COURSEWORK 2:

Product Documentation

Team Purple:

Ahmed Ahmadu, John Chan, Aditya Gaosindhe, Oliver Hamler, Zach Howard, Maximilian Mekiska, Hugh Surdeau, Yiming Yuan

MSc Computer Science
University of Bath

26/10/2020

Contents

List of Figures

1 Installation Guide

1.1 Windows, Linux and macOS Installation guide for developers

1.2 System Requirements

1.2.1 Setting up your development environment

Install Python

1. There are various ways to install Python on your machine. This guide demonstrates one way, so for alternative solutions, please refer to: https://docs.python.org/3/using/windows.html

2. Navigate to https://python.org/downloads

3. Download the latest Python version by clicking the yellow button labelled "Download Python". This library requires Python 3.8 or higher. However, since 09/10/2020 a new version of the arcade library got released which prevents Linux and macOS user from using a Python version higher than 3.8.

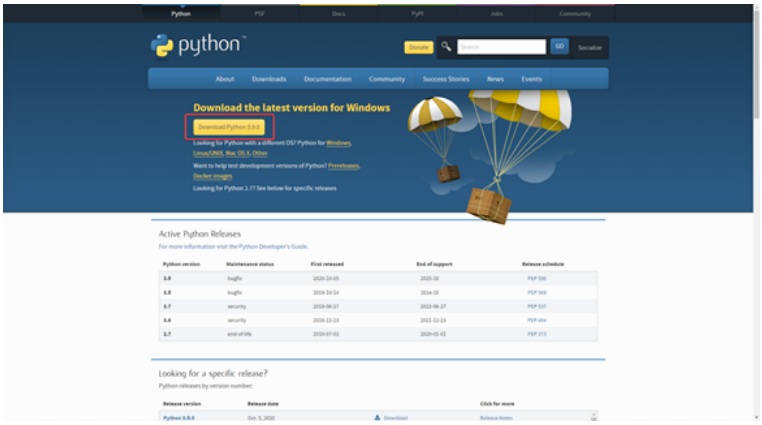

Figure 1: Python.org download

4. Make sure to check the box at the bottom labelled "Add Python to PATH" at the bottom before pressing "Install Now" so that you can run Python from the Windows command line.

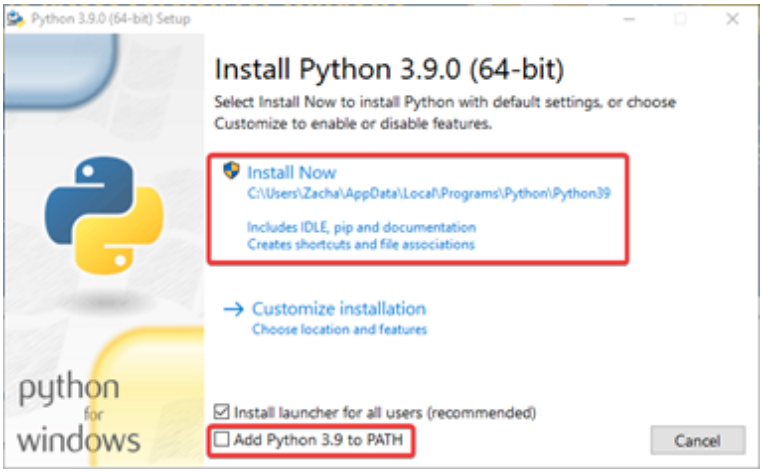

Figure 2: Add Python 3.9 to PATH

Install Git

1. Git installer can be downloaded at: https://git-scm.com/downloads

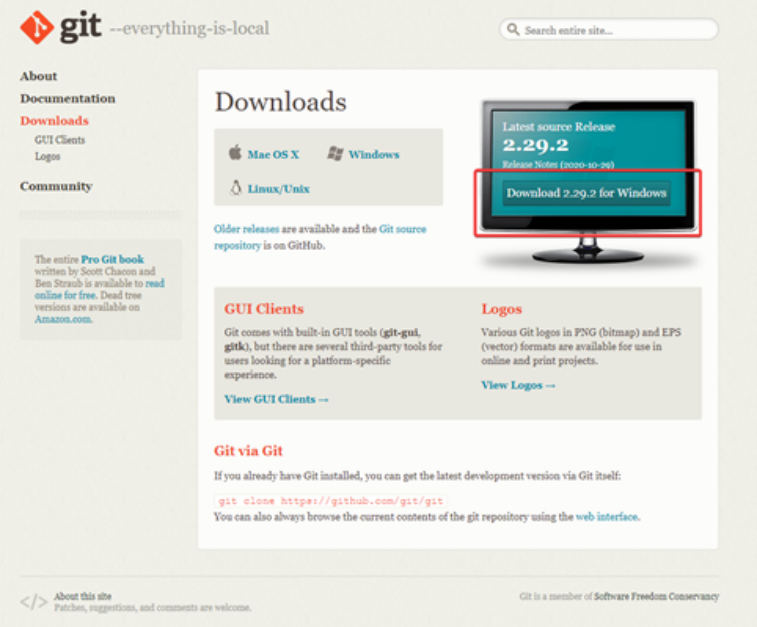

Figure 3: Git download

2. Navigate through the installer until installation is completed.

1.3 Setting up your virtual environment

Open the Windows command line and navigate to the directory in which you would like to install the repository.

```
cd path/to/directory
```

Clone the repository

```
git clone git@github.bath.ac.uk:hs706/DrillDungeonGame.git
```

Navigate to the created directory.

```
cd DrillDungeonGame
```

Make sure pip is up to date.

Windows:

```
py -m pip install --upgrade pip
```

Linux or macOS:

```
python3 -m pip install --user --upgrade pip
```

Install the virtual environment package to your base Python installation

Windows:

```
py -m pip install --user virtualenv
```

Linux or macOS:

```
python3 -m pip install --user virtualenv
```

Create a virtual environment

Windows:

```
py -m virtualenv -p py .venv
```

Linux or macOS:

```
python3 -m virtualenv .venv
```

Activating your virtual environment

Windows:

```
.\.venv\Scripts\activate
```

Linux or macOS:

```
source .venv/bin/activate
```

Installing requirements in your virtual environment

Windows:

```
py -m pip install -r requirements.txt
```

Linux or macOS:

```
python3 -m pip install -r requirements.txt
```

Running the game

Windows:

```
py main.py
```

Linux or macOS:

```
python3 main.py
```

Running the tests

Windows, Linux or macOS:

```
pytest tests/
```

1.4 Windows, Linux and macOS Installation guide for users

To install and play the Drill Dungeon Game, please download the .exe file either from the official Drill Dungeon Game website or the GitHub repository. Please make sure that your computer has a working version of Python 3.8.5 or higher installed. For the installation of Python, please refer to the explanation given above at 1.2.1 ”Install Python”.

2 User Manual

2.1 Introduction

This guide will take you through every step necessary to start up and enjoy the Drill Dungeon Game. We hope that you will enjoy your adventures through the depths of an exciting but dangerous dungeons.

2.2 The Objectives

Before diving into the game, it is important to highlight the objectives of your journey. You are herewith appointed Commander of your own drill. Congratulations, Commander! Your target is to explore and uncover the secrets buried in the endless depths of a new discovered planet. But be aware, this will be no ordinary reconnaissance mission. Our scanners have picked up unknown structures deep below the surface. We are assuming that hostile life forms will be present as well.

However, there is also good news, Commander. Spectral analysis of the crust layers has shown that minerals such as gold and coal are present. This is excellent news, since your drill will be running on coal to keep you and your crew alive. Gold on the other hand will surely come in handy to buy new upgrades from local non-hostile life forms. Our scientists have ensured us that your drill features the newest attack laser turret and modern plasma shield generator. These will be sure to protect you and your crew during your descent into the unknown.

Your main object herewith is to explore and scout the depths of the novel planet and report back all gathered data. We expect you to reach the core of the planet and come back safely. Good luck, Commander! We and your crew are counting on you!

2.3 How to start the game

Make sure that you have followed the Installation Guide before continuing with this section. Now, please navigate to the Drill Dungeons Folder and start up the main.py python file:

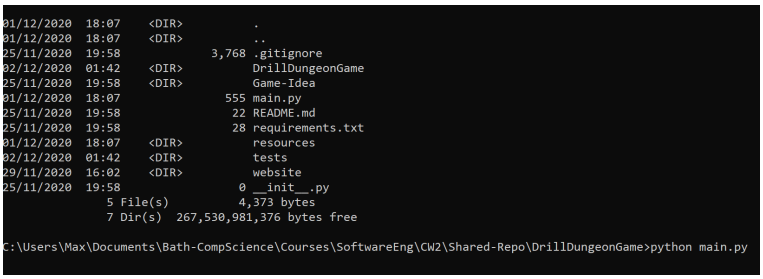

Figure 4: Start main.py

You should now see a new window opening up which will bring you directly to the Drill Dungeon Game Main Menu:

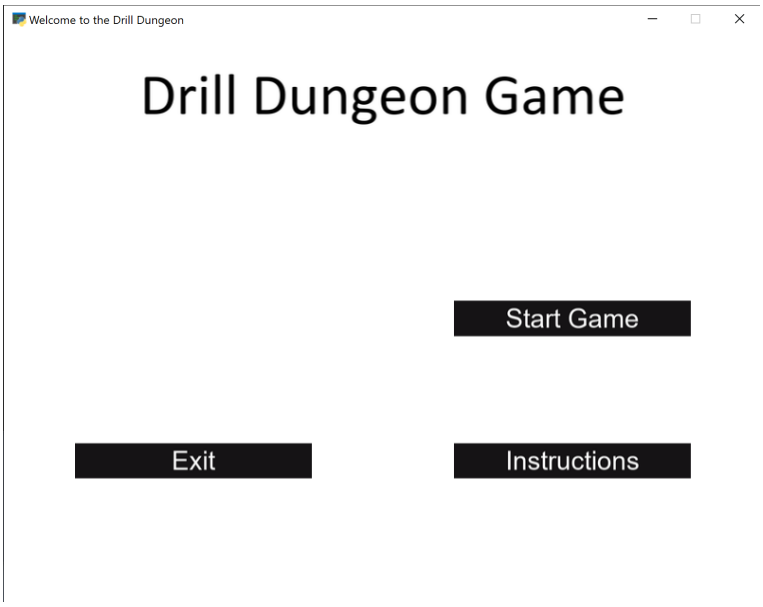

Figure 5: Main Menu

On the main menu, you are able to click on the instruction button to display a short summary of all basic controls, which we will shortly go over in detail. Furthermore, from the instruction menu you are able to click on another summary to display the main objectives of the game. If you wish to exit the game, just click on the Exit button.

Please continue by clicking on the ”Start Game” button. The game will now begin.

2.4 Drill Operators Manual

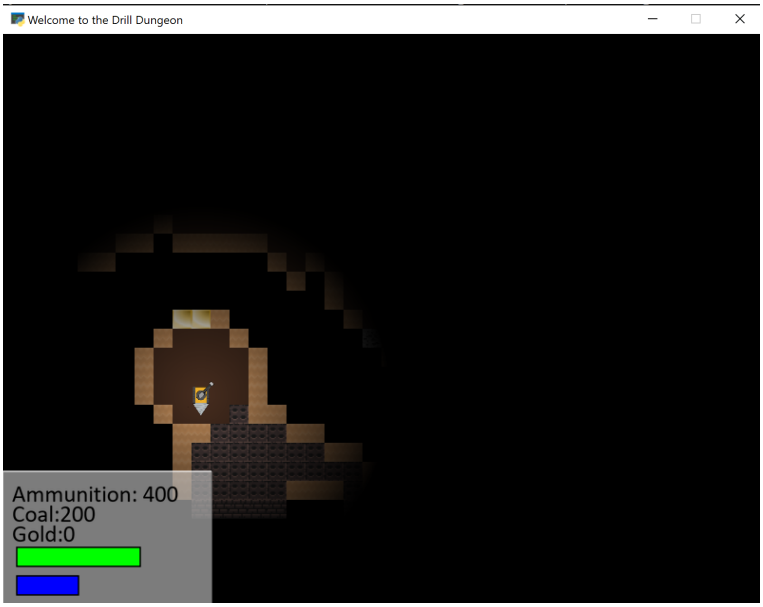

Figure 6: Start Game

Please have a look at the lower left corner. This is your HUD which will provide you with vital mission information and is the best metric with which to make tactical decisions during your adventure. The green bar represents your drill’s health. If the green bar is completely depleted your drill will explode and your mission failed:

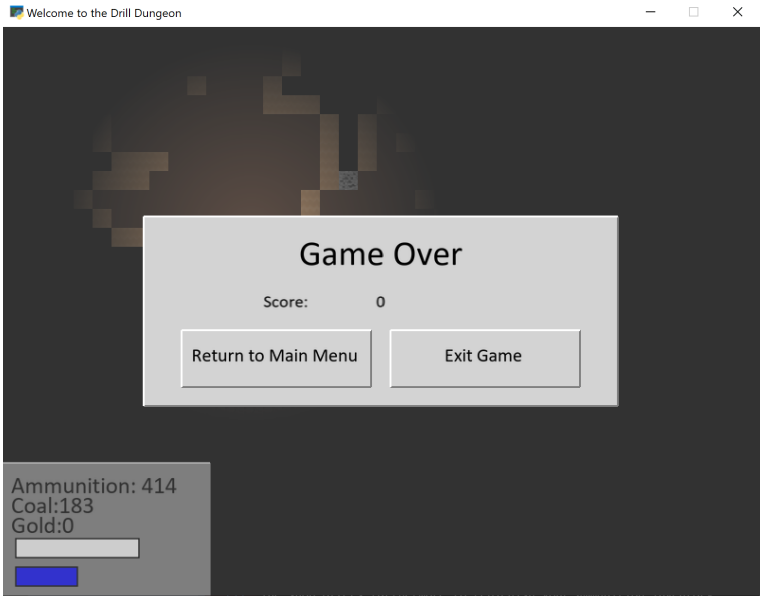

Figure 7: Game over menu

So keep an eye on this metric before deciding to engage any further enemies. Below the green bar you will find your energy level that will power your shield. The shield will protect your drill from damage - however it will not last forever. When depleted, the shield will disintegrate and leave you vulnerable to damage from enemies. But don't worry - your shield will recharge over time.

The values above health and the energy level will inform you about your drills inventory. Your inventory consists of three resources. Ammunition supplies your turret with bullets. Coal provides fuel for your drill. Gold can be used later in the game to purchase upgrades for your drill.

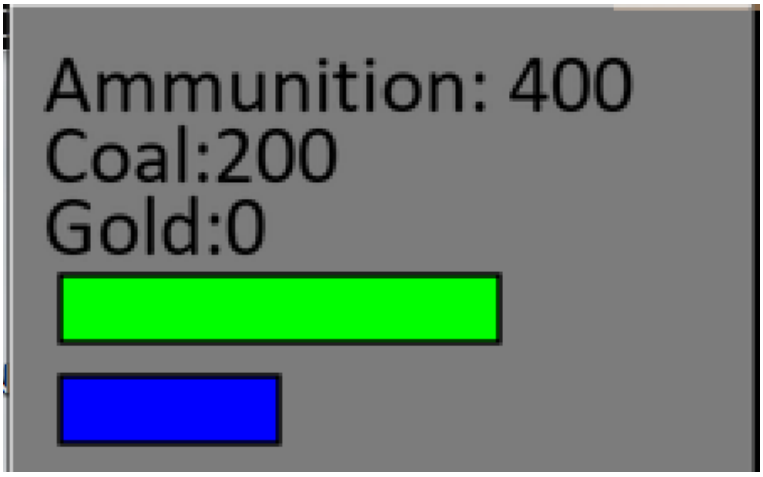

Figure 8: Player's HUD

Having familiarised yourself with the HUD, its now time to learn how to drive the drill. By pressing the keys W, A, S, D, you will be able to navigate the drill forward, left, backward and right. Your drill is also capable of drilling diagonally which can be done by pressing a pair of the aforementioned keys. As an example, if you wish to drill diagonally up to the right, press W and D simultaneously. As you start driving and exploring the dungeon, you will be able to drill through the dirt blocks to uncover new paths and elements.

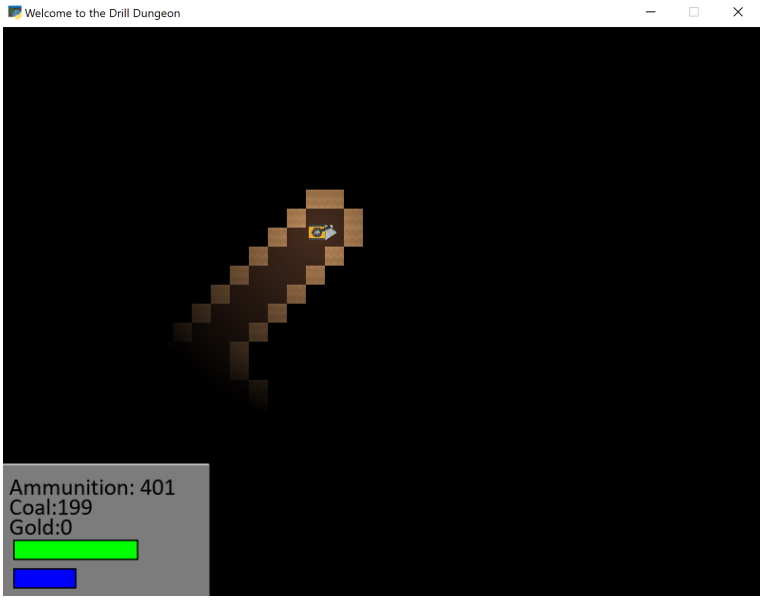

Figure 9: Navigating the Drill

Have you noticed something in the HUD?... That's correct! Navigating the drill through the dungeon slowly depletes your coal supply. So make sure to drill for new coal because if your coal supply is fully depleted, your drill will not move and it's Game Over. However, there is something else, if you look closely, ammunition has increased! That's good news! Driving will recharge not only the energy for your shield but also add bullets to your ammunition inventory.

Navigating through the dungeon will sooner or later lead to discoveries of resources. These resources are coal and gold. Coal, as explained before, will serve as your drill's fuel. Gold on the other hand will serve later as currency for new drill upgrades.

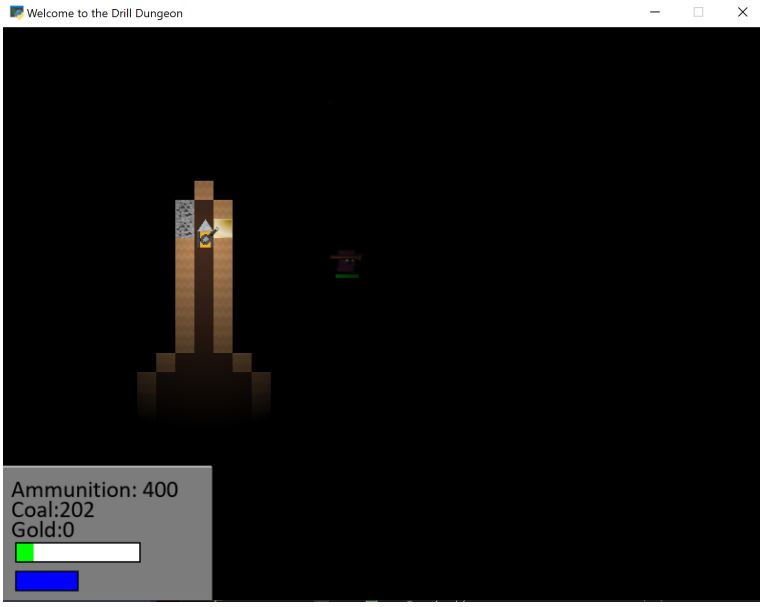

Figure 10: Resources

To collect these precious resources you simply need to drill through them. Next, you will need to learn how to defend yourself. If you are spotted by a hostile, it is essential for your survival to know how to use your energy turret. To fire the turret, you need to aim by pointing the mouse into the direction you want to fire and click the left mouse button to fire the turret. By default, the turret will fire a single shot

for each click on the left mouse button:

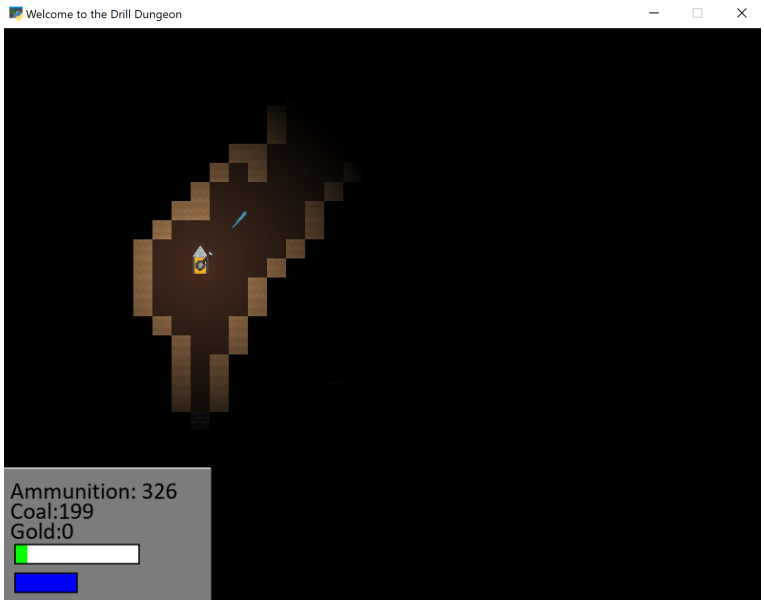

Figure 11: Single shot mode

For close quarters combat your turret is capable of changing the fire mode to buck shot. This is done by clicking the B key:

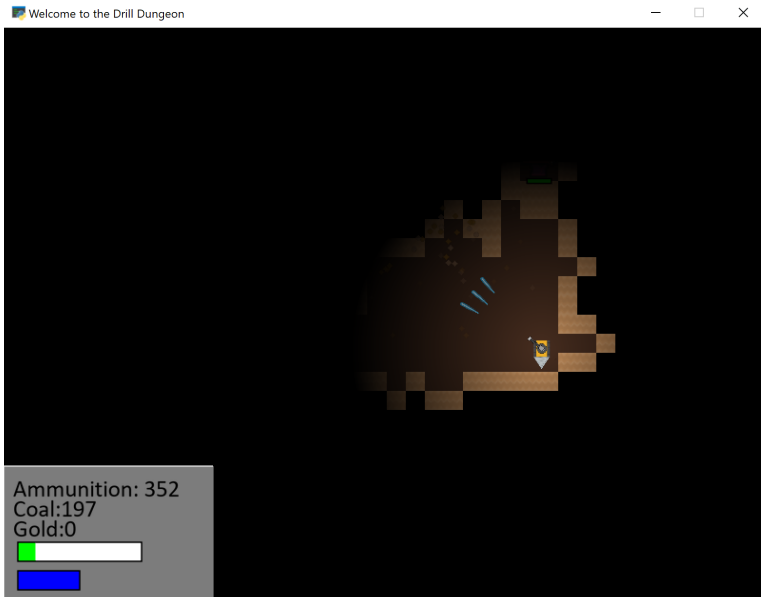

Figure 12: Buck shot

A word of caution, bullets will not only defend from enemies but also destroy valuable gold and coal blocks. Be careful where you shoot, you might find you have destroyed the very blocks you were fighting for!

Having learned the basics of shooting, now is the time to learn more about the enemies you will encounter during the game. Enemies will be hiding in the dark and might catch you by surprise, so be careful while exploring. However, sometimes you might see the enemy before it can see you. This will give you a strategic advantage. As a tip, before taking the decision to engage the enemy, have a look at your inventory, health and energy shield levels.

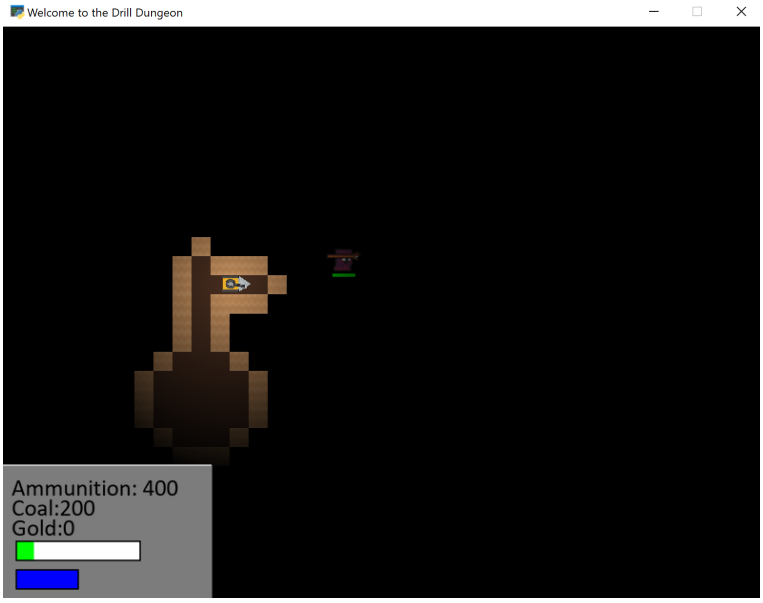

Figure 13: Enemy

As you can see in the above picture, the enemy hasn't detected us yet and we might decide to avoid combat because of our low health level. If you look closely, you will notice a similar health bar below the enemy. In combat you want to make sure that this health bar runs out faster than your own. In case it doesn't go as planned, it might be necessary to consider a tactical retreat. As a tip, if you need to retreat, use the shield to buy enough time to outrun the enemy. You can activate your shield by holding the right mouse button:

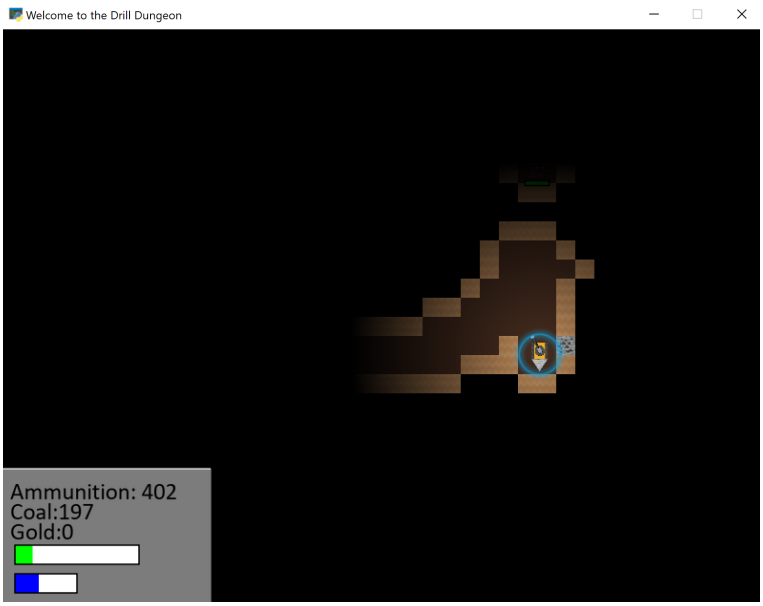

Figure 14: Shield

Having learned the basics of combat, let's have a look at the options you have to aid you in the quest of exploring the depths of the dungeons. During the game, you will be able to encounter friendly locals in the form of shops:

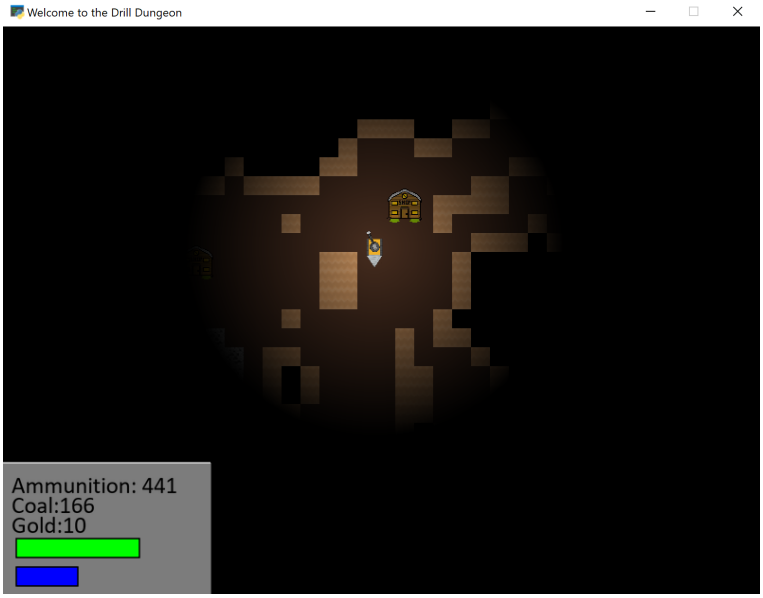

Figure 15: Shop

To interact with the shop, drive in its proximity and click on it with a left mouse click. A shop menu will open up and offer you a variety of purchasable upgrades:

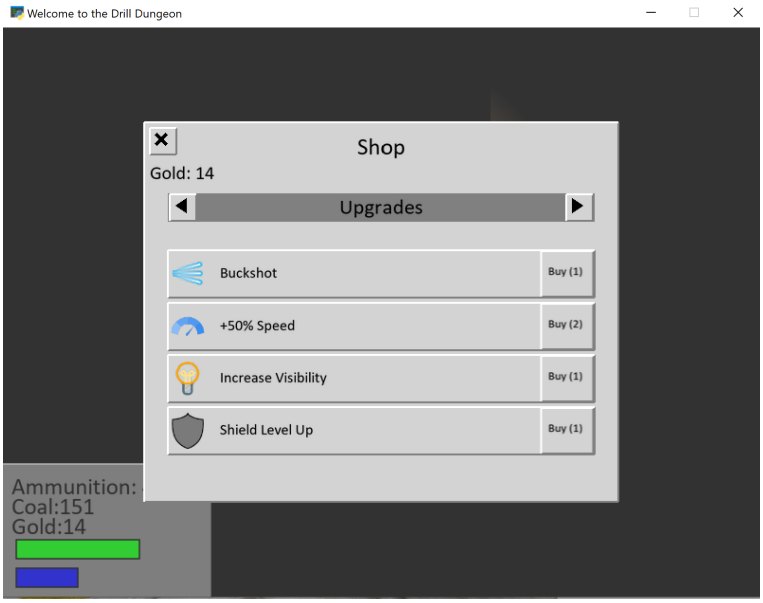

Figure 16: Shop menu upgrades

The shop also offers replenishment for your ammunition inventory:

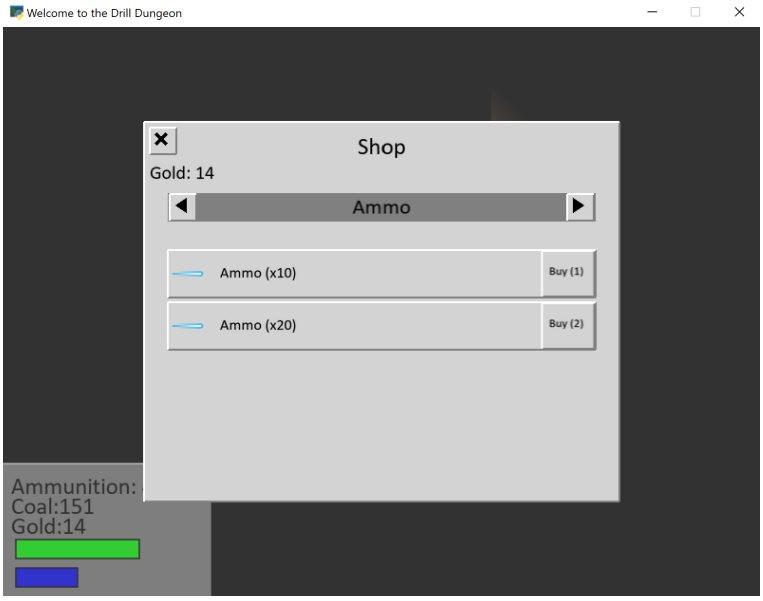

Figure 17: Shop menu ammunition

During your journeys you will encounter not only natural caves and resources but also enemy-built structures. These structures are usually made out of material that cannot be destroyed by shooting at it or drilling through it. Have a look at the example below:

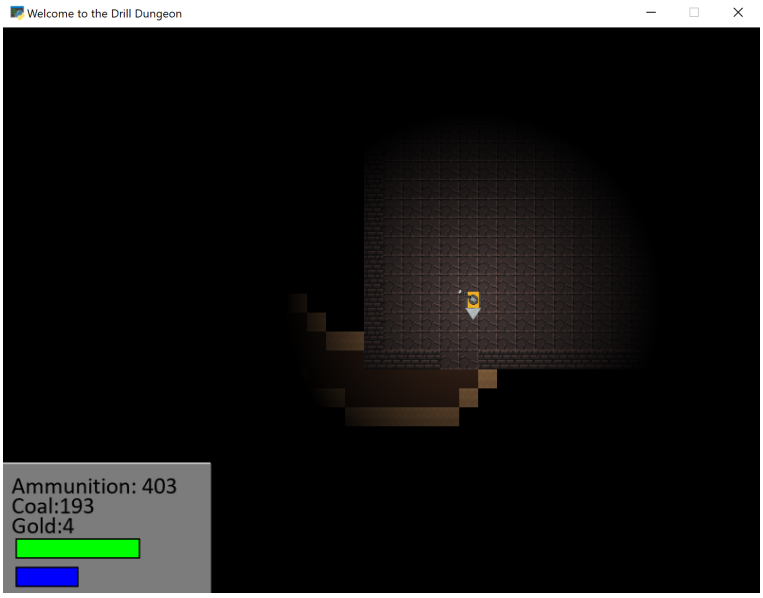

Figure 18: Dungeon example

The walls of this artificial dungeon are made out of a material you will not be able to destroy. Make sure to keep this in mind when facing enemies. It might come in handy or pose additional challenges.

Next, to advance further to the core of the planet you will need to find a drill-friendly surface:

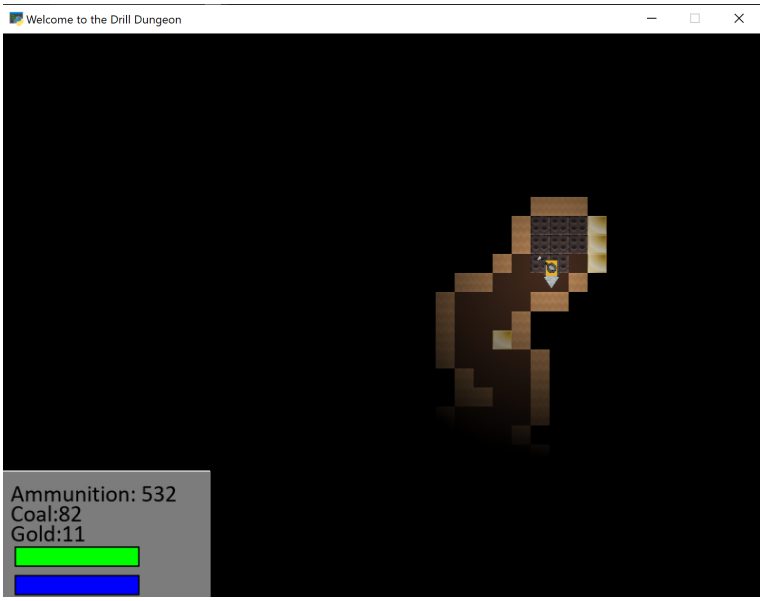

Figure 19: Tiles to drill down/up

Once you have found these specific tiles, you will be able to drill one more level down by pressing the T key. But drilling down will have a price. You will lose 30 coal and conditions on the next level will be more harsh. Your view will decrease and the speed of the drill will be reduced as well. Make sure to visit a shop and upgrade your drill before drilling into the next layer.

Furthermore, if you decide to return back into the prior layer you will be able to do so by searching for drill-friendly tiles and pressing the U key. Keep in mind that resources on this level will be depleted from your drilling.

Lastly, if you need a break or wish to exit the game, you can pause the game by pressing the ESC key. This will open up the pause menu:

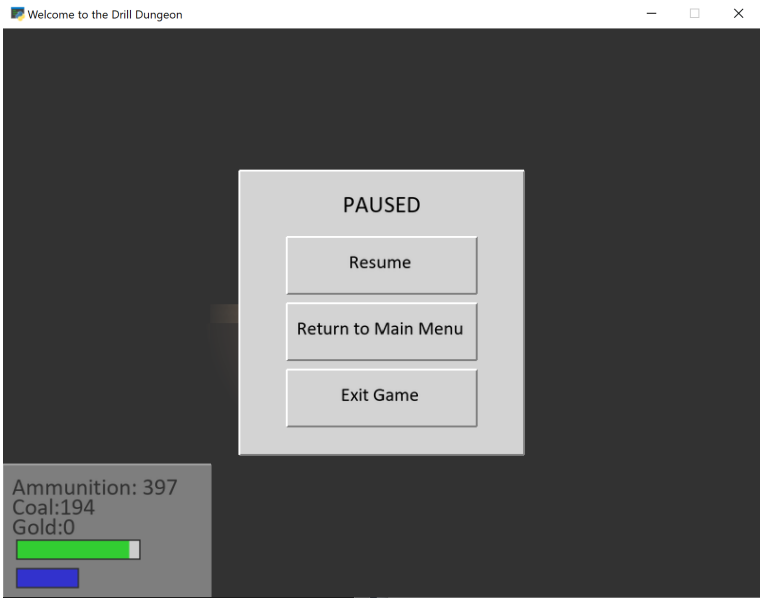

Figure 20: Pause Menu

Best of luck on your adventures, Commander! Defeat those aliens and bring back the gold!

3 Maintenance Guide

3.1 Overview

Drill Dungeon is made up of three major components: blocks, entities and menus. Blocks are the sprites that make up the map, and many of them interact in similar ways with the player. Entities consist of the player itself and any enemies that the player encounters. Menus include the start game menu, the pause menu and the shop menu.

3.1.1 Map Generation

All files pertaining to map generation are contained in the map directory. The visual map seen on screen when the game starts is generated in several stages. First, the configuration of the blocks (their types, where they are located) is generated in the dungeon_generator.py file. Next, blocks adjacent to air blocks are loaded into a container for all active sprites on the map, held in the block_map.py file. Finally, these blocks are rendered on the screen in the level.py file, which handles the visualisation of the map.

The dungeon_generator.py file contains the MapLayer class, which generates randomised levels for the drill when the game is loaded or the player drills down. This is initially done through the creation of a map layer matrix: a matrix of all the block types and their position relative to one another in a grid. First, the map layer matrix is created and completely filled with dirt blocks. The width and height of the map layer matrix (in blocks) is determined by the constructor method of the MapLayer class, with default values of 128 for each. From this base template, different components of the map are added to the matrix by different methods.

Features that are generated in clumps randomly throughout the map (coal, gold, drillable zones and cave dungeons) are generated through their respective methods (with names of the form generate_gold()). This is done by selecting a random block in the matrix and changing that to the desired type. A randomised patch size is also created using a Poisson distribution. Then, a random walk direction is taken from update_dungeon_coords() and get_walk_direction(), which assign it a random block adjacent to the current block and changes that to the desired type too. This step is repeated until a patch of the desired size has been generated.

After this is complete, border walls are added. The first and last rows of the matrix are converted from all dirt to all undrillable walls, followed by the first and last blocks of each row in between. Finally, one of the map's prefabricated dungeons is loaded row-by-row using the generate_advanced_dungeons() method. With the map layer matrix generated, a final step creates a map layer configuration, which is identical to the matrix except each component is a tuple containing the block type and its X and Y coordinates.

The map layer configuration is converted into sprites and managed through the BlockGrid class in the block_grid.py file. This class takes the layer and loads all blocks that are adjacent to air blocks, so the majority of blocks are not rendered at any time. The configuration generated in the MapLayer class is used to load all the initial block sprites required, and manages the deletion and rendering of new blocks. New blocks are rendered using the break_block() method, which loads in any blocks adjacent to it that were not already rendered in the map layer configuration using the initialise_blocks_adjacent_to_air() method.

Blocks loaded from BlockGrid are stored in an instance of the SpriteContainer method, contained in the sprite_container.py file. This class is purely a wrapper for the many types of blocks that are used in the game. The class contains lists for groups of blocks that need to behave differently, as the arcade package used in creating Drill Dungeon requires sprite lists for collision detection. Each level is thus ultimately represented by a SpriteContainer, which is used in the levels.py file to generate the map.

3.1.2 Entities

The entity subdirectory, contained within the DrillDungeonGame package contains all classes responsible for creating, containing and moving sprites with logic. Currently, this includes the Drill class, as

well as all enemies and bullets. In the future this should also contain friendly AI. All entities inherit from the base class: Entity, or through the ChildEntity class which extends Entity. Examples of when the ChildEntity class should be used over Entity is if it forms any moving part for that base entity such as a turret, hand, or bullet. This is essential for bullets so that they do not collide with the entity that fired the bullet as it launches from the barrel.

To implement the simplest Entity possible, all you should need to do is sub-class Entity and pass several parameters to the super() method, namely a string containing a path to the sprite, the scale of that sprite as well as X and Y coordinates. The only step left to have this entity drawn to the screen is to call the update() function followed by the draw() function on that entity in each game loop iteration. Both of these functions also recursively update or draw all entity's that are children to parent that it is called upon. This is made possible through the get_all_children() function, and the opposite is possible by calling get_all_parents() on any given ChildEntity.

The current entity package structure allows for additional logic to be abstracted into mix-in classes, reducing the need for repeated code in different enemies that require identical logic. Some Mixin classes require direct instantiation within the entity's __init__() function, while others simply require inheriting to gain access to additional methods. Should the mix-in require instantiation, this can be possible by calling MixinName.__init__(self, *args, **kwargs). By passing the self-parameter to the mix-in, the mix-in can also access any attributes and methods in the entity class such as health, speed, inventory and so on. Similar to how all children entities are updated when calling the parent.update() method, this also calls the update() function in all mix-ins of every child to the entity it is called upon. This makes using these mix-in classes very low effort and means more time can be spent on developing new features for them. Current functionality which has seen this abstraction to mix-in classes include a PathFindingMixin, DiggingMixin, ShootingMixin and ControllableMixin.

Firstly, the path finding mix-in provides functionality to calculate a path to another given entity or specific position and maintain that path until it is complete or otherwise cancelled. This is done through a path_to_position() method which takes an X and Y coordinate as an argument as well as a SpriteList of blocking sprites and a boolean denoting whether diagonal movement is permitted. This method is also wrapped in a path_to_entity() method, which takes another Entity object as a parameter instead of an X, Y coordinate and extracts its position. Both functions update an attribute called 'path' that contains a list of coordinates to denoting the path to the coordinate requested to path-find to. Each time the mix-in is then updated, it checks to see if this list is populated and if so, pops the first item from the list and calls the move_towards() method, which updates the velocity of the entity to that position.

Next, the digging mix-in allows the entity to break certain blocks which it collides with. This is generally restricted to dirt and ores. The controllable mix-in allows the entity to listen and react to keyboard or mouse presses. Abstracting this movement logic from outside the Drill class provides the foundation required to implement controllable bullets or other controllable friendlies. This mix-in does not require instantiation and simply functions by calling the update method every game loop.

Lastly, the shooting mix-in can be inherited to allow that entity to shoot a projectile. This is made possible with very few functions, namely an aim() function to aim at a given x, y coordinate, as well as a pull_trigger() and release_trigger() function to fire at the position aimed at. The shoot() function can also be used to instantly fire a bullet, but note that this bypasses fire rate limit.

3.2 Running Unit Tests

By running the following command:

Windows

```
py -m unittest discover tests
```

Linux, macOS

```
python -m unittest discover tests
```

Unit test can be run. All unit tests are contained in the tests directory in the top level directory.

3.3 Debugging Features

To make it easier to on the code, debugging features are added. These features can be activated in-game by pressing specific keys. The debugging features can be found at the following location in the source code: `DrillDungeonGame/drill_dungeon_game.py`.

```
# DEBUGGING CONTROLS
elif self.keys_pressed['O']:
    self.vignette.increase_vision()

elif self.keys_pressed['L']:
    self.vignette.decrease_vision()

elif self.keys_pressed['K']:
    self.vignette.blind()

elif self.keys_pressed['SEMICOLON']:
    self.vignette.far_sight()

elif self.keys_pressed['M']:
    self.window.show_view(self.window.shop_view)
```

3.4 Extending the Code

3.4.1 Adding Additional Block Types

All map blocks are represented in a map layer matrix, which stores the type of block and its relative position. Block type is represented as a short string (for example, 'X' for dirt or ' ' for an air block). This map layer matrix is later loaded into a map layer configuration, which also stores the X and Y coordinates of the block, which is then loaded into the BlockGrid class, which manages the sprites displayed on the map.

In order to add additional block types, first a new string must be assigned to the block type. A method is then required in order to add the string representing the new block to the map layer matrix. How exactly this is to be implemented depends on the nature of the block and is thus up to the maintainer. As an example, gold and coal blocks are generated in random patches in the methods generate_coal() and generate_gold() in the MapLayer class. However it is implemented, make sure the method to load the blocks into the map layer matrix is called in the get_full_map_layer_configuration() method, as this is what is called when loading in the map layer to the game.

Once the block type string is loaded into the map layer configuration, a new block type has to be defined. The block.py file contains all the block classes, which extend the main block class. Create a new block class following the format of the other ones, ensuring the 'char' attribute is set to the same one that was loaded into the map layer matrix. File is the location of the image that the new block type will take as its sprite, while scale changes the size of the block. Make sure that the block will scale to 20x20 pixels. Finally, add the new block class to the _Block class at the end of the file, which allows for the block classes to be called.

The new block type then needs to be added to the BlockGrid class in the block_grid.py file. The exact implementation of this depends on what the intended behaviour of the block is. If the block is purely for visual purposes and never needs to interact with the drill, then it is classified as an air block. To add the new block type as an air block, simply append BLOCK.<NEWTYPE> to the if statement to the following for loop on line 102:

```
    def initialise_blocks_adjacent_to_air(self, sprites):
        for x in range(self.width):
            for y in range(self.height):
                block = self.blocks[x][y]
                if any(type(adjacent_block) in (BLOCK.AIR, BLOCK.DRILLDOWN,
                                                BLOCK.FLOOR) for adjacent_block in
                                                self._get_adjacent_blocks_to(block)):
                    if type(block) in (BLOCK.FLOOR, BLOCK.AIR):
                        self.air_blocks.append(block)
                    elif type(block) == BLOCK.DRILLDOWN:
                        self.air_blocks.append(block)
                        if block not in sprites.all_blocks_list:
                            self._add_block_to_lists(block, sprites)
                    else:
                        if block not in sprites.all_blocks_list:
                            self._add_block_to_lists(block, sprites)
```

This checks if the block being iterated over is meant to be an air block, and initialises it as such.

If the block needs to interact with the drill, then an elif statement needs to be appended to the _add_block_to_list() method in block_grid.py file:

```
    def _add_block_to_lists(self, block: Block, sprites) -> None:
        if type(block) == BLOCK.DIRT:
            sprites.destructible_blocks_list.append(block)
            sprites.all_blocks_list.append(block)

        elif type(block) == BLOCK.COAL:
            sprites.destructible_blocks_list.append(block)
            sprites.all_blocks_list.append(block)

        elif type(block) == BLOCK.GOLD:
            sprites.destructible_blocks_list.append(block)
            sprites.all_blocks_list.append(block)

        elif type(block) == BLOCK.SHOP:
            sprites.shop_list.append(block)
            sprites.indestructible_blocks_list.append(block)
            sprites.all_blocks_list.append(block)

        elif type(block) == BLOCK.BORDER:
            sprites.indestructible_blocks_list.append(block)
            sprites.all_blocks_list.append(block)
            sprites.border_wall_list.append(block)

        elif type(block) == BLOCK.WALL:
            sprites.indestructible_blocks_list.append(block)
            sprites.all_blocks_list.append(block)
            sprites.border_wall_list.append(block)

        elif type(block) == BLOCK.DRILLDOWN:
            sprites.drill_down_list.append(block)
            sprites.all_blocks_list.append(block)
```

```python
        else:
            raise ValueError(f'Incorrect block type: {type(block)}!')
```

Depending on if the block can be broken or not, it should be appended to the sprites' indestructible_block_list or the destructable_blocks_list. Either way, it should also be added to all_blocks_list.

If the block requires some sort of special interaction with the drill, it may require that a sprite list be appended to the SpriteContainer class. This list can then be called in other methods which will allow just that type of block to be checked for collision or other interactions. To add this list, simply extend the constructor method of the SpriteContainer class in the sprite_container.py file, adding a new sprite list as an argument and class attribute.

3.4.2 Adding Additional Prefabricated Dungeons

To add additional prefabricated dungeons please navigate to `DrillDungeonGame/map/prefab_dungeon_rooms.py`. Any dungeon added into this file will later on be displayed in the exact same format as defined here. Prefabricated dungeons are entered via a simple array:

```python
entrance_room_one =
[['W', 'W', 'W', 'W', 'F', 'F', 'W', 'W', 'W','W', 'W','W'],
['W', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'F','F', 'F','W'],
['W', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'F','F', 'F','W'],
['W', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'F','F', 'F','W'],
['W', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'F','F', 'F','W'],
['W', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'F','F', 'F','W'],
['W', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'F','F', 'F','W'],
['W', 'F', 'F', 'F', 'F', 'F', 'F', 'F', 'F','F', 'F','W'],
['W', 'W', 'W', 'W', 'F', 'F', 'W', 'W', 'W','W', 'W', 'W']]
```

In more detail, the letters in the matrix above stand each for a specific tile in the game. 'W' represents a wall and will be displayed as an indestructible wall when rendered in the game. The other letter, 'F', represents a simple floor tile which matches the style of the indestructible wall tiles, but behaves like an air block.

3.4.3 Automated Dungeon Generation

Automated dungeon generation is more of an open ended addition, and can be implemented in several ways. In theory, the only thing required is some way to alter the map layer matrix to add some floor and dungeon wall tiles, represented by characters 'F' and 'W' respectively.

Two methods were attempted for this iteration of the game, but were both shelved due to balancing difficulties. These can be re-added by the maintainer and improved if they so wish. The first method used the same basic principle of the generate_coal() and generate_gold() methods, instead adding floor blocks. Then, the map layer matrix should be iterated over, adding wall blocks to any blocks adjacent to floors which are not also floor tiles.

The other method which was attempted was constructing new dungeons from several prefabricated rooms. This is similar to the prefabricated method, but more open ended.

3.4.4 Extending the Entity class

The entity class may be extended to include extra features such as animations. There are 2 main default animations you can add to the entity which are the idle animations and moving animations. The idle animation loop when the character is not moving while the moving animations loop when character is in motion. To add any of these include a list of the images to loop through in the subclass initialisation. This list should then be passed when initialising the parent class Entity. The idle images should be passed

as idle_textures and the moving images should be passed as moving_textures. Lastly a time value would also need to be passed to the Entity initialisation as 'time_between_animation_texture_updates'. This determines the time it takes before switching the pictures.

More animations may be added but this would require overriding the update_animation method. To implement them, add the list with images as before but instead of calling it with the parent initialisation create an attribute for the textures. The images can be loaded into the texture list using 'load_mirrored_textures', which loads all images into the created attribute. The 'load_mirrored_textures' needs to be imported from `DrillDungeonGame/utility.py`. The update_animation method can then be adjusted to change the animations based on the certain conditions for the animation being added.

An Entity can have multiple children including a Turret object used by most enemies and the drill. The Entity class has a children attribute which is a list to store all ChildEntity's.

To add specific functionality to the class the update method can be overridden. When this is done ensure to call the parents update method. In this update specific functions such as checking line of sight, aiming, firing can be added.

The Enemy class is a subset of the Entity class. It adds functionality to the Entity class such as, a health bar displayed under the character, sounds for when they attack or are attacked, and initialised variables to be used for bot implementation such as '_has_line_of_sight_with_drill'.

To create an enemy in the game a subclass of the Enemy class should be made instead of the Entity. The subclass can then also inherit the mix-ins DiggingMixin and PathFindingMixin. The PathFindingMixin is necessary in eneabling them to find a route to the drill and the DiggingMixin enables them to remove dirt blocks in their way. To enable this the update function needs to be overridden to include code that checks if enemy has a line of sight with the drill and also code to follow, aim and fire at the drill.

3.4.5 Adding Additional Explosion Effects

Creating additional tailored explosion effects is recommended when new blocks, enemies or other objects are added to the game. To do so, please navigate to `DrillDungeonGame/particles/explosion.py` file. The file begins by defining a list of constants that control how the explosion will be rendered onto the screen. These constants are universally used by all explosion classes. If this behaviour is not desired for a particular explosion, it is recommended to create a new constant and manually add it into the specific particle class. As an example, this specific tailoring approach is applied to the explosion particles colours. Each currently available block in the game has its own explosion particle colour list. Please visit:`https://arcade.academy/arcade.color.html` to see the full library provided colour list.

The general structure of the file consists of a general Smoke class and multiple individual particle classes. The smoke class can be used by each particle class to cause a smoke effect upon a particle explosion. A good example of a particle class using the smoke class is the ParticleCoal class. In the update() method of the ParticleCoal class, a smoke object is created which in turn generates the smoke effect:

```
if random.random() <= SMOKE_CHANCE:
    smoke = Smoke(5)
    smoke.position = self.position
    self.my_list.append(smoke)
```

To generate a new particle explosion, follow the example of the other particle classes. If you wish to add the smoke effect, add the above code block into the newly created particle classes update() method.

3.4.6 Main Menu Modifications

Modifying or adding new elements to the main menu is possible by navigating to the following python file: `DrillDungeonGame/views.py`. The general structure of this file consists first of button

classes and second of view classes. First, button classes (inherent from arcade.gui.UIFlatButton) define the behaviour of what happens when a particular button is pressed. Second, View classes define (inherent from arcade.View) the general structure of the window and graphical representation. Each view class contains a setup() method that is used to place the buttons in the preferred location. It also creates a button object to add the preferred logic to the buttons placed onto the window.

3.4.7 Adding Shop Items and Tabs

In order to add a new item to the shop menu the `DrillDungeonGame/in_game_menus.py` file needs to be edited. In this file the ShopMenu class inherits from a InGameMenu class. The InGameMenu class is an arcade.view class which fogs the game screen and displays a grey box for a menu. The InGameMenu class is initialised in the ShopMenu class with dimentions for the shop menu.

To add an item to the shop a ShopItem object needs to be added. The ShopItem class needs to be initialised in ShopMenu, in the on_show method, with the following:

- shop_menu, which is the shop menu the item would be added to.
- center_x, which is the x-axis screen location for the item
- item_name, the name of the item you want to add.
- cost, how much the item would cost
- image, the image used when displaying item
- reusablility, set this to true if item can be bought multiple times
- button_function, add method to be executed when item bought
- function_inputs, if method requires input it here otherwise it's set to none.

An item needs to be added to a tab. Currently there are 2 tabs in the game, upgrades and ammo. To add a new tab a ShopTab object needs to be added . The ShopTab class needs to be initialised in ShopMenu, in the on_show method, with the following:

- tab_name, name of the tab
- start_center_y, the y-axis location to start listing items in tab.

With the tab and item objects defined in the on_show method, the items then need to be added to the tabs using the tabs add_item method. After all items have been added to their respective tabs the tab_list attribute of the ShopMenu class needs to be extended with the tabs created.