Panhia Vang

ISDS 558 Final Project

1/20/2020

Learning Pygame: Broken Crossing

**Introduction**

Video games have played a huge role throughout my childhood and into my early adult years. As I grew up and began to understand more complex ideas, it was clear that I really enjoyed the programming aspect of video games. For my final project, I wanted to challenge myself in python and learn more about the available resources to produce a product with a graphics or user interface versus a command line prompt. During my research, I discovered a python module, Pygame, which is designed for video game programming and perfect for what I wanted to do. With this discovery, I started planning out a simple two-dimensional (2D) game.

**Game Summary**

Broken Crossing is a single player fighter game. The player goes through different maps (levels) and must defeat all monsters to find a map and move on.

**Graphics**

The game was designed using free and open source resources. Because I am not very skilled in this area, I utilized packages that have been created and manipulated them to fit my requirements.

Fighters



There are two types of fighters in the game: character, the protagonist, and enemy, monsters who must be defeated (see Figure 1). These were designed using an open source character generator called Universal LPC Sprite Sheet Character Generator[4]. This generator allowed me to customize character configurations to import into the game. Once the sprite sheet for each fighter is created, I decoded the files to determine which rows executed a specific action. This resource was helpful in making animated movements for fighters.

Figure 1 Fighters: Enemy (Left) and Protagonist (Right)

Map

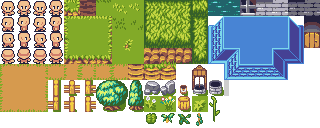
The tiles used to design the map was retrieved from a game development market [2]. It is a classic role-playing game (RPG) tile set. I needed a tile set that was 2D and not a platform-based design. In addition, I wanted each tile to be the same size, which is easier to scale (up or down) and to make calculations when programming. To create the game, I downloaded a software application, Tiled, which is a map editor tool [8]. This allows me to import the map tile set, specify the tiles, and then design my own custom map using the tiles. For this project, I only created the first level.

Figure 2 Classic RPG Tile Set

Items



Items play a big role in the game by allowing the protagonist to pick up randomly spawned or dropped items that can potentially boost or hinder the character. This aspect of the game makes it more interesting and can make each level unpredictable based on the protagonist’s actions. Items are automatically activated when the protagonist steps into the tile where the envelope is shown. For the graphics, I wanted to utilize something that was simple but detailed enough to convey the type of item (good or bad) without needing an explanation. I used a package called Ravenmore’s Icon Pack 2.0 which contains weapons and armor icons that are scalable [7]. There are four types of items implemented in the game: health (heart), poison (green potion), map and damage increase (sword). Additionally, an additional default item display is shown to hide the actual item from the protagonist using an envelope.

Figure 3 Ravenmore's Icon Pack 2.0

**Sounds**

Sound effects can help enhance the game and make it more engaging as the protagonist progresses through the game. To do so, I utilized several resources to help with animating actions and making it more interesting. For the background music, I used a song created by an acquaintance, Paul Piller, called Hair Gel [5]. This song was perfect for the game to make it lively and fit the landscape. Additionally, I wanted sounds for certain actions like attacking, looting, winning, or losing. I used a role playing game sound effect package by Kenney for creating some of these sounds [1]. Additional resources for sound effects came from freesound.org [3].

**User Interface**

A player can interact within the game using 6 keys: up, down, left, right, enter, and escape (esc). The arrow keys allow the user to move around the map. The escape key is used to exit out of the game. The enter (return) key is used to progress to the next level. This key does not do anything unless the screen renders a text indicating that the key can be pressed.

**Game Flow**

The game will automatically load all resources onto the screen. The protagonist will always spawn at the top left corner known as location (0, 0) on the grid. Once the resources load, all objects are stored to determine how the game will flow. If the game is on idle for long enough, it will cancel the program. Otherwise, the game waits for the player to select a valid key (refer to **User Interface** section for valid keys). Upon selecting a valid key, the game will determine if the user has pressed escape. If escape is pressed, the game will close. Otherwise, the game checks to see if the player is still alive. If the player is still alive, it will then check to see if the player has obtained a key (map). If the player has the key, the screen will render text indicating to the player that he/she can progress to the next level by pressing the enter key. If the player has not found the key, the game will check to see if there are any enemies on the map. If there are still enemies on the map, the game will interpret the key that was pressed and determine if there are an items or enemies on the tile that the player wants to move to. It will either activate a fight if there is an enemy or activate an item if there is an item. Once all actions are completed, the game waits again for the player to select a key to repeat the process. A workflow has been developed and can be referenced in the appendix on the game flow.

**Mission**

Defeat the enemies on the map without dying.

**Level Completed**

The player is alive and there are no enemies on the map. The last enemy will drop a map to move onto next level.

**Success**

The player is alive and there are no enemies on the final level.

**Failure**

The player reaches 0 health at any point in the game.

**Combat**

The current combat between the player and enemy is automatic. Whenever there is an enemy present on the tile, the game will initiate a fight (player attacks first and then the enemy). The player can only move onto the tile once the enemy is defeated and destroyed.

**Modules and Reference Files**

Pygame

Pygame is a python module that is designed for game programming. This module is used extensively for many aspects of the game including initializing the game, rendering objects, images, and text and interpreting key press events.

Time

This module is used to keep track of time. It also creates timers to show animations.

Json

This module is used to read the json map file. This is very important to read the map design and determine if there are any blocks that the player cannot move to.

Random

The random module is used to get random numbers. This technique was also paired with rounding and/or multipliers to generate random numbers.

Utils.py

This is a file containing functions that are used across multiple files. It helped reduce the number of times I would have to define certain functions that were used often. The code for this file is in the appendix.

*get\_resource\_path(relative\_path) –* This function is used to return the absolute path of a resource file. Because there were many files being referenced or used in the game, it was easier to get the location by using this function.

*clamp(num, min\_value, max\_value)* – This is a common way to limit the value in a given range. This method was used to prevent the player from moving off screen.

*round\_whole(number, multiple)* – This function is used to round a number based on a multiple that has been supplied.

*rand\_range(min, max)* – This function returned a random number based on the range provided.

*play\_sound(file, volume=None)* – This function is used to play a sound effect.

Constants.py

This file is used to store constants to make it easily accessible by other files. The code for this file is located in the appendix.

*ANIMATION\_DURACTION* – Sets the length of duration for animations

*SCREEN\_SIZE* – Sets the size of the screen for the game window

*FRAME\_TIMEOUT* – Sets the frame timeout refresh rate

*TILE\_SIZE* – Sets the tile size of each grid

*BIG\_FONT* – Sets the pygame.font to a large font size

*REG\_FONT* – Sets the pygame.font to a regular font size

*BLACK\_COLOR* – Sets a tuple equivalent to color black

*RED\_COLOR* – Sets a tuple equivalent to color red

*BEIGE\_COLOR* – Sets a tuple equivalent to color beige

*BLUE\_COLOR* – Sets a tuple equivalent to color blue

*CENTER\_POS* – Sets a tuple that is half the size of the screen (x, y)

*Directions* – Defines a class of the different directions for Fighters

*Actions* – Defines a class of the different action types for Fighters

**Classes**

All class source codes have been placed in the appendix.

Timer

Timer is a stand-alone class that is used to keep track of time. It is primarily used to keep track of time when the game is running (to time out the game if it is not being played). It is also used in the FloatingAnimation class to animate an object that floats and fades away.

Collision

Collision is a class used for determining the objects that are loaded to the level. It has three members: x, y, and invalid\_position. X and Y are integers to determine where the object is located on the level. Invalid\_position is a Boolean variable to determine if the location of the object is a valid position. A position is only valid if there is nothing (enemy, item, or colliding tile) at the specified location.

There are also local variables used to keep track of this information.

*colliding\_tiles* – A set used to read a json file to determine which areas on the map are not allowed for the player to step on.

*enemy\_tiles* – A dictionary used to store all enemies that are spawned on the level.

*item\_tiles* – A dictionary used to store all items that are spawned on the level.

*width* – Determine the location of the item on the x axis.

*height* – Determine the location of the item on the y axis.

All the methods in this class are static. This is because this class is used across different classes but do not particularly share any related properties. Additionally, no instances of Collision will be created in the game.

Renderable

Renderable is a parent class to several sub-classes (Animation, LevelManager, FloatingAnimation, and Item). This class is used to instantiate and render objects. It is inherited and used in many ways including moving the player around the map, revealing items, and so on.



LevelManager

LevelManager is a child class of Renderable. It is used to manage all the levels implemented in the game through storing the levels in a list. It also manages the progression of levels using this list which is helpful in instantiating different levels.

Level

Level is a child class of LevelManager. It is used to create map objects.

Level1

Level1 inherits from Level, which inherits from LevelManager. This class creates the first level of the map.

Level2

Level2 inherits from Level, which inherits from LevelManager. This class creates the second level of the map.

Level3

Level3 inherits from Level, which inherits from LevelManager. This class creates the third level of the map.



FloatingAnimation

The class inherits from Renderable. It is a class used to render hidden objects (Item class) and reveal the actual item behind the envelope for a specified time period. The rendering of this object was designed to show above the player and float up while fading away.

Item

Item inherits from Renderable. This class has one member: hidden\_image which is a string of the item that is hidden behind the envelope. Additionally, it overrides the render method from Renderable and defines its own.

*on\_collide(self)* – A method called when the player collides with an item. This method creates an animation and renders it on the window. It then activates the item and removes it from the list.

*render(self, screen)* – This method is overridden to show the default image for items.

*activate(self) –* This method is not defined on purpose. Because each item has its own way of being activated, it will be defined within Item’s child classes.

HealthItem

HeathItem is a class that inherits from Item. It is one of the items that can be activated when a player collides with an envelope. This item will add 25 health to the player.

PoisonItem

PoisonItem is a class that inherits from Item. It is one of the items that can be activated when a player collides with an envelope. This item will reduce 25 health from the player.

SwordItem

SwordItem is a class that inherits from Item. It is one of the items that can be activated when a player collides with an envelope. This item will increase the player’s maximum damage by 10.

MapItem

MapItem is a class that inherits from Item. It is one of the items that can be activated when a player collides with an envelope. This item will indicate that the player has found the key to proceed to the next level or complete the game.



Animation

Animation is a class that inherits from Renderable. This class is the base of designing how to animate things within the game. To do so, several members and methods were implemented. The class uses a timer to pace the rendering of different images in order to make it appear animated in the game. It also uses a direction member to determine where

CharacterAnimation

CharacterAnimation is a class that inherits from Animation, which inherits from Renderable. It is a very short class to instantiate the Character png file for rendering images of the character actions.

Fighter

Fighter is a parent class used to define things that can fight. It has two members: health and damage. It also has three methods: *get\_damage(self)*, to determine the amount of damage a fighter does; *is\_alive(self)*, to determine if the fighter is still alive; and *fight(self, enemy)*, to call the fighter to fight an enemy.

Character

Character is a class that inherits from Fighter and CharacterAnimation. This class is used to create the player, set the player position, render the player’s image by overriding Renderable’s render method, and read the player’s movements based on key presses.

Enemy

Enemy is a class that inherits from Fighter and Animation. This class is used to create enemies for the player. At the same time, it defines a method, *drop\_item(self)*, which rewards the player with a random item once it is defeated.

Skeleton

Skeleton is a class that inherits from Enemy. This class is used to create an enemy type called skeleton.

Orc

Orc is a class that inherits from Enemy. This class is used to create an enemy type called orc. It has different traits compared to Skeleton. At the same time, it redefines a method, *drop\_item(self)*, which rewards the player with a random item once it is defeated.

**Results**

Everything described is put together in the file called *\_\_init\_\_.py* which is called from *\_\_main\_\_.py*. The main function describes how the game goes. I first initialize pygame and set some settings including the logo, icon, and caption. Additionally, the background music is started here. I then set the size of the screen and start the timer. Once this has been completed, I render all my objects including the collisions, map, and character. From there I think start a loop to run the game until it is stopped. This will then link to the other classes to determine what actions the program should take.

****

**Conclusion and Future Improvements**

I knew I was a bit ambitious when I started the project, but I have thoroughly enjoyed it. The process took very long, and I was spending a lot of time each day to make progress. However, I have learned a lot from this. I was able to apply more concepts related to classes and inheritance. Additionally, I was finally able to play with a module that really helped make the process of building a game much easier (pygame). However, with my time constraints, I do think there were things I was not able to implement. I wanted to have many levels but the timing was just not enough. Additionally, I wanted to have music and some sound effects. I also planned to have animated movements and would probably take the route of trying to make it read keys continuously. All in all, it was very fun.

**Bibliography**

[1] 50 RPG Sound Effects by Kenney | OpenGameArt.org. Retrieved from https://opengameart.org/content/50-rpg-sound-effects

[2] Classic RPG Tileset | GaveDev Market. Retrieved from https://www.gamedevmarket.net/asset/classic-rpg-tileset/

[3] Freesound - "success.wav" by grunz | freesound.org. Retrieved from https://freesound.org/people/grunz/sounds/109662/

[4] Mehl, Luke., Sjolund, Johannes., van de Steeg, Marcel. Riecke, Manuel., Brimhall, Thane., Nagel, Lori A., Zimmerman, Lanea., Challener, Stephen., Eddeland, Daneil., Williams, Shaun., White, Joe., Weyer, Mark., Rivera, Babara., ReimanJaidyn., and Nila122. Universal LPC Spritesheet Character Generator. Retrieved from http://gaurav.munjal.us/Universal-LPC-Spritesheet-Character-Generator/#

[5] Piller, Paul. Hair Gel.

[6] Pygame Documentation. Retrieved from https://www.pygame.org/docs/

[7] Ravenmore. Fantasy Icon Pack by Ravenmore | OpenGameArt.org. Retrieved from https://opengameart.org/content/fantasy-icon-pack-by-ravenmore-0

[8] Tiled Map Editor | A flexible level editor. Retrieved from https://www.mapeditor.org/

**Appendix**

*File Structure*

BrokenCrossing/

├── \_\_init\_\_.py

├── \_\_main\_\_.py

├── Character.py

├── Collision.py

├── Constants.py

├── Enemy.py

├── Fighter.py

├── FloatingAnimation.py

├── Item.py

├── Level.py

├── LevelManager.py

├── Renderable.py

├── Timer.py

├── Utils.py

└── resources/

├── \_\_init\_\_.py

├── Character.png

├── envelope.png

├── Hair-Gel-By-Paul-Piller.mp3

├── heart.png

├── map.png

├── Orc.png

├── potionGreen.png

├── scroll.png

├── Skeleton.png

├── sword.png

├── win.wav

├── map/

│ ├── ClassicRPG\_Sheet.png

│ ├── ClassicRPG\_Sheet.tsx

│ ├── level\_1.json

│ ├── level\_1.tmx

│ ├── level\_1.png

│ ├── level\_2.json

│ ├── level\_2.tmx

│ ├── level\_2.png

│ ├── level\_3.json

│ ├── level\_3.tmx

│ └── level\_3.png

└── OGG/

├── chop.ogg

├── cloth2.ogg

├── creak2.ogg

├── creak3.ogg

└── dropLeather.ogg

*Python Codes*

**\_\_init\_\_.py**

'''

module for isds 558

'''

import pygame

import time

from .Item import \*

from .Timer import \*

from .Enemy import \*

from .Utils import \*

from .Constants import \*

from .Collision import \*

from .Character import \*

from .Renderable import \*

from .LevelManager import \*

# to do's here-- add music

# add sound effects

# add win sound

# add lose sound

# loop animation(like walking) or no(attack)

# play attack animation during attacks

# prevent additional actions while in attack

def main():

'''

main function for module

'''

# initialize the pygame module

pygame.mixer.pre\_init(44100, 16, 2, 4096)

pygame.init()

pygame.mixer.init()

# play music infinitely

pygame.mixer.music.load(get\_resource\_path("Hair-Gel-By-Paul-Piller.mp3"))

pygame.mixer.music.set\_volume(0.4)

pygame.mixer.music.play()

# load and set the logo

logo = pygame.image.load(get\_resource\_path("scroll.png"))

pygame.display.set\_icon(logo)

pygame.display.set\_caption("Broken Crossing")

# create a surface on screen

screen = pygame.display.set\_mode(SCREEN\_SIZE)

# start time

timer = Timer()

# initialize objects

# collission

Collision.init()

# background

bg = pygame.Surface(SCREEN\_SIZE)

bg.fill(BLACK\_COLOR)

bg\_renderable = Renderable(bg)

Renderable.add\_renderable(0, bg\_renderable)

# map

level\_mgr = LevelManager()

LevelManager.instance = level\_mgr

Renderable.add\_renderable(1, level\_mgr)

# character

character = Character()

Character.instance = character

Renderable.add\_renderable(100, character)

# define a variable to control the main loop

running = True

# main loop

while running:

# event handling, gets all event from the event queue

for event in pygame.event.get():

# only do something if the event is of type QUIT

if event.type == pygame.QUIT:

# change the value to False, to exit the main loop

running = False

if event.type == pygame.KEYDOWN:

# if user presses "ESC"

# close the game

if event.key == pygame.K\_ESCAPE:

running = False

# otherwise, determine movement

# check timer

if timer.is\_expired():

# timeout

timer.start(FRAME\_TIMEOUT)

# Handle inputs

character.on\_button\_press()

# render all renderables

for r in Renderable.get\_renderables():

r.render(screen)

pygame.display.flip()

time.sleep(0.001)

**\_\_main\_\_.py**

from BrokenCrossing import \*

main()

**Animation.py**

import pygame

from .Timer import \*

from .Constants import \*

from .Renderable import \*

class Animation(Renderable):

def \_\_init\_\_(self, image):

Renderable.\_\_init\_\_(self, image, False)

self.timer = Timer()

self.timer.start(ANIMATION\_DURATION)

self.active\_frame = 0

self.direction = Directions.DOWN

self.action = Actions.WALK

self.frame\_size = 64

self.\_\_is\_frozen = True

def render(self, screen):

if self.\_\_is\_frozen:

self.active\_frame = 0

elif self.timer.is\_expired():

if not (self.action == Actions.HURT and self.active\_frame == Actions.NUM\_FRAMES[Actions.HURT] - 1):

self.timer.start(ANIMATION\_DURATION)

self.active\_frame += 1

self.active\_frame %= Actions.NUM\_FRAMES[self.action]

if self.active\_frame == 0 and self.action != Actions.WALK:

self.set\_action(Actions.WALK)

self.set\_frozen(True)

crop\_x = self.active\_frame \* self.frame\_size

crop\_y = ((Directions.NUM\_DIRECTIONS \* self.action) + self.direction) \* self.frame\_size

cropped = pygame.Surface((self.frame\_size, self.frame\_size), pygame.SRCALPHA)

cropped.blit(self.image, (0, 0), (crop\_x, crop\_y, self.frame\_size, self.frame\_size))

cropped = pygame.transform.scale(cropped, (TILE\_SIZE, TILE\_SIZE))

screen.blit(cropped, (self.x\_position, self.y\_position))

def set\_action(self, action):

if self.action != action:

self.action = action

self.active\_frame = 0

if self.action == Actions.HURT:

self.direction = Directions.UP

def set\_direction(self, direction):

if self.direction != direction and self.action != Actions.HURT:

self.direction = direction

self.active\_frame = 0

def set\_frozen(self, frozen):

if self.\_\_is\_frozen != frozen:

self.\_\_is\_frozen = frozen

if not self.\_\_is\_frozen:

self.timer.start(ANIMATION\_DURATION)

def get\_frozen(self):

return self.\_\_is\_frozen

class CharacterAnimation(Animation):

def \_\_init\_\_(self):

Animation.\_\_init\_\_(self, "Character.png")

**Character.py**

import pygame

import random

from .Timer import \*

from .Fighter import \*

from .Collision import \*

from .Constants import \*

from .Renderable import \*

from .LevelManager import \*

# Character class

# used for the user

class Character(Fighter, CharacterAnimation):

instance = None

# initialize character

def \_\_init\_\_(self):

Fighter.\_\_init\_\_(self, 150, 15)

CharacterAnimation.\_\_init\_\_(self)

self.inactive\_time = 1.0

# user location will always be top left

self.row = 0

self.col = 0

self.y\_position = 0

self.has\_key = False

self.input\_timer = Timer()

self.input\_timer.start(ANIMATION\_DURATION)

self.inactive\_timer = Timer()

self.inactive\_timer.start(self.inactive\_time)

self.had\_key = False

self.previous\_lost = False

self.continue\_level = False

# render character info

def render(self, screen):

# show character object

CharacterAnimation.render(self, screen)

# WIN STATUS

# check if there are any enemies on map

#if num\_enemies == 0:

if self.has\_key:

if not self.had\_key:

play\_sound('win.wav', 1)

if LevelManager.instance.has\_remaining\_level():

Renderable.show\_complete(screen)

else:

Renderable.show\_win(screen)

self.had\_key = True

# if there are enemies

# check if user still alive

if self.is\_alive():

# render user's health and damage at bottom left

health\_display = 'Health: {} / 150'.format(self.health)

damage\_display = 'Damage: {} / 80'.format(self.damage)

health\_text = REG\_FONT.render(health\_display, True, BLUE\_COLOR, BEIGE\_COLOR)

health\_text\_rect = health\_text.get\_rect()

health\_text\_rect.topleft = (0, SCREEN\_SIZE[2] - (TILE\_SIZE))

damage\_text = REG\_FONT.render(damage\_display, True, BLUE\_COLOR, BEIGE\_COLOR)

damage\_text\_rect = damage\_text.get\_rect()

damage\_text\_rect.topleft = (health\_text\_rect.left, health\_text\_rect.top + health\_text\_rect.height)

screen.blit(health\_text, health\_text\_rect)

screen.blit(damage\_text, damage\_text\_rect)

else:

# if there are enemies

# and user is not alive

Renderable.show\_lose(screen)

if not self.previous\_lost:

self.set\_action(Actions.HURT)

play\_sound('OGG\\creak2.ogg')

play\_sound('OGG\\creak3.ogg')

self.previous\_lost = not self.is\_alive()

# when button is pressed for up, down, left, right

def on\_button\_press(self):

row = self.row

col = self.col

move\_x = 0

move\_y = 0

direction = self.direction

if self.action == Actions.WALK:

self.set\_frozen(self.inactive\_timer.is\_expired())

if self.action != Actions.WALK:

return

if self.input\_timer.is\_expired():

self.input\_timer.start(ANIMATION\_DURATION)

else:

return

# verify that user is alive

# if not alive, exit

if not self.is\_alive():

return

# check to see if character will collide when moved

# if no collision then move the character

keys = pygame.key.get\_pressed()

if not self.has\_key:

if keys[pygame.K\_LEFT]:

col -= 1

move\_x = -TILE\_SIZE

direction = Directions.LEFT

elif keys[pygame.K\_RIGHT]:

col += 1

move\_x = TILE\_SIZE

direction = Directions.RIGHT

elif keys[pygame.K\_UP]:

row -= 1

move\_y = -TILE\_SIZE

direction = Directions.UP

elif keys[pygame.K\_DOWN]:

row += 1

move\_y = TILE\_SIZE

direction = Directions.DOWN

else:

if keys[pygame.K\_RETURN]:

self.continue\_level = True

if direction != self.direction:

self.set\_direction(direction)

# check to see if user is moving into a tile

# that will collide with an object

if not Collision.check\_collision(row, col):

item = Collision.check\_item(row, col)

enemy = Collision.check\_enemy(row, col)

is\_item\_present = item is not None

is\_enemy\_present = enemy is not None

# trigger for if there is an item on tile

if is\_item\_present:

item.on\_collide()

# trigger for if there is an enemy on tile

elif is\_enemy\_present:

# initiates fight

# play sword sound

play\_sound('OGG\\chop.ogg')

self.fight(enemy)

enemy.fight(self)

self.set\_action(Actions.THRUST)

self.set\_frozen(False)

enemy\_direction = Directions.get\_opposite(self.direction)

enemy.set\_direction(enemy\_direction)

enemy.set\_action(Actions.THRUST)

enemy.set\_frozen(False)

# once enemy dies

if not enemy.is\_alive():

# enemy drops item

# enemy removed from screen

enemy.set\_action(Actions.HURT)

Collision.remove\_enemy(row, col)

# if enemy is not on the tile

# allow user to move to tile

if not is\_enemy\_present and (move\_x or move\_y):

self.move(move\_x, move\_y)

self.row = row

self.col = col

self.inactive\_timer.start(self.inactive\_time)

def add\_health(self, health):

self.health = clamp(self.health + health, 0, 150)

def add\_damage(self, damage):

self.damage = clamp(self.damage + damage, 0, 80)

**Collision.py**

import json

from .Utils import \*

# static class used to read json file

# json file includes data on background

# collision objects

# item objects

class Collision:

# set used to store all tiles which cannot be collided on (like a tree or wall)

colliding\_tiles = set()

# dictionary of where all enemies located

enemy\_tiles = {}

# dictionary of all items located

item\_tiles = {}

width = 0

height = 0

@classmethod

def empty\_all(cls):

cls.colliding\_tiles = set()

for enemy in cls.enemy\_tiles.values():

enemy.remove()

cls.enemy\_tiles = {}

for item in cls.item\_tiles.values():

item.remove()

cls.item\_tiles = {}

# static method

# set the location of where the object will be located

@classmethod

def get\_spawn\_loc(cls):

invalid\_position = True

# loop while the position determined is not good

while invalid\_position:

# randomly determine coordinates based on random and size of screen

x = random.random() \* (SCREEN\_SIZE[0])

y = random.random() \* (SCREEN\_SIZE[2])

# round the coordinates to whole numbers

x = round\_whole(x, TILE\_SIZE)

y = round\_whole(y, TILE\_SIZE)

(row, col) = (int(y / TILE\_SIZE), int(x / TILE\_SIZE))

# check if there is already an object on the location

invalid\_position = cls.check\_collision(row, col) or (row, col) == (0, 0) or Collision.check\_item(row, col) or Collision.check\_enemy(row, col)

return (row, col, x, y)

# static method

@classmethod

def init(cls):

# read file

path = get\_resource\_path('map\\level\_1.json')

with open(path, 'r') as map\_file:

data = map\_file.read()

# parse file

# find object layer

obj = json.loads(data)

collision\_layer = None

for layer in obj['layers']:

if layer['name'] == 'Objects':

collision\_layer = layer

break

# if object layer exists

# set width and height

# determine row and col of each collision object

if collision\_layer is not None:

cls.width = collision\_layer['width']

cls.height = collision\_layer['height']

for i, d in enumerate(collision\_layer['data']):

row = int(i / cls.width)

col = i % cls.width

if d != 0:

cls.colliding\_tiles.add((row, col))

# static method

# check to see if render items are on collision objects

@classmethod

def check\_collision(cls, row, col):

collides = ((row, col) in cls.colliding\_tiles or

row < 0 or

row >= cls.height or

col < 0 or

col >= cls.width)

return collides

# static method

# add the enemy tile into the dictionary

@classmethod

def add\_enemy\_tile(cls, row, col, enemy):

cls.enemy\_tiles[(row, col)] = enemy

# static method

# check if there is an enemy at the location

@classmethod

def check\_enemy(cls, row, col):

return cls.enemy\_tiles.get((row, col))

# static method

# remove the enemy from the dictionary

@classmethod

def remove\_enemy(cls, row, col):

cls.enemy\_tiles.pop((row, col))

# static method

# get the total number of enemies in the dictionary

@classmethod

def get\_num\_enemies(cls):

return len(cls.enemy\_tiles)

# static method

# add an item into the dictionary

@classmethod

def add\_item\_tile(cls, row, col, item):

cls.item\_tiles[(row, col)] = item

# static method

# check if item exists at location

@classmethod

def check\_item(cls, row, col):

return cls.item\_tiles.get((row, col))

# static method

# remove item from dictionary

@classmethod

def remove\_item(cls, row, col):

cls.item\_tiles.pop((row, col))

**Constants.py**

import pygame

# constants used in game

ANIMATION\_DURATION = 0.2

# screen size for game

SCREEN\_SIZE = (1280, 768)

# frame timeout for refresh

FRAME\_TIMEOUT = 1/30

# size of each tile (grid outlined in game)

TILE\_SIZE = 64

# font & sizes

pygame.init()

BIG\_FONT = pygame.font.Font('freesansbold.ttf', 72)

REG\_FONT = pygame.font.Font('freesansbold.ttf', 24)

# rgb colors

BLACK\_COLOR = (0, 0, 0)

RED\_COLOR = (255, 0, 0)

BEIGE\_COLOR = (239, 228, 176)

BLUE\_COLOR = (0, 0, 128)

CENTER\_POS = (SCREEN\_SIZE[0]/2, SCREEN\_SIZE[2]/2)

class Directions:

UP = 0

LEFT = 1

DOWN = 2

RIGHT = 3

NUM\_DIRECTIONS = 4

@classmethod

def get\_opposite(cls, direction):

opposites = {

cls.UP: cls.DOWN,

cls.DOWN: cls.UP,

cls.LEFT: cls.RIGHT,

cls.RIGHT: cls.LEFT

}

return opposites.get(direction, cls.DOWN)

class Actions:

SPELL\_CAST = 0

THRUST = 1

WALK = 2

SLASH = 3

SHOOT = 4

HURT = 5

NUM\_ACTIONS = 6

NUM\_FRAMES = {

SPELL\_CAST: 7,

THRUST: 8,

WALK: 9,

SLASH: 6,

SHOOT: 13,

HURT: 6

}

**Enemy.py**

import pygame

import random

from .Utils import \*

from .Fighter import \*

from .Constants import \*

from .Collision import \*

from .Animation import \*

from .Renderable import \*

# Enemy class

# used to handle all enemy objects

class Enemy(Fighter, Animation):

# initialize enemy at a random spot

# that is not in a collision coordinate

def \_\_init\_\_(self, enemy, health, damage):

Fighter.\_\_init\_\_(self, health, damage)

Animation.\_\_init\_\_(self, enemy)

(row, col, x, y) = Collision.get\_spawn\_loc()

self.set\_position(x, y)

Collision.add\_enemy\_tile(row, col, self)

def render(self, screen):

if self.action == Actions.HURT and Actions.NUM\_FRAMES[Actions.HURT] - 1 == self.active\_frame:

self.removed = True

self.drop\_item()

else:

Animation.render(self, screen)

# enemy will have ability to drop an item

def drop\_item(self):

from .Item import MapItem, HealthItem, SwordItem

play\_sound('OGG\\dropLeather.ogg')

num = rand\_range(0, 100)

# get enemy current position

(row, col) = self.get\_coordinates()

x = self.x\_position

y = self.y\_position

# randomly drop a health or sword item

if len(Collision.enemy\_tiles) == 0:

item = MapItem(row, col, x, y)

elif num <= 50:

item = HealthItem(row, col, x, y)

else:

item = SwordItem(row, col, x, y)

Renderable.add\_renderable(49, item)

class Skeleton(Enemy):

def \_\_init\_\_(self):

Enemy.\_\_init\_\_(self, "Skeleton.png", 75, 10)

class Orc(Enemy):

def \_\_init\_\_(self):

Enemy.\_\_init\_\_(self, "Orc.png", 85, 15)

# drops something more special by chance

def drop\_item(self):

from .Item import MapItem, PoisonItem, SwordItem

play\_sound('OGG\\dropLeather.ogg')

num = rand\_range(0, 100)

# get enemy current position

(row, col) = self.get\_coordinates()

x = self.x\_position

y = self.y\_position

# randomly drop a health or sword item

if len(Collision.enemy\_tiles) == 0:

item = MapItem(row, col, x, y)

elif num <= 75:

item = PoisonItem(row, col, x, y)

Renderable.add\_renderable(51, item)

else:

item = SwordItem(row, col, x, y)

Renderable.add\_renderable(49, item)

**Fighter.py**

import random

import pygame

from .FloatingAnimation import \*

from .Renderable import \*

# Fighter class

# used for child classes that can fight

class Fighter:

# initialize fighters (characters and enemies)

def \_\_init\_\_(self, health, damage):

self.health = health

self.damage = damage

# default attack method

def fight(self, enemy):

# check if obj is alive

if self.is\_alive():

# check if obj has an enemy

if enemy is not None:

# determine damage to enemy

damage = self.get\_damage()

enemy.health -= damage

# show damage on screen

damage\_text = REG\_FONT.render('-{}'.format(damage), True, RED\_COLOR)

# make damage text on screen fade

animation = FloatingAnimation(damage\_text)

animation.set\_position(enemy.x\_position, enemy.y\_position)

Renderable.add\_renderable(101, animation)

# random damage generator for fighting

def get\_damage(self):

jiggle\_amount = self.damage / 10

half\_jiggle = jiggle\_amount / 2

jiggle\_amount \*= random.random()

jiggle\_amount -= half\_jiggle

damage = self.damage + jiggle\_amount

return int(damage)

# check if fighter is alive

def is\_alive(self):

return self.health > 0

# determine health of fighter

def add\_health(self, points):

self.health += points

self.health = min(self.health, 150)

return self.health

# determine dmg attack of fighter

def add\_damage(self, points):

self.damage += points

self.damage = min(self.damage, 80)

return self.damage

**FloatingAnimation.py**

from .Renderable import \*

from .Timer import \*

# FloatingAnimation class

# used to deal with animations that will float and disappear in the game

class FloatingAnimation(Renderable):

# initializer

def \_\_init\_\_(self, target):

Renderable.\_\_init\_\_(self, target)

self.animation\_time = 1

self.timer = Timer()

self.timer.start(self.animation\_time)

# rendering on screen

# reveals hidden image

def render(self, screen):

# if time is not expired

if not self.timer.is\_expired():

temp = self.image.copy()

# this works on images with per pixel alpha too

# animate the object to float above the character

# and make transparency increase over time

ratio\_animation\_done = self.timer.get\_time\_elapsed() / self.animation\_time

transparency = 255 - (ratio\_animation\_done \* 255)

transparency = int(transparency)

temp.fill((255, 255, 255, transparency), None, pygame.BLEND\_RGBA\_MULT)

y = self.y\_position

y -= TILE\_SIZE \* ratio\_animation\_done

screen.blit(temp, (self.x\_position, y))

else:

# remove item

self.removed = True

**Item.py**

import pygame

from .Timer import \*

from .Utils import \*

from .Collision import \*

from .Constants import \*

from .Character import \*

from .Renderable import \*

from .FloatingAnimation import \*

# Item class used for usable items in game

# inherits from Renderable

class Item(Renderable):

# initializer

def \_\_init\_\_(self, image, hidden\_image, row=None, col=None, x=None, y=None):

Renderable.\_\_init\_\_(self, image)

# get a spawn location if it is not specified when initialized

if not (row is not None and col is not None and x is not None and y is not None):

(row, col, x, y) = Collision.get\_spawn\_loc()

# set spawn location

self.set\_position(x, y)

# add item into items list

Collision.add\_item\_tile(row, col, self)

# set actual item image

self.hidden\_image = hidden\_image

# actions when character collides with item

def on\_collide(self):

# play open sound

play\_sound('OGG\\cloth2.ogg', 1)

# reveal the hidden image (hidden item)

animation = FloatingAnimation(self.hidden\_image)

animation.set\_position(self.x\_position, self.y\_position)

Renderable.add\_renderable(101, animation)

# use item

self.activate()

# set remove trait to true

self.removed = True

(row, col) = self.get\_coordinates()

# remove item from items list

Collision.remove\_item(row, col)

# show item on screen

def render(self, screen):

Renderable.render(self, screen)

# to be defined based on item type (child classes)

def activate(self):

raise NotImplementedError

# HealthItem class

# info for spawning health object

class HealthItem(Item):

# initializer

def \_\_init\_\_(self, row=None, col=None, x=None, y=None):

# this initializer is designed sort of like an overloaded method

# it will spawn randomly at start of game

# or the location for item to spawn is specified when enemy is killed

Item.\_\_init\_\_(self, 'envelope.png', 'heart.png', row, col, x, y)

# use health item (adds health to character)

def activate(self):

Character.instance.add\_health(25)

# PoisonItem class

# info for spawning poison object

class PoisonItem(Item):

# initializer

def \_\_init\_\_(self, row=None, col=None, x=None, y=None):

# this initializer is designed sort of like an overloaded method

# it will spawn randomly at start of game

# or the location for item to spawn is specified when enemy is killed

Item.\_\_init\_\_(self, 'envelope.png', 'potionGreen.png', row, col, x, y)

# use poison item (deducts health from character)

def activate(self):

Character.instance.add\_health(-15)

# SwordItem class

# info for spawning sword object

class SwordItem(Item):

# initializer

# this initializer is designed sort of like an overloaded method

# it will spawn randomly at start of game

# or the location for item to spawn is specified when enemy is killed

def \_\_init\_\_(self, row=None, col=None, x=None, y=None):

Item.\_\_init\_\_(self, 'envelope.png', 'sword.png', row, col, x, y)

# use sword item (increases damage attack for character)

def activate(self):

Character.instance.add\_damage(10)

class MapItem(Item):

# initializer

# this initializer is designed sort of like an overloaded method

# it will spawn randomly at start of game

# or the location for item to spawn is specified when enemy is killed

def \_\_init\_\_(self, row=None, col=None, x=None, y=None):

# to do -- change png image

Item.\_\_init\_\_(self, 'envelope.png', 'map.png', row, col, x, y)

def activate(self):

Character.instance.has\_key = True

**Level.py**

import pygame

from .Enemy import \*

from .Utils import \*

from .Constants import \*

from .Renderable import \*

# Level class is to set the map for the game

class Level(Renderable):

# initialize map

def \_\_init\_\_(self, level):

Renderable.\_\_init\_\_(self, level, False)

self.image = pygame.transform.scale(self.image, SCREEN\_SIZE)

def is\_complete(self):

from .Character import Character

return Character.instance.continue\_level

def get\_json\_file(self):

raise NotImplementedError('Level.get\_json\_file')

def load(self):

level\_json = self.get\_json\_file()

# read file

path = get\_resource\_path(level\_json)

with open(path, 'r') as map\_file:

data = map\_file.read()

# parse file

# find object layer

obj = json.loads(data)

collision\_layer = None

for layer in obj['layers']:

if layer['name'] == 'Objects':

collision\_layer = layer

break

# if object layer exists

# set width and height

# determine row and col of each collision object

if collision\_layer is not None:

Collision.width = collision\_layer['width']

Collision.height = collision\_layer['height']

for i, d in enumerate(collision\_layer['data']):

row = int(i / Collision.width)

col = i % Collision.width

if d != 0:

Collision.colliding\_tiles.add((row, col))

# initilize enemies (random number each time)

for e in range(rand\_range(7, 13)):

#for e in range(rand\_range(1, 5)):

enemies = [Skeleton, Orc]

enemy\_type = random.choice(enemies)

enemy = enemy\_type()

Renderable.add\_renderable(50, enemy)

from .Item import HealthItem, PoisonItem, SwordItem

# randomly initialize items

for i in range(rand\_range(15, 20)):

num = rand\_range(0, 100)

if num <= 40:

health = HealthItem()

Renderable.add\_renderable(51, health)

elif num <= 80:

poison = PoisonItem()

Renderable.add\_renderable(51, poison)

elif num <= 100:

sword = SwordItem()

Renderable.add\_renderable(51, sword)

# first level of game

class Level1(Level):

def \_\_init\_\_(self):

Level.\_\_init\_\_(self, "map\\level\_1.png")

def get\_json\_file(self):

return 'map\\level\_1.json'

# second level of game

class Level2(Level):

def \_\_init\_\_(self):

Level.\_\_init\_\_(self, "map\\level\_2.png")

def get\_json\_file(self):

return 'map\\level\_2.json'

# third level of game

class Level3(Level):

def \_\_init\_\_(self):

Level.\_\_init\_\_(self, "map\\level\_3.png")

def get\_json\_file(self):

return 'map\\level\_3.json'

**LevelManager.py**

from .Level import \*

from .Collision import \*

from .Renderable import \*

class LevelManager(Renderable):

instance = None

def \_\_init\_\_(self):

Renderable.\_\_init\_\_(self, None)

self.active\_level = None

self.all\_levels = [Level1, Level2, Level3]

def render(self, screen):

from .Character import Character

if (self.active\_level is None or self.active\_level.is\_complete()) and len(self.all\_levels) > 0:

if self.active\_level is not None:

self.active\_level.remove()

Character.instance.has\_key = False

Character.instance.had\_key = False

Character.instance.continue\_level = False

self.clean\_level()

self.active\_level = self.all\_levels.pop(0)()

self.active\_level.load()

Renderable.add\_renderable(1, self.active\_level)

Character.instance.set\_position(0, 0)

Character.instance.row = 0

Character.instance.col = 0

def clean\_level(self):

Collision.empty\_all()

def has\_remaining\_level(self):

return len(self.all\_levels) > 0

**Renderable.py**

import pygame

from .Utils import \*

from .Constants import \*

# Renderable is a class used to render objects onto the screen

class Renderable:

# renderables to store different renderable items

# and which key to indicate when to render

renderables = {}

# initialize renderable

def \_\_init\_\_(self, image, normalize\_size=True):

# the object image that is passed in could be a string or layer

# if image is a string

if type(image) is str:

self.image = pygame.image.load(get\_resource\_path(image))

if normalize\_size:

self.image = pygame.transform.scale(self.image, ((TILE\_SIZE, TILE\_SIZE)))

else:

# if image is layer

self.image = image

# default position

self.x\_position = 0

self.y\_position = 0

self.removed = False

# show render item on screen

def render(self, screen):

screen.blit(self.image, (self.x\_position, self.y\_position))

# move render item

def move(self, x, y):

self.x\_position += x

self.y\_position += y

# set position of render item

def set\_position(self, x, y):

self.x\_position = x

self.y\_position = y

# get coordinates of current render item

def get\_coordinates(self):

return (int(self.y\_position / TILE\_SIZE), int(self.x\_position / TILE\_SIZE))

def remove(self):

self.removed = True

# static method

# add render items to list based on key

# key will be a number specified in \_\_init\_\_

# higher number = outer layer

@classmethod

def add\_renderable(cls, depth, renderable):

renderable\_layer = cls.renderables.get(depth, [])

renderable\_layer.append(renderable)

cls.renderables[depth] = renderable\_layer

# static method

# get list of renderables

@classmethod

def get\_renderables(cls):

# sort the renderables

keys = list(cls.renderables.keys())

keys.sort()

all\_renderables = []

# loop each key

for key in keys:

# put into temp variable

renderable\_layer = cls.renderables[key]

current\_renderables = []

for renderable in renderable\_layer:

# put each renderable into the list

# if removed trait is set to false

if not renderable.removed:

current\_renderables.append(renderable)

all\_renderables.append(renderable)

cls.renderables[key] = current\_renderables

return all\_renderables

@classmethod

def show\_win(cls, screen):

# render text onto center screen

win\_display = 'YOU WIN.'

win\_text = BIG\_FONT.render(win\_display, True, RED\_COLOR, BEIGE\_COLOR)

win\_text\_rect = win\_text.get\_rect()

win\_text\_rect.center = CENTER\_POS

screen.blit(win\_text, win\_text\_rect)

@classmethod

def show\_lose(cls, screen):

# render text onto center screen

lose\_display = 'YOU LOSE.'

lose\_text = BIG\_FONT.render(lose\_display, True, RED\_COLOR, BEIGE\_COLOR)

lose\_text\_rect = lose\_text.get\_rect()

lose\_text\_rect.center = CENTER\_POS

screen.blit(lose\_text, lose\_text\_rect)

@classmethod

def show\_complete(cls, screen):

# render text onto center screen

level\_complete\_display = 'LEVEL COMPLETE.'

level\_complete\_text = BIG\_FONT.render(level\_complete\_display, True, RED\_COLOR, BEIGE\_COLOR)

level\_complete\_text\_rect = level\_complete\_text.get\_rect()

level\_complete\_text\_rect.center = CENTER\_POS

continue\_display = 'PRESS ENTER TO CONTINUE.'

continue\_text = REG\_FONT.render(continue\_display, True, RED\_COLOR, BEIGE\_COLOR)

continue\_text\_rect = continue\_text.get\_rect()

continue\_text\_rect.midtop = level\_complete\_text\_rect.midbottom

screen.blit(level\_complete\_text, level\_complete\_text\_rect)

screen.blit(continue\_text, continue\_text\_rect)

**Timer.py**

import time

# Timer class used to track in game time and object rendering

class Timer:

# initializer

def \_\_init\_\_(self):

self.start\_time = 0

self.expired\_time = 0

# start the timer

def start(self, timeout):

self.start\_time = time.clock()

self.expired\_time = self.start\_time + timeout

# determine if time has expired

def is\_expired(self):

current\_time = time.clock()

return current\_time >= self.expired\_time

# get the time that has elapsed

def get\_time\_elapsed(self):

current\_time = time.clock()

return current\_time - self.start\_time

**Utils.py**

# functions used in multiple files

import os

import random

from .Constants import \*

from . import resources

import importlib.resources as pkg\_resources

# find the path location for a resource and return it

def get\_resource\_path(relative\_path):

abs\_path = ''

#print(relative\_path)

this\_dir, this\_filename = os.path.split(\_\_file\_\_)

DATA\_PATH = os.path.join(this\_dir, "resources", relative\_path)

return DATA\_PATH

with pkg\_resources.path(resources, relative\_path) as path:

abs\_path = str(path.absolute())

return abs\_path

# limit val in given range

# used to prevent objects from moving off screen

def clamp(num, min\_value, max\_value):

return max(min(num, max\_value), min\_value)

# round a number based on the multiple

def round\_whole(number, multiple):

return number - (number % multiple)

# get a random range based on 2 inputs

def rand\_range(min, max):

return int(min + (random.random() \* (max - min)))

def play\_sound(file, volume=None):

sound = pygame.mixer.Sound(get\_resource\_path(file))

if (volume is not None):

sound.set\_volume(volume)

else:

sound.set\_volume(0.5)

sound.play()

*Resources*

|  |  |
| --- | --- |
| **File** | **Usage** |
| \_\_init\_\_.py | Enable grabbing resources |
| Character.png | Character renderable images |
| envelope.png | Envelope item image |
| Hair-Gel-By-Paul-Piller.mp3 | Background music |
| heart.png | Health item image |
| map.png | Map item image |
| Orc.png | Enemy: Orc renderable images |
| potionGreen.png | Poison item image |
| scroll.png | Icon image |
| Skeleton.png | Enemy: Skeleton renderable images |
| win.wav | Sound effect: win or level completed |
| sword.png | Damage increase item image |
| map/ | Folder |
| map/ClassicRPG\_Sheet.png | Map tile set |
| map/ClassicRPG\_Sheet.tsx | Map tile set rendered in Tiled application |
| map/level\_1.json | Level 1 map meta |
| map/level\_1.tmx | Level 1 map design |
| map/level\_1.png | Level 1 image render |
| map/level\_2.json | Level 2 map meta |
| map/level\_2.tmx | Level 2 map design |
| map/level\_2.png | Level 2 image render |
| map/level\_3.json | Level 3 map meta |
| map/level\_3.tmx | Level 3 map design |
| map/level\_3.png | Level 3 image render |
| OGG/ | Folder |
| OGG/chop.ogg | Sound effect: attack |
| OGG/cloth2.ogg | Sound effect: item activation |
| OGG/creak2.ogg | Sound effect: lose |
| OGG/creak3.ogg | Sound effect: lose |

*Game Flow Chart*



*Classes*



