the character, `*` in level files.

2.2 - Goals

Implement a new type of block that acts as a goal and allows changing between levels. The block should be constructed with an id and the name of the next level file. Your must include at least the following two types of goals:

Type	Sprite	Purpose
Flagpole		When a player collides with this, immediately take the player to the next level. If the player lands on top of the flag pole, their health should be increased.
Tunnel		By default this should act as a normal block. However, if the player presses the down key while standing on top of this block, the player should be taken to another level.

A player's current state (coins, health, etc) should not change when the level changes, apart from their position.

You may find the `GOAL_SIZES` constant defined in `app.py` useful in determining the size of the various goal types.

You may also find the `flag` and `tunnel` image files in `a3_files.zip` useful.

The flags and tunnels are represented by the characters, `I` and `=` respectively in level files.

The levels loaded by the flag and tunnel may be hardcoded until the next task is completed

2.3 - Loading Configuration

Implement the ability to load a configuration file for a game of mario. When the game is launched the user should be prompted to enter the file path for a configuration file.

The configuration file will be in a similar format to the example given below.

```

==World==
gravity : 400
start : level1.txt

==Player==
character : luigi
x : 30
y : 30
mass : 100
health : 4
max_velocity : 100

==level1.txt==
tunnel : bonus.txt
goal : level2.txt

==bonus.txt==
goal : level1.txt

==level2.txt==
tunnel : small_room.txt
goal : level3.txt

==small_room.txt==
goal : level2.txt

==level3.txt==
goal : END

```

At the minimum, a configuration file will include a `==World==` tag, a `==Player==` tag and a tag for the level specified as start in `==World==`

A `==World==` tag should have a `gravity` property which will set the gravity of a world when it is constructed. It should also contain a `start` property which will be the filepath of the first level to load.

A `==Player==` tag should have the following properties:

- `character`: This can be either mario or luigi and will change the image displayed in game.
- `x`: This is the starting x co-ordinate of the player.
- `y`: This is the starting y co-ordinate of the player.
- `mass`: This is the weight of the player set when adding the player to the world.
- `health`: This is the maximum amount of health a player will have.
- `max_velocity`: This is the maximum x velocity that a player can reach when moving.

Each of the levels should have it's own tag, e.g. `==level==` where level is the file path of that level.


A level tag should have a `goal` property which is the filename of the level to load when the player reaches a flag goal block. If the next level is END then it should prompt the user that the game is over and close the game window.

A level tag may also have a `tunnel` property which is the filename of the level to load when the player enters a tunnel block.

If the configuration file is invalid, or missing and cannot be parsed, you should alert the user via a `tkinter` error message box and then exit the game.

2.4 - Switches

Implement a new type of block that acts like a switch. When a player lands on-top of a switch, all bricks within a close radius of the switch should disappear. A player should not be able to trigger a switch by walking into the side of it (the player should stop moving as if it were any other block).

When a switch is pressed, it should remain in a 'pressed' state () for 10 seconds. During this time, the player should not be able to collide with this block (HINT: returning `False` from a collision handler will turn off collisions). After this time, the switch should revert to its original state and all invisible bricks should become visible again.

It is up to you to pick a sensible radius for the switch. It needs to be noticable when playing with the supplied level files.

You may (and should) utilise the `switch` and `switch_pressed` image files found in `a3_files.zip`

The switch is represented by the character, `s` in level files.

2.5 - High Scores

In this task you should implement a way to store the high scores for each level in a file.

The highscore information should be stored in (a) txt file(s) in a reasonable format. The exact format of the file(s) and way the data is stored is up to you but marks may be deducted for inappropriate save format.

When a player reaches a goal (and therefore finishes a level), prompt the user via a dialog for their name and store the score they had at the end of the level to the relevant file for the level (creating it if it doesn't exist). Adding a new entry to the file shouldn't remove any existing entries already in it.

Add a button to the file menu called "High Scores". When clicked this will open a new custom `tk.Toplevel` window displaying the names and scores of the top ten highest scorers for the current level, sorted in descending order by score. Note that the file might have more or less than ten entries, but you shouldn't display more than the top ten entries in the window. If the file doesn't exist, the dialog shouldn't display any entries.

Task 3 - Over to You

For this task you are expected to demonstrate your ability to intelligently extend the features of the base game. The features that you choose to implement are left up to you. For full marks in this task your features will need to be of sufficient complexity. For example, sound effects are generally *not* deemed to be of 'sufficient complexity'.

It is highly encouraged that you only attempt this task if you feel confident in your programming ability. You will need to discuss your plans for this task with a tutor in a practical or via a private question on Piazza. The tutor will be able to advise you as to whether your intended feature is of sufficient complexity to gain marks.

If you are attempting to implement this task then you must submit a feature PDF which outlines exactly what features you have implemented and how to use them when running your assignment. The PDF should include screenshots of the features and a brief description of how the features were implemented.

Hints

If you are thinking about using pygame for your task three please ensure that you discuss it with a tutor as pygame should be avoided where possible.

If you attempt task 3 and it requires additional assets then you are allowed to zip your entire project and submit that. Ensure that extracting the zip will have `app.py` at the top level. You must mention in your features.pdf document that you have submitted a zip and what additional assets were included.

If your feature wants to have different functionality for the `w` and `up` keys you may rebind those keys.

If your feature.pdf does not clearly explain your feature then you may not receive marks for that feature.

Postgraduate Task

The task for postgraduate students is to implement a class `SpriteSheetLoader` which is able to load images from a sprite sheet. A sprite sheet is an image which consists of multiple smaller images, see the spritesheets folder for reference. The

`SpriteSheetLoader` should be able to load one of the smaller images from a sprite sheet based on the smaller images location and position within the sheet.

Hint: To implement this, you will want to investigate using the [Pillow library](#)

Hint: To implement this, you may also want to investigate making a new `ViewRenderer` subclass to handle sprite sheets. Ensure that loaded images are stores in the `self._images` dictionary within the new `ViewRenderer` subclass.

In addition to loading images from a spritesheet, you will need to implement animations for the following entities.

- Player: When the player is walking, jumping or falling animate the player with the appropriate sprites from the `characters` sprite sheet.
- Mushroom Mob: When the mushroom mob is walking, animate the walk using the appropriate sprites from the `enemies` sprite sheet.
- Mushroom Mob: When the mushroom mob is jumped on, animate the squishing using the appropriate sprites from the `enemies` sprite sheet.
- Coins: Animate the coin to be spinning using the appropriate sprites from the `items` sprite sheet.
- Bounce Block: When the bounce block is used, animate the extension of the bounce block using the sprites in the `items` sprite sheet.

Marking

Your style mark will be scaled by your percentage achieved from task 1.

Style = Style Marks * Task 1 Percentage

Your total mark will be made up of functionality marks and style marks. Functionality marks are worth 15 of the 20 available marks and style marks are worth the other 5 marks.

Total = (Style Percentage * 5) + (Functionality Percentage * 15)

Functionality Assessment

Your assignment will be marked by tutors who will run your `app.py` file and evaluate the completeness and correctness of the tasks you've implemented.

The table below specifies the mark breakdown for each of the tasks for for CSSE1001 and CSSE7030 students.

Task	CSSE1001 Weighting	CSSE7030 Weighting
Task 1	35%	30%
Task 2	35%	30%
Task 3	30%	25%
Postgraduate	0%	15%

The weighting of each of the sub tasks within the tasks is given in the table below.

Task	Sub-Task	Weight
Task 1	Working Game	10%
	File Menu and Dialogs	15%
	Status Display	30%

Task	Sub-Task	Weight
Task 2	Bounce Block	20%
	Mushroom Mob	25%
	Star Item	10%
	Goals	20%
	Loading Configuration	30%
	Switches	20%
	High Scores	20%
Task 3		100%
Postgraduate	Sprite Loading	50%
	Animation	50%

Code Style

The style of your assignment will be assessed by one of the tutors, and you will be marked on the broad categories listed below.

	Description	Weight
Documentation	All classes, methods and functions have appropriate documentation in the form of docstrings. The parameters and return values where applicable have been documented appropriately including an accurate description and the types. Where there exists code whose meaning is not immediately obvious or clear upon reading, its meaning has been clarified using appropriate inline comments. Documentation is concise and is not excessive or extraneous.	10%
Algorithmic Logic & Design	All algorithmic logic is suitable for solving the intended objective of the algorithm. Algorithms are not overly complex or without purpose. All algorithms have been simplified to accomplish the objective with a succinct and clear approach.	30%
Readability	All code is clear and easy to read. Lines of code have been sufficiently abstracted and clarified through the use of good variable naming so that every line is easily interpreted and the intention is clear. Lines are not excessive so as to distract from the intention. Blocks of related code have been grouped together through vertical whitespace to produce logical groups of code. Excessive or insufficient whitespace which detracts from the structure of code is not present. Code adheres to common Python style practices such as line length, line breaks, indentation, etc.	20%
Good Object Oriented Practices	Inheritance is used appropriately to reduce duplicated code. Inheritance has been used where appropriate to succinctly and clearly extend functionality of existing classes. Classes have been designed in a way which provides a logical and clear structure of classes. Appropriate information is passed between and stored within classes. Private class attributes have only been used within the declaring class or its subclasses. Methods have been included where it improves ability to utilise a class.	40%

Assignment Submission

Your assignment must be submitted via the assignment three submission link on Blackboard. You must submit a python file, `app.py`, containing your implementation of the assignment. If you attempted task 3 then you must also submit a PDF file, `features.pdf`, containing a description of your features. Both files should be uploaded individually to blackboard. If you modify `player.py` or `level.py` then you must also upload those files. If you require additional assets for task 3, upload all your assets, python files and features pdf in a single zip archive named `a3.zip`. Once submitted download the support code and your app.py submission from blackboard and place the app.py in the extracted support code file to ensure everything works as expected.

Late submission of the assignment will **not** be accepted. Do not wait until the last minute to submit your assignment, as the time to upload it may make it late. Multiple submissions are allowed, so ensure that you have submitted an almost complete version of the assignment *well* before the submission deadline. Your latest on-time, submission will be marked. Ensure that you submit the correct version of your assignment. An incorrect version that does not work **will** be marked as your final submission.

In the event of exceptional circumstances, you may submit a request for an extension. See the [course profile](#) for details of how to apply for an extension. Requests for extensions must be made **no later** than 48 hours prior to the submission deadline. The expectation is that with less than 48 hours before an assignment is due it should be substantially completed and submittable. Applications for extension, and any supporting documentation (e.g. medical certificate), must be submitted via [my.UQ](#). You must retain the original documentation for a *minimum period* of six months to provide as verification should you be requested to do so.

Change Log

Version 1.1.0 - Sep 30

Assignment Sheet

- Added a marking criteria.
- Added a hints section to task 3.
- Fixed configuration file example to include settings for the references small_room.txt

Support Code

app.py

- World creation in `MarioApp.__init__` replaced with `MarioApp.reset_world`
- `MarioApp.reset_world` fixed when changing levels
- `MarioApp.scroll` behaviour changed

level.py

- `load_world` now accepts and passes args to `WorldBuilder.add_entity`

world.py

- New method `World.get_things_in_range` added

player.py and entities.py

- Removed `Player.is_jumping` and `Player.set_jumping`
- Added `DynamicEntity.is_jumping` and `DynamicEntity.set_jumping`