TinderBox Knight Maintenance Guide

This guide provides an overview of the Tinderbox Knight code to facilitate future development on the project.

Installation Guide

Python:

Tinderbox Knight is written in Python, using the Pygame module. If you do not already have both installed, you will need to download and install both in order to run the code.

Python can be installed here. Choose the correct version for your operating system and follow the instructions. Check it is installed running `Python --version` on a terminal (macOS, Linux) or PowerShell (Windows). You should see the version of Python you have installed displayed.

Pygame can be installed by typing `pip install pygame` in a terminal or PowerShell. More information on installing pygame can be found here.

Note: Tinderbox Knight has been developed using Python 3. If you have an older version of Python, you may need to update it.

Git:

Tinderbox Knight has been built using Git to manage version control. Git can be downloaded from here. Download the correct version for your operating system then follow the instructions on the installer.

To check you have git installed, open up a terminal or PowerShell, type `git --version`, and press enter. You should see version of git you have just installed.

You can make an empty git repository by running the terminal command `git init` while inside a folder. Connect to the TinderBoxKnight GitHub repository by running `git remote add origin https://github.com/Robzi11a/TinderBoxKnight.git`. You can verify that this has worked by running `git remote -v`.

Development Environment:

You can use any Python-compatible development environment to work on Tinderbox Knight. The most common IDE used by the development team to make the original code has been VS Code, which can be downloaded from here. Follow the download instructions, then open VS Code and install the Python Extension.

You may also have to select the Python interpreter you want to use - open the command plate using Control+Shift+P, then type Python: Select Interpreter. Select the version of version of Python you wish to use.

Getting the code:

The code is stored on a github repository at *https://github.com/Robzi11a/TinderBoxKnight*. You can either download a ZIP file of the code and extract it to your preferred folder, or you can open a terminal at the folder and run `git clone` followed by the link to the repository.

Game Overview

Tinderbox Knight is a tile-based dungeon crawling game. The aim of the game is to find the torch within each level by lighting up squares while avoiding the obstacles and monsters which populate each level.

Tinderbox Knight is written in Python, utilising the pygame module. Python is a relatively uncommon language for game development, as it does not run as quickly as languages in the C family. However, this is only a significant concern for large games with more complex graphics. For Tinderbox Knight, the ease of programming and readability of the code were more important. The pygame module is not used heavily within the game, so although familiarity with it is beneficial for making modifications to the game it is not strictly necessary. The pygame documentation can be read here.

Levels for the game are stored in text files which used a csv-like format to store the tiles in the rows and columns of the level. At the start of each level, this is read into a two-dimensional list. Almost all functionality within the game involves interacting with this list of tiles to check or change them.

Class Overview

Tinderbox Knight has been created using Object Oriented programming, in which each feature has a separate class. This section will provide an overview of the purpose and main features of each class.

main.py

Main.py is the starting point for the game, which the user will run to play Tinderbox Knight. The main function is to start the pygame engine.

game.py

Game.py is the main game file, which handles opening the pygame window and transitioning between game states. Game.py defines the State class, which is inherited by individual states that the game can be in. The Game class starts the game running. Again, as this is still high-level, it should not be necessary to modify this to add new features.

mainmenu.py

Mainmenu.py defines the menu class, which is what the player will see when opening the game. To add a new option to the menu for a new feature, follow the format used for the other menu options. This class also allows the player to use the arrow keys to select an option.

level.py

level.py is the largest file, and is responsible for the Level class, which handles all of the levels in the game.

At the top are the attributes of the class. This is primarily flags to signal than a specific action has happened that turn, as well as data relating to reading in the level.

Below this is where the game handles user input. Checks are run on each of the arrow keys for movement, as well as the S key, the F key, the S key and the space bar. As the movement and light function interact with enemies, when the player presses a check is also run to see if the player has interacted with an enemy.

Finally, there are extra methods to assist in various functionality. Most have descriptive comments to explain their functionality. Of particular note is the *read_in_level* method, which takes the text-based level and converts it to a two dimensional list by treating it like a csv file. This method also initialises many of the other objects relating to the level.

knight.py

The Knight class, defined in knight.py, handles most of the functionality directly linked to the player's character. Most importantly, this includes moving the knight around the board and checking that the knight is not moving out of bounds or into an enemy.

spider.py

Spider.py handles the cave spider's behaviour. This is quite a short class, as much of the spider's behaviour is covered by the knight class. Spider.py is therefore responsible for checking if the player has lit up a spider and resetting it afterwards.

rangedenemy.py

The ranged enemy has more complex behaviour than the spider. The Ranged_Enemy class's main method first checks whether the player is on the same row or column as the Ranged Enemy object (of which there can be multiple), and then checks whether there is a wall in between the player and the enemy. If there is not, and the player's square is light, it changes the array then returns True along with a copy of the original array to allow the squares to be reset.

bigtorch.py

The BigTorch class covers the functionality for ending the level when the player lights the torch. It first checks that the torch is visible, which is a prerequisite for being able to light it. It then checks that the player is close enough to the torch to light it. If both of these are True, then it will change the icon for the torch.

light.py

Light.py is responsible for allowing the player to light up the surrounding tiles. It does by two for loops, which loop through the indexes for the 3x3 centred on the player, switching each dark tile with its light variant by replacing the character indicating the visibility of the tile.

scan.py

The Scan class works similarly to the light, but loops through a 5x5 grid instead of 3x3. For each tile, it increments the count by one if the tile contains a hidden enemy. Scan also handles printing out the results of the scan.

tiles.py

Tiles.py is responsible for mapping each string code to a kind of tile. It contains two classes. The Tile class is for the individual tile, and contains a mapping of each string to a tile. Tiles is the class that contains the individual tile objects and is responsible for drawing them on the screen.

ground.py

This file does not contain a class or any logic but stores the image name for each kind of tile to provide a central point of reference for other classes.

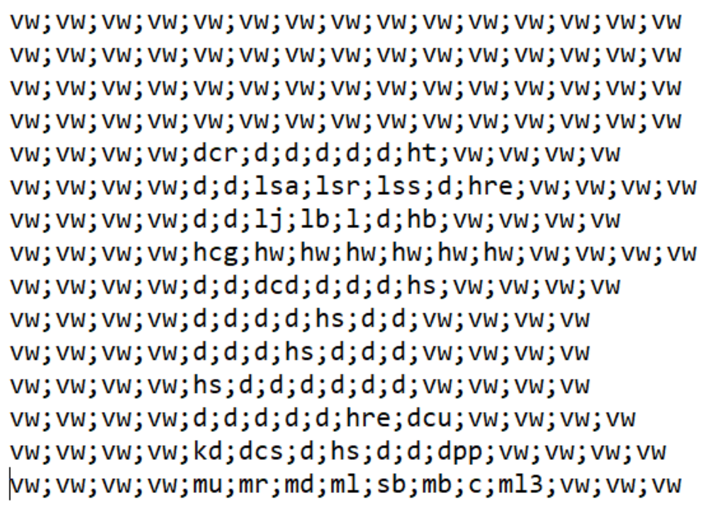
How do I… ?

This section will provide a guide to completing some of the more common modifications that you may wish to make to the game.

How do I make a new level?

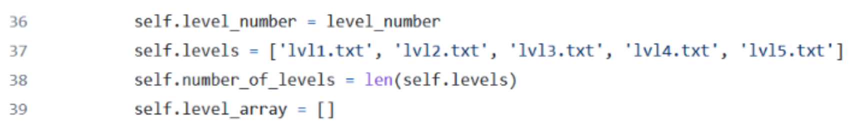
Tinderbox Knight was created with the goal of making levels as easy to create as possible. As such, creating a new level out of the existing tiles is an easy process which requires only minimal code to be changed.

Levels are stored as text files consisting of 15 lines of 15 elements, separated by a semi-colon. An example can be seen below. Each element corresponds to a different tile within the game; any tiles which are not used are simply filled out with a wall tile (vw). The maximum size for a level is 13x13, as the outer elements should always be a wall. At the bottom is a row containing the tiles for the display at the bottom of the screen. To make a new level, you can either start with a blank file or modify one of the existing text files. The meaning for each code can be found by looking in tiles.py.



```
vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw
vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw
vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw
vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw;vw
vw;vw;vw;vw;dcr;d;d;d;d;d;ht;vw;vw;vw;vw
vw;vw;vw;vw;d;d;lsa;lsr;lss;d;hre;vw;vw;vw;vw
vw;vw;vw;vw;d;d;lj;lb;l;d;hb;vw;vw;vw;vw
vw;vw;vw;vw;hcg;hw;hw;hw;hw;hw;hw;vw;vw;vw;vw
vw;vw;vw;vw;d;d;dcd;d;d;d;hs;vw;vw;vw;vw
vw;vw;vw;vw;d;d;d;d;hs;d;d;vw;vw;vw;vw
vw;vw;vw;vw;d;d;d;hs;d;d;d;vw;vw;vw;vw
vw;vw;vw;vw;hs;d;d;d;d;d;d;vw;vw;vw;vw
vw;vw;vw;vw;d;d;d;d;d;hre;dcu;vw;vw;vw;vw
vw;vw;vw;vw;kd;dcs;d;hs;d;d;dpp;vw;vw;vw;vw
vw;vw;vw;vw;mu;mr;md;ml;sb;mb;c;ml3;vw;vw;vw
```

Save the text level created in the levels folder, along with the other files. Then, add the name of the level to the self.levels list in level.py here:



```
        self.level_number = level_number
        self.levels = ['lvl1.txt', 'lvl2.txt', 'lvl3.txt', 'lvl4.txt', 'lvl5.txt']
        self.number_of_levels = len(self.levels)
        self.level_array = []
```

Then, just start the game as normal and play through. The game will automatically convert your text file to a level.

How do I add a new tile (that is only decorative)?

Adding a new tile is also a simple process but does require changing a few lines within the code. To start off with, add the image for your tile to the levels/images folder. Then, within ground.py, add the filename of the image along with a name for the tile. An example of a blank dark tile can bee seen below. For most tiles, you will need to add two names: one for when the tile is dark, and one for when it is light. The light tile should be your image, and the dark tile should be set to “DarkTile.png” as in the code below.

```

# Dark tile = ["d"]

DARK_TILE_EMPTY = "DarkTile.png"

```

Then, open the tiles.py file and add a code for the tile. Again, most tiles will need a dark and a light tile code. All tiles within the game start with one of four letters: d, l, h, or v. If a tile code starts with d or h, it indicates that it is dark; while l and v are light. Tiles the player can walk on are denoted by the d and l prefix, while h and v are tiles the player cannot walk on. It is important to use this naming system for the new tile if it is to work correctly within the game. An example of the pressure plate tile is below:

```
        #Pressure plate tiles

        elif kind_of_tile == "lpp" : filename = floor.LIT_PRESSURE_PLATE

        elif kind_of_tile == "dpp" : filename = floor.HIDDEN_PRESSUREPLATE

```

Once this is done, you can create a level as above but using your new tile code.

How do I add a new tile (that interacts with the player)?

If you have used the above instructions to add a dark and light variant of a tile, by default the player will be able to use the light function to switch between them. Extra interaction with the player comes in two forms: either the player can press a specified key to interact with the tile, or the behavior is activated if the player is in a specific location.

Either way, first define the behaviour in class in a separate file, then import it to level.py. Adding a check for a specific key press can be done as in the example below, where *pygame.k_<key>* is the key that the player must press.

```
# Scan
if key == pygame.K_s:
    self.scanner = Scanner(self.level_array, self.original_array, self.scanned_tiles, self.knight.return_position())
    self.is_scanned = True
```

If the tile should change when the player is in a specific location, add a check to each movement function to see if the player’s move will take them onto the tile. The *return_position()* method of the Knight class may be useful for this.

How do I add a new enemy?

The advantage of having a tile-based game is that all items within the game are tiles, so there is no special procedure to adding a monster tile. The above instructions will cover most cases.

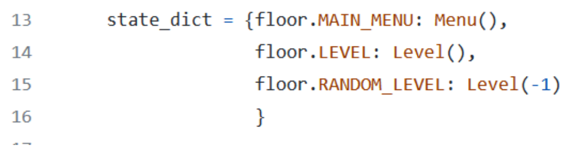
Depending on the design of the enemy, there are two further things that may need to be done. If there are multiple enemies within a level and each enemy is a separate object, you will need to create a list to store them. This can either be done by adding a check during *read_in_level* or *create_monster_objects*. Second, if the monster should attack the player when they walk into it, *check_for_enemy* within the Knight need should be updated to check for the new enemy code.

How do I add a new game mode?

Adding a new game mode is a significant undertaking, and the complete details are outside of the scope of this guide. However, there are a few items to note in integrating the mode with the existing game.

The most obvious is that the menu should be updated to allow the player to select the mode. Follow the template used for the other options, although the position of the item will obviously have to be updated.

Second, add the new game state to the *state_dict* within main.py, here:



```
state_dict = {floor.MAIN_MENU: Menu(),
              floor.LEVEL: Level(),
              floor.RANDOM_LEVEL: Level(-1)
              }
```

This will allow the game to switch to the new mode.