

Installation and Maintenance Guide

User Manual

For a clear tutorial of how to play the game, what the goal is and the core mechanics please watch the video from the link: [Tutorial](#). Alternatively, you can install the game, run it and press the “Tutorial” button on the first screen when opening the game.

<https://github.com/jrgreenway/SE-Team-Gold/blob/main/assets/Avatar%20Game%20Tutorial.mp4>

Installation Guide

In order to be able to run and play the Avatar game, users need to have [Python 3](#) installed on their machines. You can check this by following the steps below:

If On Windows:

1. Press the **Win** key and type **cmd** in the search box. Hit **Enter**.
2. Type the command: **python -version** and hit enter.
3. The output should be something like: **Python 3.x.x**

If On MacOS / Linux:

1. Open the **Terminal**.
2. Type the command: **python -version** and hit enter.
3. The output should be something like: **Python 3.x.x**

Downloading the Code

Once **Python 3** is installed on the users machines, the user should go to the following link [Avatar Game](#) and download the code. If you are familiar with **Git** you can also clone the repository but if not simply download it as a **zip**.

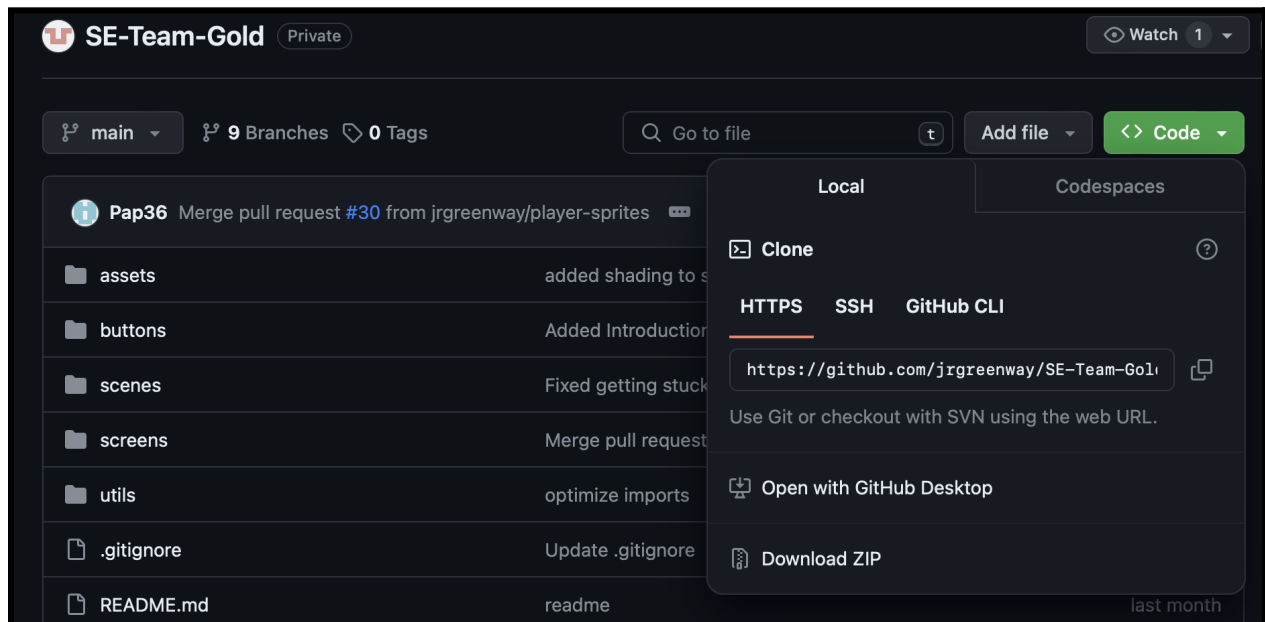


Fig 1. Download code as ZIP

Save the archive on your machine and extract its contents into a folder.

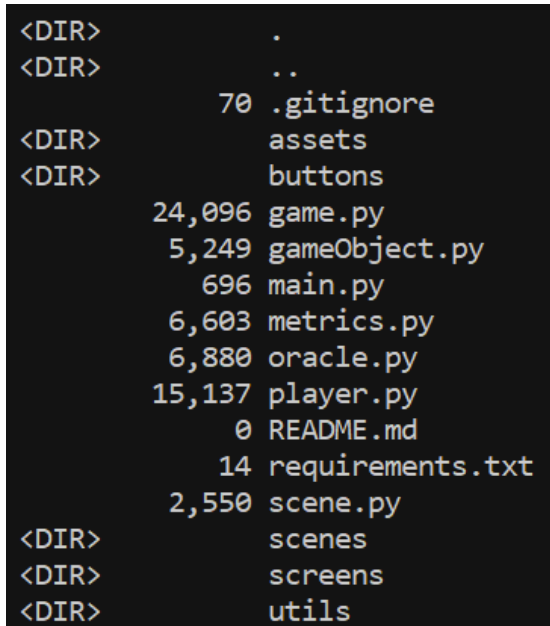
Running the Code

The first step is to navigate to the folder's location in Command Prompt for Windows or Terminal for MacOS / Linux.

If on Windows

1. Open **Command Prompt**.
2. Type the command **cd [folderPath]** to navigate to the extracted folder.
For example **cd C:/Users/John/Downloads/AvatarGame**
3. Type the command **dir** and hit **Enter**.

You should see something like this:



```
<DIR>      .
<DIR>      ..
          70 .gitignore
<DIR>      assets
<DIR>      buttons
        24,096 game.py
        5,249 gameObject.py
         696 main.py
        6,603 metrics.py
        6,880 oracle.py
       15,137 player.py
           0 README.md
          14 requirements.txt
       2,550 scene.py
<DIR>      scenes
<DIR>      screens
<DIR>      utils
```

If on MacOS / Linux

1. Open **Terminal**.
2. Type the command **cd [folderPath]** to navigate to the extracted folder.
For example **cd C:/Users/John/Downloads/AvatarGame**
3. Type the command **ls** and hit **Enter**.

You should a display like this:

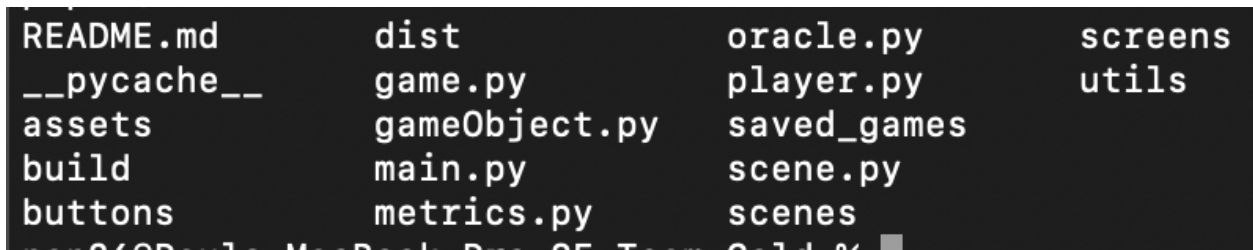


Fig 3. List of files in game folder - MacOS/Linux

Ignore `__pycache__` but make sure that all the other files or folders appear. You should also see a **requirements.txt** file as well.

Installing requirements

From the current **Command Prompt / Terminal** type the following command:

`pip install -r requirements.txt`

This will install all the required packages to help you run the game.

The user can now run the code and play the game by typing in the following command:

`python main.py`

Maintenance Guide

For the development of this game, **VS Code** has been used as the primary **IDE**. Despite this, any **IDE** of your choice is suitable for maintaining this code. Firstly, it is important to understand the code structure before we discuss how other implementations would happen. This is explained in the table below.

Note that indentation of a file or folder in the table means that this is a sub-file or subfolder of its parent. For example:

- **assets**
 - **animations**
- **screens**

Shows **animations** is a child of **assets** but **screens** is independent of it.

Also, some files in the table will have the symbol * at the beginning. This means that the entry in the table is a general template for all files resembling it.

The general layout of the repository is:

File / Folder	Brief description of its purpose
• assets	Folder with all assets used in the game.

• animations	Folder with assets used for animating the character whilst moving.
• E_F	Folder with assets used for animating the female character when moving east.
• E0.png	First sprite of the animation.
• E1.png	Second sprite of the animation.
• E2.png	Third sprite of the animation.
• E_M	Folder with assets used for animating the male character when moving east.
• E0.png	First sprite of the animation.
• E1.png	Second sprite of the animation.
• E2.png	Third sprite of the animation.
• N_F	Folder with assets used for animating the female character when moving north.
• N0.png	First sprite of the animation.
• N1.png	Second sprite of the animation.
• N2.png	Third sprite of the animation.
• N_M	Folder with assets used for animating the male character when moving north.
• N0.png	First sprite of the animation.
• N1.png	Second sprite of the animation.
• N2.png	Third sprite of the animation.
• S_F	Folder with assets used for animating the female character when moving south.
• S0.png	First sprite of the animation.
• S1.png	Second sprite of the animation.
• S2.png	Third sprite of the animation.
• S_M	Folder with assets used for animating the male character when moving south.
• S0.png	First sprite of the animation.

<ul style="list-style-type: none"> • S1.png 	Second sprite of the animation.
<ul style="list-style-type: none"> • S2.png 	Third sprite of the animation.
<ul style="list-style-type: none"> • W_F 	Folder with assets used for animating the female character when moving west.
<ul style="list-style-type: none"> • W0.png 	First sprite of the animation.
<ul style="list-style-type: none"> • W1.png 	Second sprite of the animation.
<ul style="list-style-type: none"> • W2.png 	Third sprite of the animation.
<ul style="list-style-type: none"> • W_M 	Folder with assets used for animating the male character when moving west.
<ul style="list-style-type: none"> • W0.png 	First sprite of the animation.
<ul style="list-style-type: none"> • W1.png 	Second sprite of the animation.
<ul style="list-style-type: none"> • W2.png 	Third sprite of the animation.
<ul style="list-style-type: none"> • audio 	Folder with assets for audio effects.
<ul style="list-style-type: none"> • soundtrack.mp3 	The background track of the game.
<ul style="list-style-type: none"> • *objectAsset.png 	Png asset file for a certain object displayed in the game. For example Oven.png .
<ul style="list-style-type: none"> • assetsConstants.py 	<p>Defines constants for important asset paths in order to avoid hard coding them wherever needed.</p> <p>Also contains the defined locations which appear on the map when interacting with a door.</p>
<ul style="list-style-type: none"> • buttons 	Folder relating to all buttons involved in the game.
<ul style="list-style-type: none"> • button.py 	<p>Custom implementation of a button which has a position on the screen and an action which triggers when clicked.</p> <p>Also, given the position of the mouse click, the button returns a boolean of whether the click happened on top of it or not.</p>
<ul style="list-style-type: none"> • buttonCallbacks.py 	File containing all the callbacks used in the game. Used to assign callbacks to buttons.
<ul style="list-style-type: none"> • saved_games 	Folder in which the state of the game is being

	saved either automatically or by the user.
<ul style="list-style-type: none"> • *Name, Day, Time.json 	A saved game whose format follows the fileName. It stores the state of the game as well as the state of the character in a json file.
<ul style="list-style-type: none"> • scenes 	Folder relating to storing all scenes and drawing them.
<ul style="list-style-type: none"> • sceneDrawer.py 	File which loads a given scene from the json file and draws it and its objects accordingly on the screen.
<ul style="list-style-type: none"> • scenes.json 	Json file containing all scenes in the game including the objects present in each of them and each object's effects on the character / game state.
<ul style="list-style-type: none"> • screens 	Folder containing all screens displayed in the game.
<ul style="list-style-type: none"> • *someScreen.py 	File which displays a certain screen and returns all buttons present on that screen at that time.
<ul style="list-style-type: none"> • utils 	Folder with utility functions for the game.
<ul style="list-style-type: none"> • gameLoader.py 	File which loads the state of the game and character from a json file in the saved_games folder.
<ul style="list-style-type: none"> • gameSaver.py 	File which saves the state of the game and character to a json file in the saved_games folder.
<ul style="list-style-type: none"> • game.py 	The main game class and its logic.
<ul style="list-style-type: none"> • gameObject.py 	A class representing an object from the game. Objects are defined in the construction of a scene in the scenes.json file and can have effects on either the character or the game's state. Objects can be interactable and colidable or not.
<ul style="list-style-type: none"> • main.py 	Main file which starts the game loop.
<ul style="list-style-type: none"> • metrics.py 	A class representing the player's current state represented as metrics (health, happiness, time, money).
<ul style="list-style-type: none"> • oracle.py 	The class represents the Oracle which

	provides guidance when needed. It can be triggered by clicking on its corresponding icon, through which appears a list of questions and corresponding answers the player can choose from.
<ul style="list-style-type: none"> player.py 	The player class which handles moving the player on the screen, animating its movement as well as interacting with objects.
<ul style="list-style-type: none"> requirements.txt 	Text file with all packages required by the code and its minimum versions.
<ul style="list-style-type: none"> scene.py 	A class which represents a scene to be drawn on the screen alongside its objects.

Understanding the nomenclature and the general flow of the game

To avoid confusion, it is important to understand the difference between a scene and a screen. A **Scene** is a physical location in which the player can exist and where the user can interact with objects. Currently, the game has **5** scenes: **Home**, **Office**, **Park**, **Gym** and **Tax Office**. All of these are drawn on the **Game Screen**. Note that there can be **one scene at most** drawn at any time.

On the other hand, a **Screen** is similar to a canvas and represents any other visual display apart from the scenes above. For example, the **Welcome Screen** displays the first buttons the user can press when starting the game. Similarly, the **Oracle Screen** will present a list of buttons with questions that the player can ask. It is important to mention that **Screens** can be drawn on top of each other. For example, the **Pause Screen** is drawn on top of the **Game Screen** every frame with 50% opacity. This creates the overlay effect and also displays the character in its current **Scene** behind the currently active **Pause Screen**.

Buttons and their callbacks

Every time a screen is drawn, it returns a list of buttons which are currently displayed on it.

```
def draw_oracle_question_screen(
    screen: pygame.Surface,
    questions: list[str],
    questionCB: Callable,
    backCB: Callable
) -> list[Button]:
```

Note how for the oracle screen, only one callback is passed for the questions, even though there are multiple questions. This is because specific parameters for each callback are defined and passed in the **game.py** file. Depending on the **currentScreen** there are always two

methods of the **Game** class which get triggered every frame - **draw** and **handleEvents** - both specific to the **currentScreen**.

```
def drawQuestionScreen(self, events) -> None:
    ''' Game.drawQuestionScreen(events) -> None
    Draws the question screen and handles mouse clicks on the buttons
    '''

    draw_game_screen(self.screen, self.currentScene)
    self.player.draw(self.currentDay)
    buttons = draw_oracle_question_screen(
        self.screen,
        self.oracle.getQuestions(),
        self.buttonCBs['clickOracleQuestion'],
        self.buttonCBs['back']
    )
    self.handleQuestionScreenEvents(events, buttons)

def handleQuestionScreenEvents(self, events, buttons) -> None:
    ''' Game.handleQuestionScreenEvents(events, buttons) -> None
    Handles mouse clicks on the question screen
    '''
    for event in events:
        if event.type == pygame.MOUSEBUTTONDOWN:
            clicked = [button for button in buttons if button.rect.collidepoint(event.pos)]
            index = buttons.index(clicked[0]) if len(clicked) > 0 else -1
            if len(clicked) > 0:
                self.running, self.currentScreen = clicked[0].onClick(
                    game=self,
                    currentScreen=self.currentScreen,
                    oracle=self.oracle,
                    question=self.oracle.getQuestions()[index] if index < len(self.oracle.getQuestions()) else None
                )
```

Fig 5. Functions handling the Oracle Question Screen

Firstly, the screen is drawn and buttons are passed forward as a parameter so that the mouse events can be handled. Depending on which screen we are on and which button is pressed, different parameters can be passed to the **onClick()** method which are then retrieved as **key value based arguments (kwargs)**.

```
def clickOracleQuestionCB(**kwargs) -> tuple[bool, str]:
    kwargs['oracle'].setQuestion(kwargs['question'])
    return True, ORACLE_ANSWER_SCREEN
```

Moving the player

When moving the player a temporary position is computed depending on the keys which are currently being pressed. Furthermore, the objects in the current scene are passed as arguments to the **move** method of the **Player** class. This enables collision checking between the temporary position and the objects. Should a collision happen, then the position is not updated.

Interacting with objects

Similar to moving the player, every frame in the **Game Screen** checks if the current player's position is close enough to an object for an interaction to happen. If this is the case, then a pop-up is displayed in the **Player** class which is triggered by the **interact** method. One important thing to mention here is that some objects can trigger new screens to appear. For

example, a **Bed** object would have the **openMap** attribute set to true, in which case the **return** statement from **interact()** would return true. This in turn triggers the transition from the current **Game Screen** to **Map Screen**.

```
return object.getOpenMap() if canInteract else None
```

Return statement from Player.interact()

```
showMapScreen = self.player.interact(self.holdingKeys, self.giveInteractable())
if showMapScreen:
    self.currentScreen = MAP_SCREEN
    return
```

Transition to Map Screen from Game Screen

Scenes

Scenes are all defined in **scenes.json** and each scene has the following attributes:

- **id:** integer
- **name:** string
- **texture:** string
- **objects:** list

These are all mostly intuitive. Note that the texture is a string representing the **path** to the **asset** of a **tile**. The screen size can be split in a grid of **35** tiles (7 x 5). Thus the background of the scene is the repeated asset found at the path specified by the **texture** attribute.

The objects in the list all have the following attributes (note that not all of them are mandatory):

- **id:** integer
- **texture:** string
- **position-relative:** {
 - **direction:** string
 - **relativeTo:** integer}
- **position-absolute:** {
 - **x:** integer
 - **y:** integer}
- **interactive:** boolean (default false)
- **isCollidable:** boolean (default true)
- **happiness-effect:** integer (default 0)
- **health-effect:** integer (default 0)
- **time-effect:** integer (default 0)
- **money-effect:** integer (default 0)
- **openMap:** boolean (default false)
- **next-day:** boolean (default false)

Each object is equivalent to 1 tile, so the texture the path points to will be drawn on that specific tile. Either **position-relative** or **position-absolute** must be specified. The first one allows the object to be positioned relative to another object (identified by its **id**) in the specified **direction** (**E, S, N, W**). On the other hand, **position-absolute** must specify the exact **x, y** coordinates where the top left corner of the tile is. Note that **0, 0** represents the top left coordinate of the screen.

Interactive is a boolean representing whether the object can be interacted with. Similarly, **isCollidable** defines if the player can walk over the object or not. Following these are the effects an interaction of the object has on the player's metrics. Finally, the two attributes **openMap** and **next-day** trigger new screen transitions, namely, **Map Screen** and **Next Day Screen**.

Adding a new Scene

It is not enough for a player to define a new scene in the **scenes.json** file. **Scenes** change based on the user's interaction with the **Map Screen**. Namely, when a user clicks on one of the displayed locations. As a result of this, we also need to add the new scene in the list of locations displayed on the **Map Screen**. This is done in **assetsConstants.py**:

```
LOCATIONS = {  
    "Home": (0, 80, 140, "assets/home.png"),  
    "Office": (1, 300, 200, "assets/office.png"),  
    "Park": (2, 600, 250, "assets/park.png"),  
}
```

Locations is a dictionary of the following format:

“Scene Name”: (id, x, y, path)

The **id** is defined in the **scenes.json** file for the new scene. **x, y** are the coordinates of where on the screen the sprite of this new location to be drawn on the **Map Screen**. Finally, the **path** points to the asset drawn on the **Map Screen** for this location.