Hacker

Assignment 3
Semester 2, 2021
CSSE1001/CSSE7030

Due date: 16:00, 29th of October, 2021 GMT+10

1 Introduction

The third assignment will be a culmination of all your existing knowledge of the course content. Your task will be to make a game with a Graphical User Interface (GUI). The game of Hacker is conceptually similar to Space Invaders, the player is fixed vertically while entities progress vertically towards the player (See Figure 1).

This assignment is split into 3 tasks of increasingly difficulty. As tasks increase, their total marks decrease. Task 3 is for CSSE7030 students only. Student's are expected to follow the Apple Model-View-Controller (MVC) design pattern. You are not allowed to import any modules

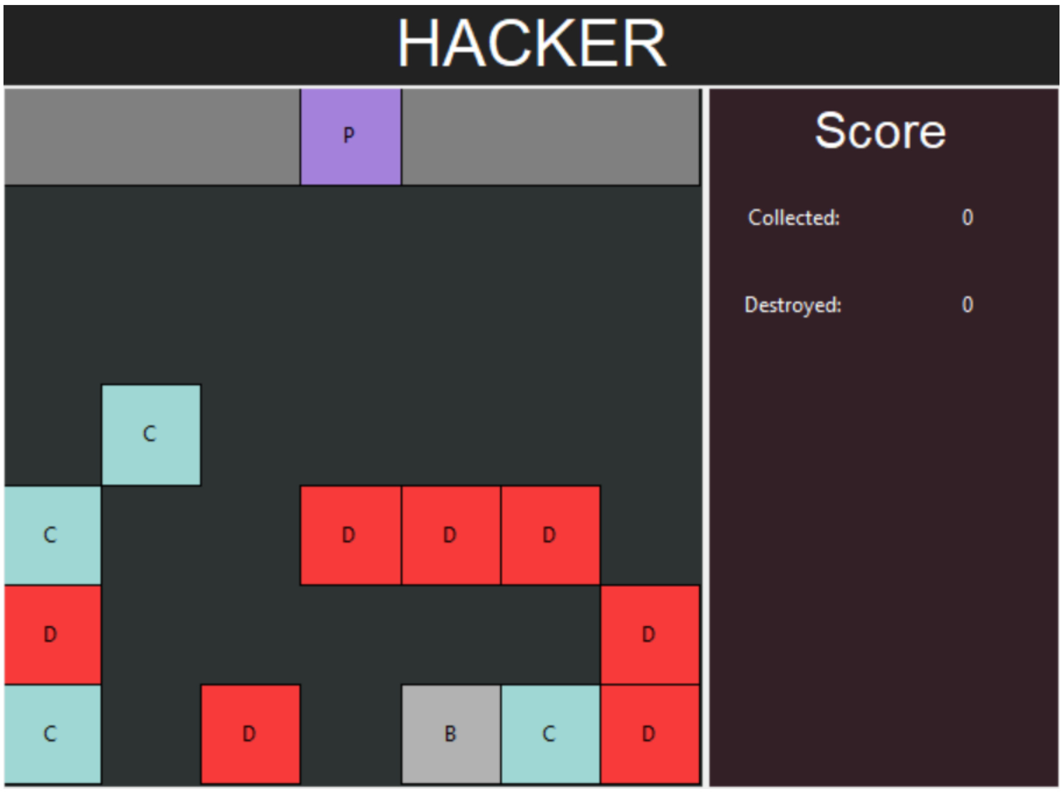

Figure 1: A preview of an in-progress game. The Player is represented by a purple square with 'P'. Other entities are indicated with 'B', 'C', and 'D' squares.

other than the following:

- `tkinter` (and all `tkinter` submodules)
- `random`
- `Pillow`
- `a3_support`

Importing any module not listed here will result in a deduction of up to 100%

2 Terminology

Destroyable (D) Red entity that players must destroy before it reaches the top row where the player is located.

Collectable (C) Blue entity that players must collect. By default, collecting 7 Collectables will cause the Player to win the game.

Blocker (B) Grey entity that players cannot shoot through.

Shot type There are 2 shot types in the game - destroy and collect. Rules on what can be destroyed and collected can be seen in the Model Class descriptions (Section 5.2).

3 Overview and Game Rules

The goal of the game is to collect a specified amount of collectable entities without having a destroyable entity reach the player's row. If a red destroyable entity reaches the player's row then the game is over. If any other entity reaches the top row then the game continues as normal.

Players mainly interact with the game by moving non-player entities on the grid. It is important to note that the player never moves and remains centred on the grid in the top row. Entities are rotated left and right and *wrap* around to the other side of the grid if they are moved off an edge (See Figure 2).

Every 2 seconds, non-player entities step 1 space closer to the player and a new bottom row containing randomly generated entities will appear. This is known as a `'step event'`.

4 Getting Started

Download `a3.zip` from Blackboard — this archive contains the necessary files to start this assignment. Some support code has been included to assist with implementing the program. Once extracted, the `a3.zip` archive will provide the following files/directories:

a3.py
File to run your program. *Write your code in this file. Do not change any provided functions found in this file.* Note: You are required to use the methods provided in this file.

a3_support.py
Constants, functions, and classes you *may* find helpful for writing the engine. Read the docstrings in order to understand how to use them in your solution. *Do not* make changes to this file.

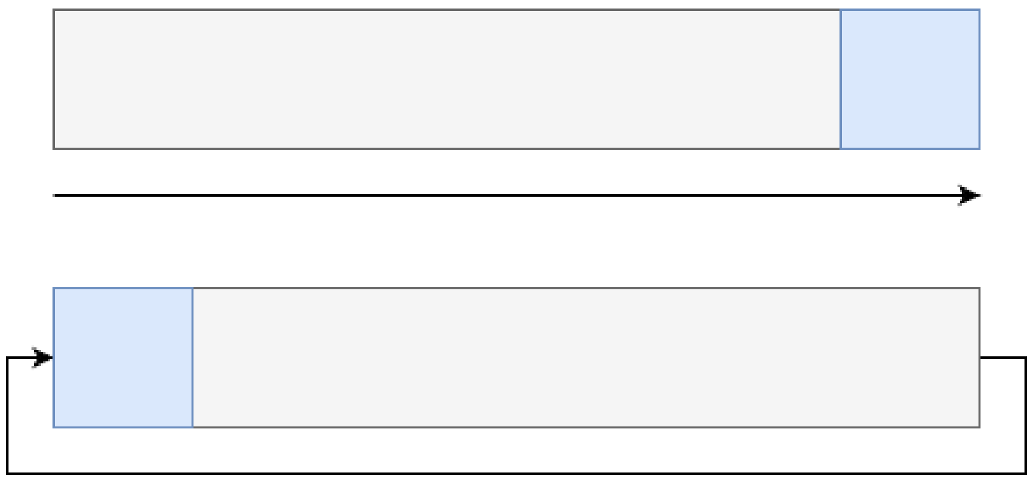

Figure 2: Before and after of rotating the grid right. Upon rotating right, the blue entity is moved to the other edge as if these edges were connected.

images

Folder containing images used for Task 2 and 3 functionality. Note: Task 3 is the CSSE7030 task.

5 Task 1 - Basic Model and View

The following sub-sections outline the gameplay requirements (Section 5.1), as well as the recommended structure for your code (5.2, 5.3). Remember to test individual methods as you write them. Within this section we outline methods that you must write for the model, and some methods that may be useful to write in the view classes for Task 1. In the view classes, type hints are omitted, as it is up to you to determine what these should be.

5.1 Basic Gameplay

User input events should cause behaviour as per Table 1.

Event	Behaviour
Key Press: ‘A’ or ‘a’	All non-player entities rotate left one square.
Key Press: ‘D’ or ‘d’	All non-player entities rotate right one square.
Key Press: ‘SPACE’	Shoot a destroy shot.
Key Press: ‘ENTER’	Shoot a collect shot.

Table 1: Task 1 events and their corresponding behaviours.

The entities can rotate independently of the step event. That is, the entities can rotate multiple times within the space of a single step event.

When the player wins or loses the game they should be informed of the outcome via a messagebox that pops up on the screen. Once this messagebox is closed, the game is ended and the application should terminated.

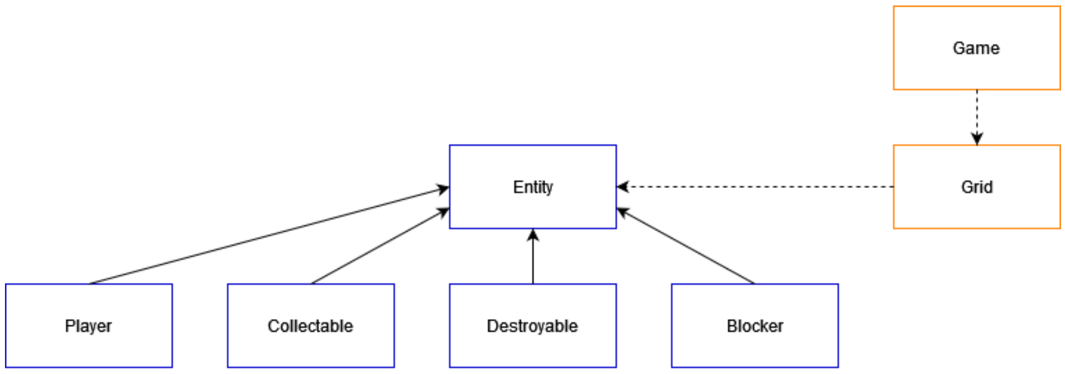

Figure 3: Basic class diagram.

5.2 Model Classes

This section outlines classes and functions you need to implement that represent the model of the game. The class diagram below depicts these classes with the basic relationships between them (See Figure 3).

- Solid arrows indicate inheritance (i.e. the “is-a” relationship).
- Dashed arrows indicate composition (i.e. the “has-a” relationship).

5.2.1 Entity

Entity is an abstract class that is used to represent any element that can appear on the game’s grid.

- `display(self) -> str:`
  Return the character used to represent this entity in a text-based grid. An instance of the abstract Entity class should never be placed in the grid, so this method should only be implemented by subclasses of Entity. To indicate that this method needs to be implemented by subclasses, this method should raise a NotImplementedError.

- `__repr__(self) -> str}:`
  Return a representation of this entity. This should be inherited by each subclass but should instead display the class name of that subclass rather than always Entity’.

```
>>> entity = Entity()
>>> repr(entity)
        'Entity()'
```

5.2.2 Player

A subclass of Entity representing a Player within the game.

- `display(self) -> str:`
  Return the character representing a player: `'P'`

5.2.3 Destroyable

A subclass of Entity representing a Destroyable within the game. A destroyable *can* be destroyed by the player but not collected.

- `display(self) -> str:`
  Return the character representing a destroyable: `'D'`

5.2.4 Collectable

A subclass of Entity representing a Collectable within the game. A collectable *can* be destroyed *or* collected by the player.

- `display(self) -> str:`
  Return the character representing a collectable: `'C'`

5.2.5 Blocker

A subclass of Entity representing a Blocker within the game. A blocker *cannot* be destroyed *or* collected by the player.

- `display(self) -> str:`
  Return the character representing a blocker: `'B'`

5.2.6 Grid

The Grid class is used to represent the 2D grid of entities. The top left position of the grid is indicated by (0, 0).

- `__init__(self, size: int) -> None:`
  A grid is constructed with a size representing the number of rows (equal to the number of columns) in the grid. Entities should be stored in a dictionary. Initially a grid does not contain any entities.

- `get_size(self) -> int:`
  Return the size of the grid.

- `add_entity(self, position: Position, entity: Entity) -> None:`
  Add a given entity into the grid at a specified position. This entity is only added if the position is valid.

  If an entity already exists at the specified position, this method will replace the current entity at the specified position.

- `get_entities(self) -> Dict[Position, Entity]:`
  Return the dictionary containing grid entities.

  Updating the returned dictionary should have no side-effects.

- `get_entity(self, position: Position) -> Optional[Entity]:`
  Return a entity from the grid at a specific position or None if the position does not have a mapped entity.

- `remove_entity(self, position: Position) -> None:`
  Remove an entity from the grid at a specified position.

- `serialise(self) -> Dict[Tuple[int, int], str]:`
  Convert dictionary of Position and Entities into a simplified, serialised dictionary mapping tuples to characters, and return this serialised mapping.

  Tuples are represented by the x and y coordinates of a Positions and Entities are represented by their ‘display()‘ character.

- `in_bounds(self, position: Position) -> bool:`
  Return a boolean based on whether the position is valid in terms of the dimensions of the grid. Return True iff:

  - ⋆ $x \geq 0$ and $x <$ grid size
  - ⋆ $y \geq 1$ and $y <$ grid size

- `__repr__(self) -> str`: Return a representation of this Grid.

```
>>> grid = Grid(5)
>>> repr(grid)
        'Grid(5)'
```

5.2.7 Game

The Game handles the logic for controlling the actions of the entities within the grid.

- `__init__(self, size: int) -> None:`
  A game is constructed with a size representing the dimensions of the playing grid. A game should be constructed with at least the following variable:
  - ⋆ Flag representing whether the game is won or lost
- `get_grid(self) -> Grid:`
  Return the instance of the grid held by the game.
- `get_player_position(self) -> Position:`
  Return the position of the player in the grid (top row, centre column). This position should be constant.
- `get_num_collected(self) -> int:`
  Return the total of Collectables acquired.
- `get_num_destroyed(self) -> int:`
  Return the total of Destroyables removed with a shot.
- `get_total_shots(self) -> int:`
  Return the total of shots taken.
- `rotate_grid(self, direction: str) -> None:`
  Rotate the positions of the entities within the grid depending on the direction they are being rotated. Valid directions are specified by the constants found in the `a3_support` file. Entity positions rotate as seen in Figure 2.
  - Left rotation moves all entities by an offset of (-1, 0)
  - Right rotation moves all entities by an offset of (1, 0)
- `_create_entity(self, display: str) -> Entity:`
  Uses a display character to create an Entity. Raises a NotImplementedError if the character parsed into as the display is not an existing Entity.
- `generate_entities(self) -> None: (provided)`
  Generate random entities for each step of the game.
- `step(self) -> None:`
  This method moves all entities on the board by an offset of (0, -1). Once entities have been moved, new entities should be added to the grid (using `generate_entities`). Entities should not be re-added to the grid if they have moved out of bounds.

- `fire(self, shot_type: str) -> None:`
  Handles the firing/collecting actions of a player towards an entity within the grid. A shot is fired from the players position and iteratively moves down the grid.

  `shot_type` refers to whether a collect or destroy shot has been fired (refer to Entity descriptions for how different entities react to being hit by different types). Valid `shot_type` constants can be found in the `a3_support` file.

- `has_won(self) -> bool:`
  Return True if the player has won the game.

- `has_lost(self) -> bool:`
  Returns True if the game is lost (a Destroyable has reached the top row).

5.3 View Classes

You must implement view classes for both a) the game grid and b) the game statistics (left and right sections of Figure 1 respectively). Because these widgets can both be represented by grids of rectangles, you should create an abstract class to factor out the shared functionality. Note: View classes should not be resizable.

5.3.1 AbstractField

`AbstractField` is an abstract view class which inherits from `tk.Canvas` and provides base functionality for other view classes. An `AbstractField` can be thought of as a grid with a set number of rows and columns, which supports creation of text at specific positions based on row and column. The number of rows may differ from the number of columns, and the cells may be non-square. You must define the constructor for the `AbstractField` class as:

- `__init__(self, master, rows, cols, width, height, **kwargs)`: The parameters `rows` and `cols` are the number of rows and columns in the grid, `width` and `height` are the width and height of the height of the grid (and hence the width and height of the tk.Canvas, measured in pixels) and `**kwargs` signifies that any additional named arguments supported by `tk.Canvas` should also be supported by `AbstractField`.

The following methods may be useful to include in the `AbstractField` class.

- `get_bbox(self, position)`: Returns the bounding box for the position; this is a tuple containing information about the pixel positions of the edges of the shape, in the form `(x_min, y_min, x_max, y_max)`.

- `pixel_to_position(self, pixel)`: Converts the (x, y) pixel position (in graphics units) to a (row, column) position.

- `get_position_center(self, position)`: Gets the graphics coordinates for the center of the cell at the given (row, column) position.

- `annotate_position(self, position, text)`: Annotates the center of the cell at the given (row, column) position with the provided text.

5.3.2 GameField

`GameField` is a visual representation of the game grid which inherits from `AbstractField`. Entities are drawn on the map using coloured rectangles at different (row, column) positions. You must annotate the rectangles of all entities with their `display()` characters (as per Figure 1). You

must use the `create_rectangle` and `create_text` methods from tk.Canvas to achieve this. The colours representing each entity are found in `a3_support`. The `FIELD_COLOUR` constant indicates the background colour of the view.
You must define the constructor for the `GameField` class as:

- `__init__(self, master, size, width, height, **kwargs)`: The `size` parameter is the number of rows (= number of columns) in the grid, `width` and `height` are the width and height of the grid (in pixels) and `**kwargs` signifies that any additional named arguments supported by `tk.Canvas` should also be supported by `GameField`.

It may be useful to add the following methods to the `GameField` class:

- `draw_grid(self, entities)`: Draws the entities (found in the Grid's entity dictionary) in the game grid at their given position using a coloured rectangle with superimposed text identifying the entity (this includes the Player entity).

- `draw_player_area(self)`: Draws the grey area a player is placed on. This colour is found in `a3_support` file as the `PLAYER_AREA` constant.

5.3.3 ScoreBar

`ScoreBar` is a visual representation of shot statistics from the player which inherits from `AbstractField`. For Task 1, this bar displays the number of collectables acquired and the number of destroyables shot as seen in Figure 1. The ScoreBar's statistics values must update in real time. You must define the constructor for the `ScoreBar` class as:

- `__init__(self, master, rows,  **kwargs)`: `rows` is the number of rows contained in the ScoreBar canvas. This should match the size of the grid. By default, columns should be set to 2. Relevant support file constants should be used for window sizing and `**kwargs` signifies that any additional named arguments supported by `tk.Canvas` should also be supported by `ScoreBar`. Finally, the background colour of the `ScoreBar` can be found in `a3_support` as `SCORE_COLOUR`.

5.3.4 HackerController

`HackerController` acts as the controller for the Hacker game. The constructor should be defined as:

- `__init__(self, master, size)`: The parameter `master` represents the master window and `size` represents the number of rows (= number of columns) in the game map. This method should draw the title label (see `a3_support` for font, colour and size details), initialise the Game model, and instantiate and pack the `GameField` and `ScoreBar`. Finally, this method should initiate the game's '`step`'

The following are suggested methods to include in this class:

- `handle_keypress(self, event)`: This method should be called when the user presses any key during the game. It must handle error checking and event calling and execute methods to update both the model and the view accordingly.

- `draw(self, game)`: Clears and redraws the view based on the current game state.

- `handle_rotate(self, direction)`: Handles rotation of the entities and redrawing the game. It may be easiest for the `handle_keypress` method to call `handle_rotate` with the relevant arguments.

- `handle_fire(self, shot_type)`: Handles the firing of the specified shot type and redrawing of the game. It may be easiest for the `handle_keypress` method to call `handle_fire` with the relevant arguments.
- `step(self)`: The `step` method is called every 2 seconds. This method triggers the step method for the game and updates the view accordingly. Note: The `.after` method for tkinter widgets may be useful when trying to get this method to run every 2 seconds.

6 Task 2 - Images, StatusBar and Filemenu Extensions

Task 2 requires you to add additional features to enhance the game's look and functionality. Figure 4 gives an example of the game at the end of Task 2.

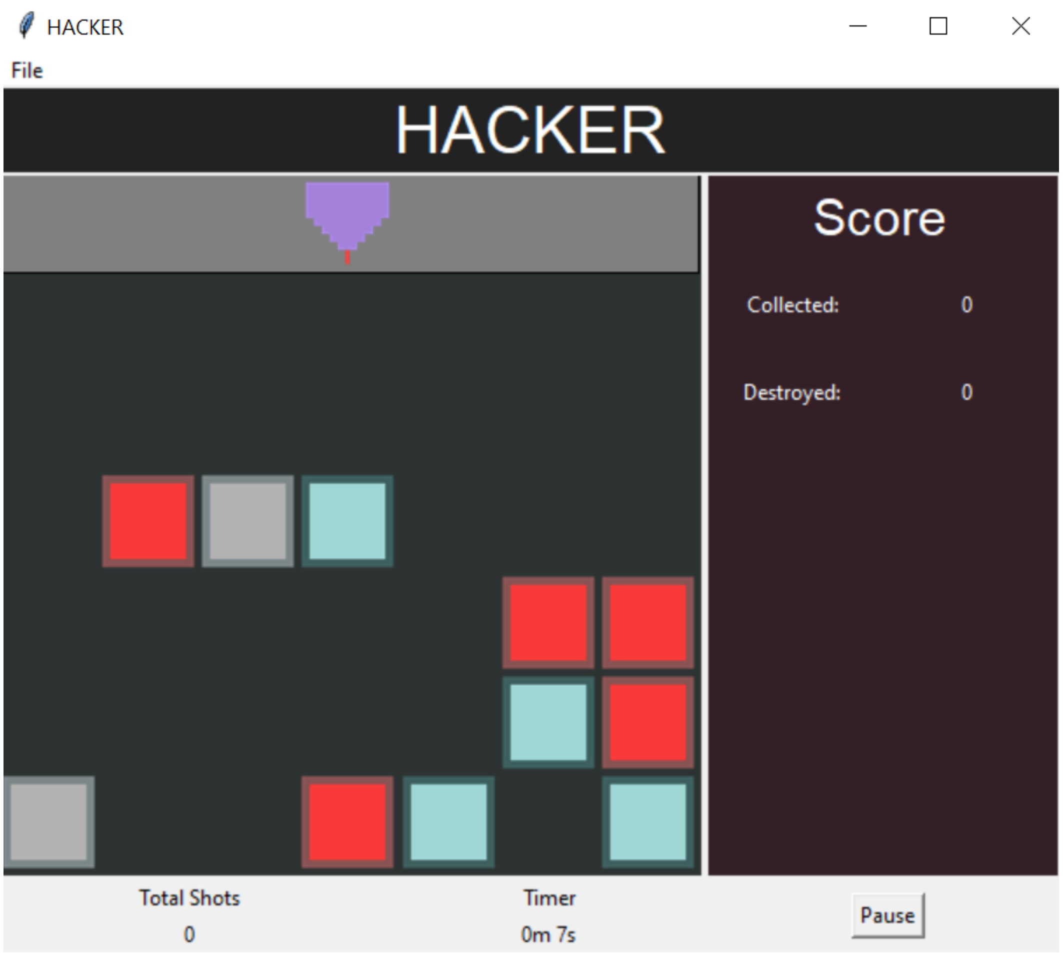

Figure 4: Task 2 GUI

Note: Your Task 1 functionality must still be testable. When your program is run with the `TASK` constant in `a3_support.py` set to 1, the game should display **only** Task 1 features. When your program is run with the `TASK` constant set to 2, the game should display all attempted task 2 features. There should be **no** Task 2 features visible when running the game in Task 1 mode. To separate Task 1 and 2, you must create a new interface class that extends the functionality of the HackerController class named AdvancedHackerController.

6.1 Images

Create a new view class, ImageGameField, that extends your existing GameField class. This class should behave similarly to GameField, except that images should be used to display each square rather than rectangles (see the provided images folder). The game map should be set up as an ImageGameField in Task 2, but you should still provide a functional GameField class that allows us to test your Task 1 functionality when TASK=1

6.2 StatusBar

Add a `StatusBar` class that inherits from `tk.Frame`. In this frame, you should include:

- A shot counter, displaying how many shots overall the player has made in the current game.
- A game timer displaying the number of minutes and seconds the user has been playing the current game.
- A 'Pause/Play' button, which either pauses or resumes the game. (Button text will toggle on every press)

6.3 File Menu

Add a file menu with the options described in Table 2. Note that on Windows this will appear at the top of the game window, whereas on Mac this will appear at the top of your screen. For saving and loading files, you must design an appropriate text (.txt) file format to store information about game state. Reasonable actions by the user (e.g. trying to load a non-game file) should be handled appropriately (e.g. with a pop-up to inform the user that something went wrong). Any error output messages caused by reasonable behaviour in the GUI will be marked down.

Option	**Behaviour**
New game	Start a new Hacker game
Save game	Prompt the user for the location to save their file (using an appropriate method of your choosing) and save all necessary information to replicate the current state of the game.
Load game	Prompt the user for the location of the file to load a game from and load the game described in that file.
Quit	Prompt the player via a messagebox to ask whether they are sure they would like to quit. If no, do nothing. If yes, quit the game (window should close and program should terminate).

Table 2: File menu options.

7 Task 3 - CSSE7030 (Masters) Task

7.1 Model

This section outlines the class and functions you need to extend to the original model outline in Task 1.

7.1.1 Bomb

A subclass of Entity representing a `Bomb` within the game that player's can destroy other entities with. A bomb removes all entities within a 'splash damage' radius. The `SPLASH` constant refers to the specific offsets. A bomb *can* be destroyed but *cannot* be collected.

- `display(self) -> str:`
  Return the character representing a bomb: `'B'`

7.1.2 Game

The relevant function in `Game` will need to be extended to accommodate for the Bombs splash damage functionality. This method will need to check for TASK=3 in order to work and the commented section in `generate_entities()` will need to be uncommented.

The `Game` class will also be extended to allow players to have one extra life. This means players will start with an extra life.

7.2 View

The updates to the model seen in the previous section will be visualised as seen in Figure 5. Defining another subclass would be beneficial for the implementation of the extended features. Note: These features should only be displayed when TASK=3.

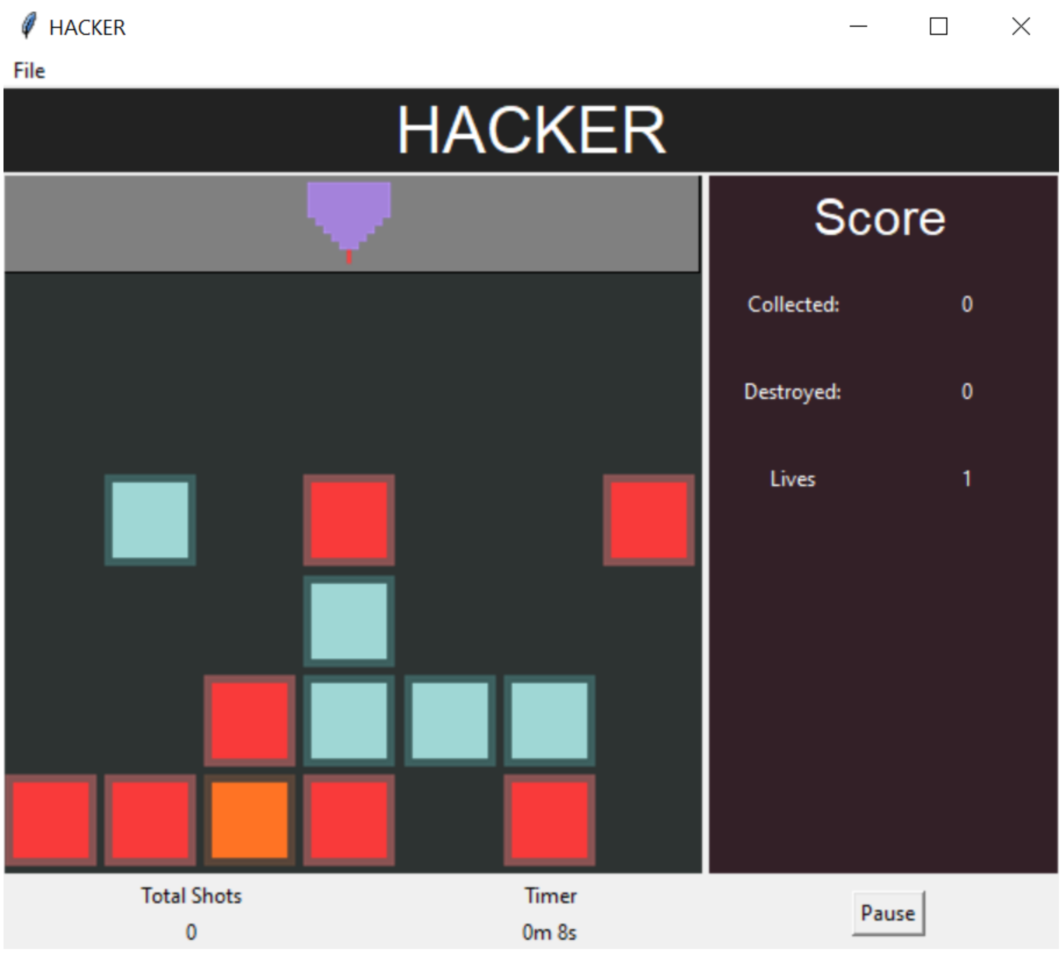

Figure 5: Complete Masters game

8 Assessment and Marking Criteria

8.1 Marking Breakdown

Your total grade for this assessment piece will be a combination of your functionality and style marks. You are **not** required to attend an interview for this assignment.

8.2 Functionality Marking

Your program's functionality will be marked out of a total of 50 marks. As in assignment 0, your assignment will be put through a series of tests and your functionality mark will be proportional to the number of tests you pass. You will be given a *subset* of the functionality tests before the due date for the assignment.

Your assignment will be tested on the functionality of gameplay features. The automated tests will play the game and attempt to identify components of the game, how these components operate will then be tested. Well before submission, run the functionality tests to ensure components of your application can be identified If the autograder is unable to identify components, you will be marked accordingly, even if your assignment is functional. The tests provided prior to submission will help you ensure that all components can be identified by the autograder.

You need to perform your *own* testing of your program to make sure that it meets *all* specifications given in the assignment. Only relying on the provided tests is likely to result in your program failing in some cases and you losing some functionality marks.

Your program must run in the Python interpreter (the IDLE environment). Partial solutions will be marked, but if there are errors in your code that cause the interpreter to fail to execute your program, you will get zero for functionality marks. If there is a part of your code that causes the interpreter to fail, comment out the code so that the remainder can run. Your program must run using the Python 3.9 interpreter. If it runs in another environment (e.g. Python 3.8 or PyCharm) but not in the Python 3.9 interpreter, you will get zero for the functionality mark.

8.3 Style Marking

The style of your assignment will be assessed by a tutor. Style will be marked according to the style rubric provided with the assignment. The style mark will also be out of 50.

8.4 Assignment Submission

This assignment follows the assignment submission policy as assignment 0. Please refer to the assignment 0 task sheet.

You must submit your assignment as a single Python file called `a3.py` (use this name – all lower case), and *nothing else*. Your submission will be automatically run to determine the functionality mark. If you submit a file with a *different name*, the tests will *fail* and you will get *zero* for functionality. Do *not* submit the `a3_support.py` file, or any other files. Do *not* submit any sort of archive file (e.g. zip, rar, 7z, etc.).

8.5 Plagiarism

This assignment follows the same plagiarism policy is as per assignment 0. Please refer to the assignment 0 task sheet.