Contents

1 introduction

2 Basic idea

A dungeon crawler set in a modern day office space (with cubicles, cantinas etc.) instead of fighting I need another way of handling engagement.

3 Actor ACTOR

An actor is anything that is supposed to have sentience and behaviour ingame.

3.1 "Races" RACES

- Zzzombie - the constantly sleepy coworker, never seems to be quite awake nor have they had a good nights sleep in their life.
- Gobbo - The coworker who handles mails, you're not sure if they've ever had a shower, but the greasy strands of hair makes you think not.
- Creeps -
- Emotional vampire
- Ghost

- Troglodytes

3.2 Classes

- Financial
- IT Wizard
- Assistant
- Intern
- Secretary
- Janitor
- Manager
- HR
- Boss

3.3 Stats

Stats in an RPG is like... something... really... hmm (ed: come back with a zinger here). Usually the stats in a traditional roguelike might be something akin to - Constitution, Strength, Wisdom, Intelligence etc. basically if you've seen a rulebook for a TTRPG you can probably recognize most of these AND THEY WORK! that's why people use them, they really REALLY work. I'm not choosing differently because I think I can do better, but because, well since the game is already set in a modern office environment, there's little reason to not have some weird quirk with the stats again.

3.3.1 List of stats to consider:

- Neuroticism: how panicked and alert a player is about their surrounding.
- Openness: how open they can allow themself to be.
- Reasoning: how good they are at logically deducting things, (maybe reasoning and neuroticism can kindda benefit and be a detriment at the same time)

- Madness: How mad the character is (remember! you don't have to be mad to work here, but it helps!)
- Agreeableness: How much other characters like you.
- Laziness: How much (or little :/) energy a character has.

I Really REALLY wanted to put "Agility" into this stat list, but not agility as it is traditionally used, but more as in the agile method that is usually claimed by every company under the sun that has too much money and not enough sensible managers/CEOs. Though I feel like that'll be an unnescessary dig at alot of people I have tremendous respect for. (though maybe they don't care what a nobody on the internet does or say... I dunno :))

So that's the N.O.R.M.A.L. stat system (you'd think I'd have picked these words carefully... and you'd be right) patent pending yadda yadda.

3.3.2 In game stats

Health

- Phys-Health
  - is made out of Laziness (the lower the laziness the higher the health), Openness
- Mental-Health
  - is made out of Madness, Reasoning.
- Social-Health
  - Agreeableness, Neuroticism
- HR-cana
  - Is defined by Madness, Agreeableness
-

3.4 Spells

A magic system wouldn't really work in a contemporary world. There is however something more arcane and mysterious that I can employ which is: HR guidelines.

3.5 HR Guidelines

The HR guidelines should, like all magic systems. Affect stats in some way.

4 Code

How neat would it be if we could quit here? alas now we have to try and code this :S Ah well into the fray.

```
from __future__ import annotations
from abc import ABC, abstractmethod
import pygame
import pprint
from pygame.locals import *
import numpy
from collections.abc import Callable
from functools import reduce, cache, wraps, partial
import json
from dataclasses import dataclass, field
import operator
import rx
```

5 Globals

Global variables are frowned upon by virtually everyone and their mom, which is fair, I don't like them either, but I don't have a better idea.

Global variables are the following

- cell size, i.e. how large the cell that fills the screen surface should be. Then I can later just divide the screenwidth or screenheight of the drawing destination surface by the cell size to know how big the are is. (so I don't draw outside the screen/surface), this is also where I'm placing the filenames for fonts which I might need/want to change.

```
CELLSIZE = 32
FONTFILE = "terminal8x8_gs_ro.png"
```

6 Positional handling POSITION

Positional functions all tackles different aspect of onscreen positioning. Things lige moving, checking collisions. multiplying a vector tuple so that it uses the correct

6.1 Cell function — **Coordinate**

The cell function handles converting the direction vectors into coordinates that can be used on the screen

```
def Cell(cell : (int, int)) -> (int, int):
    return tuple(map(lambda n : n * CELLSIZE, cell))
```

6.2 Collision — **Collision**

The collision function gets two position tuples and checks if they are the same, if they are it returns true if not it returns false

```
def collision(xy, _xy):
    return xy == _xy
```

6.3 Move — **Update**

Move function is just meant to take tqwo positional arguments, the current position and the destination, and return a new tuple with the new current position. I believe It **could** theoretically maybe, potentially handle diagonal movement - ish but this is just you grandmas 4 directional moves.

```
def move(xy : (int,int), _xy : (int, int)) -> (int, int):
    acc = []
    for n, _n in zip(xy, _xy):
        if n > _n:
            n = n - 2
        elif n < _n:
            n = n + 2
        acc.append(n)
    return tuple(acc)
```

6.4 Vector — **Datacontainer::Coordinate**

The vector class is basically a copy of a tuple datatype, however it has the added "add" and "sub" methods defined. I don't really need (at the moment) to handle more "advanced" features like vector dot product and so on.

```
@dataclass(frozen=True)
class Vector:
    x : int
    y : int
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)
```

7 Input IO

7.1 Input player movement

for now I just handle the input through a simple function that checks whether or not a valid key has been pressed. if not it returns a (0,0) vector.

```
def getInput(ev):
    inputList = { pygame.K_UP : (0,-1),
                  pygame.K_DOWN : (0,1),
                  pygame.K_LEFT : (-1, 0),
                  pygame.K_RIGHT : (1, 0),
    }
    return inputList.get(ev.key, (0,0))
```

7.2 Ressource handler File

Another euqually import (or more important input) is the different ressources. For now it is only a config json file and a tilesheet, I'm interested in, but it could expand to more tilesheets, or even premade levels (or templates). The ressources are being loaded into a file class that holds the different information.

```
def loadFiles() -> ConfigFile:
    _fontImage = pygame.image.load("terminal8x8_gs_ro.png")
    _fontImage.set_colorkey((0,0,0))
    with open('conf.json', 'r') as _file:
        config = json.load(_file)
    return ConfigFile(_fontImage, config)
```

7.2.1 File class DATACONTAINER

The file class is an immutable data container that holds the information needed for the game to function

```
@dataclass(frozen=True)
class ConfigFile:
    _image : pygame.Surface
    _conf : dict
    def image(self) -> pygame.Surface:
        return self._image
    def config(self) -> dict:
        return self._conf
```

7.2.2 Tilesheet class DATACONTAINER

```
@dataclass(frozen=True)
class TileSheet:
    _tiles = [pygame.Surface]
    def __init__(self,
                 file : pygame.Surface,
                 width : int,
                 height : int,
                 rows : int,
                 columns : int):
        for x in range(rows):
            for y in range(columns):
                self._tiles.append(
                    pygame.transform.scale(
                        file.subsurface(y * width, x * height, width, height),
                        (CELLSIZE, CELLSIZE)
                    )
                )
```

8 Time

time where did you go?.... aaaand that's all I had time for

In the game there's going to be two kinds of time. Firstly a realtime module that just makes sure we don't call the draw function more than 60 times a second (at most) because let's be honest... it's a text based engine so it's kindda dumb to blow up the GPU for **that**

But the second kind of time is the ingame time. That's based on turns. Every turn everything that can (and should) move, moves.

8.1 Turn DATACONTAINER

The turn class is a data container class (get used to be seeing those around btw.) that takes care of storing whatever information a turn need.

```
@dataclass(frozen=Turn)
class Turn:
    turn : int
```

9 Window handler

The window class is where the initializing is going to happen, as well as where the pygame window.

```
class Window():
    _size = (int, int)
    _surface = pygame.Surface
    def __init__(self, width, height) -> None:
        self._surface = pygame.display.set_mode((width, height), 0, 32)


    def surface(self) -> pygame.Surface:
        return self._surface
```

10 View MIGHT BE REMOVED

The view holds a surface and a pygame.Rect. The rect is moved around to "slice" a subsurface from the map.

```
class View:
    _camera : pygame.Rect
    _trackingObject = None
    def __init__(self,
                 topx : int,
                 topy : int,
                 cameraWidth : int,
                 cameraHeight : int,
                 trackingObject = None):
        self._camera = pygame.Rect(topx, topy, topx+cameraWidth, topycameraHeight)
        if trackingObject is not None:
            self._trackingObject = trackingObject

    def slice(self, surface : pygame.Surface):
        return surface.Subsurface(self._camera)

    def trackObject(self, surface : pygame.Surface):
        pass

    def checkCenterObject(self):
        pass

    def update(self) -> pygame.Surface:
        pass
```

The view function takes the relevant actor and centers the map on it.

```
def View(actor : Actor,
         surface : pygame.Surface,
         view : pygame.Rect) -> pygame.Surface:
    _view = view
    _view.center = actor.currxy()
    return surface.subsurface(_view)
```

11 Drawing Onscreen

11.1 Drawmap Map

Drawmap function is only called to draw the surface of the static map.

```
def drawMap(map : str,
            pos : [(int, int)],
            tiles : [pygame.Surface],
            destination :pygame.Surface):
    _drawingList : [(pygame.Surface,(int,int))] = []
    x = 0
    y = 0
    for ind, c in enumerate(map):
        _drawingList.append((tiles[ord(c)+1], pos[ind]))
    destination.blits(_drawingList)
```

11.2 Drawing Characters

The drawing function takes one or more characters and draws them to the screen.

```
def drawing(chars : str,
            pos : [(int, int)],
            tiles : [pygame.Surface],
            destination :pygame.Surface):
    _drawingList : [(pygame.Surface,(int,int))] = []
    x = 0
    y = 0

    for ind, c in enumerate(chars):
        _drawingList.append((tiles[ord(c)+1], pos[ind]))
    destination.blits(_drawingList)
```

12 Map Map

the map is for now just a container class for a premade "dungeon", this is to test whether or not the drawing function can handle the sheer drawing calls. AND that it can handle the various characters.

It's supposed to just have a giant, static (more or less static) image of the map.

```
class Map:
    _str : str
    _pos : [(int,int)] = []
    _map : pygame.Surface
    def __init__(self, tiles):
        # TEMP map
        self._str = ""
        tempMap = [ "###############################",
                    "#       #             #        #",
                    "#       #             #        #",
                    "#       #             #        #",
                    "#       #             #        #",
                    "#                              #",
                    "#                              #",
                    "#           y                  #",
                    "#      Hello                   #",
                    "#       p                      #",
                    "#                              #",
                    "#                              #",
                    "#                              #",
                    "#                              #",
                    "#                              #",
                    "################################"
        ]
        w = len(tempMap[0]) * CELLSIZE
        h = len(tempMap) * CELLSIZE
        self._map = pygame.Surface((w, h))

        for i, s in enumerate(tempMap):
            self._str = self._str + s
            for string_i, _ in enumerate(s):
                self._pos.append((string_i*CELLSIZE, i*CELLSIZE))
        drawMap(self._str, self._pos, tiles, self._map)

    def map(self):
        return self._map
```

13 Actor list **ACTOR**

The actor list is where a list of actors are being created.

```
def makeActors(amount : int, playerIndex : int) -> [Player]:
    pass
```

14 Game Loop function

The Game loop is where the structure of the game is at.

```
def GameLoop(window, _map, tiles):
    running = True
    player = Actor('@',(32,32), (32,32))
    currVec = player.currxy()
    nxtVec = player.nxtxy()
    log = []
    cl = pygame.time.Clock()
    while(running):
        movVec = (0,0)
        for event in pygame.event.get():
            match event.type:
                case pygame.QUIT:
                    running = False
                case pygame.KEYDOWN:
                    movVec = getInput(event)
        if player.arrived():
            nxtVec = tuple(map(lambda n, _n: n + ( _n * CELLSIZE), nxtVec, movVec))
        currVec = move(player.currxy(), nxtVec)
        player = Actor( '@',currVec, nxtVec)
        window.surface().blit(_map.map(), (0,0))
        drawing(str(player), [player.currxy()], tiles._tiles, window.surface())
        pygame.display.flip()
        log.append("current vector : " + str(currVec) + "nxtVector : " + str(nxtVec) +
        cl.tick_busy_loop(60)
    pprint.pprint(_map.map())
```

15 TODO Text Text

15.1 TODO Textprinting Onscreen

15.2 TODO Textformatting

15.3 TODO Textinput

16 Actor Actor

The player class is, for now just a place holder, keeping the player char (literally a char value), position and that's it. It does also use a couple of standard operators that I've overloaded to return just a string.

16.1 Actor class — Datacontainer

```
@dataclass(frozen=True)
class Actor:
    _c : chr = field(init=True)
    xy : (int, int) = field(init=True)
    _xy : (int, int) = field(init=True)
    def arrived(self) -> bool:
        return self.xy == self._xy

    def currxy(self):
        return self.xy

    def nxtxy(self):
        return self._xy

    def __repr__(self):
        return self._c
```

16.2 Update Actors

The update actor function is going to map onto the list of actors in the game.

```
def updateActor(lstOfActors : [Actor]) -> [Actor]:
    # retLst = []
    # #check collision
    # lstCollision
    # for nxt in lstOfActors:
    #     lstCollision.append(nxt.
    # for actor in lstOfActors:
    #     currVec = move(actor.currxy(), actor.nxtxy())
    pass
```

17 Main

In the main function I initialize the different components, like the window, the gameloop function, etc.

```
def main():
    pygame.init()
    pygame.font.init()
    # -------
    _files = loadFiles()
    _tiles = TileSheet(_files.image(), 8, 8, 16, 16)
    window = Window(800, 600)
    map = Map(_tiles._tiles)
    GameLoop(window, map, _tiles)
    # -------
    pygame.quit()

if __name__=="__main__":
    main()
```