
THE LEGEND OF PYZELDA:
BREATH OF THE SNAKES

Введение

Привет, читатель! Прежде чем создать свою игру на питоне, мне бы хотелось обсудить несколько вопросов:

1. «Как человеку заняться геймдевом?»
2. Зачем писать игру на питоне, если есть unity и unreal engine?

Обычно есть два пути, как человек попадает в геймдев:

1. Умышленный путь. Человек, играющий в игры, увлекается ими настолько, что у него возникает желание их создавать. Тогда он выбирает, где и как ему учится. Он учится работать с игровыми движками и изучает C-образные языки.
2. Путь программистов. Так или иначе везде, где есть код — есть компьютер. Цифровые развлечения, в том числе игры, не проходят мимо тех людей, которые постоянно пишут код. Поэтому некоторым программистам становится интересно попробовать свои силы в разработке собственной игры. Об этом пути мы и поговорим.

Конечно можно сказать, что если ты хочешь работать — иди учись и получай нужные навыки. Осваивай Unity или Unreal и работай в крутой геймдев-компании. В этом есть доля правды. Однако, превращать любое своё хобби в работу — дело не благоразумное. Выгорание скажет об этом достаточно понятно. Тогда вам могут предложить сделать либо мод на игру (например, на движке Source) или поиграть в Грёзы, которая тоже очень популярна. Такой вариант вполне подойдёт, но появляется новая проблема: "Выбор варианта работы".

Выбор варианта работы

Есть три варианта работы над игрой:

1. Создать игру на известном движке-конструкторе
2. Сделать мод на существующий движок
3. Создать свой движок и написать игру

Каждый из этих вариантов хорош по-своему, но и имеет ряд недоработок. Например, выбрав путь создания игры на популярном движке вы столкнётесь с проблемами самого движка, а главное — с проблемами монетизации проекта. Если выбрать модопуть — вы сразу столкнётесь с тем, что не каждый движок открыт для написания игр, а также, не каждый движок позволяет продавать мод хоть за какую-то цену. Путь написания своего движка вызывает в геймдеве дикий припадок, мол "Да зачем опять идти очень сложно, когда сроки горят, а сам ты ещё не доработал концепцию?". Данный путь весьма не популярен для общества из-за сложности реализации и проблем с набором хоть какой-то команды, ведь если вы пишете движок, кому-то с ним работать.

Так зачем выбирать "сложный" путь? Во-первых, не для всех он сложнее. Лично мне сложнее перейти на C# с использованием Unity, чем написать новый проект на Python. Во-вторых, люди не всегда понимают сам движок. Сейчас я преподаю в институте "Бизнеса и Дизайна". Это одно из первых профильных учреждений, где обучают геймдеву. Я спросил у студентов: «зачем им движок?». Каково было моё удивление, что далеко не все понимаю, вообще зачем он нужен. Спойлер, не из-за пресетов.

Зачем вам движок?!

По сути, движок решает всего лишь три, но очень важных вопроса:

1. **Физика**. Если вы окончили физмат, вы понимаете насколько сложно определить и прописать траекторию падения стеклянного шара весом в 2 килограмма на доску из дуба, которая лежит под углом 32 градуса. Также, нужно не забыть прописать с какой высоты должен упасть шар, чтобы разбиться. Движок прописывает эту физику за вас, а вы лишь обращаетесь к тем или иным объектам.
2. **Тайлсеты**. Тайл — это маленькие картинки для прорисовки графики (чаще всего, карт). Процесс создания тайлов весьма энергозатратный, а наложение этих тайлов достаточно простое дело. Просто выбрать где лежит камень, а где земле или вода. В движках типа Unreal или Unity можно выбрать слой карты и нарисовать клеточками из тайтлов карту.
3. **Анимация**. Самый спорный пункт. В движках все элементы на карте — это объекты. Каждый объект имеет форму по координатам x, y, z и их можно менять прямо в самом движке путём перемещения, сжатия или растягивания объекта. Но важное уточнение, прорисовывать анимацию по x, y, z — не самая лучшая идея. Куда надёжнее, красивее и вообще правильнее прописывать каждый шаг объекта путём перемещения по пресету персонажа по графике.

Иными словами, сам игровой движок — это автоматизирующая система, которая позволяет упростить разработку. Однако, при использовании движка теряется часть гибкости в настройках. Моя задача — описать как можно более подробный процесс написания игры по самому сложному пути, хотя Python и упрощает часть вещей.

Я решил писать игру на языке Python, потому что я его хорошо знаю и он для меня удобен. Я обожаю игры, но выучивать новый язык только для того, чтобы написать игру - на мой взгляд, очень муторно. Своим примером я хочу показать, что нет ничего невозможного. Даже с питона можно начать свой путь в игровую индустрию!

Эта книга будет интересна всем, кто начал своё знакомство с языком Python. В книге будет объясняться все вопросы, которые затрагивают игровые механики. Минимальны порог входа для прочтения данной книги — пройденный курс "Поколение Python": курс для начинающих.

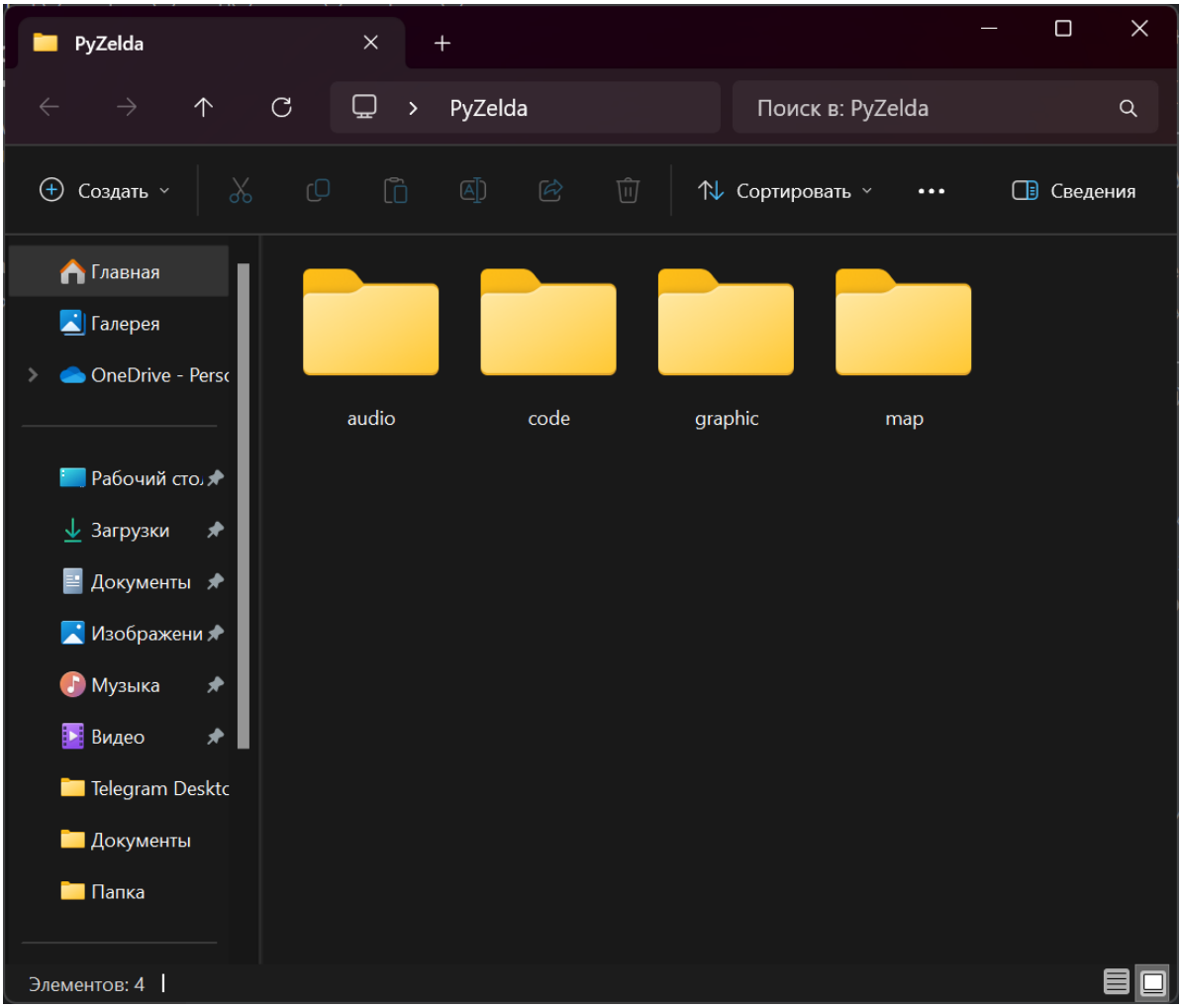
Автор книги - Валерий Линьков: более 8 лет в IT-образовании от сетей до змей!

Редактор - Анастасия Линькова: более 7 лет объясняет сложные вещи простыми словами.

Если вы нашли ошибки - напишите на почту admin@montirovka.com . Пришлем вам презент за вашу внимательность :)

Подготовка

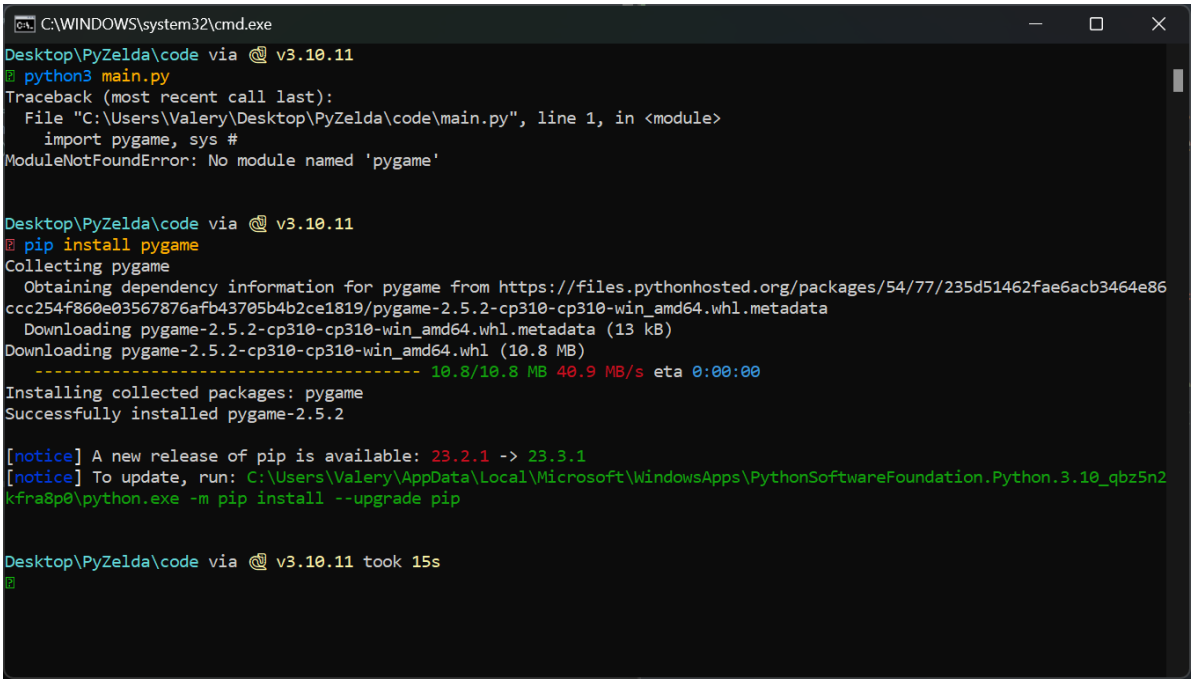
Сначала создадим папку проекта. Я назвал проект PyZelda и в папке проекта есть ещё 4 директории: audio, code, graphic и map.



Далее в папке code создадим три файла:

- main.py — основной файл игры
- settings.py — настройки игры: тут мы укажем настройки полей и основных данных по окну и прорисовке.
- debug.py — файл дебагинга

Перед работой, не забудьте скачать библиотеку PyGame (`pip install pygame`)



Файл main:

```
import pygame, sys #импортируем библиотеки PyGame и Sys
from settings import * #импорт из файла settings

class Game: #основной класс игры
    def __init__(self): #создаём конструктор класса
        pygame.init() #конструктор использует конструкции из библиотеки PyGame
        self.screen = pygame.display.set_mode((WIDTH,HEIGTH)) #забирает из
нашего проекта экран в виде размеров в ширину и высоту
        pygame.display.set_caption("PyZelda") #Устанавливаем название нашего
окна
        self.clock = pygame.time.Clock() #а также, забирает из проекта время

    def run(self): #функция запуска игры
        while True: #до выхода из игры она активна
            for event in pygame.event.get(): #просмотр событий в игре
                if event.type == pygame.QUIT: #сейчас мы можем только выйти и
при выходе:
                    pygame.quit() #вызываем метод закрытия игры
                    sys.exit() #и закрываем окно системы
            self.screen.fill('green') #помимо событий, указываем цвет экрана
            pygame.display.update() #обновляем экран
            self.clock.tick(FPS) #запрашиваем FPS

if __name__ == '__main__': #запуск игры только из main-файла
    game = Game() #Если файл main, то сама игра вызывает класс...
    game.run() #...и запускает функцию run из класса
```

В файле main мы запускаем игру. Создаём конструктор на базе PyGame и работаем с входными файлами и данными.

Файл setting.py:

```
WIDTH = 1280 #Ширина экрана
HEIGTH = 720 #Высота экрана
```

```
FPS = 60 #Число FPS
TILESIZE = 64 #Размер тайла (квадрата текстуры)

WORLD_MAP = [ #Карта мира игры. Она состоит из 20x20 квадратов
['x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x',
'x', 'x', 'x', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',', ',',
',', ',', ',', 'x'],
['x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x',
'x', 'x', 'x', 'x'],
]
```

В settings.py будут все настройки игры. Стандартные ширина и высота окна, а также число FPS (частота обновления окна) и самая интересная часть — размер тайла. Как я говорил ранее, тайл — это квадрат текстуры и именно этот квадрат мы описываем размерами. Высота и ширина кадра измеряется в пикселях (я установил 1280 на 720), как и сам тайл, но есть важный момент — тайл всегда квадратный. Поэтому, мы прописываем размер квадрата тайла (он у нас 64 на 64 пикселя). Далее мы создаём карту мира. Сейчас она 20 на 20 тайлов (далее в проекте будем рисовать всё тайлами). Я обозначил, что `'x'` — это конец карты, а `','` — пустое место для игры.

Далее файл debug.py:

```
import pygame #Снова обращаемся к PyGame
pygame.init() #Используем базовый конструктор
font = pygame.font.Font(None, 30) #Указываем шрифт

def debug(info, y = 10, x = 10):  #Сама функция дебага
    display_surface = pygame.display.get_surface() #Получаем ссылку на текущую установленную поверхность отображения в игре
    debug_surf = font.render(str(info), True, "Red") #Рендерим текст (я сделал его красным)
    debug_rect = debug_surf.get_rect(topleft = (x, y)) #Указывамем место отображения инфы на экране (левый верхний угол)
    pygame.draw.rect(display_surface, "White", debug_rect) #Делаем микро-консоль в виде прямоугольника (и да он белый, будет белая консоль с красным текстом)
    display_surface.blit(debug_surf, debug_rect) #Собираем нашу микро-консоль с параметрами текста
```

В этом файле мы создадим дебаг игры. Это поле, которое мы выводим вверху слева экрана, и в нём показываем нужный нам текст. Для запуска дебага нужно импортировать debug в main-файл и вывести что-то после отрисовки окна. Примерно так (файл main.py):

```
import pygame, sys #импортируем библиотеки PyGame и Sys
from settings import * #импорт из файла settings
from debug import debug #импортируем дебаг <------------------

class Game: #основной класс игры
    def __init__(self): #создаём конструктор класса
        pygame.init() #конструктор использует конструкции из библиотеки PyGame
        self.screen = pygame.display.set_mode((WIDTH,HEIGTH)) #забирает из нашего проекта экран в виде размеров в ширину и высоту
        pygame.display.set_caption("PyZelda") #Устанавливаем название нашего окна
        self.clock = pygame.time.Clock() #а также, забирает из проекта время

    def run(self): #функция запуска игры
        while True: #до выхода из игры она активна
            for event in pygame.event.get(): #просмотр событий в игре
                if event.type == pygame.QUIT: #сейчас мы можем только выйти и при выходе:
                    pygame.quit() #вызываем метод закрытия игры
                    sys.exit() #и закрываем окно системы
            self.screen.fill('green') #помимо событий, указываем цвет экрана
            debug('Чё как? Я дебаг!') #Дебажим <------------------
            pygame.display.update() #обновляем экран
            self.clock.tick(FPS) #запрашиваем FPS

if __name__ == '__main__': #запуск игры только из main-файла
    game = Game() #Если файл main, то сама игра вызывает класс...
    game.run() #...и запускает функцию run из класса
```

Получим феерическую игру цветов:

Стартовый набор файлов для скачивания.

Пишем уровень

Для начала нужно разобраться в понятии "класс" в концепции объектно-ориентированного программирования - ООП. Сейчас мы создадим класс `Level`. В этом классе есть всё необходимое для работы с объектами внутри. Класс `Level` и есть наша игра, так как именно этот класс собирает все **спрайты** (игрок, враги, карта мира и т.д.). Помимо самих спрайтов нас интересует набор элементов, связанных с подсчётом тех или иных пунктов. Как я говорил ранее, движок нужен для прописывания физики, а физика — это математический способ описать поведение объекта. Таким образом, мы взаимодействуем с **числами** и **параметрами**. Это тоже находится в классе `Level`. Например, если игрока ударит враг, то игрок потеряет 5% здоровья — это и есть наши параметры.

Но есть одна проблема – объектов очень много. Чтобы не искать в коде конкретный камень №67, будем разбивать код группами объектов. Для этого создадим две группы: `visible_sprites` и `obstacle_sprites`. Как вы уже поняли, группа `visible_sprites` отвечает за все объекты, которые пользователь видит (например, карту мира и персонажа), а `obstacle_sprites` — это объекты, которые нужны для технического использования. Все объекты можно сослать на конкретные группы или на набор групп. Так, объект "кот" может быть в группе объектов движущихся, но не взаимодействующих с игроком. Если же добавить кота в обе группы, он будет отображён и активен.

Приступим к коду. Создадим новый файл `level.py`. Пропишем в нём следующий код:

```
import pygame

class Level: #Создали класс
    def __init__(self): #Базовые параметры
        self.visible_sprites = pygame.sprite.Group() #Создали группу видимых
элементов
        self.obstacle_sprites = pygame.sprite.Group() #Создали группу
технических элементов

    def run(self): #Создали метод в классе
        pass
```

Мы создали класс и в нём 2 группы, а затем в классе создали метод вызова данного класса. Далее нужно вернуться в main-файл и импортировать класс `Level` (`from level import Level`), объявить класс `Level` (`self.level = Level()`), а таже запустить его в самом run-методе в main (`self.level.run()`). Я решил пока убрать дебаг из проекта, так как сейчас дебажить нечего и финальный код в main-файле выглядит так:

```
import pygame, sys #импортируем библиотеки PyGame и Sys
from settings import * #импорт из файла settings
from level import Level #+++ импорт из файла level класс Level +++

class Game: #основной класс игры
    def __init__(self): #создаём конструктор класса
        pygame.init() #конструктор использует конструкции из библиотеки PyGame
        self.screen = pygame.display.set_mode((WIDTH,HEIGTH)) #забирает из
нашего проекта экран в виде размеров в ширину и высоту
        pygame.display.set_caption("PyZelda") #Устанавливаем название нашего
окна
```

```
        self.clock = pygame.time.Clock() #а также, забирает из проекта время
        self.level = Level() #+++ объёвили Level +++

    def run(self): #функция запуска игры
        while True: #до выхода из игры она активна
            for event in pygame.event.get(): #просмотр событий в игре
                if event.type == pygame.QUIT: #сейчас мы можем только выйти и
при выходе:
                    pygame.quit() #вызываем метод закрытия игры
                    sys.exit() #и закрываем окно системы
            self.screen.fill('green') #помимо событий, указываем цвет экрана
            self.level.run() #+++ запустили функцию run в файле level в классе
Level +++
            pygame.display.update() #обновляем экран
            self.clock.tick(FPS) #запрашиваем FPS

if __name__ == '__main__': #запуск игры только из main-файла
    game = Game() #Если файл main, то сама игра вызывает класс...
    game.run() #...и запускает функцию run из класса
```

Давайте пропишем что-то в классе Level. Таким образом, мы создадим подложку из зелёного фона, а сверху наложим новые объекты. Для этого мы воспользуемся простым методом, как в и в дебаге: `display_surface = pygame.display.get_surface()`. В коде с Level оно выглядит так:

```
import pygame

class Level: #создали класс
    def __init__(self): #базовые параметры
        self.display_surface = pygame.display.get_surface() #создали новый слой
объектов
        self.visible_sprites = pygame.sprite.Group() #создали группу видимых
элементов
        self.obstacle_sprites = pygame.sprite.Group() #создали группу
технических элементов

    def run(self): #создали метод в классе
        pass
```

Сейчас ничего не поменяется, так как мы объявили что будем что-то рисовать, но ничего не нарисовали. Приступим к процессу отрисовки: для этого создадим новый файл `tile.py`.

```
import pygame
from settings import *

class Tile(pygame.sprite.Sprite):
    def __init__(self, pos, groups):
        super().__init__(groups) #наследуем все группы
        self.image = pygame.image.load('../graphic/box.png').convert_alpha()
#указываем адрес картинки
        self.rect = self.image.get_rect(topleft = pos) #указываем позицию
отрисовки (левый верхний угол)
```

Интересного в этом коде мало. Мы импортируем pygame (снова и снова). Из файла `settings.py` мы берём всю информацию по разметке поля – поэтому мы его оставляем здесь. Далее, мы создаём новый класс `Tile`. В нём создаём параметры себя, позицию и группу наследования. Далее, командой `super()` мы включаем наследование. Далее указываем картинки самих предметов и `rect` данных картинок. Из интересного, `convert_alpha()`. Эта функция пайгейма создает новую копию поверхности с желаемым форматом пикселей. Суровая необходимость, чтобы пайгейм понял что мы делаем.

Такой же код создадим в файле `player.py`:

```
import pygame
from settings import *

class Player(pygame.sprite.Sprite):
    def __init__(self, pos, groups):
        super().__init__(groups)
        self.image = pygame.image.load('../graphic/link.png').convert_alpha()
        self.rect = self.image.get_rect(topleft = pos)
```

Отличие данного кода лишь в двух местах:

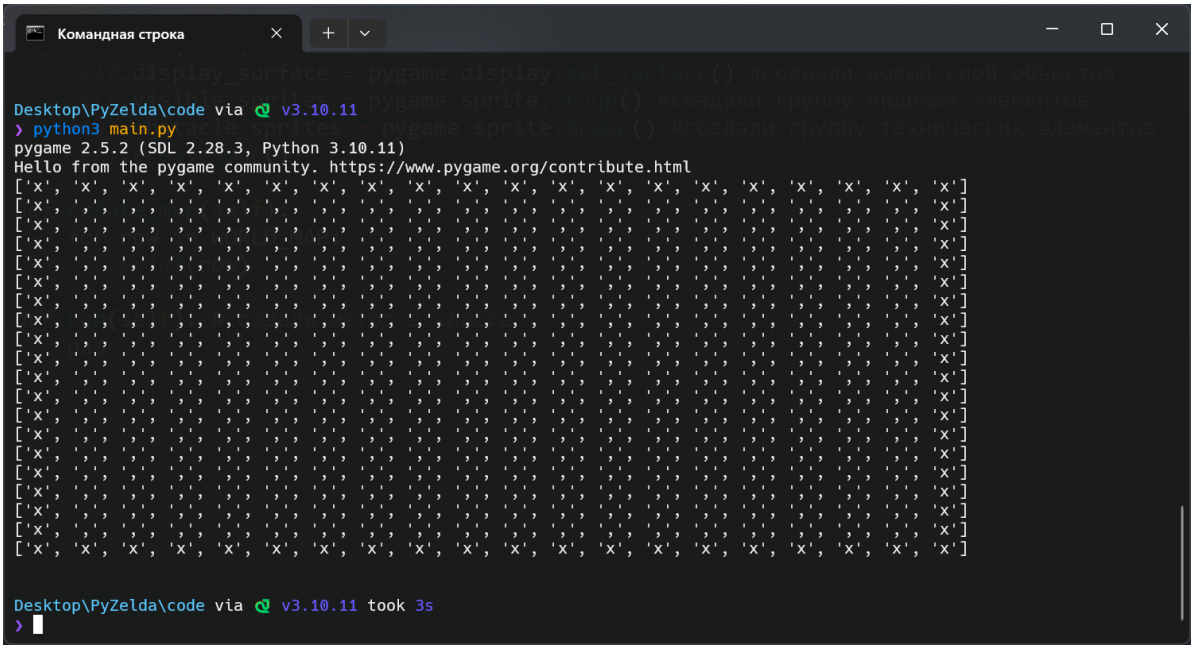
- Я создал класс `Player` вместо Tile
- Заменили картинку с коробок на героя Хайрула :)

Все пресеты картинок я взял из проекта NinjaAdventure. Там есть способ скачать бесплатные сеты графики. Обрезал картики в фотошопе и получил мини-изображения.

Теперь отобразим нашу мировую карту. Делать мы это будем в файле `level.py`, так как именно в нём мы храним всю нужную информацию про конкретный уровень. Сразу импортируем `settings.py` и обратимся к карте. Финал работы будет выглядеть так:

```
import pygame
from settings import *

class Level: #создали класс
    def __init__(self): #базовые параметры
        self.display_surface = pygame.display.get_surface() #создали новый слой
объектов
        self.visible_sprites = pygame.sprite.Group() #создали группу видимых
элементов
        self.obstacle_sprites = pygame.sprite.Group() #создали группу
технических элементов
        self.create_map()

    def create_map(self):
        for row in WORLD_MAP:
            print(row)

    def run(self): #создали метод в классе
        pass
```

После запуска main-файла игра никак не поменялась, но мы в терминале отпечатали карту.

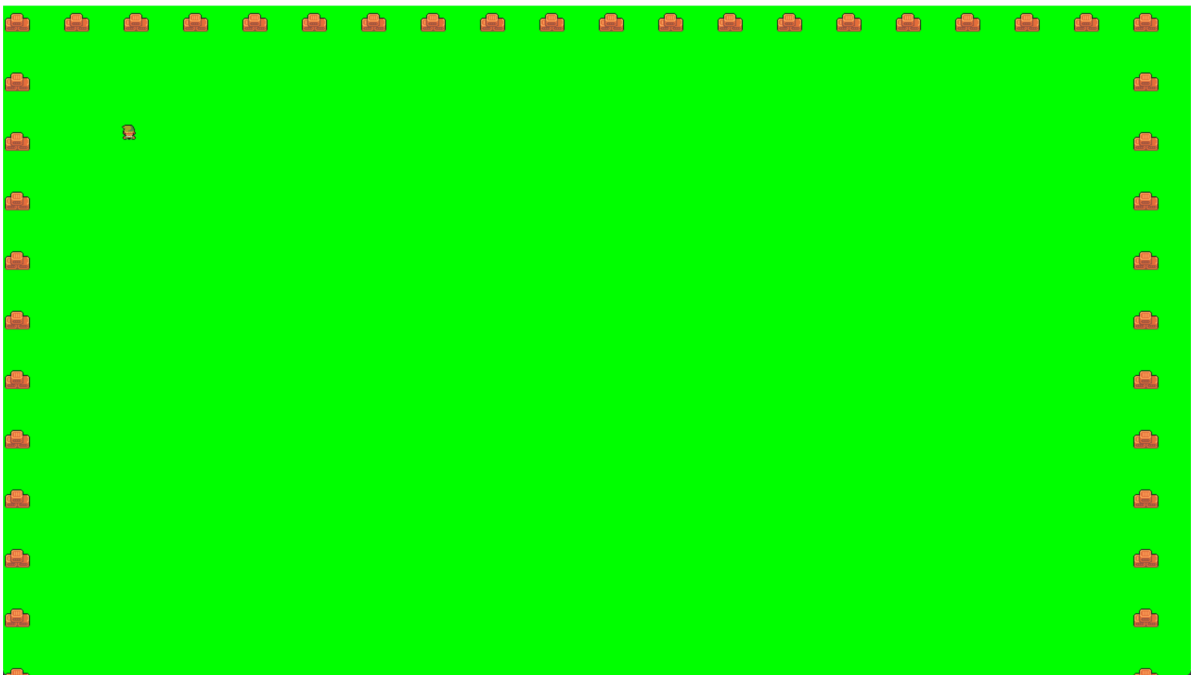


Про карту. Для упрощения работы я создал клетку (напомню, что она размером 64 на 64 пикселя), и указываю символами что куда ставить. Поставил карту я так: 'x' — это коробки, ',' — пустое место, а 'p' — это наш герой.

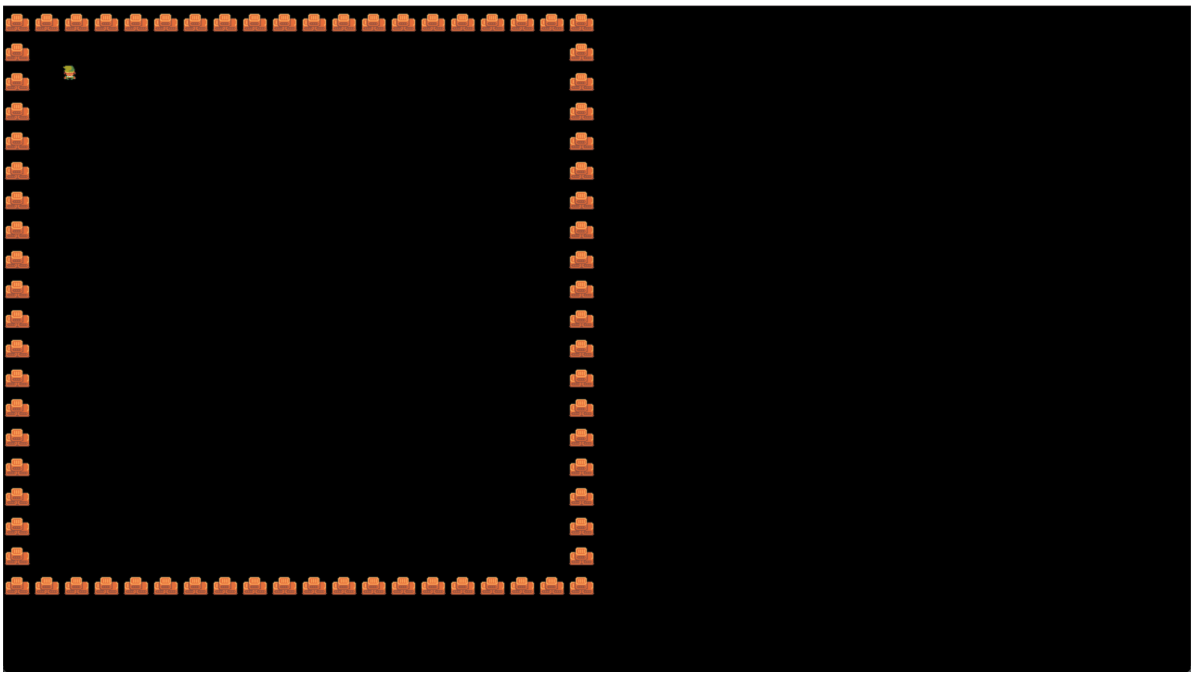
Это, конечно, круто, но вот я хотел бы нарисовать карту в игре, а не в терминале. Для этого, модернизируем код. Работаем только с файлом `level.py`:

```
import pygame
from settings import *
from tile import Tile
from player import Player

class Level: #создали класс
    def __init__(self): #базовые параметры
        self.display_surface = pygame.display.get_surface() #создали новый слой
объектов
        self.visible_sprites = pygame.sprite.Group() #создали группу видимых
элементов
        self.obstacle_sprites = pygame.sprite.Group() #создали группу
технических элементов
        self.create_map()

    def create_map(self):
        for row_index, row in enumerate(WORLD_MAP):
            for col_index, col in enumerate(row):
                x = col_index * TILESIZE
                y = row_index * TILESIZE
                if col == 'x':
                    Tile((x, y), [self.visible_sprites])
                if col == 'p':
                    Player((x, y), [self.visible_sprites])

    def run(self): #создали метод в классе
        self.visible_sprites.draw(self.display_surface)
```

Всё что я делаю — прохожу массив данных и умножаю элемент на 64, так как размер моего тайла равен 64 пикселям. Далее, если элемент равен `'x'` заменяю его на коробку, а если `'p'`, то на героя. Затем я прописал в методе запуска метод отрисовки поля. Также на карте я заменил одну `','` на `'p'`. И результат:



В main-файле, заменив цвет фона на чёрный, а размер тайла на 32 пикселя, я получил это:



Файлы данного этапа для скачивания. Сразу оговорюсь, что отображение не совсем корректное из-за самих картинок. Я не подбивал их к размеру 64 на 64 пикселя. Далее я исправил это и вернул размер тайла в 64 на 64 пикселя.

Создаём игрока

Игрок должен уметь передвигаться и сталкиваться с существующими объектами. Сначала запишем его движения: переходим в `player.py` и прописываем функцию перемещения. Код в `player.py`:

```
import pygame
from settings import *

class Player(pygame.sprite.Sprite):
    def __init__(self, pos, groups):
        super().__init__(groups)
        self.image = pygame.image.load('../graphic/link.png').convert_alpha()
        self.rect = self.image.get_rect(topleft = pos)

        self.direction = pygame.math.Vector2() #обращаемся к направлению через
вектор

    def input(self): #варьируем кнопки
        keys = pygame.key.get_pressed()

        if keys[pygame.K_UP]:
            self.direction.y = -1
        elif keys[pygame.K_DOWN]:
            self.direction.y = 1
        else:
            self.direction.y = 0

        if keys[pygame.K_LEFT]:
            self.direction.x = -1
        elif keys[pygame.K_RIGHT]:
            self.direction.x = 1
        else:
            self.direction.x = 0

    def update(self):
        self.input()
```

Тут с кодом достаточно просто. В функции `self.direction = pygame.math.Vector2()` мы задаём вектор точке. Точка — это наш герой, а его вектор болтается в диапазоне от -1 до 1. Функция `keys = pygame.key.get_pressed()` позволяет продолжать нажатие и тогда, герой должен разгоняться. Если нажатия нет — вектор равен 0 и герой тормозит. Это очень условное, но приписывание инерции.

Ещё в файле `level.py` в функции `run` я записал обновление всех спрайтов. Это строчка кода вида:

```
self.visible_sprites.update()
```

Ещё я переписал в том же файле вызов героя. Мне это нужно для глобализации позиции героя. Из строки

```
Player((x, y), [self.visible_sprites])
```

Я сделал

```
self.player = Player((x, y), [self.visible_sprites])
```

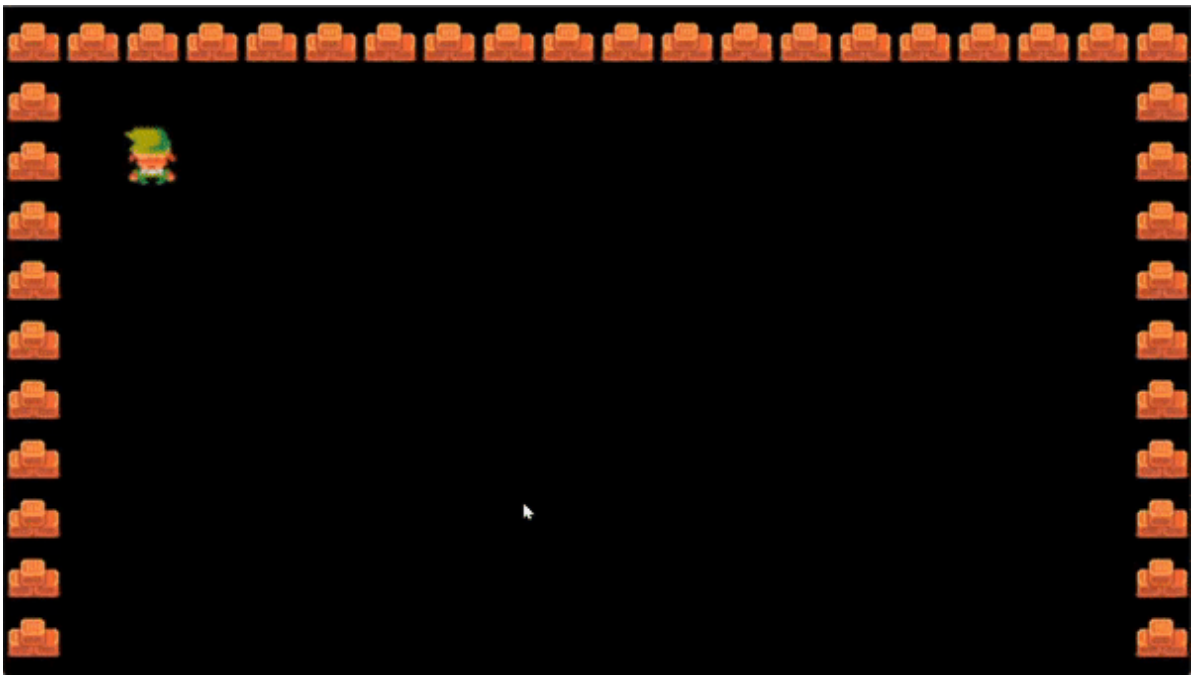
Продолжим прорисовку движений в файле `player.py`. Создадим ещё один параметр — скорость. Для этого в глобальные параметры (`__init__`) добавим строчку `self.speed = 5`

Также напишем функцию `move`:

```
def move(self, speed):
        self.rect.center += self.direction * speed
```

И в update-функции пропишем движения персонажа: `self.move(self.speed)`.

Результат:



Если приглядеться, то по диагонали Линк бежит чуть быстрее. Дело в том, что он бежит по диагонали со скоростью корень из двух, что примерно равно 1.4. Мы прописали скорость вверх, вниз, влево и вправо равной 1, а вот по диагонали из правил математики, можно понять, что длина гипотенузы равна сумме квадратов длин оснований квадрата под корнем, то есть **корень из (1^2 + 1^2)**. Исправим данный баг нормализацией от PyGame (да-да, как в Unity). Исправленная функция движения:

```
def move(self, speed):
    if self.direction.magnitude() != 0:
        self.direction = self.direction.normalize()
    self.rect.center += self.direction * speed
```

Теперь вторая проблема. Линк — танк. Он сбивает всё на своём пути. Нам нужны объекты, с которыми он будет сталкиваться Не забывайте, что все объекты у нас — квадраты тайлами.

Сейчас есть проблема — файл `player.py` не знает о наличии тайлов, которые отображаются через файл уровня. Поэтому дадим файлу новый аргумент. Добавим его в `init` и назовём `obstacle_sprites`. Также не забудьте в файле уровня сослаться на `obstacle_sprites` в отрисовке точки игрока. Для этого в файле `level.py` замените строку:

```
self.player = Player((x, y), [self.visible_sprites])
```

На строку:

```
self.player = Player((x, y), [self.visible_sprites], self.obstacle_sprites)
```

В файле игрока создадим метод столкновений под название `collision`. Функция:

```
def collision(self, direction):
    if direction == 'horizontal':
        for sprite in self.obstacle_sprites:
            if sprite.rect.colliderect(self.rect):
                if self.direction.x > 0: #двигаем вправо
                    self.rect.right = sprite.rect.left
                if self.direction.x < 0: #двигаем влево
                    self.rect.left = sprite.rect.right

    if direction == 'vertical':
        for sprite in self.obstacle_sprites:
            if sprite.rect.colliderect(self.rect):
                if self.direction.y > 0: #двигаем вниз
                    self.rect.bottom = sprite.rect.top
                if self.direction.y < 0: #двигаем вверх
                    self.rect.top = sprite.rect.bottom
```

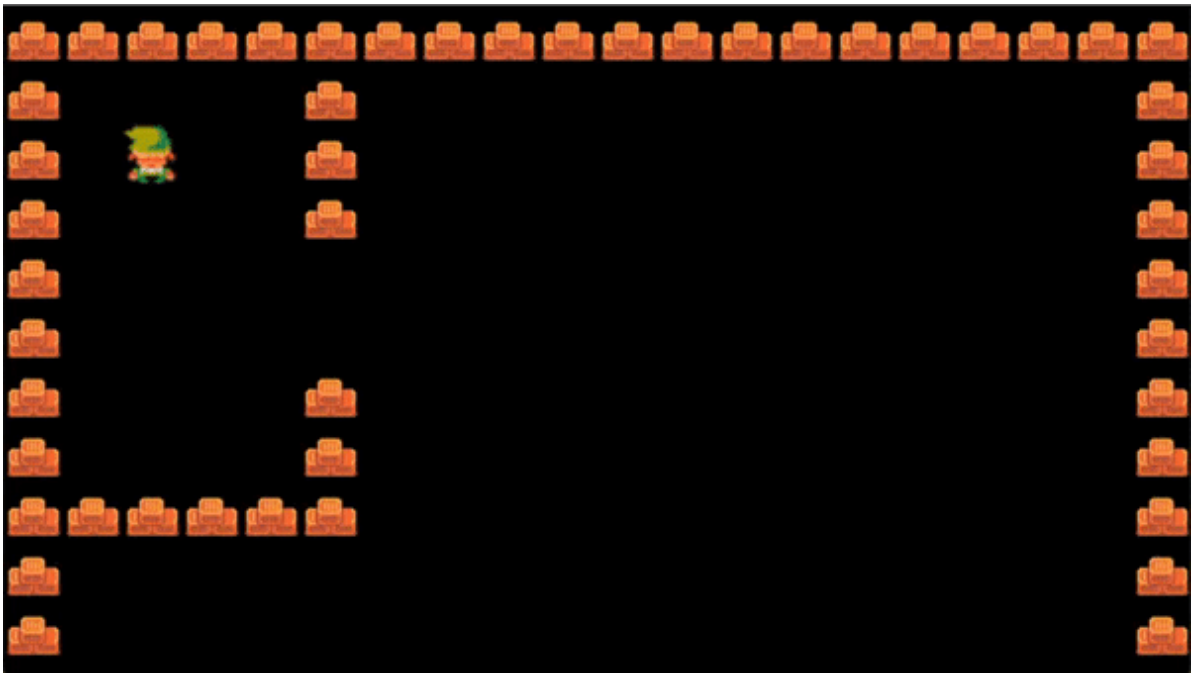
В ней всё разделено на две координаты. По горизонтальной оси — x, по вертикальной — y. Если столкновение произошло по горизонтали, то делаем смещения (вправо или влево). Тоже самое по вертикали, но там вниз и вверх. Последний шаг — разделить метод в движении на два варианта: вертикальный и горизонтальный. То есть из строки:

```
self.rect.center += self.direction * speed
```

Делаем структуру:

```
self.rect.x += self.direction.x * speed
self.collision('horizontal')
self.rect.y += self.direction.y * speed
self.collision('vertical')
```

Получаем героя Хайрула без наклонностей в приведение:



Из предыдущей гифки видна следующая задача — создание камеры. Этим и займёмся. Ссылка на этот этап.

Создание камеры

Предлагаю создать камеру с учётом вектора направления движения персонажа. Мы будем двигать камеру вместе с персонажем. Переходим к файлу уровня и создаём новый класс.

```
class YSortCameraGroup(pygame.sprite.Group):
    def __init__(self):
        super().__init__()
```

Всё, что я сделал — создал новый пустой класс с наследованием всего и вся. Далее применяем его вместо `pygame.sprite.Group()` в `visible_sprites`, дабы не тавтологироваться, а ссылаться на класс с более тонкими настройками. Теперь к тонкостям:

```
class YSortCameraGroup(pygame.sprite.Group):
    def __init__(self):
        super().__init__()
        self.display_surface = pygame.display.get_surface()

    def custom_draw(self):
        for sprite in self.sprites():
            self.display_surface.blit(sprite.image, sprite.rect) #прорисуем
новую поверхность и отрисуем её
```

Здесь добавлены те же отображения новых поверхностей в главного демона с наследованием и создан метод ручной отрисовки (`custom_draw`).

Демон — это программа (или часть программы), которая запускается в фоновом режиме (без терминала или пользовательского интерфейса), ожидая событий и предлагая какие-то службы для их выполнения.

В коде самый интересный пункт в `display_surface.blit`. Surface создаёт новый слой объектов, а blit отрисовывает их. Далее мы передаём картинки и фигуры. Скоро и это перепишем, так как нам нужны векторы. Этим и займёмся. Вектор знаком вам по ходьбе Линка. Далее снова улучшим класс:

```
class YSortCameraGroup(pygame.sprite.Group):
    def __init__(self):
        super().__init__()
        self.display_surface = pygame.display.get_surface()
        self.offset = pygame.math.Vector2()

    def custom_draw(self):
        for sprite in self.sprites():
            offset_pos = sprite.rect.topleft + self.offset #определяем позицию
путём сравнения верхнего левого угла с вектором направления
            self.display_surface.blit(sprite.image, offset_pos) #прорисуем новую
поверхность и отрисуем её
```

Из нового только одна конструкция в методе. Она направлена на определение верхнего левого угла, отрисовку и последующую сумму с вектором направления. Далее в run-методе, не забудьте поменять отрисовку с базовой на нашу кастомную. Примерно так:

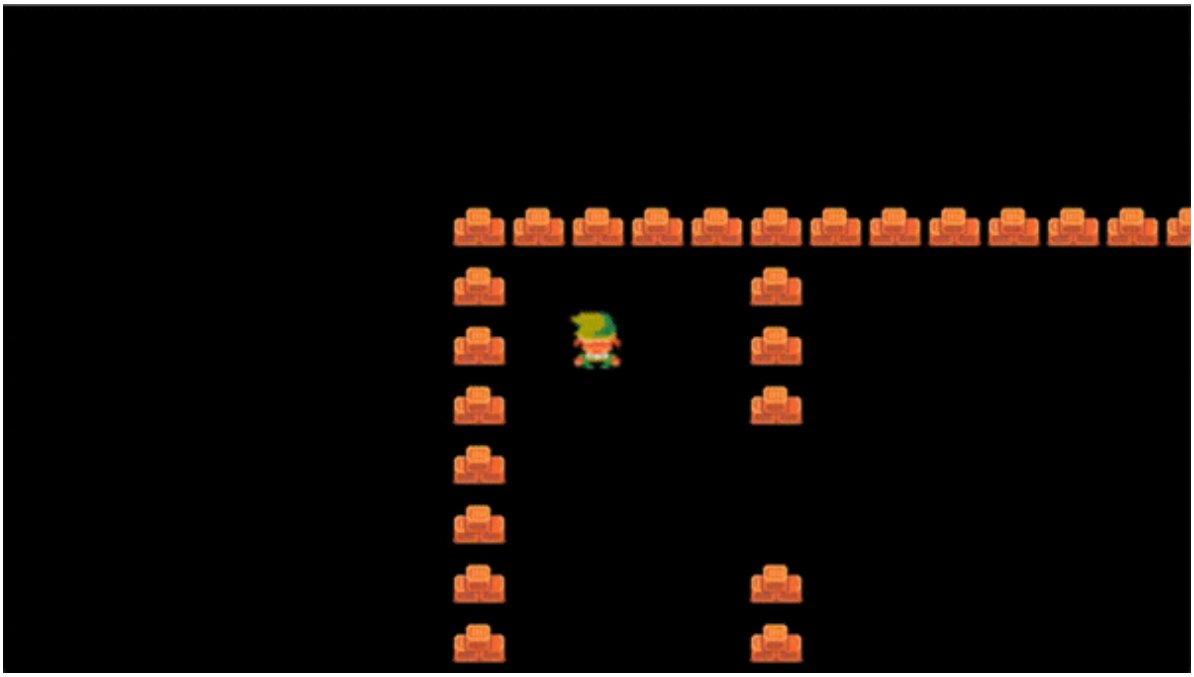
```
self.visible_sprites.custom_draw()
```

Если вы сделали всё верно, вставив в вектор 2 значения (x и y), вы увидите перемещение карты. Я поставил -150 и -150 и получил это:

Осталось перемещаться, зацепившись за героя. Его мы оставим по центру экрана (да здравствует эпохе Dendy). Наш код по доработке:

```
class YSortCameraGroup(pygame.sprite.Group):
    def __init__(self):
        super().__init__()
        self.display_surface = pygame.display.get_surface()
        self.half_width = self.display_surface.get_size()[0] // 2 #середина
отрисованного экрана по ширине
        self.half_heigth = self.display_surface.get_size()[1] // 2 #середина
отрисованного экрана по высоте
        self.offset = pygame.math.Vector2()

    def custom_draw(self, player):
        self.offset.x = player.rect.centerx - self.half_width #координата x
Линка
        self.offset.y = player.rect.centery - self.half_heigth #координата y
Линка
        for sprite in self.sprites():
            offset_pos = sprite.rect.topleft - self.offset #определяем позицию
путём сравнения верхнего левого угла с вектором направления
            self.display_surface.blit(sprite.image, offset_pos) #прорисуем новую
поверхность и отрисуем её
```

В базовых демонов добавили размеры экрана и центр экрана (нацело делим пополам). В нашу кастомную рисовалку передадим координаты игрока и по x, y офсетам высчитаем их. Чтобы избавиться от "пьяной камеры", я вычитаю вектор направления, а не прибавляю его. Не забудьте run-методе передать `self.player`. Результат:



Теперь нужно разобраться с хитбоксами. Сейчас у нас графика из Денди, где каждый объект стоит по конкретным клеткам (тайлам). Нужно это исправить. Для начала перейдем к настройкам тайла и допишем один демон-элемент:

```
self.hitbox = self.rect.inflate(0, -10)
```

Таким образом, мы уменьшили хитбокс на 10 пикселей. Так как отрисовка идёт из центра — мы уменьшаем хитбокс сверху и снизу на 5 пикселей. В демонах игрока пропишем тот же код, но уменьшим хитбокс на 26 пикселей. Далее, наша задача двигаться и проверять коллизии через хитбоксы. Для этого объекты с `rect.x` заменим на `hitbox.x`. Тогда move-функция выглядит так:

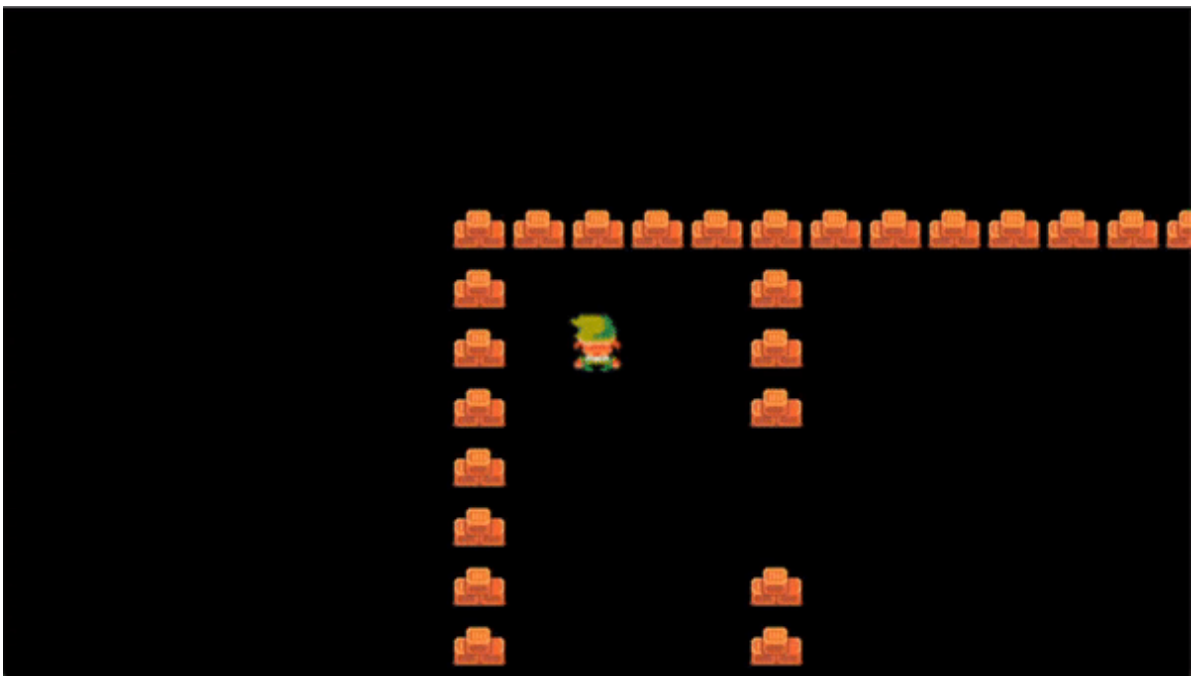
```
def move(self, speed):
    if self.direction.magnitude() != 0:
        self.direction = self.direction.normalize()
    self.hitbox.x += self.direction.x * speed
    self.collision('horizontal')
    self.hitbox.y += self.direction.y * speed
    self.collision('vertical')
    self.rect.center = self.hitbox.center
```

Починим коллизии:

```
def collision(self, direction):
    if direction == 'horizontal':
        for sprite in self.obstacle_sprites:
            if sprite.hitbox.colliderect(self.hitbox):
                if self.direction.x > 0: #двигаем вправо
                    self.hitbox.right = sprite.hitbox.left
                if self.direction.x < 0: #двигаем влево
                    self.hitbox.left = sprite.hitbox.right

    if direction == 'vertical':
        for sprite in self.obstacle_sprites:
            if sprite.hitbox.colliderect(self.hitbox):
```

```
                if self.direction.y > 0: #двигаем вниз
                    self.hitbox.bottom = sprite.hitbox.top
                if self.direction.y < 0: #двигаем вверх
                    self.hitbox.top = sprite.hitbox.bottom
```

Результат:



Линку отрывает шапку. Это происходит из-за того, что изображения хаотично находятся на поверхности. Какие-то выше, какие-то ниже. Исправим это через класс YSort (не просто же так мы мы назвали класс YSortCameraGroup). Поправим только одну строку:

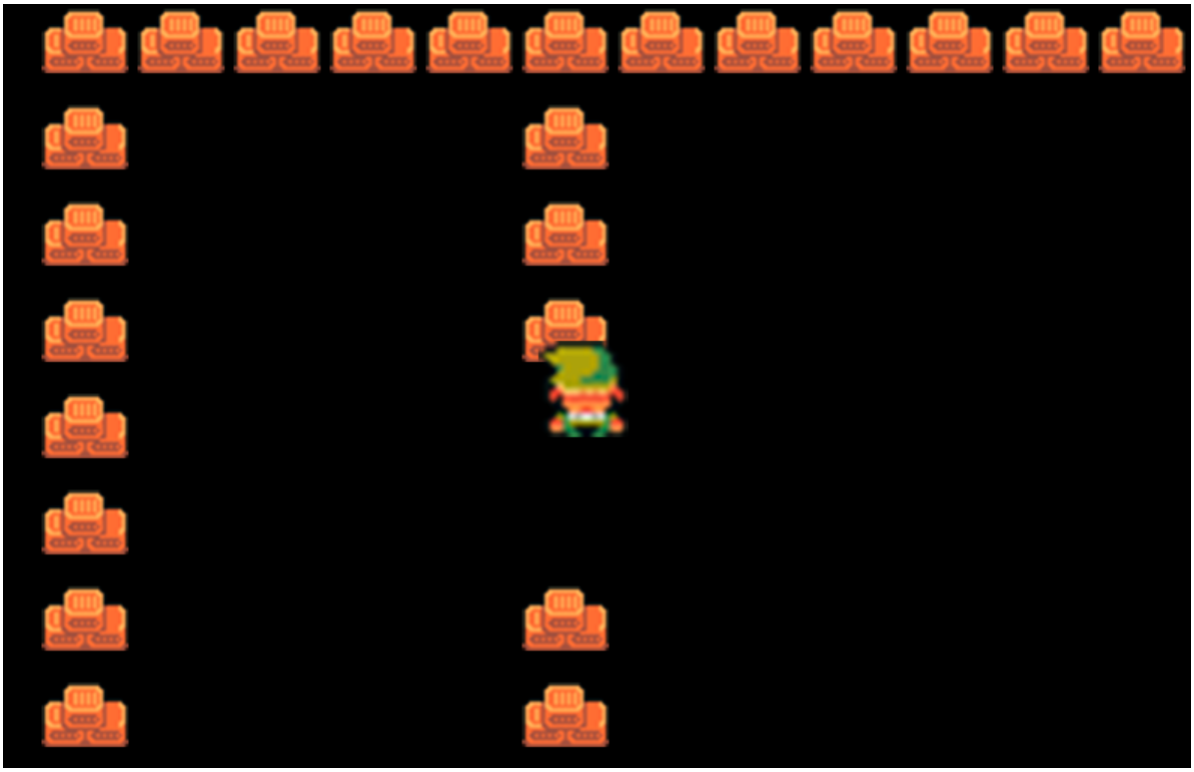
```
for sprite in self.sprites():
```

На строку:

```
for sprite in sorted(self.sprites(), key = lambda sprite: sprite.rect.centery):
```

Мы сделали сортировку по y-координате. Про лямбду можно отдельно почитать, но если сказать грубо — лямбда-функция — функция, которая работает с анонимными функциями.

Итог:

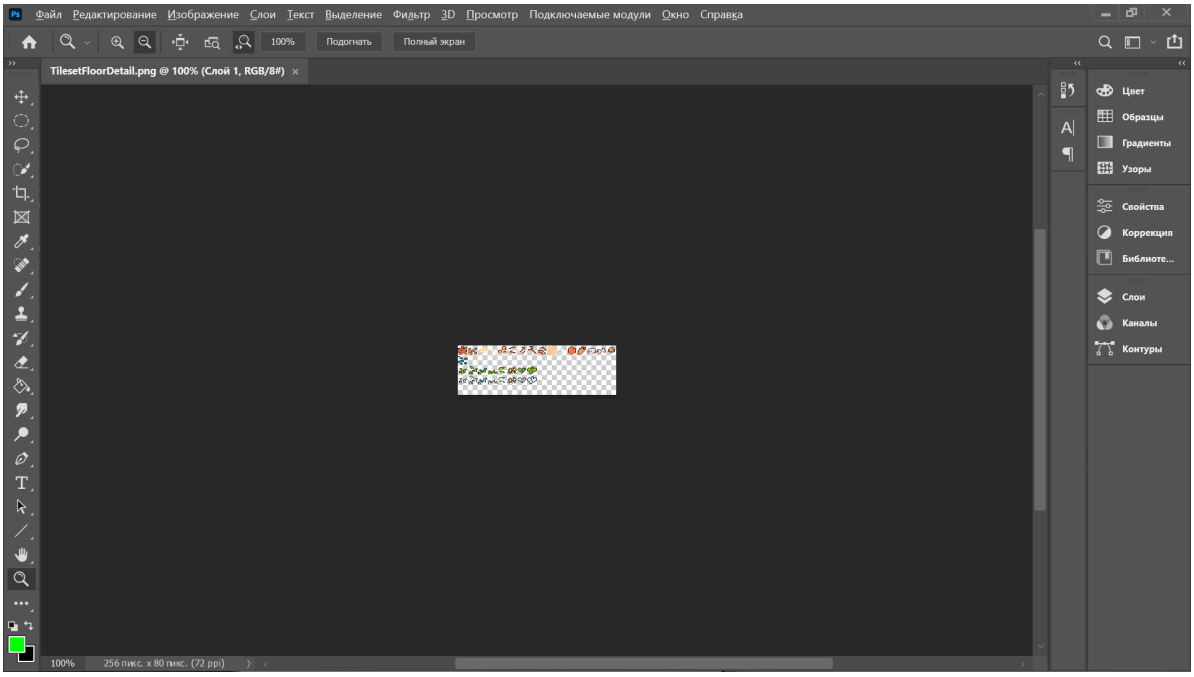


Ссылка на этап проекта.

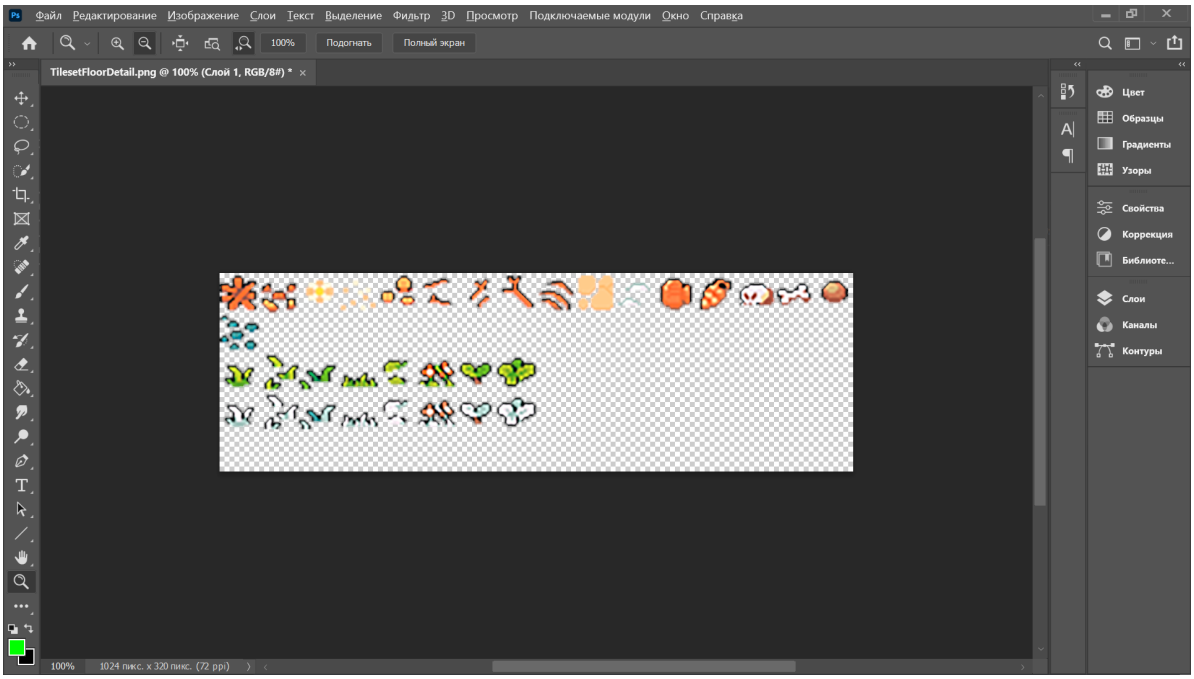
Графика

Мне пришлось освоить для себя новую программу. Она была очень дружелюбной, и со второго раза я смог нарисовать поле игры. Программа называется Tiled. В ней можно легко настраивать карты. Сразу оговорюсь, что это мой первый опыт работы с картами. Это сыграло со мной злую шутку. Дважды.

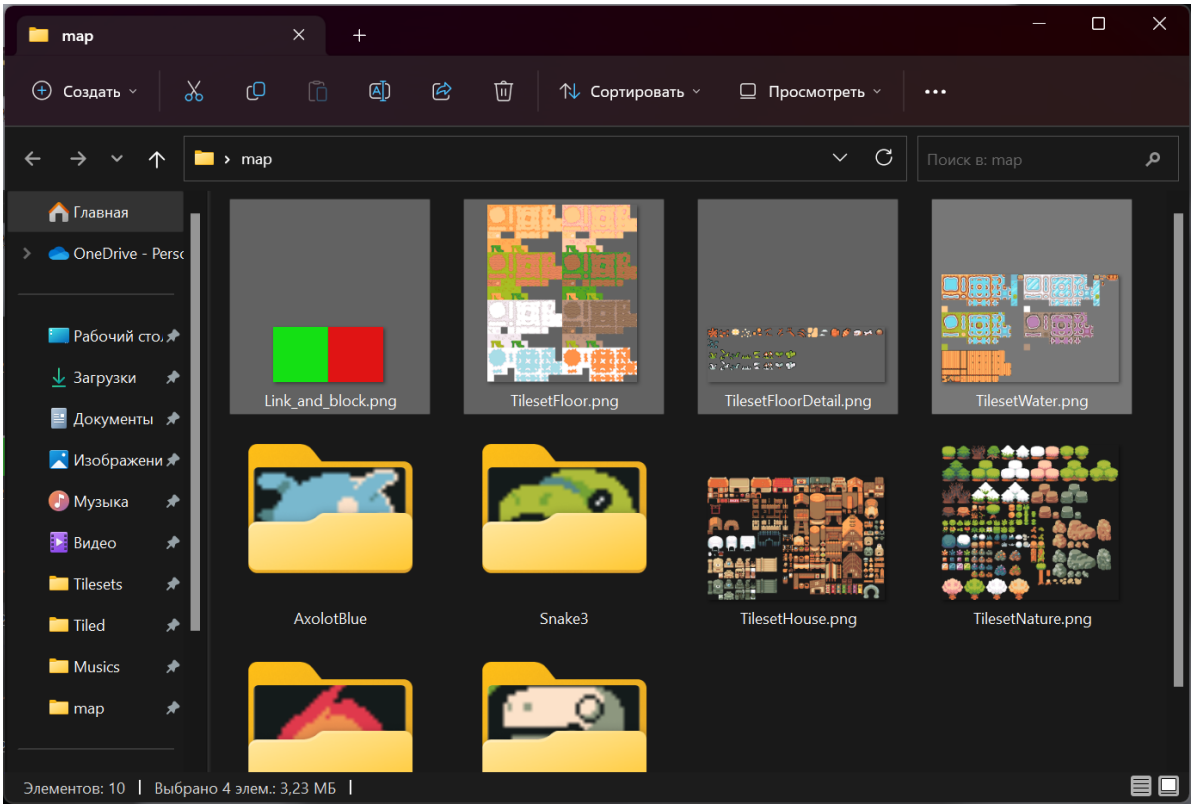
Чуть ранее я писал про проблемы с ассетами. Дело в том, что я выбрал размер тайла 64 пикселя. Это стандартное разрешение для игр в ряд, но я не учёл один момент — все ассеты, которые были у Ninja Adventure были в разрешении 16 на 16 пикселей. Это нормально для пиксельной графики. В общем, была задача поправить это расхождение. Поправка была простой — я увеличил размер нужных мне тайлов в 4 раза в фотошопе. В итоге, получилось из этого:



Сделать это:



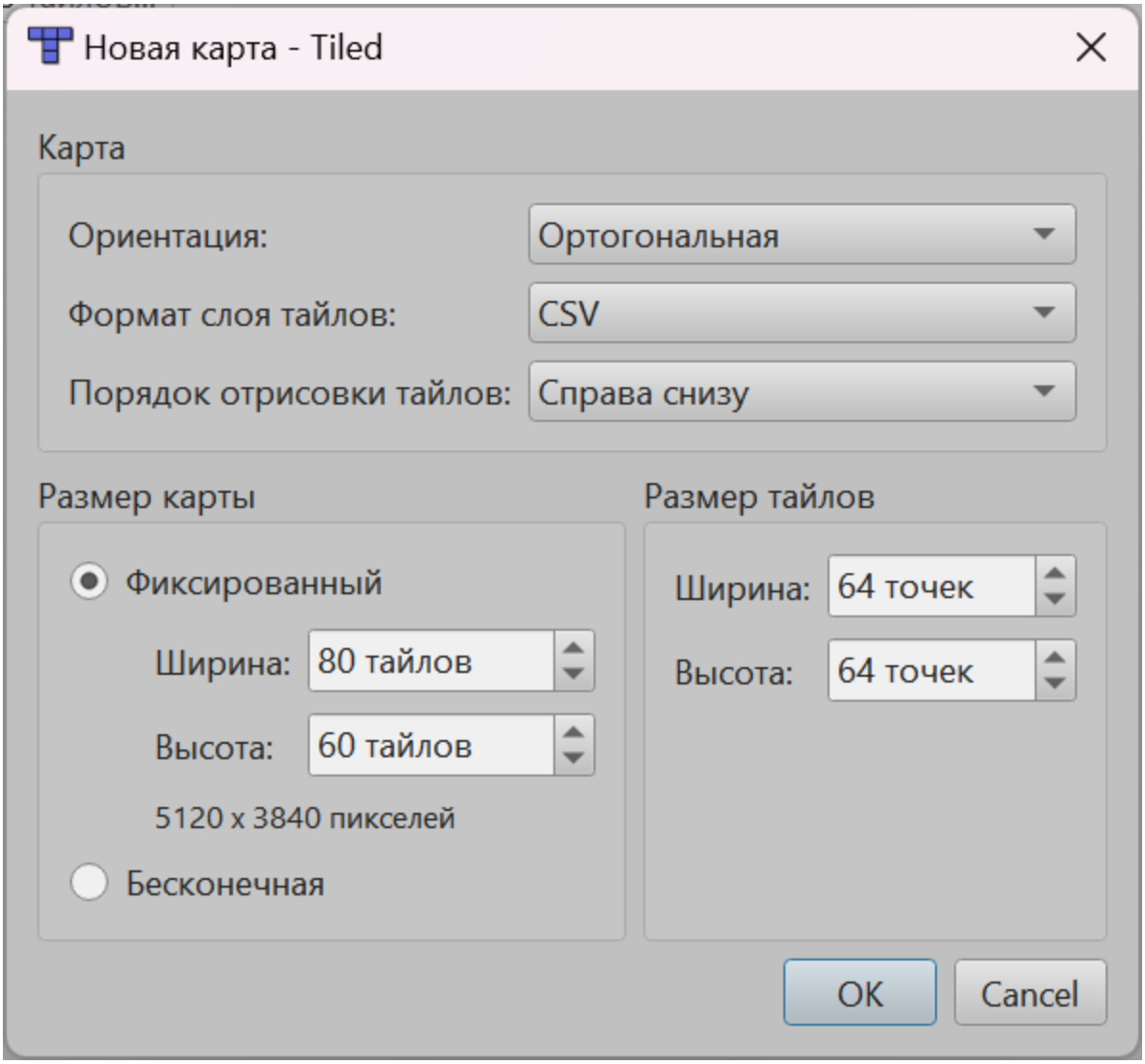
Для отрисовки карты, я взял 3 файла (TilesetFloor.png, TilesetFloorDetails.png и TilesetWater.png):



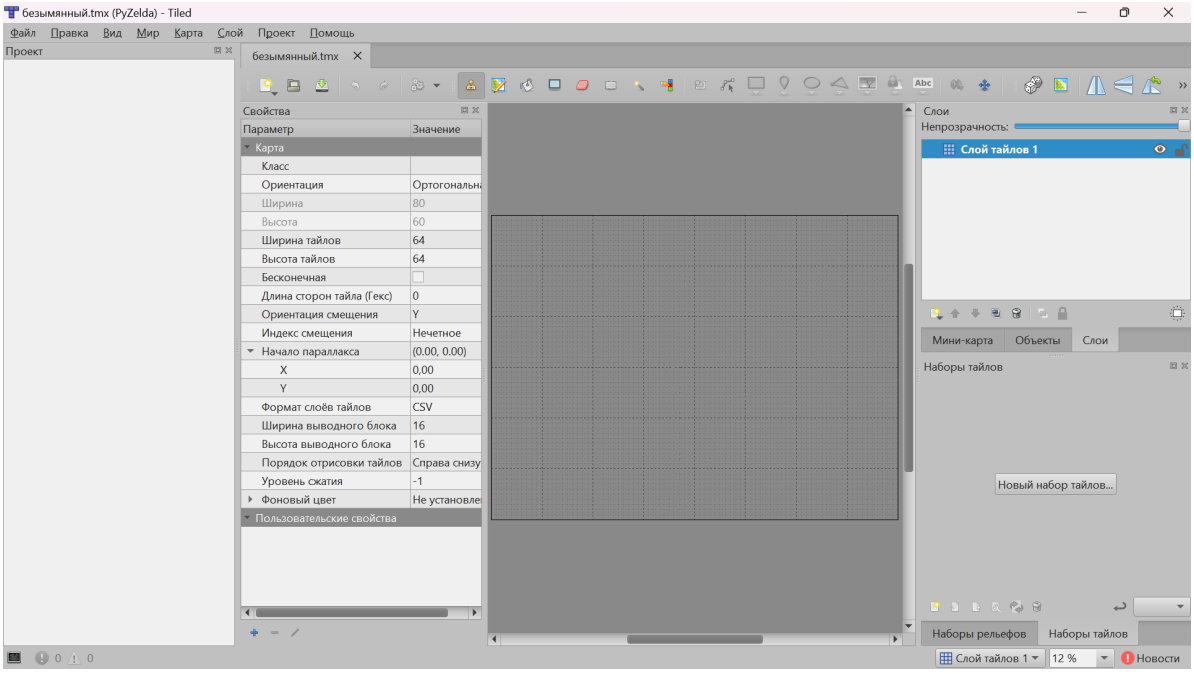
> **Что-то типа ворнинга**. Тут будьте осторожны. Мне пришлось перерисовать карту и сделать из разных картинок с тайлами одну. Не повторяйте моих ошибок и сначала прочитайте что происходит ниже в разделе "Монстры".

Помимо прочего, я создал зелёный квадрат 64 на 64 и такой же красный с прозрачностью в 20%, чтобы поставить точку спауна Линка и невидимые стены для него. Приступим к отрисовке карты.

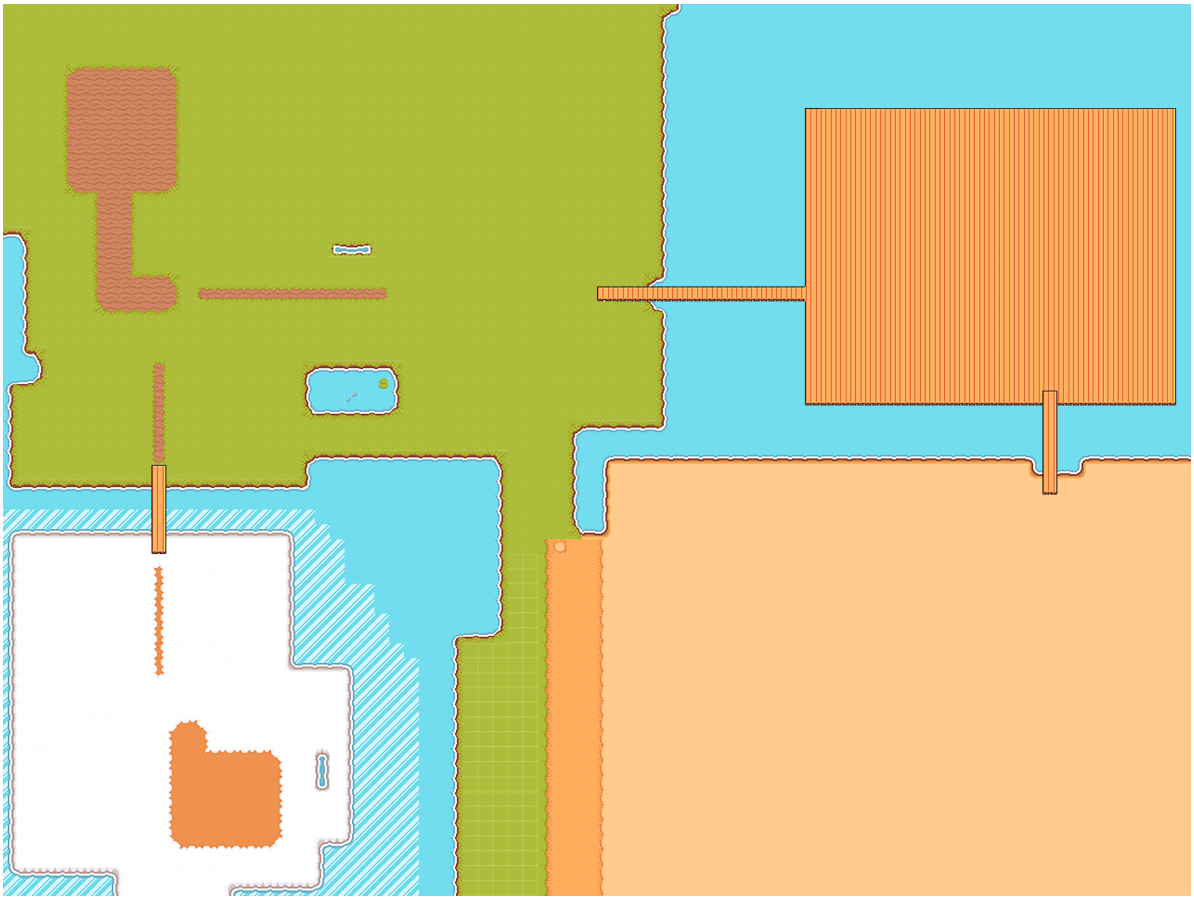
Открываем Tiled и выставляем следующие настройки: ортогональная карта лежит в координатах x и y, что нам и подходит. Слой будет выводиться в формате CSV и отрисовываться справа снизу. Размеры карты будут 80 на 60 тайлов, а это значит 80 на 64 = 5120 и 60 на 64 = 3840 пикселей.



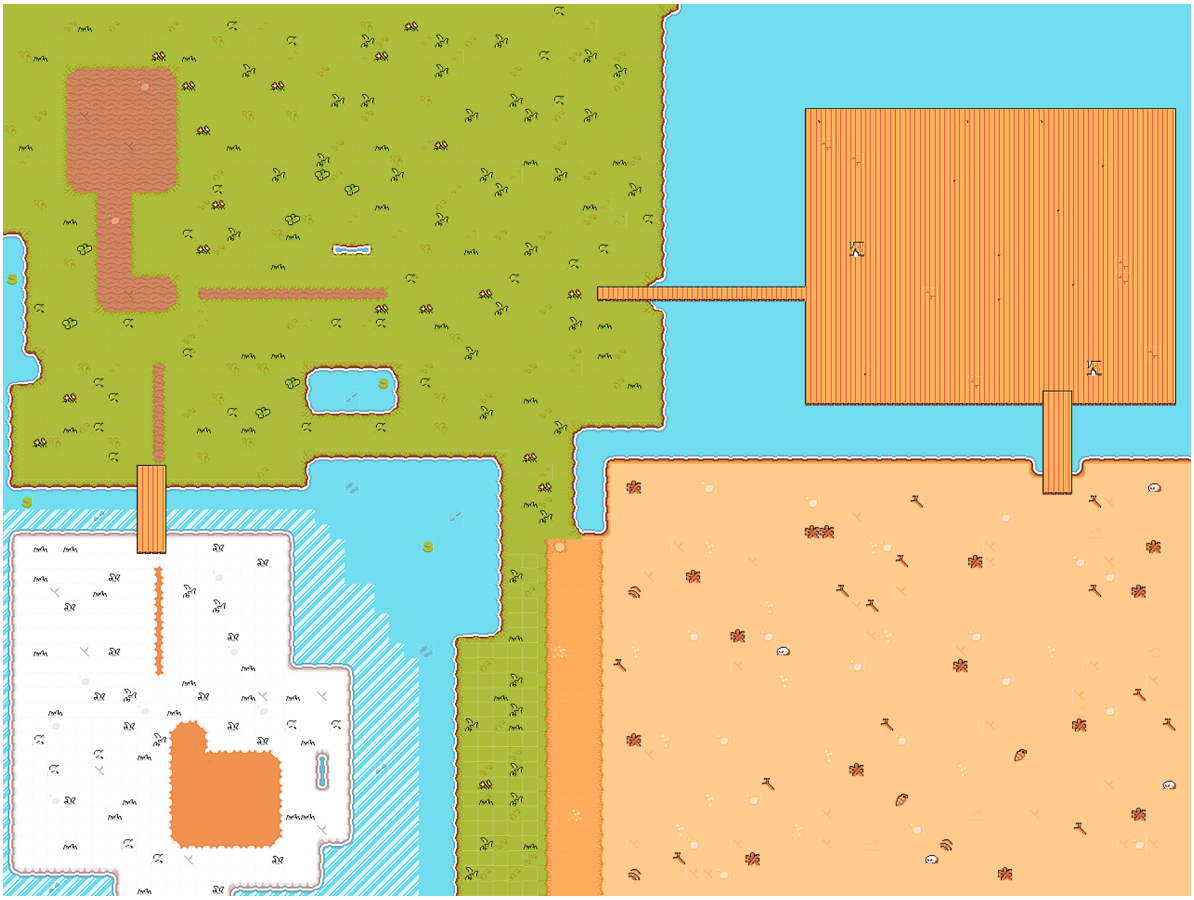
Создав карту, получим своё рабочее поле и приступим к рисованию:



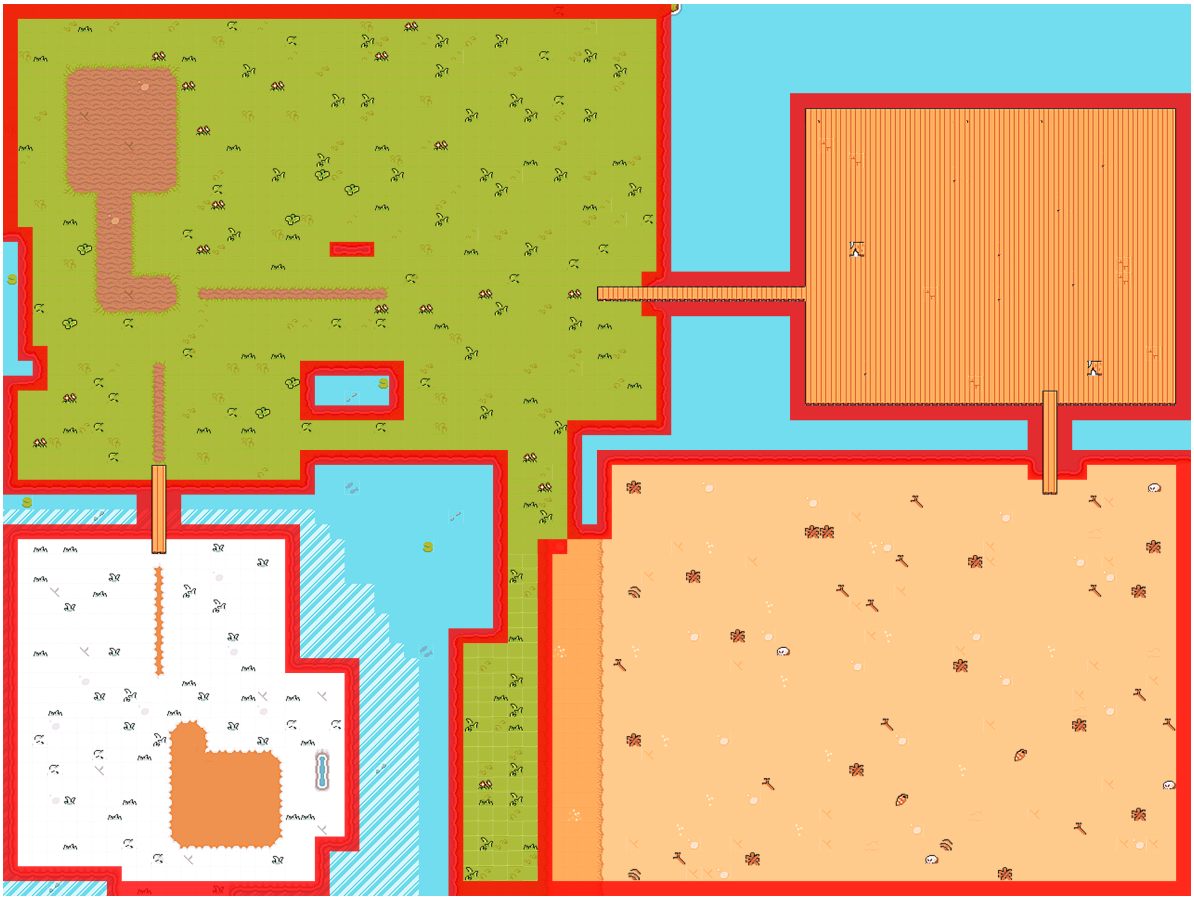
Час работы и результат:



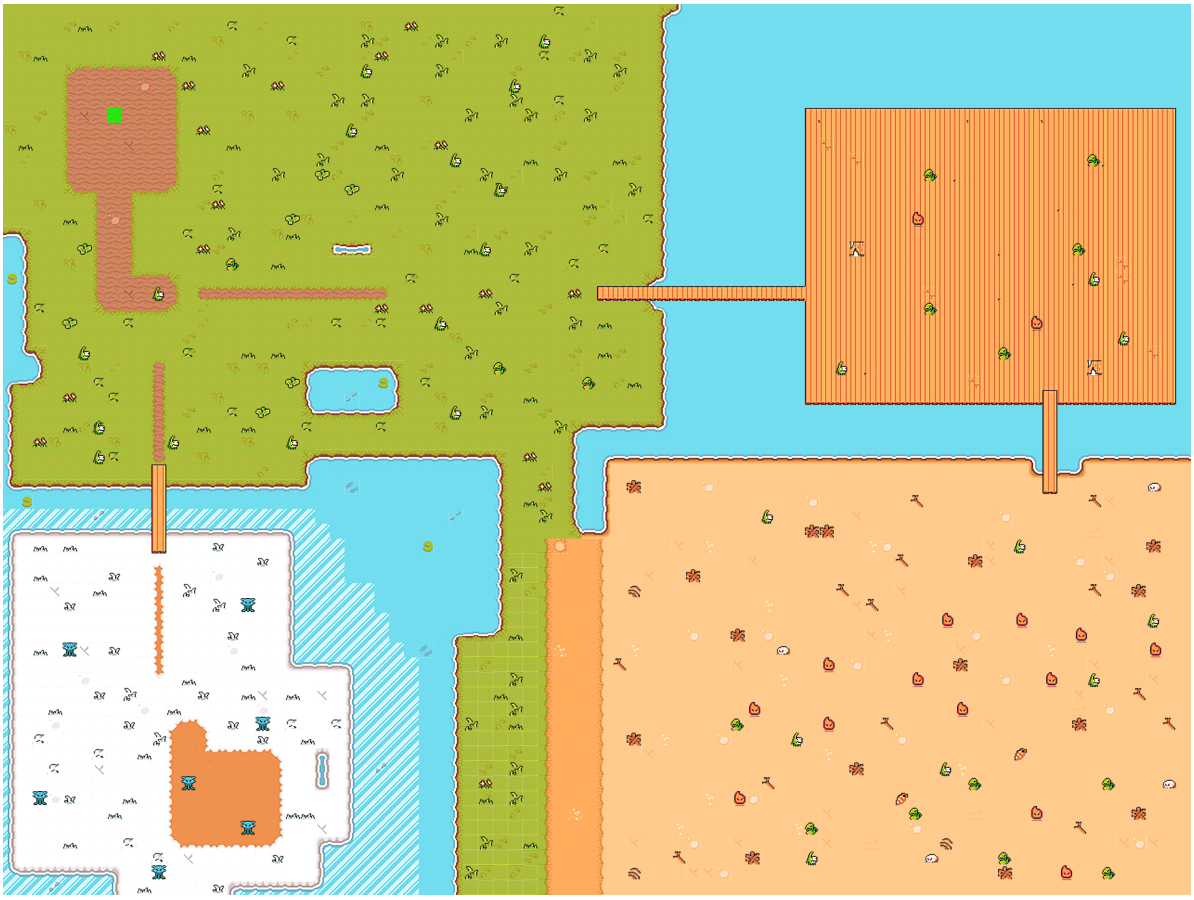
Сейчас работа очень грубая. Нужно добавить деталей. Займёмся этим. Вот что получилось спустя минут 20:



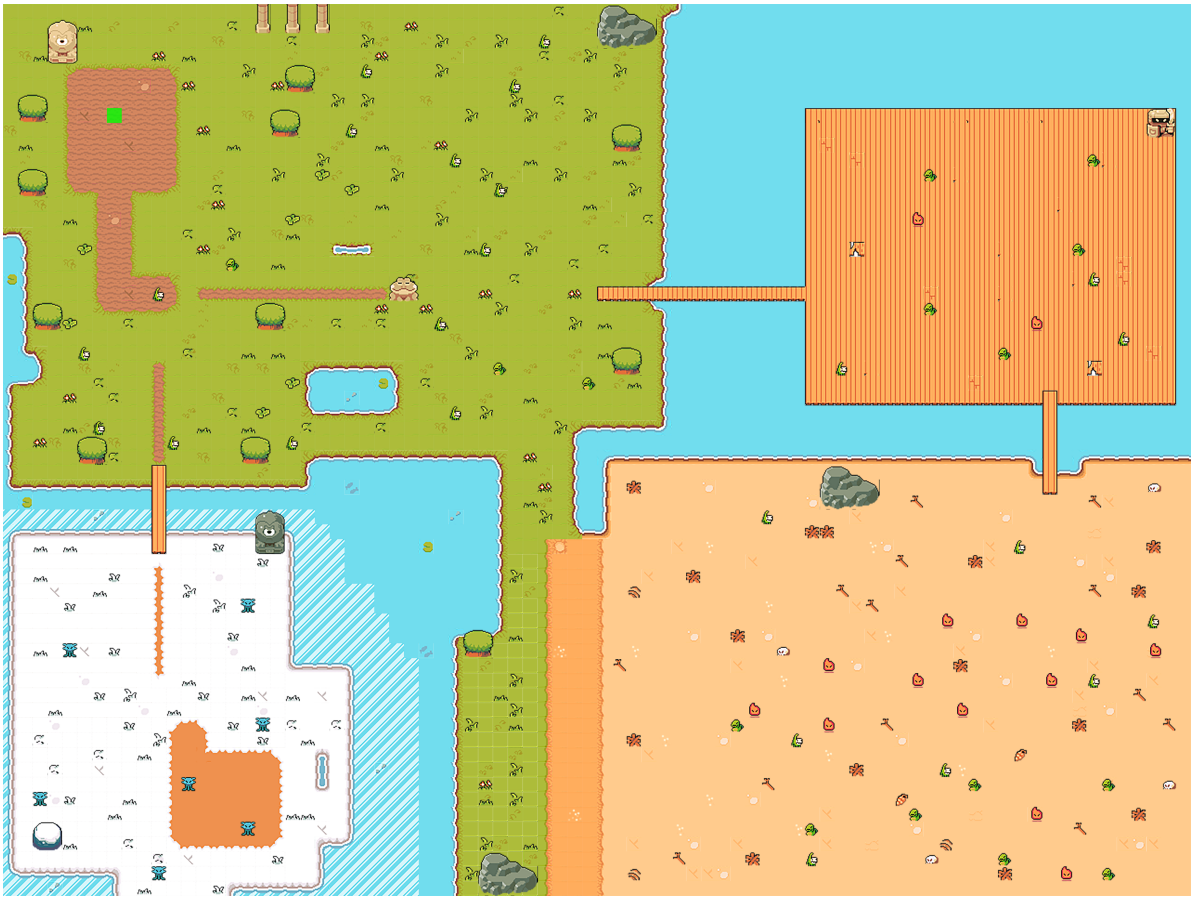
Далее я добавил тайлы-блоки, чтобы за них Линк не выбрался (куда ж без невидимых стен). Главное — все поля должны быть замкнуты, чтобы Линк внезапно не залетал. У меня получилось как-то так:



Потом я выключил данный слой, чтобы он не отвлекал. Далее, проставим места спауна монстров (бебов) и главного героя – Линка:



Добавим пару объектов:



Конечно, карта может быть детальнее проработана, но это не моя задача. Моя задача — проверить как там игродев на змейке.

Всю графику я перенёс в папку graphic, а также выгрузил карту в `.csv`-формате в папку map.

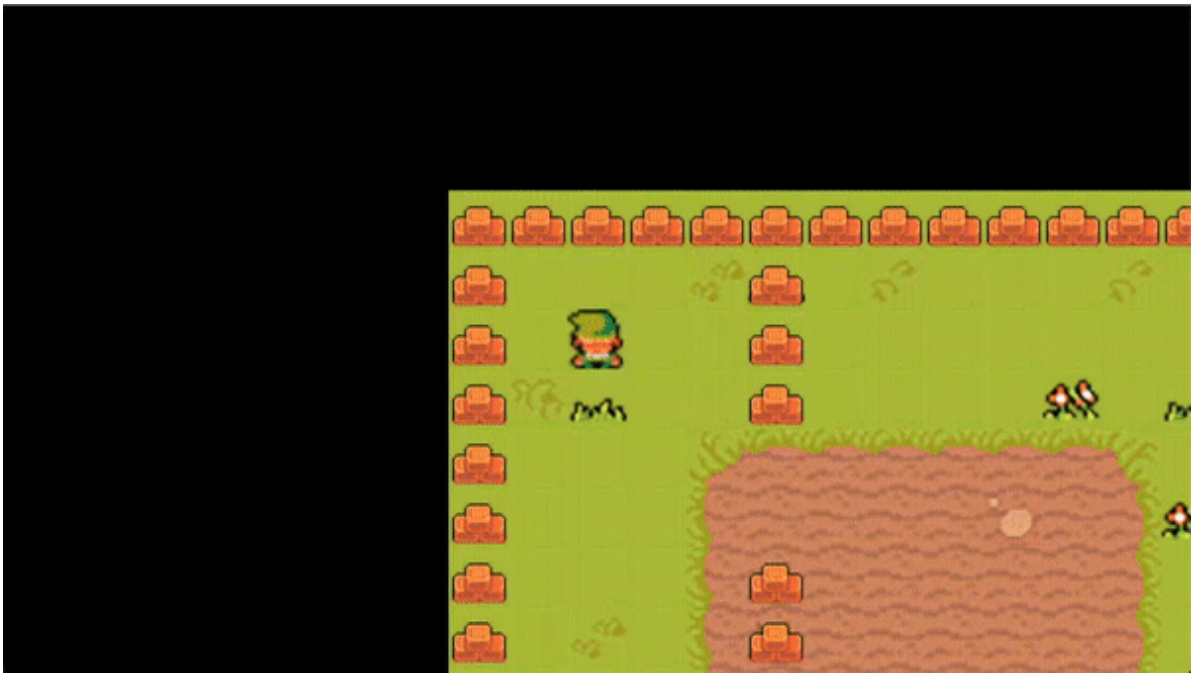
Приступим к коду. Для начала, я скинул в папку Test нашего Линка и коробку и поправил пути в проекте в файлах `player.py` и `title.py`. После запуска ничего не поменялось. Далее я решил положить подложкой на чёрный экран карту. Я экспортировал картинку без спауна и дополнительных объектов и установил её выше чёрного экрана, но ниже Линка с коробками. Для этого в файле `level.py` я объявил о наличии таких файлов двумя строками кода в демонах класса `YSortCameraGroup`:

```
self.floor_surf = pygame.image.load('../graphic/map+det.png').convert()
#добавили задник карты
self.floor_rect = self.floor_surf.get_rect(topleft = (0, 0)) #отрисовка карты с
левого верхнего угла
```

Далее в методе `custom_draw` вычисляем отрисовку с офсетами для камерами (всё ровно также, как и ранее):

```
floor_offset_pos = self.floor_rect.topleft - self.offset
self.display_surface.blit(self.floor_surf, floor_offset_pos)
```

Результат:



Происходит наслоение старой карты и новой. Исправим это: удалим всё в методе `create_map` в `level.py`. Немного переписав метод, я привёл его к следующему виду:

```
def create_map(self):
    self.player = Player((1000, 1000), [self.visible_sprites],
self.obstacle_sprites)
```

Всё что я оставил — это отрисовку места Линка на карте. Просто указал координату.

Теперь наша задача провзаимодействовать с файлом `tile.py` Ему нужны более детальные настройки для прорисовки объектов и картинок конкретного размера. Приступим.

Для начала помимо прочих элементов, демонам нужно передавать новые два аргумента `sprite_type` и `surface = pygame.Surface((TILESIZE, TILESIZE))`. Думаю, по названию понятно, что они будут передавать тип спрайта и его размер из файла `settings.py`. Ещё немного переработаем файл и получим упрощение конструкции с ссылкой на себя:

```
import pygame
from settings import *

class Tile(pygame.sprite.Sprite):
    def __init__(self, pos, groups, sprite_type, surface =
pygame.Surface((TILESIZE, TILESIZE))):
        super().__init__(groups) #наследуем все группы
        self.sprite_type = sprite_type
        self.image = surface
        self.rect = self.image.get_rect(topleft = pos) #указываем позицию
отрисовки (левый верхний угол)
        self.hitbox = self.rect.inflate(0, -10) #делаем по 5 пискселей сверху и
снизу от самого объекта, до хитбокса
```

Вернёмся в `creat_map`. Создадим новый словарь на основе csv-файлов:

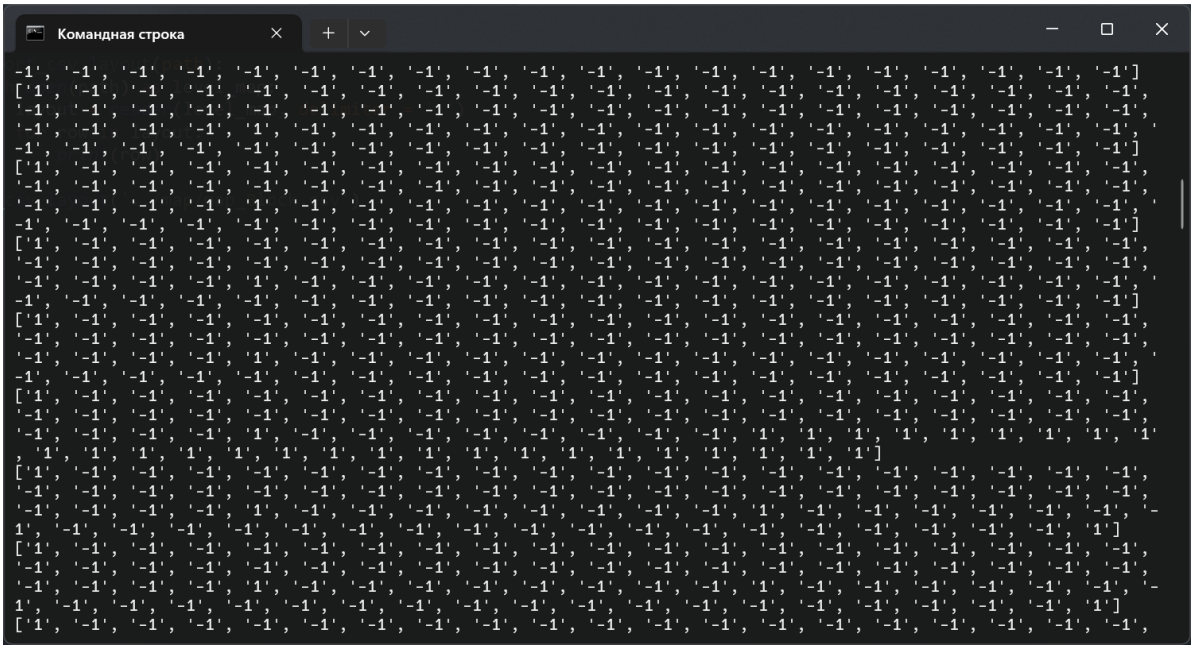
```
layout = {
        'boundary': import_csv_layout('../map/map_Block.csv')
}
```

Метод `import_csv_layout` нам не знаком. Напишем его в новой файле `support.py`.

```
from csv import reader

def import_csv_layout(path):
    with open(path) as level_map:
        layout = reader(level_map, delimiter = ',')
        for row in layout:
            print(row)

import_csv_layout('../map/map_Block.csv')
```

В данном коде идёт чтение csv-файла. А при чтении "1" — означает, что объект там есть, а "-1" — его там нет.

Результат вывода `support.py`:



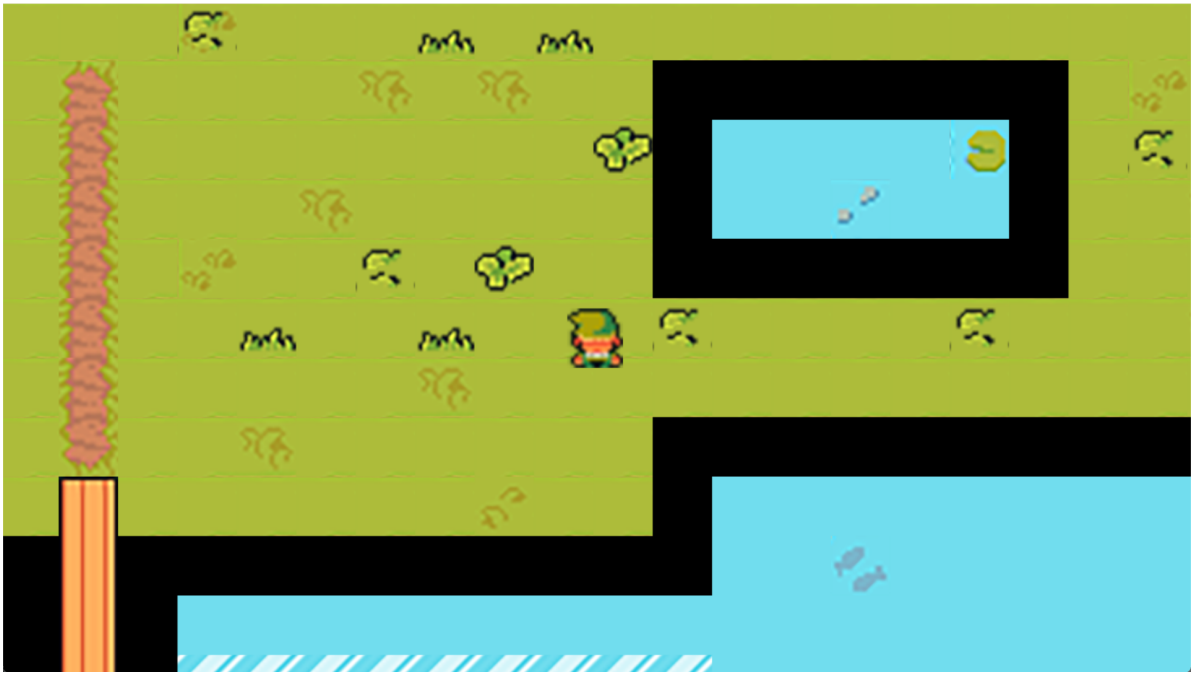
Чуть подправим, чтобы у нас на выходе была матрица (массив из выведенных строк) и функция готова:

```
from csv import reader

def import_csv_layout(path):
    terrain_map = []
    with open(path) as level_map:
        layout = reader(level_map, delimiter = ',')
        for row in layout:
            terrain_map.append(list(row))
        return terrain_map
```

Теперь, вернёмся к `creat_map` и допишем функцию очень похожую на то, что было ранее (не забудьте импортировать support):

```
def create_map(self):
    layout = {
            'boundary': import_csv_layout('../map/map_Block.csv')
    }
    for style, layout in layout.items():
        for row_index, row in enumerate(layout):
            for col_index, col in enumerate(row):
                if col != '-1':
                    x = col_index * TILESIZE
                    y = row_index * TILESIZE
                    if style == 'boundary':
                        Tile((x, y), [self.visible_sprites,
self.obstacle_sprites], 'invisible')
    self.player = Player((1000, 1000), [self.visible_sprites],
self.obstacle_sprites)
```

Итог:

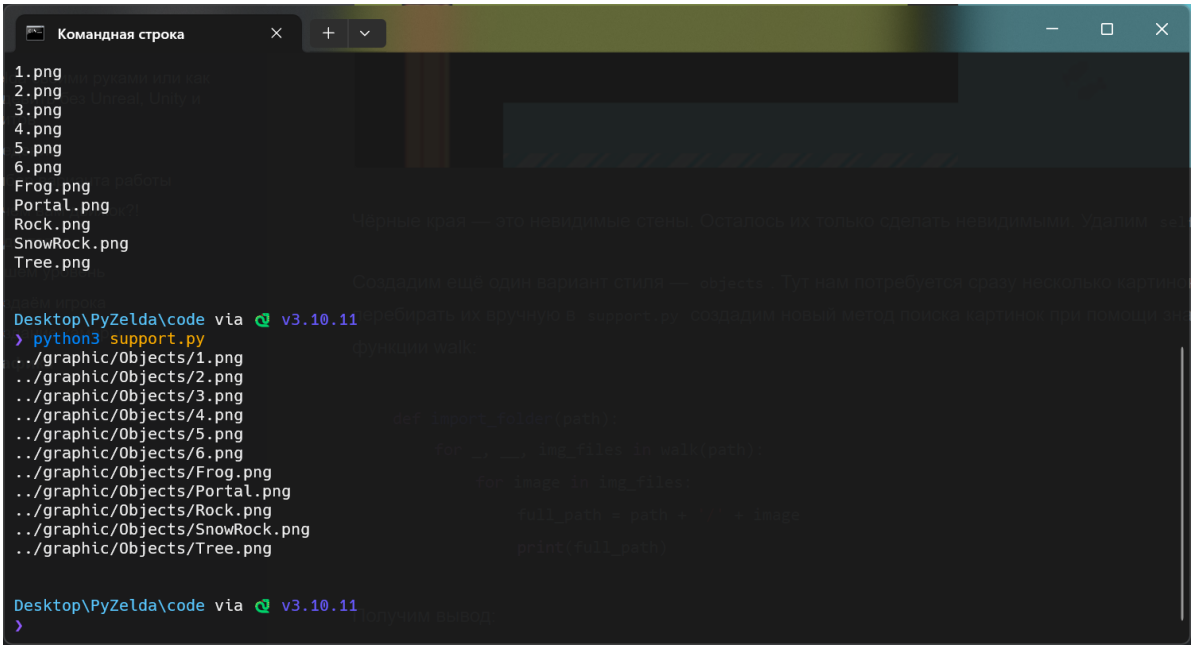


Чёрные края — это невидимые стены. Осталось их только сделать невидимыми. Удалим `self.visible_sprites`.

Создадим ещё один вариант стиля — `objects`. Тут нам потребуется сразу несколько картинок. Чтобы не перебирать их вручную в `support.py` создадим новый метод поиска картинок при помощи знакомым многим функции walk:

```
def import_folder(path):
    for _, __, img_files in walk(path):
        for image in img_files:
            full_path = path + '/' + image
            print(full_path)
```

Получим вывод:



Немного перепишем метод под pygame:

```
def import_folder(path):
    surface_list = []
    for _, __, img_files in walk(path):
        for image in img_files:
            full_path = path + '/' + image
            image_surf = pygame.image.load(full_path).convert_alpha()
            surface_list.append(image_surf)
    return surface_list
```

Метод отрисовки объектов:

```
if style == 'objects':
    surf = graphics['objects'][int(col)]
    Tile((x, y), [self.visible_sprites, self.obstacle_sprites], 'object', surf)
```

Перебором по всем файлам выставляем объекты. Но есть одна проблема. Если объекты были друг за другом — они хаотично ставятся. Исправим это в `tile.py`:

```
if sprite_type == 'object':
    self.rect = self.image.get_rect(topleft = (pos[0], pos[1] - TILESIZE))
else:
    self.rect = self.image.get_rect(topleft = pos) #указываем позицию отрисовки
(левый верхний угол)
```

В этом моменте мы отключили перекрытие объектом других объектов. Итог — Линк гуляет в лесу!

Результат работы по ссылке. Помимо прочего, оставляю проект в Tiled, чтобы вы сами могли "поиграться" с ним. Далее мы поговорим об анимации.

Анимация Линка

Сначала нужно добавить действие и кнопку. На атаку будет стоять кнопка "Пробел". Пропишем это в `player.py`:

```
if keys[pygame.K_SPACE] and not self.attacking:
    self.attacking = True
```

Непонятно, что такое `self.attacking`. В демоны файла я добавил три параметра: статус атаки (`self.attacking = False`), кулдаун после атаки (`self.attack_cooldown = 400`) и время атаки (`self.attack_time = None`).

Добавим время между атаками. После успешной атаки создадим конструкцию: `self.attack_time = pygame.time.get_ticks()` сразу после объявление атаки флагом `True`. Далее перейдём к настройке нового метода `cooldowns`. Сам метод:

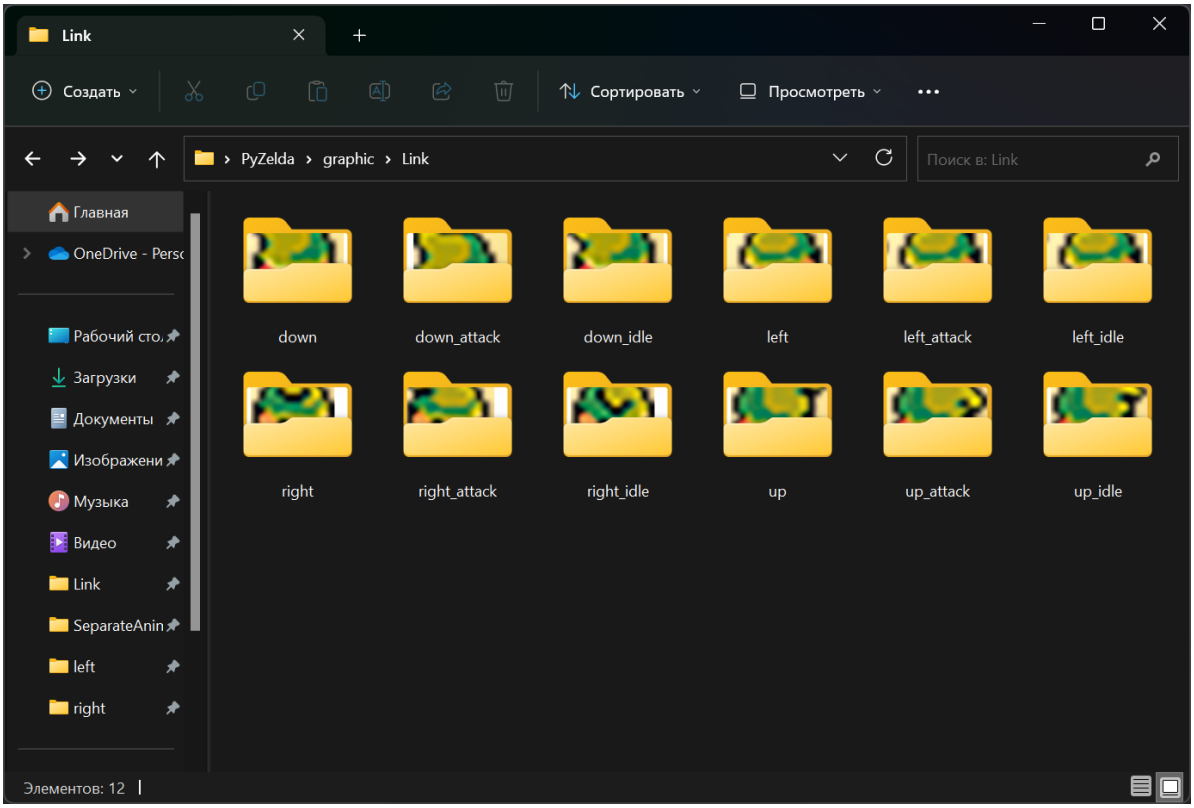
```
def cooldowns(self):
    current_time = pygame.time.get_ticks()

    if self.attacking:
        if current_time - self.attack_time >= self.attack_cooldown:
            self.attacking = False
```

Конструкция очень проста. Если разница во времени после нажатия меньше кулдауна атаки (в моём случае 400 мс), запрещать атаку. Не забудьте закинуть функцию cooldowns в update-функцию.

Далее, пропишем метод `import_player_assets`:

```
def import_player_assets(self):
    character_path = '../graphic/Link/'
    self.animations = {'up': [], 'down': [], 'left': [], 'right': [],
    'right_idle': [], 'left_idle': [], 'up_idle': [], 'down_idle': [],
    'right_attack': [], 'left_attack': [], 'up_attack': [], 'down_attack': []}
```

В нём прописаны все возможности передвижения. Они лежат в папке "Link":



Не забудьте закинуть в базовых демонов Player-класса наши ассеты: `self.import_player_assets()`. Далее, "пробежимся" по нашим папкам благодаря методу `import_folder` в `support.py`:

```
for animation in self.animations.keys():
    full_path = character_path + animation
    self.animations[animation] = import_folder(full_path)
```

Сейчас мы можем открыть все файлы, но есть дополнительная проблема — нужно прописать статусы и анимации при нажатии на кнопку. Статус по умолчанию укажем в демоне и по умолчанию Линк будет смотреть вниз (`self.status = 'down'`). Но этого мало. Нужно получать статус автоматически, а это значит новый метод:

```
def get_status(self):
    if self.direction.x == 0 and self.direction.y == 0:
        self.status = self.status + '_idle'
```

Прописываем название файла "стояния" героя. Стояние — `_idle`, а статус будет меняться от нажатия кнопок. Закидываем `get_status()` в update-метод.

Пропишем статусы для наших методов хождения:

```
if keys[pygame.K_UP]:
    self.direction.y = -1
    self.status = 'up'
elif keys[pygame.K_DOWN]:
    self.direction.y = 1
    self.status = 'down'
else:
    self.direction.y = 0
```

```
if keys[pygame.K_LEFT]:
    self.direction.x = -1
    self.status = 'left'
elif keys[pygame.K_RIGHT]:
    self.direction.x = 1
    self.status = 'right'
```

Тут всё достаточно прозрачно. Движение = статусу. Подправим функцию статуса:

```
if not 'idle' in self.status:
    self.status = self.status + '_idle'
```

Если мы отпускаем кнопку — статус равен idle, а если держим просто название направления.

Теперь окончательно проапгрейдим код для атак и комбинаций с "_idle":

```
def get_status(self):
    if self.direction.x == 0 and self.direction.y == 0:
        if not 'idle' in self.status and not 'attack' in self.status:
            self.status = self.status + '_idle'
    if self.attacking:
        self.direction.x = 0 #координаты по x
        self.direction.y = 0 #координаты по y
        if not 'attack' in self.status: #если нет подписи "attack"
            if 'idle' in self.status: #но есть "idle"
                self.status = self.status.replace('_idle', '_attack') #убираем
_idle, но оставляем _attack
            else:
                self.status = self.status + '_attack' #если idle не было, просто
стави attack
    else:
        if 'attack' in self.status:
            self.status = self.status.replace('_attack', '') #удаляем attack при
завершении статуса
```

Наконец, заанимируем Линка. Установим два новых демона `self.frame_index = 0` — индекс первой картинки и скорость смены картинок — `self.animation_speed = 0.15`. Затем создадим метод `animate()`:

```
def animate(self):
    animation = self.animations[self.status] #узнаём статус для ссылки на нужный
файл
    self.frame_index += self.animation_speed #добовляем нашу скорость и когда
добавится единица (из 0.15), сменяем картинку
    if self.frame_index >= len(animation): #при вылете из массива
        self.frame_index = 0 #возвращаемся к начальной картинке и тем самым
зацикливаемся
    self.image = animation[int(self.frame_index)] #указываем картику
    self.rect = self.image.get_rect(center = self.hitbox.center) #указываем
хитбокс
```

Сейчас мы перебираем все наши картинки и главное — их зациклить, как в рилсах. Не забываем в update-функцию `self.animate()`.

Тут у меня залогала анимация атаки, так как кнопка продолжала нажиматься. Исправим это в input-методе:

```
def input(self): #варьируем кнопки
    if not self.attacking:
        keys = pygame.key.get_pressed()

        if keys[pygame.K_UP]:
            self.direction.y = -1
            self.status = 'up'
        elif keys[pygame.K_DOWN]:
            self.direction.y = 1
            self.status = 'down'
        else:
            self.direction.y = 0

        if keys[pygame.K_LEFT]:
            self.direction.x = -1
            self.status = 'left'
        elif keys[pygame.K_RIGHT]:
            self.direction.x = 1
            self.status = 'right'
        else:
            self.direction.x = 0

        if keys[pygame.K_SPACE]:
            self.attacking = True
            self.attack_time = pygame.time.get_ticks()
```

Итог:

Далее прорисуем оружие. Файлы этого этапа.

Оружие героя

Карта немного поменялась, так как Линк врезался в мосты, которые соединяют нижние острова. Изменения делал непосредственно в Tiles и сохранял карту и csv-файлы. Чтобы всё было канонично, Линку выдадим меч. Спрайты на мечи я тоже отрисовал. Он лежит в папке weapons/sword. Там 5 картинок, где full обозначает сам меч, а остальные 4 — направления меча. Приступим к коду.

В файле setting.py, создаем словарь из оружий. У меня будет только один меч, но вы можете добавить больше оружия. Сам словарь:

```
weapon_data = {'sword': {'cooldown': 300, 'damage': 15,
'graphic':'../graphic/weapons/sword/full.png'}}
```

Далее, создадим новый файл с настойкой оружия `weapon.py`:

```
import pygame

class Weapon(pygame.sprite.Sprite):
    def __init__(self, player, groups):
        super().__init__(groups)
        self.image = pygame.Surface((40, 40)) #сама графика
        self.rect = self.image.get_rect(center = player.rect.center) #место
отрисовки
```

Тут всё традиционно и без нового. Мы центруемся от самого игрока последней строкой в коде. Сейчас посередине будет рисоваться чёрный квадрат 40 на 40 пикселей. Импортируем новый файл в Level и создаём новый метод `create_attack`.

```
def create_attack(self):
    Weapon(self.player, [self.visible_sprites])
```

Далее в самой отрисовке игрока, добавим наш метод как ссылку на него:

```
self.player = Player((1000, 1000), [self.visible_sprites],
self.obstacle_sprites, self.create_attack)
```

Далее перейдём в файл player.py и там передадим create_attack и создадим демона. Также, при нажатии на пробел, добавим метод `self.create_attak()`. Теперь можем вернуться к отрисовке нашего оружия:

```
import pygame

class Weapon(pygame.sprite.Sprite):
    def __init__(self, player, groups):
        super().__init__(groups)
        direction = player.status.split('_')[0] #обрезаем строку по "_" чтобы
понимать куда он смотрит
        full_path = f'../graphic/weapons/{player.weapon}/{direction}.png' #адрес
оружия
        self.image = pygame.image.load(full_path).convert_alpha() #сама графика
```

```
        if direction == 'right':
            self.rect = self.image.get_rect(midleft = player.rect.midright +
pygame.math.Vector2(-10, 16))
        elif direction == 'left':
            self.rect = self.image.get_rect(midright = player.rect.midleft +
pygame.math.Vector2(10, 16))
        elif direction == 'down':
            self.rect = self.image.get_rect(midtop = player.rect.midbottom +
pygame.math.Vector2(-15, 0))
        else:
            self.rect = self.image.get_rect(midbottom = player.rect.midtop +
pygame.math.Vector2(-15, 0))
```

Интересные моменты: во-первых, обрезаем строку по "_" чтобы понять куда смотрит герой (неважно с "attack" или без). Затем отрисовка. Нам нужно чтобы меч рисовался в руке у Линка. Я выбрал следующий метод, если меч слева, то "приклеиться" он должен справа от Линка, то есть `midleft = player.rect.midright + pygame.math.Vector2(-10, 16)`. Далее, плюсуем вектор направления чтобы меч был ровно в руке. Далее всё повторяется в зависимости от направления.

Добавим в player.py новых демонов:

```
self.create_attak = create_attak #создали атаку
self.weapon_index = 0 #номер оружия (если у вас будет несколько орудий пыток
монстров)
self.weapon = list(weapon_data.keys())[self.weapon_index] #выбрали конкретное
оружие и все его параметры
```

Теперь есть меч, но есть проблема. Мечи не исчезают:

Создадим новый метод в `level.py`: `destroy_weapon`. Помимо прочего, нужно немного переработать код в самом `level.py`. Создадим нового демона `self.current_attack = None`, а также переработаем метод `create_attack`.

```
def create_attack(self):
    self.current_attack = Weapon(self.player, [self.visible_sprites])

def destroy_weapon(self):
    if self.current_attack:
        self.current_attack.kill()
    self.current_attack = None
```

Таким образом, при наличии атаки, мы убиваем процесс и обнуляем указатель атаки (`current_attak`). Нужен ещё кулдаун. Также, не забудьте сослаться на `self.destroy_weapon` в player:

```
self.player = Player((1000, 1000), [self.visible_sprites],
self.obstacle_sprites, self.create_attack, self.destroy_weapon)
```

Далее пропишем метод в player.py и внесём в кулдаун новую функцию: `self.destroy_weapon()`. Результат:

Файлы этапа можно скачать здесь. Теперь настроим интерфейс.

Интерфейс

Наша задача — сделать интерфейс игры. Первым делом, нужно создать такие простые вещи как базовые параметры. У нас уже есть один параметр. Это скорость. Сейчас допишем остальные параметры. В демоны в файле `player.py` запишем следующие параметры:

```
self.stats = {'health': 100, 'energy': 60, 'attack': 10, 'speed': 5} #все
параметры героя
self.health = self.stats['health'] #соотношение со здоровьем
self.energy = self.stats['energy'] #соотношение с энергией
self.exp = 4221 #количество очков
self.speed = self.stats['speed'] #скорость игрока
```

Все параметры указаны и можно удалить параметр скорости из демона ранее. Особенно хорошо, что Линк уже набрал 4221 очка за просмотр себя на ютюбе у Артура из Уфы. В файле settings.py Добавим немного параметров отображения:

```
BAR_HEIGHT = 20 #толщина всех панелек
HEALTH_BAR_WIDTH = 200  #длина панельки здоровья
ENERGY_BAR_WIDTH = 140 #длина панельки энергии
ITEM_BOX_SIZE = 80 #размер значка под предмет
UI_FONT = '../graphic/font/joystix.ttf' #основной шрифт
UI_FONT_SIZE = 18 #кегель шрифта

UI_BG_COLOR = '#222222' #цвет задника
UI_BORDER_COLOR = '#111111' #цвет обводки
TEXT_COLOR = '#EEEEEE' #цвет текста

HEALTH_COLOR = 'red' #цвет здоровья
ENERGY_COLOR = 'green' #цвет энергии
```

Тут самый важный момент — добавить шрифт. Самый его большой плюс в том, что он поддерживает русский язык. Теперь в настройках уровня укажем нового демона `self.ui = UI()` и импортируем сам файл в проект. Добавим в run-метод правило отрисовки:

```
self.ui.display(self.player)
```

Теперь создадим новый файл `ui.py`:

```
import pygame
from settings import *

class UI:
    def __init__(self):
        self.display_surface = pygame.display.get_surface() #прорисовка самого
экрана
        self.font = pygame.font.Font(UI_FONT, UI_FONT_SIZE) #добавление
стандартного шрифта
        self.healt_bar_rect = pygame.Rect(10, 10, HEALTH_BAR_WIDTH, BAR_HEIGHT)
#панелька для здоровья
        self.energy_bar_rect = pygame.Rect(10, 34, ENERGY_BAR_WIDTH, BAR_HEIGHT)
#панелька для энергии
```

```
    def show_bar(self, current, max_amountm, bg_rect, color):
        pygame.draw.rect(self.display_surface, UI_BG_COLOR, bg_rect) #отрисовка
задника панельки
        ratio = current / max_amountm
        current_with = bg_rect.width * ratio
        current_rect = bg_rect.copy()
        current_rect.width = current_with
        pygame.draw.rect(self.display_surface, color, current_rect) #рисование
панельки
        pygame.draw.rect(self.display_surface, UI_BORDER_COLOR, bg_rect, 3)
#рисование обводки панельки

    def display(self, player):
        self.show_bar(player.health, player.stats['health'],
self.healt_bar_rect, HEALTH_COLOR) #обращение к отрисовке панели здоровья
        self.show_bar(player.energy, player.stats['energy'],
self.energy_bar_rect, ENERGY_COLOR) #обращение к отрисовке панели энергии
```

Из интересного, тут есть строки рассчётов:

```
ratio = current / max_amountm
current_with = bg_rect.width * ratio
current_rect = bg_rect.copy()
current_rect.width = current_with
```

Мы переводим значения нашего здоровья в проценты прорисовки в поле пикселей. Нам нужно перевести даже 140 очков здоровья в 100%. Для этого, мы запрашиваем `current` — очки здоровья и делим их на длину нашего поля здоровья (`max_amountm`), далее перемножаем это с `bg_rect.width` для отображения в поле. Я поставил в player.py значение здоровья на -10 и -50 на энергию:

```
self.health = self.stats['health'] - 10 #соотношение со здоровьем
self.energy = self.stats['energy'] - 50 #соотношение с энергией
```

Результат:

Нарисуем же наши очки. Переходим опять к `ui.py` и создадим новый метод `show_exp`:

```
def show_exp(self, exp):
    text_surf = self.font.render(str(int(exp)), False, TEXT_COLOR) #рендер шрифта с экспой, без сглаживания и с цветом
    x = self.display_surface.get_size()[0] - 20 #отступ снизу на 20 пикселей по x
    y = self.display_surface.get_size()[1] - 20 #отступ снизу на 20 пикселей по y
    text_rect = text_surf.get_rect(bottomright = (x, y)) #цепляемся за низ экрана справа

    pygame.draw.rect(self.display_surface, UI_BG_COLOR, text_rect.inflate(20, 20)) #цепляемся за низ экрана справа
    self.display_surface.blit(text_surf, text_rect) #отображаем панельку
    pygame.draw.rect(self.display_surface, UI_BORDER_COLOR, text_rect.inflate(20, 20), 3) #обводка панели
```

Из интересного: тут стоит False в методе `self.font.render`. Данный метод выключает сглаживание, так как у меня пиксельный шрифт. А также, pygame сделали крутой метод `text_rect.inflate`, который помогает вписать текст в панельку. Результат удивительно прекрасен:

Последний шаг — прорисовка меча Линка. Для начала, нужно прописать в базовых демонах наш список оружия. У меня в наличии только одно оружие, так что этот кусок кода можно упростить, но я оставил возможность подбирать оружие и добавить его в словарь оружия, а затем и в этот список:

```
self.weapon_graphics = []
for weapon in weapon_data.values():
    path = weapon['graphic']
    weapon = pygame.image.load(path).convert_alpha()
    self.weapon_graphics.append(weapon)
```

Далее пишем простой метод для отображения предметов:

```
def weapon_overlay(self, weapon_index):
    bg_rect = pygame.Rect(10, 630, ITEM_BOX_SIZE, ITEM_BOX_SIZE)
    pygame.draw.rect(self.display_surface, UI_BG_COLOR, bg_rect)
    weapon_surf = self.weapon_graphics[weapon_index]
    weapon_rect = weapon_surf.get_rect(center = bg_rect.center)

    self.display_surface.blit(weapon_surf, weapon_rect)
```

В конце не забудьте добавить отображение оружия в display-методе: `self.weapon_overlay(player.weapon_index)`.

Результат:

Файлы проекта и дальше будем писать монстры (более известных как бебы).

Монстры

Для начала, я прописал референсы для монстры. Они хранятся в папке `graphic/monsters`. Также я добавил звуки ударов и закинул их в папку `audio/attack` Далее, в setting.py прописаны их входные данные:

```
monster_data = {
    'axalot': {'health': 200, 'exp': 400, 'damage': 40, 'attack_type': 'slash',
'attack_sound': '../audio/attack/slash.wav', 'speed': 3, 'resistance': 3,
'attack_radius': 80, 'notice_radius': 300},
    'lizard': {'health': 50, 'exp': 100, 'damage': 15,'attack_type': 'claw',
 'attack_sound': '../audio/attack/claw.wav', 'speed': 2, 'resistance': 3,
'attack_radius': 100, 'notice_radius': 400},
    'snake': {'health': 100,'exp':100,'damage': 10,'attack_type': 'claw',
'attack_sound': '../audio/attack/claw.wav', 'speed': 4, 'resistance': 3,
'attack_radius': 80, 'notice_radius': 350},
    'spirit': {'health': 150,'exp':200,'damage': 15,'attack_type': 'claw',
'attack_sound': '../audio/attack/claw.wav', 'speed': 3, 'resistance': 3,
'attack_radius': 100, 'notice_radius': 400}}
```

Пробежимся по параметрам:

- `health` — здоровье монстра
- `exp` — сколько очков за смерть монстра
- `damage` — какой урон он нанесёт герою
- `attack_type` — тип атаки
- `attack_sound` — звук удара
- `speed` — скорость зверька
- `resistance` — на сколько монстр отлетит после нашего удара
- `attack_radius` — радиус, с которого монстр опасен и может ударить
- `notice_radius` — радиус зрения монстра

Далее, создадим новый файл сущностей (`entity.py`) и добавим туда супер демона с наследованием групп:

```
import pygame

class Entity(pygame.sprite.Sprite):
    def __init__(self, groups):
        super().__init__(groups)
```

Скопируем методы `move` и `collision` из `player.py`. Теперь удаляем из `player.py` эти методы и будем ссылаться в классе не на `pygame.sprite.Sprite`, а на `Entity` (не забывайте импортировать файл). Эти небольшие танцы с бубном нужны для того, чтобы не переписывать каждый раз правила движений для нашего героя и монстров. Все они — одинаковые сущности. Также я перенёс демонов скорости анимации, фрейма и определения вектора скорости. Когда всё сделаете, перепроверьте, что всё работает.

Затем, наконец, создадим файл `enemy.py`:

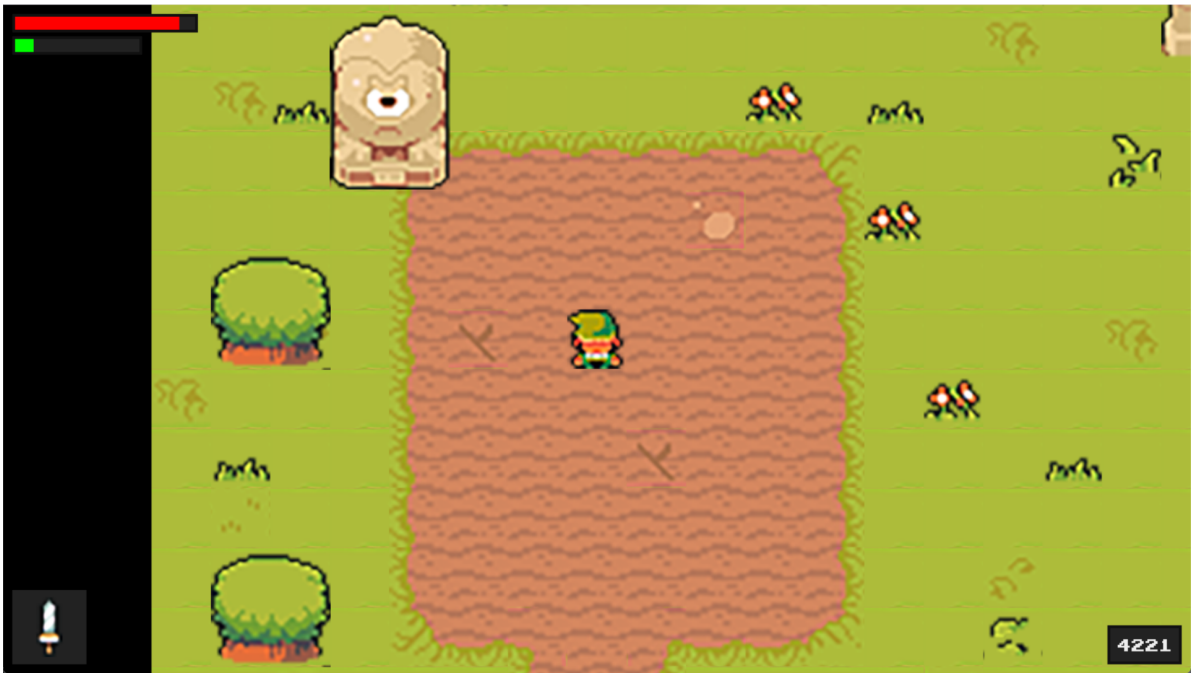
```
import pygame
from settings import *
from entity import Entity

class Enemy(Entity):
    def __init__(self, monster_name, pos, groups):
        super().__init__(groups)
        self.sprite_type = 'enemy' #новый тип спрайтов – враги
        self.image = pygame.Surface((64, 64)) #наш традиционный размер тайла
        self.rect = self.image.get_rect(topleft = pos) #традиционная отрисовка
```

Перейдём к настройке уровня. Для начала, сделаем так, чтобы наш герой спаунился там, где надо (зелёный квадрат на карте). Для этого, нам нужно импортировать новый csv-файл из уже созданной карты `'entities': import_csv_layout('../map/map_Spawn.csv')`. И пропишем в методе `creat_map` новый объект:

```
if style == 'entities':
    if col == '8':
    self.player = Player((x, y), [self.visible_sprites], self.obstacle_sprites,
self.create_attack, self.destroy_weapon)
```

Результат вас удивит:

Герои Хайрула размножились. Проблема в том, что я прорисовывал карту разными наборами элементов. Не повторяйте моих ошибок и давайте всё исправлять. Дело в том, что номер тайла автоматически рассчитывается программой Tiles, а я указал, разные тайлы и номера задублировались, так как у меня был отдельный файл Bebs.png для спауна врагов и Link_and_block.png для Линка и блоков-стен. Теперь я объединил два набора тайлов и присвоил линку номер 16. Результат:



Линк на своём законном месте. Давайте заспаумим врагов, добавив лишь один else:

```
else:
    Enemy('monster', (x, y), [self.visible_sprites])
```

Монстры отобразились, но теперь нужно отобразить их верно. Работаем с файлом `enemy.py`:

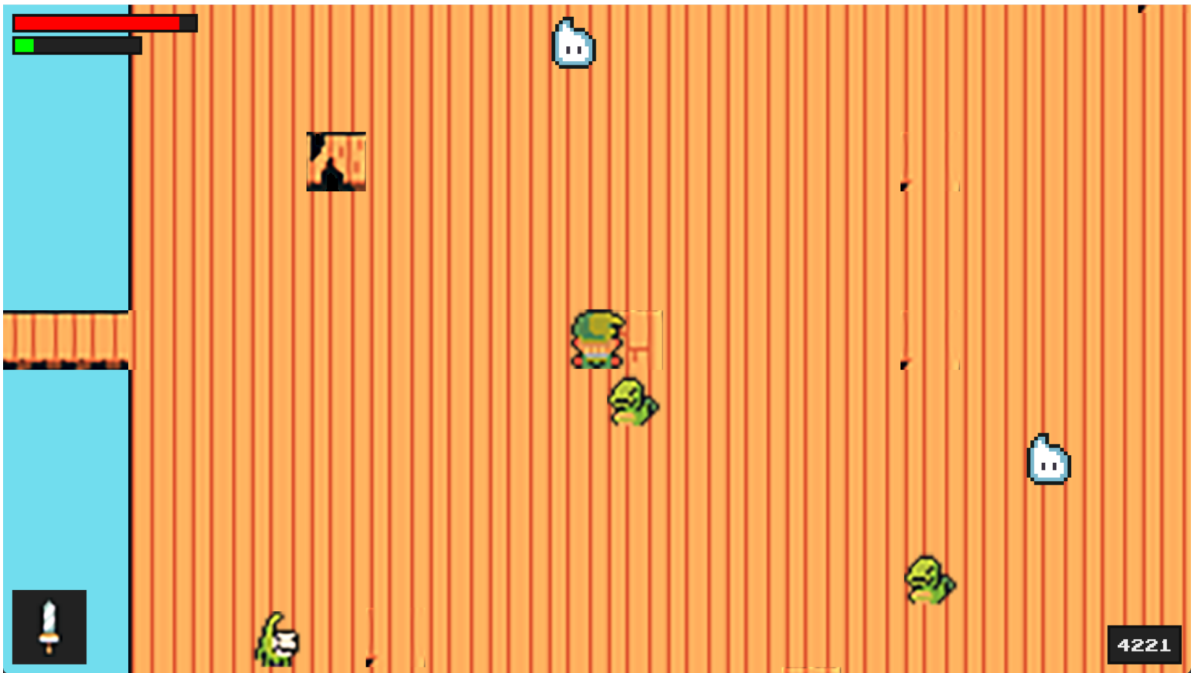
```
import pygame
from settings import *
from entity import Entity
from support import *

class Enemy(Entity):
    def __init__(self, monster_name, pos, groups):
        super().__init__(groups)
        self.sprite_type = 'enemy' #новый тип спрайтов – враги
        self.import_graphics(monster_name) #обращаемся к новой функции перебора
картинок
```

```
        self.status = 'idle' #установим базовый статус
        self.image = self.animations[self.status][self.frame_index] #перебираем
номер фрейма в папке из функции ниже
        self.rect = self.image.get_rect(topleft = pos) #традиционная отрисовка

    def import_graphics(self, monster_name):
        self.animations = {'idle': [], 'move': [], 'attack': []} #перебираем
возможные варианты анимаций в папках
        main_path = f'../graphic/monsters/{monster_name}/' #обращаемся к монстру
по имени :)
        for animation in self.animations.keys(): #перебираем все картинки
            self.animations[animation] = import_folder(main_path + animation)
#перебор благодаря support-файлу
```

Тут мы делаем всё ровно также, как и ранее, но если в файле `level.py` мы внесём имя любого монстра, то получим картинку монстра на карте:

Осталось только перебрать монстров по их номерам тайлов на карте:

```
else:
    if col == '0':
        monster_name = 'axolot'
    elif col == '4':
        monster_name = 'lizard'
    elif col == '8':
        monster_name = 'snake'
    else:
    monster_name = 'spirit'

    Enemy(monster_name, (x, y), [self.visible_sprites])
```



Теперь, добавим аргумент `obstacle_sprites` в нашу конструкцию, чтобы монстры могли взаимодействовать с Линком. Далее, создадим update-метод для файла `enemy.py`:

```
def update(self):
    self.move(self.speed)
```

Далее, нужно прописать несколько статусов для наших плохишей. Добавим их в демонов данного файла:

```
self.monster_name = monster_name #имя монстра
monster_info = monster_data[self.monster_name] #перехват данных монстра по имени
self.health = monster_info['health']
self.exp = monster_info['exp']
self.speed = monster_info['speed']
self.attack_damage = monster_info['damage']
self.resistance = monster_info['resistance']
self.attack_radius = monster_info['attack_radius']
self.notice_radius = monster_info['notice_radius']
self.attack_type = monster_info['attack_type']
```

Тут мы ссылаемся на файл settings.py и перехватываем все параметры монстров оттуда. Теперь наша задача прописать метод определения дистанции до объекта. Я думал, что эта задача непроста, так как координаты объекта рассчитываются с верхнего левого угла, у них есть свои векторы (скорости), да ещё и нужна нормализация для предотвращения "диагонального чита" (как это было у Линка). Собственно весь метод:

```
def get_player_distance_direction(self, player):
    enemy_vec = pygame.math.Vector2(self.rect.center) #координата врага
    player_vec = pygame.math.Vector2(player.rect.center) #координата Линка
    distance = (player_vec - enemy_vec).magnitude() #Евклидова величина
    if distance > 0:
        direction = (player_vec - enemy_vec).normalize() #вычисление вектора
сближения
    else:
        direction = pygame.math.Vector2() #точка, мы друг в друге
    return(distance, direction)
```

Я искренне не ожидал, что это **так** просто. По сути, все сложные методы вычисления Евклидовой величины по поиску дистанции мы переложили на функцию `magnitude()`, а с читерской функцией `normalize()` вы уже знакомы. И зачем я учил математику? Далее, пропишем метод определения статуса монстра по отношению к Линку:

```
def get_status(self, player):
    distance = self.get_player_distance_direction(player)[0]
    if distance <= self.attack_radius:
        self.status = 'attack'
    elif distance <= self.notice_radius:
        self.status = 'move'
    else:
        self.status = 'idle'
```

Тут мы отсекаем изнутри во вне "окружности" зрения (близко — атака, средняя дистанция — преследование, далеко — idle), но чтобы оно заработало, нам нужно обновлять данные в файле `level.py`:

```
def enemy_update(self, player):
    enemy_sprites = [sprite for sprite in self.sprites() if
hasattr(sprite,'sprite_type') and sprite.sprite_type == 'enemy']
    for enemy in enemy_sprites:
        enemy.enemy_update(player)
```

Тут самая интересная строка — строка прорисовывания спрайтов для врага. Тут можно как в анекдоте: "Потерялся атрибут? Ничего страшного! Всегда есть метод `hasattr`". Далее, в run-методе пропишем отрисовку спрайтов врага:

```
self.visible_sprites.enemy_update(self.player)
```

Теперь мы сможем замкнуть врага на игрока, а игрока на уровень. Для этого пропишем новый метод в `enemy.py`:

```
def enemy_update(self, player):
    self.get_status(player)
```

Теперь, у нас есть способ получения методов, но мы с ними не взаимодействуем. Исправим это новым методом:

```
def actions(self, player):
    if self.status == 'attack':
        print('attack') #тут мы только пишем в терминале атаку
    elif self.status == 'move':
        self.direction = self.get_player_distance_direction(player)[1]
#нанюхивать Линка
    else:
        self.direction = pygame.math.Vector2() #остановиться по координатам
```

В этом методе всё так же, как было ранее, но не забудьте закинуть его вызов в `enemy_update`-функцию командой `self.actions(player)`. Тетерь пропишем анимацию. Она полностью аналогична анимации Линка:

```
def animate(self):
    animation = self.animations[self.status]
    self.frame_index += self.animation_speed
    if self.frame_index >= len(animation):
        self.frame_index = 0
    self.image = animation[int(self.frame_index)]
    self.rect = self.image.get_rect(center = self.hitbox.center)
```

Также, добавьте animate в update-функцию. Теперь можно получить ачивку: "Собрал всех чушпанов с района":

Но есть проблема. Они атакуют несчастного Линка толпой без остановки. Это нужно исправить, а значит время нового метода и нового кулдауна. Сначала я добавил нового демона `self.can_attack = True`. Это флаг, который будет указывать на то, что монстр может пнуть Линка. Соответственно, нужно подправить условие атаки и помимо дистанции, указать данный флаг. Если вы добавили флаг на `True`, то обязательно сразу нужно прописать ситуацию, когда он будет опускаться (положение `False`). Запишем этот пункт в методе анимации:

```
if self.frame_index >= len(animation):
    if self.status == 'attack':
        self.can_attack = False
```

Немного объясню происходящее. Анимация атаки не должна прерывать анимацию перехода и если мы завершили весь цикл из переходов от картинки к картинке, то только тогда можно менять флаг на опущенное состояние. Простыми словами, все враги могут ударить нас только 1 раз, так как флаг не поднимается обратно. Поднимать тот самый флаг мы будем по кулдауну через паузу. То есть, я добавлю два демона, которые будут обозначать время атаки и кулдаун после атаки:

```
self.attack_time = None
self.attack_cooldown = 400
```

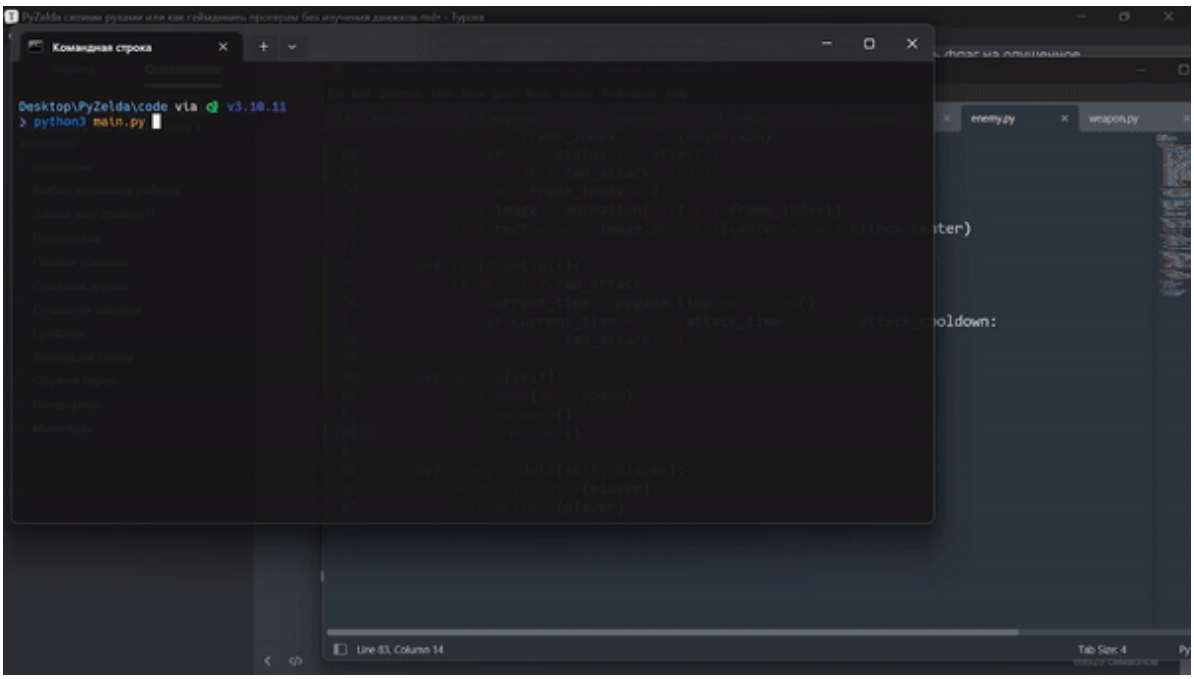
Теперь пропишем сам метод кулдауна по вычислению разницы текущего времени и времени задержки:

```
def cooldown(self):
    if not self.can_attack:
        current_time = pygame.time.get_ticks()
        if current_time - self.attack_time >= self.attack_cooldown:
            self.can_attack = True
```

Тут самое главное, не забыть про место старта времени, то есть про установку времени на момент атаки:

```
self.attack_time = pygame.time.get_ticks()
```

После этого, не забудьте закинуть метод в update-метод. Результат:



Мы не закончили работу с монстрами, но давайте оставлю бэкап проекта сейчас и в следующей части создадим методы взаимодействия нас с монстрами и монстров с нами.

Драки с монстрами

Итак, для начала создадим два новых демона для атак в `level.py`:

```
self.attack_sprites = pygame.sprite.Group() #атакующий спрайт
self.attackable_sprites = pygame.sprite.Group() #атакуемый спрайт
```

Думаю, по названиям понятно, что демоны нужны для обозначения процесса атаки. Дополним метод вызова врагов помимо видимых спрайтов, атакуемыми спрайтами (`self.attack_sprites`):

```
Enemy(monster_name, (x, y), [self.visible_sprites, self.attackable_sprites],
self.obstacle_sprites)
```

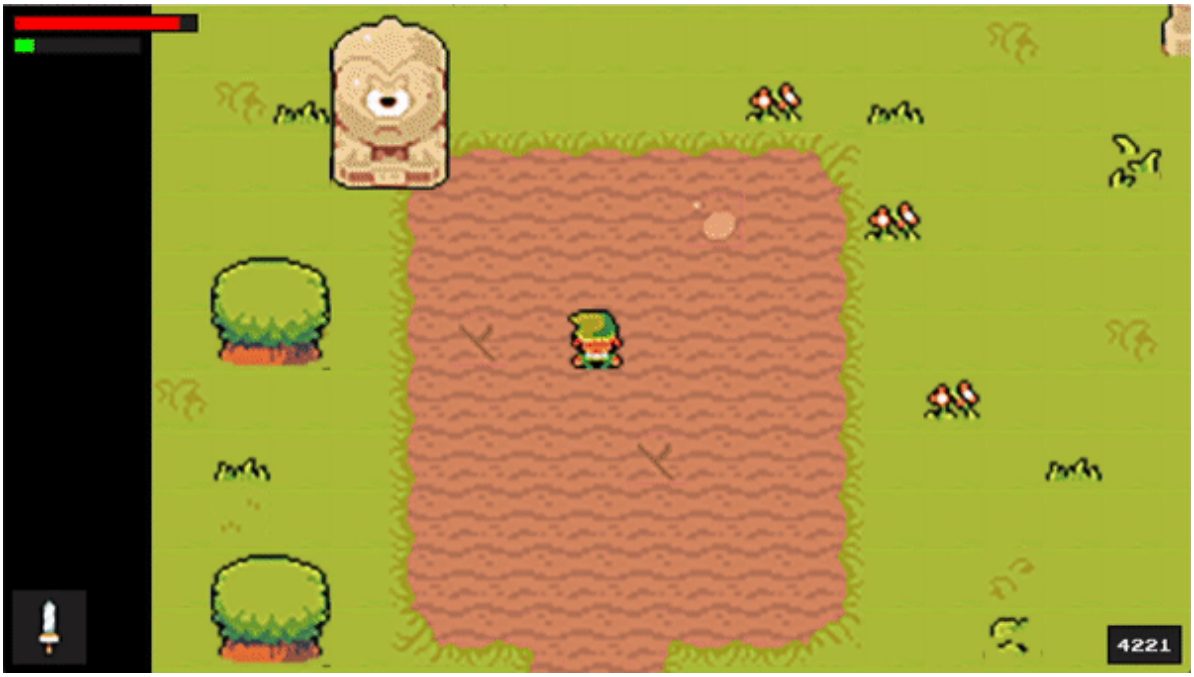
Также, дополним метод `create_attack` атакующим спрайтом (`self.attack_sprites`):

```
def create_attack(self):
    self.current_attack = Weapon(self.player, [self.visible_sprites,
self.attack_sprites])
```

Далее пропишем новый метод с логикой атаки игрока:

```
def player_attack_logic(self):
    if self.attack_sprites:
        for attack_sprites in self.attack_sprites:
            collision_sprites = pygame.sprite.spritecollide(attack_sprites,
self.attackable_sprites, False)
            if collision_sprites:
                for target_sprite in collision_sprites:
                    target_sprite.kill()
```

Самая интересная строка тут, это строка с методом пайгейма `pygame.sprite.spritecollide`. Данный метод позволяет удалять спрайт из группы. Первый аргумент функции — спрайты для атаки, второй — группа спрайтов, из которой мы будем удалять спрайт, третий — DoKill. Если у DoKill установлено значение `True`, все спрайты, которые сталкиваются, будут удалены из группы. Далее, в run-методе пропишем наш метод. Исход:



Немного улучшим метод `player_attack_logic`:

```
if target_sprite.sprite_type == 'enemy':
    target_sprite.get_damage(self.player, attack_sprites.sprite_type)
```

Мы стали сопоставлять наши спрайты по типам. В моём проекте типов только два (enemy и weapon). Я прописал в файле `weapon.py` в демоне строку для присваивания ему нового типа:

```
self.sprite_type = 'weapon'
```

Далее нам не нужно удалять врага при ударе. Нам нужно прописывать ему урон от нашего оружия. Собственно, теперь нужно в файле enemy.py прописать новый метод — `get_damage`:

```
def get_damage(player, attack_type):
    if attack_type == 'weapon':
        self.health -= player.get_full_weapon_damage()
```

Тут мы прописываем новый метод (`get_full_weapon_damage`), который должен высчитывать сумму урона от оружия и от силы самого Линка (прямо как в Dark Souls). Пропишем же данный метод в `player.py`:

```
def get_full_weapon_damage(self):
    base_damage = self.stats['attack'] #урон самого Линка
    weapon_damage = weapon_data[self.weapon]['damage'] #урон от выбранного
оружия
    return base_damage + weapon_damage
```

Находим и складываем уроны из наших списков. Возможно, тут встанет вопрос: "Зачем так сложно, если у нас только один меч и всё?". Я хотел бы сделать проект так, чтобы вы могли самостоятельно с ним "поиграться". Собственно и цель книги не номинация "Игра года" в The Game Awards, а лишь попытка продемонстрировать работоспособность языка Python как неплохого движка. Ну да вернёмся к коду. Помимо урона, я решил сразу прописать кулдаун оружия и заменил строку:

```
if current_time - self.attack_time >= self.attack_cooldown
```

На строку:

```
if current_time - self.attack_time >= self.attack_cooldown +
weapon_data[self.weapon]['cooldown']
```

Напишем новый метод в `enemy.py` на проверку смерти монстра:

```
def check_death(self):
    if self.health <= 0:
        self.kill()
```

Тут я даже не знаю что ещё подсветить в коде :) Не забудьте добавить данный метод в update-метод. Теперь один удар приводит к смерти врага. Таким образом, PyGame считает, что пока оружие соприкасается с врагом (вызывается метод коллизий), удары наносятся один за другим. Как итог — Линк танк, который уничтожает всё на своём пути. Исправим это. Для начала, создадим новых демонов `enemy.py`:

```
self.vulnerable = True #флаг уязвимости
self.hit_time = None #время удара
self.invincibility_duration = 300 #продолжительность неуязвимости
```

Тут достаточно прозрачные демоны. Важный момент — флаг уязвимости. Если он поднят — враг может получать урон. Теперь встроим их в `get_damage`:

```
def get_damage(self, player, attack_type):
    if self.vulnerable:
        if attack_type == 'weapon':
            self.health -= player.get_full_weapon_damage()
        self.hit_time = pygame.time
```

Далее необходимо дополнить код cooldown-метода:

```
if not self.vulnerable:
    if current_time - self.hit_time >= self.invincibility_duration:
        self.vulnerable = True
```

Тут, как я говорил ранее, сразу добавляем вариацию флага. У нас было место, где флаг опускается и теперь в cooldown-методе он поднимается по истечению указанного времени неуязвимости. Теперь все враги убиваются весьма приятно. Теперь, нужно добавить отбивание врага на дистанцию, которая указана у каждого врага. Создадим ещё метод:

```
def hit_reaction(self):
    if not self.vulnerable:
        self.direction *= -self.resistance
```

Осталось вычислить положение в методе `get_damage`:

```
self.direction = self.get_player_distance_direction(player)[1]
```

Теперь добавим мерцание во время удара, чтобы понять что удар был сделан. В PyGame все сигнатуры (синусоидные функции) лежат в диапазоне от -255 до 255, а позиции удобно брать из синусоид, так как интерпретация в Python будет работать на основе степеней (как и любой калькулятор), а затем будет процесс получения точки на синусоиде. Этот процесс я описывал дольше, чем будет писаться метод отображения пульсации:

```
def wave_value(self):
    value = sin(pygame.time.get_ticks())
    if value >= 0:
        return 255
    else:
        return 0
```

Данный метод написан в `entity.py`. Не забудьте импортировать sin-метод из библиотеки math. Теперь вызовем данный метод при ударе по врагу в методе `animate`:

```
if not self.vulnerable:
    alpha = self.wave_value()
    self.image.set_alpha(alpha)
else:
    self.image.set_alpha(255)
```

Сейчас мы бьём врагов абсолютно верно и можем отследить когда монстры атакуют нас (в терминал приходит сообщение "attack"). Осталось сделать метод, который делает урон Линку. Пропишем его в `level.py`:

```
def damage_player(self, amount, attack_type):
    if self.player.vulnerable:
        self.player.health -= amount
        self.player.vulnerable = False
        self.player.hurt_time = pygame.time.get_ticks()
```

Дополним в `create_map` в строку вызова монстров нанесение урона Линку:

```
Enemy(monster_name, (x, y), [self.visible_sprites, self.attackable_sprites],
self.obstacle_sprites, self.damage_player)
```

Добавим в Enemy-класс параметр (`damage_player`) и пропишем нового демона — `self.damage_player = damage_player`. Теперь вместо простого вывода сообщения "attack" выполним вызов нашего метода:

```
self.damage_player(amount, attack_type)
```

Добавим параметры таймеров в демонов нашего player-класса:

```
self.vulnerable = True
self.hurt_time = None
self.invulnerablity_duration = 500
```

Они аналогичны демонам в `enemy.py`. Далее традиционно пропишем смену флага в наш кулдаун-метод:

```
if not self.vulnerable:
    if current_time - self.hurt_time >= self.invulnerablity_duration:
        self.vulnerable = True
```

Теперь пропишем наше мерцание. Тут всё также, как и ранее. Прописывать будем в animate-методе:

```
if not self.vulnerable:
    alpha = self.wave_value()
    self.image.set_alpha(alpha)
else:
    self.image.set_alpha(255)
```

Теперь нам нужно восстанавливать энергию (повторим механику из Dark Souls). Для этого создадим метод `energy_recover`:

```
def energy_recover(self):
    if self.energy < self.stats['energy']:
        self.energy += 0.1
    else:
        self.energy = self.stats['energy']
```

Далее пропишем, что при ударе у нас теряется 10 очков стамины, а если её не хватает — атака не проходит:

```
if keys[pygame.K_SPACE]:
    if self.energy >= 10:
        self.energy -= 10
        self.attacking = True
        self.attack_time = pygame.time.get_ticks()
        self.create_attak()
        self.weapon_attack_sound.play()
```

Последнее, что я хотел бы сделать — добавить экспу за убийство врага в `level.py`:

```
def add_xp(self, amount):
    self.player.exp += amount
```

Добавим этот же метод для `create_map` в `Enemy`. Далее повторим всё, что делали ранее с `damage_player`.

Файлы данного шага тут. Приступим к последнему шагу — музыке.

Музыка

Мы на финишной прямой. В `player.py` добавим демонов:

```
self.weapon_attack_sound = pygame.mixer.Sound('../audio/attack/slash.wav')
self.weapon_attack_sound.set_volume(0.4)
```

Первый демон укажет на название файла, а второй на громкость. Далее, вызовем звук при нажатии на клавишу:

```
self.weapon_attack_sound.play()
```

Повторим успех с ударом от монстра:

```
self.hit_sound = pygame.mixer.Sound('../audio/attack/claw.wav')
self.hit_sound.set_volume(0.4)
```

```
self.hit_sound.play()
```

Данный метод я вызвал в get_damage после успешного попадания. Теперь мы слышим попадания при ударе. А монстр? Добавим ещё пару демонов и вызов:

```
self.attack_sound = pygame.mixer.Sound(monster_info['attack_sound'])
self.attack_sound.set_volume(0.3)
```

```
self.attack_sound.play()
```

Вызов функции в методе actions-методе в `enemy.py`. Осталось только главная музыка игры. Переходим в `main.py`:

```
main_sound = pygame.mixer.Sound('../audio/main.wav')
main_sound.play(loops = -1)
```

Это два новых демона. Отличие только одно — метод `play(loops = -1)`. Тут мы создаём музыку, которая будет играть снова и снова во время игры и не закончится до выхода.

В конце, я забыл, что игра не заканчивается. Я исправил это функцией `final` в `level.py`:

```
def final(self):
    if self.player.health <= 0:
        print('\n', '\n', 'Линк умер. Хана Хайрулу')
        exit()
    if self.player.exp > 5000:
        print('\n', '\n', 'Линк победил')
        exit()
```

Тут всё просто. При получении более 5000 очков — вы победили, менее — проиграли. Ну и конечно, вернул здоровье и энергию на 100%, а экспу на 0. Все файлы с доработками оставлю тут. Вот что получилось:

Деплой

Как и любой шедевр, игру должен увидеть свет. Если сейчас скачать все файлы с доработками — можно поиграть в игру, но с несколькими "но":

1. Мы не сможем запустить игру без ПК на Windows или Linux. Это нормально, учитывая, что мы писали игру под компьютер.
2. Нужно установить python на машину, а для этого хоть немного понимать в программировании и не бояться терминала и командной строки. Это уже весомый минус.
3. Нужно скачать все библиотеки. Так как мы использовали только pygame, то её нам и нужно скачать. Для питониста — ничего сложного, а вот для геймера — трудность и шанс скинуть игру в ту, которую он не будет смотреть.

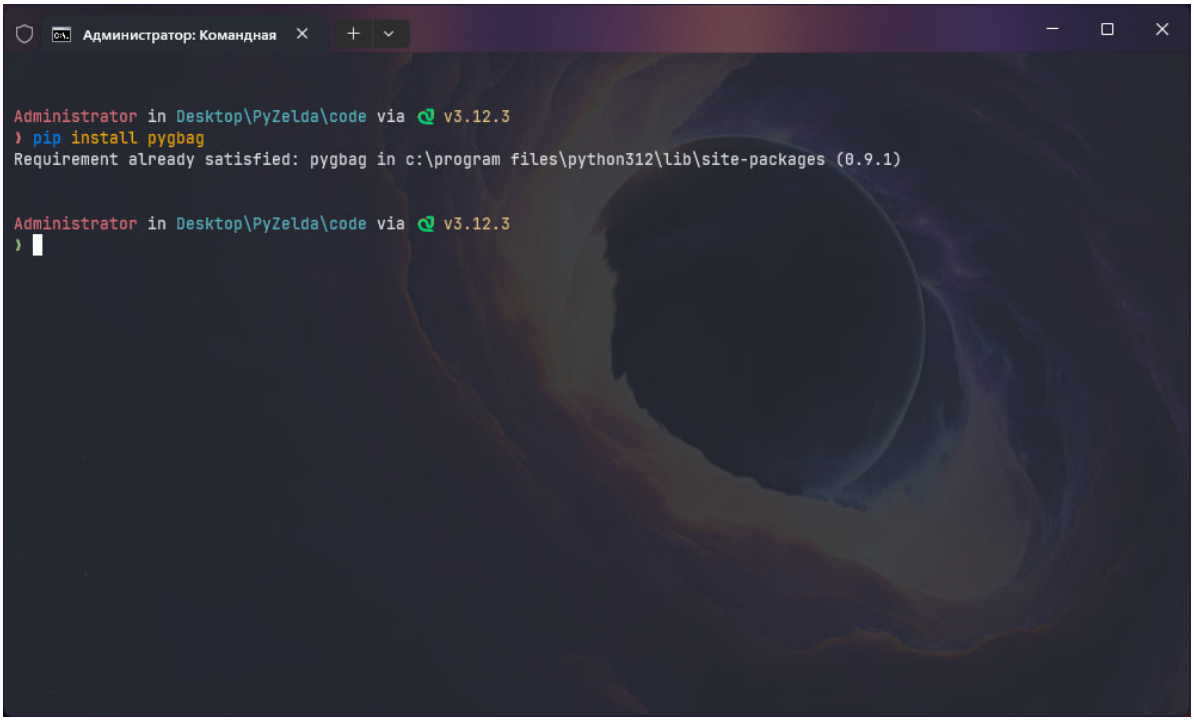
Из-за этого, было принято решение (спасибо за идею Стасу Фомину) отдеплоить Зельду на web-интерфейс.

Деплой — это процесс выгрузки проекта в сеть Интернет. Игры в Интернете ушли с уходом Adobe Flash Player, так как писать игры на флеше было бы глупо, учитывая, что Adobe запретили использовать их утилиту в сети из-за отсутствия безопасности. Из-за этого все онлайн-игры перешли либо на мобильные устройства, либо на язык, который проприетарно был нужен для работы с вебом (JavaScript), либо студии пишут свою платформу и архитектуру для своих же серверов. Писать сервер на питоне — невозможно, так как питон является высокоуровневым языком программирования и написать драйвера (программы для работы с оборудованием) на нём невозможно (давайте не будет пытаться запустить на стороннем ПК интерпретатор, который будет посылать запросы на ассемблере серверу). Возрождать флеш — тоже не самая лучшая идея, так как смахивает на некрофилию и пережитки технологий (что мертво - умереть не может, поэтому на просторах Интернета можно всегда найти версии флеш-плеера). На JS я не хочу идти, так как не по-геройски конкурентов поддерживать. Решение было предоставлено Веб Ассамблеей.

WebAssembly — язык программирования низкого уровня, призванный внести программируемость туда, где нужны кроссплатформенность, эффективность и безопасность, в первую очередь на клиентскую сторону Всемирной паутины. Программирование идёт на обычных статически типизированных языках, таких как Си, C++, C#, Rust, Go. Стековая виртуальная машина, исполняющая инструкции бинарного формата wasm, может быть запущена как в среде браузера, так и в серверной среде. Код на wasm — переносимое абстрактное синтаксическое дерево, что обеспечивает как более быстрый анализ, так и более эффективное выполнение в сравнении с JavaScript.

Как вы уже могли понять, wasm также работает с Python. Я буду запускать код на своём ПК, wasm будет его переводить в "околобинарный" вид. Приставка"около", так как мы работаем в псевдопростанстве, где невозможно перевести всё в бинарники, но можно написать правила взаимодействия с нашими стандартами сжатия и работы с высокоуровневыми системами. Более того, мы даже не запускаем код, а запускаем отпечаток, который запустился на моём ПК и этот отпечаток деплоим в веб. Этот подход "съест" настраиваемость, но даст прирост в производительности. Код работает быстрее, чем на JS (Хах!). Ну да хватить разглагольствовать, давайте приступать.

Первым делом устанавливаем библиотеку pygbag.



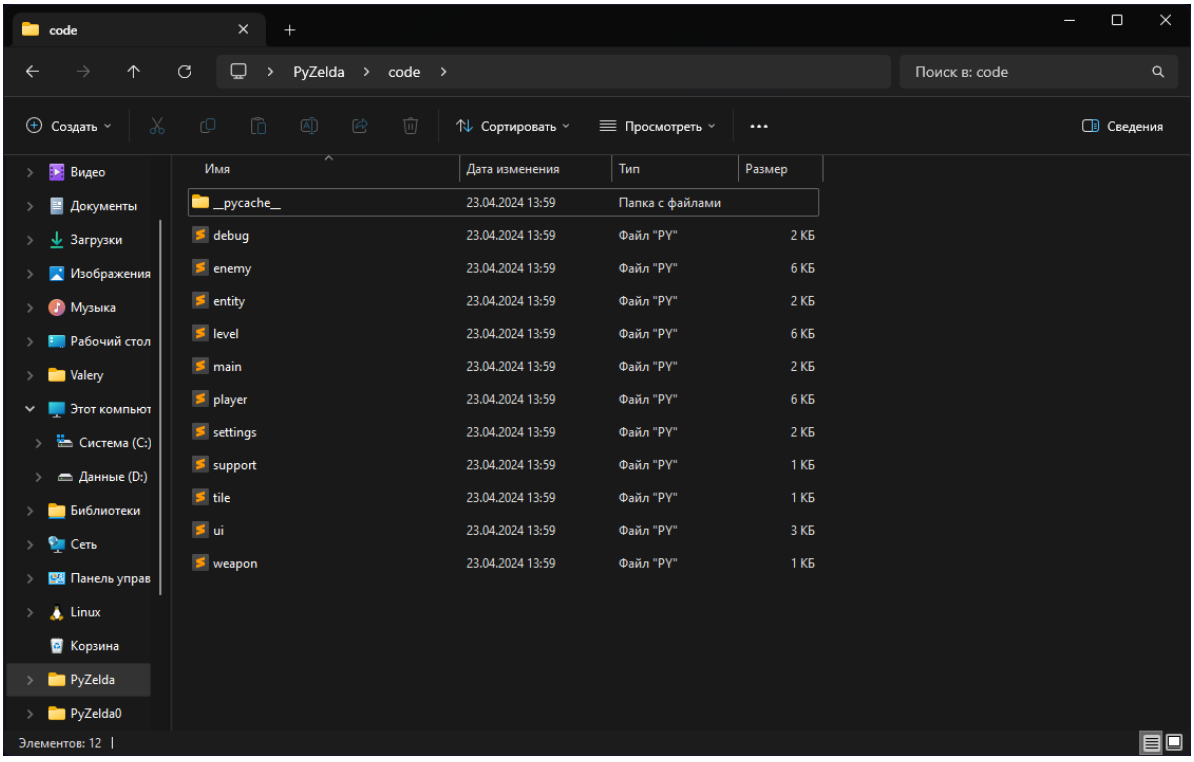
Тут всё просто: `pip install pygbag`. Это вторая библиотека, которую мы используем за всё время работы (не считая базовые). Вы можете этого не делать, если хотите оставить проект в самом первозданном виде. Я же хочу, чтобы в игру поиграли другие пользователи и выложу её в сеть.

Теперь, как написано в официальной документации, мне нужно пересобрать проект по следующим позициям:

1. Перенести файл запуска игры в корень. Я решил убрать директорию "code".
2. Внутри main-функции нужно создать отслеживание потока.
3. Нужно сжать всё, что можно сжать. В моём случае, это .wav-фалы. Их я сожму до .ogg.
4. Переписать все пути внутри всех файлов (самая кропотливая работа).

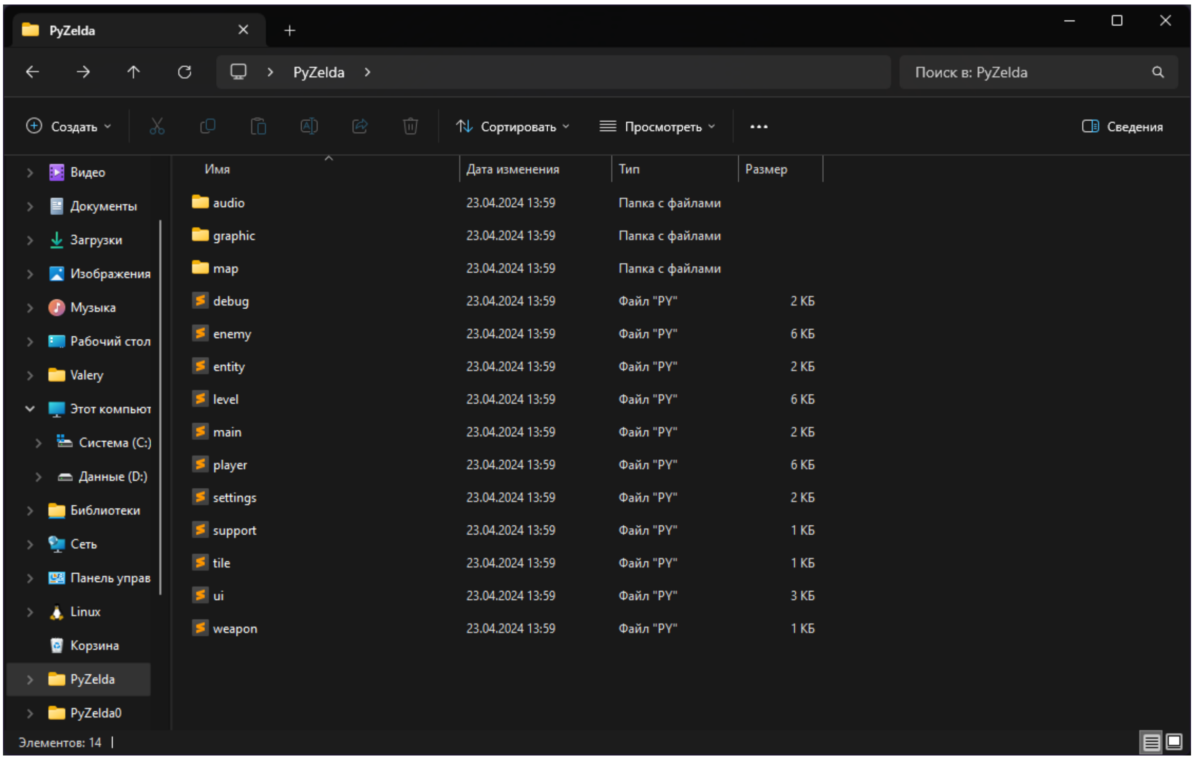
Приступим.

Сейчас моя папка "code" выглядит так:

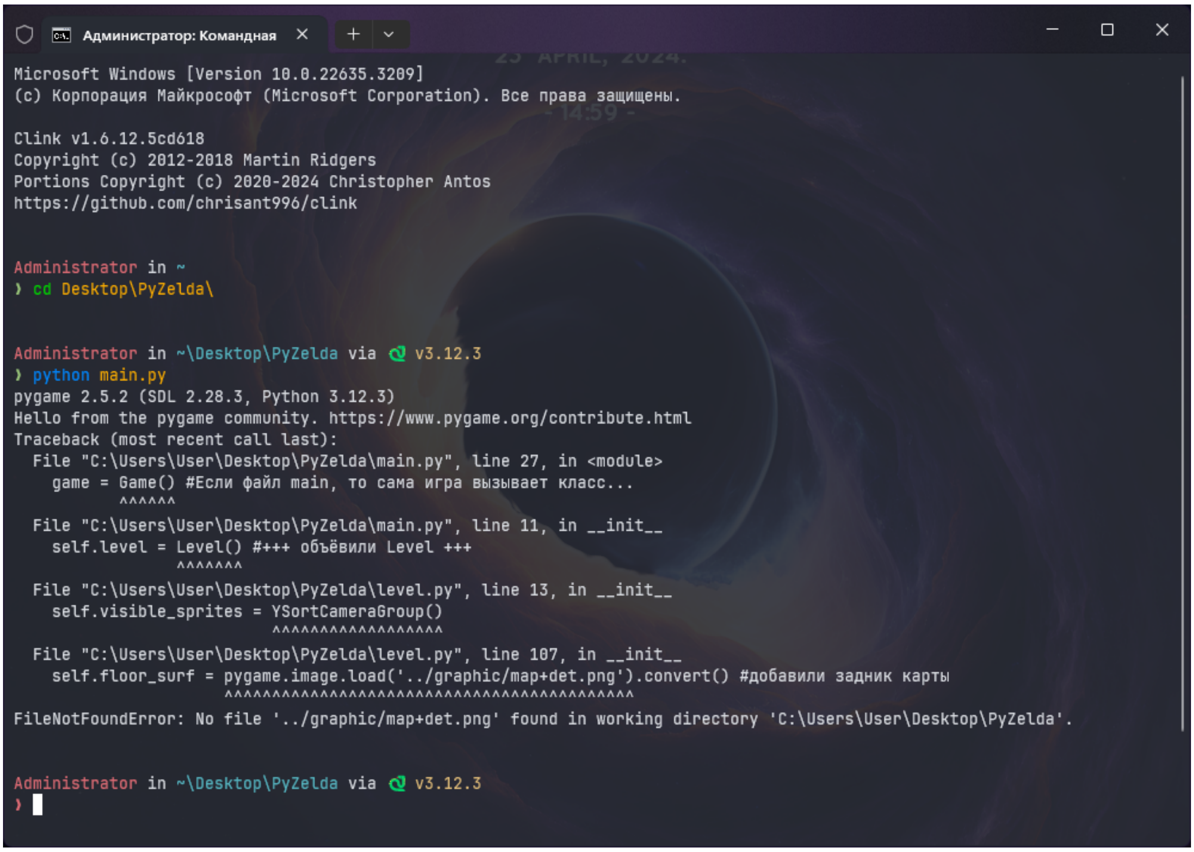


Папку кеша (`__pycache__`) можно удалить. Она нужна для ускорения запуска игры и запоминает временные файлы. Сейчас они все изменятся, так как пути будут недействительны, так что она нам не нужна.

Теперь файлы проекта все в одном месте:



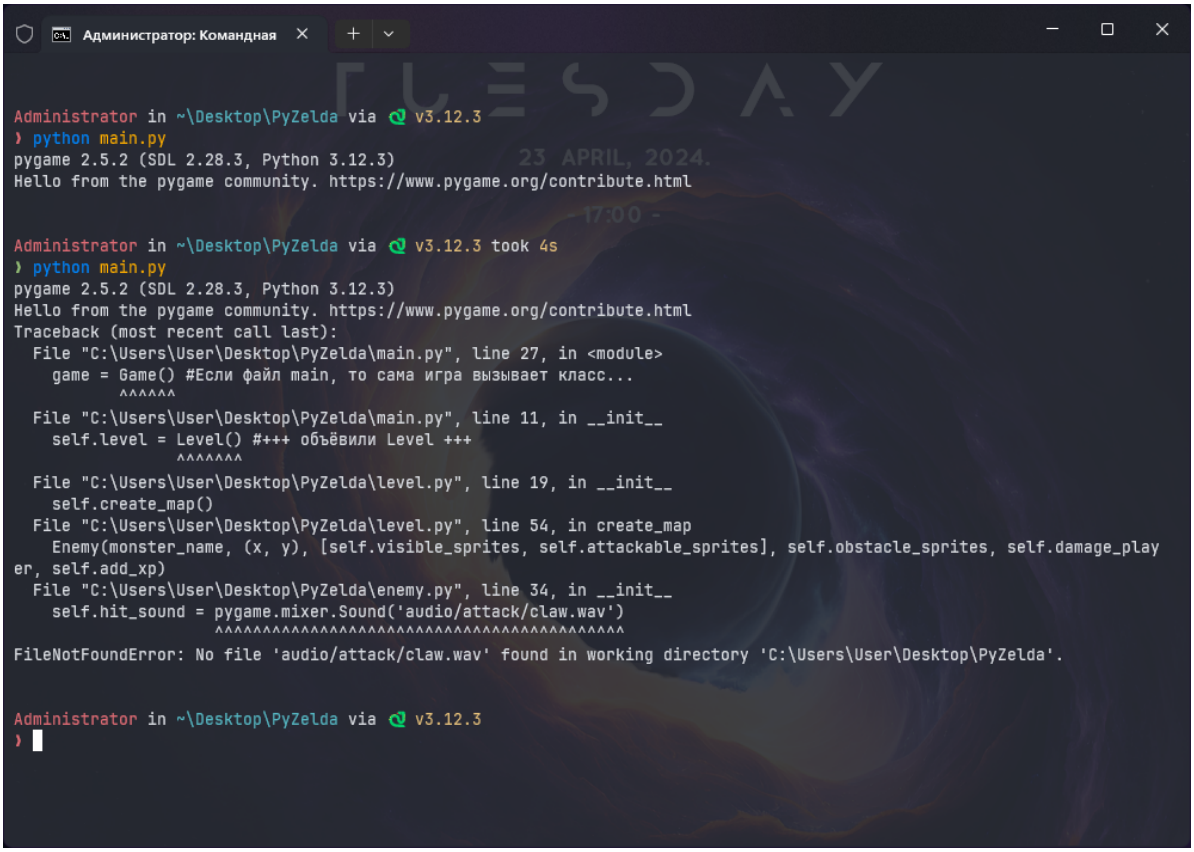
При попытке запуска, мы получим следующее:



Проблема с путями. Нужно все пути с "`../`" (предыдущая директория), поменять на текущую директорию, так как для запуска файла, раньше нам нужно было подняться из папки "code" в корень и оттуда спуститься, например в директорию "graphic", теперь же -- просто спуститься в директорию "graphic". Ctrl+F нам в помощь. У меня исправлять пришлось файлы:

- weapon.py
- enemy.py
- level.py
- main.py
- player.py
- settings.py

После всех исправлений, запустим `main.py` и всё должно работать. Теперь заменим все файлы звуков с `.wav` на `.ogg`. Не забудьте заменить расширения в коде, чтобы не получить это:

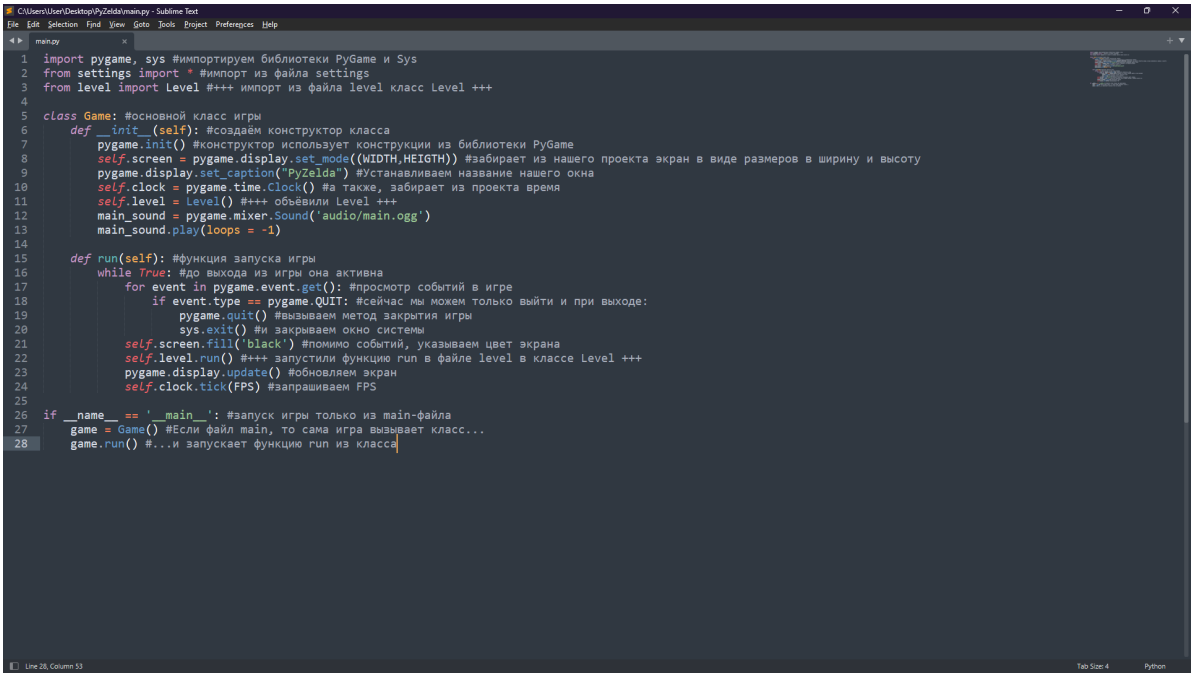


Я заменял формат звуков в следующих файлах:

- enemy.py
- main.py
- player.py
- settings.py

Когда замена произойдёт, вы не почувствуете разницы в звучании, но файлы "ужались" в 10 раз. Нужно это было для грамотной работы веба.

Осталось сделать отслеживание потока в main-файле в main-функции. Сейчас наш код в файле `main.py` выглядит так:

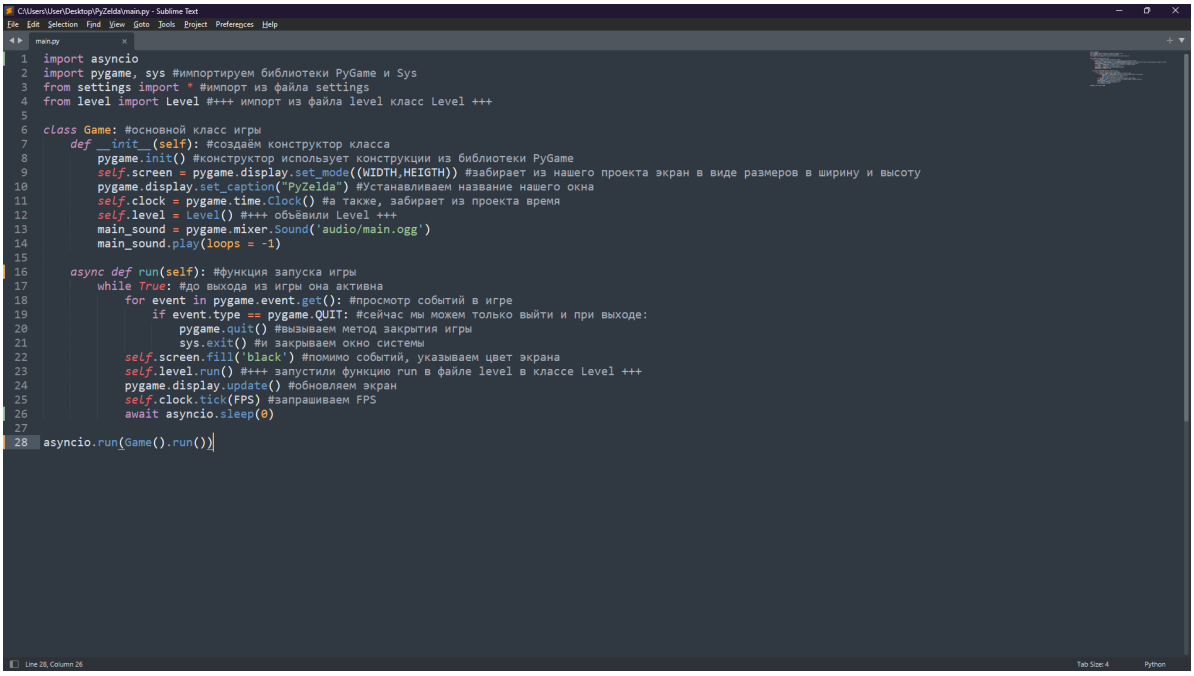


```
import pygame, sys #импортируем библиотеки PyGame и Sys
from settings import * #импорт из файла settings
from level import Level #+++ импорт из файла level класс Level +++

class Game: #основной класс игры
    def __init__(self): #создаём конструктор класса
        pygame.init() #конструктор использует конструкции из библиотеки PyGame
        self.screen = pygame.display.set_mode((WIDTH,HEIGTH)) #забирает из нашего проекта экран в виде размеров в ширину и высоту
        pygame.display.set_caption("PyZelda") #Устанавливаем название нашего окна
        self.clock = pygame.time.Clock() #а также, забирает из проекта время
        self.level = Level() #+++ объёвили Level +++
        main_sound = pygame.mixer.Sound('audio/main.ogg')
        main_sound.play(loops = -1)

    def run(self): #функция запуска игры
        while True: #до выхода из игры она активна
            for event in pygame.event.get(): #просмотр событий в игре
                if event.type == pygame.QUIT: #сейчас мы можем только выйти и при выходе:
                    pygame.quit() #вызываем метод закрытия игры
                    sys.exit() #и закрываем окно системы
            self.screen.fill('black') #помимо событий, указываем цвет экрана
            self.level.run() #+++ запустили функцию run в файле level в классе Level +++
            pygame.display.update() #обновляем экран
            self.clock.tick(FPS) #запрашиваем FPS

if __name__ == '__main__': #запуск игры только из main-файла
    game = Game() #Если файл main, то сама игра вызывает класс...
    game.run() #...и запускает функцию run из класса
```

Исправим несколько моментов:

- Импортируем библиотеку `asyncio`
- `run`-метод делаем потоковым
- Завершения данного метода — `await`-функция
- Проверку if-ом заменим на асинхронный запуск метода

Итог:

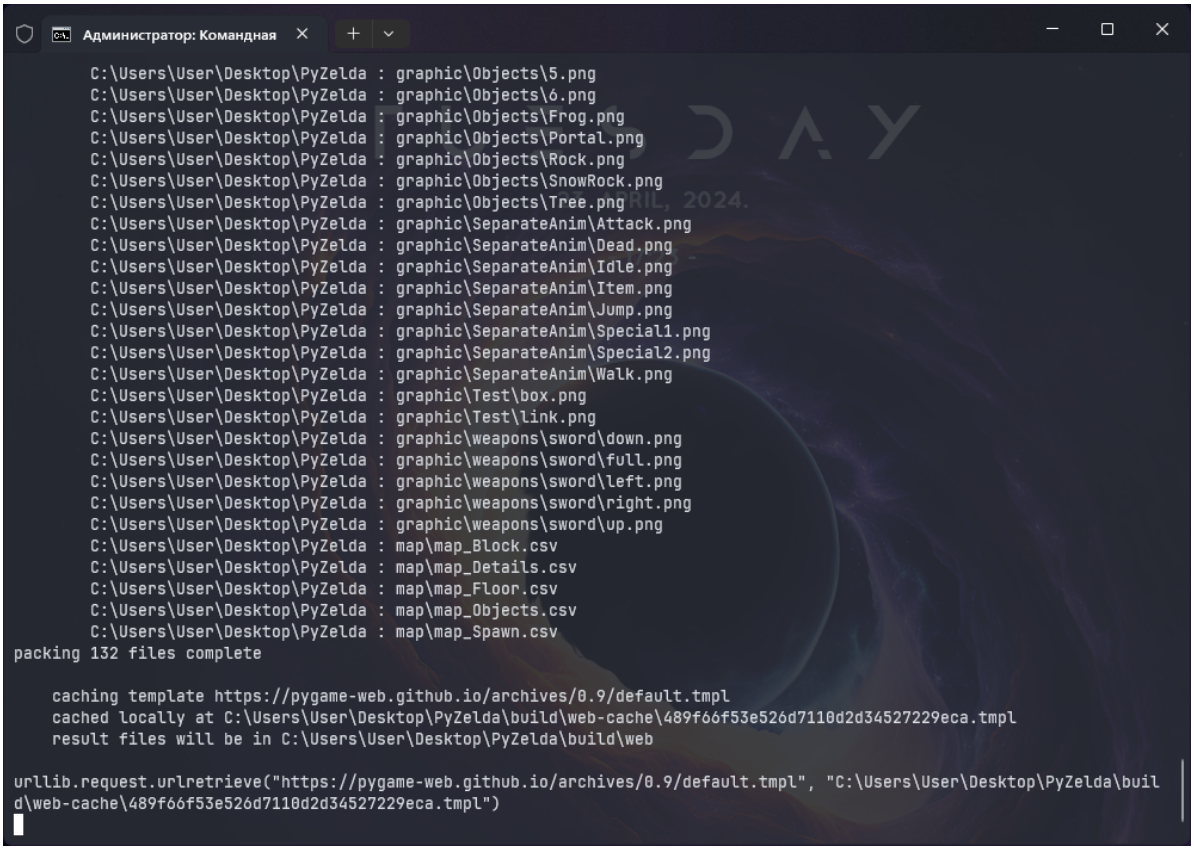


```
import asyncio
import pygame, sys #импортируем библиотеки PyGame и Sys
from settings import * #импорт из файла settings
from level import Level #+++ импорт из файла level класс Level +++

class Game: #основной класс игры
    def __init__(self): #создаём конструктор класса
        pygame.init() #конструктор использует конструкции из библиотеки PyGame
        self.screen = pygame.display.set_mode((WIDTH,HEIGTH)) #забирает из нашего проекта экран в виде размеров в ширину и высоту
        pygame.display.set_caption("PyZelda") #Устанавливаем название нашего окна
        self.clock = pygame.time.Clock() #а также, забирает из проекта время
        self.level = Level() #+++ объёвили Level +++
        main_sound = pygame.mixer.Sound('audio/main.ogg')
        main_sound.play(loops = -1)

    async def run(self): #функция запуска игры
        while True: #до выхода из игры она активна
            for event in pygame.event.get(): #просмотр событий в игре
                if event.type == pygame.QUIT: #сейчас мы можем только выйти и при выходе:
                    pygame.quit() #вызываем метод закрытия игры
                    sys.exit() #и закрываем окно системы
            self.screen.fill('black') #помимо событий, указываем цвет экрана
            self.level.run() #+++ запустили функцию run в файле level в классе Level +++
            pygame.display.update() #обновляем экран
            self.clock.tick(FPS) #запрашиваем FPS
            await asyncio.sleep(0)

asyncio.run(Game().run())
```

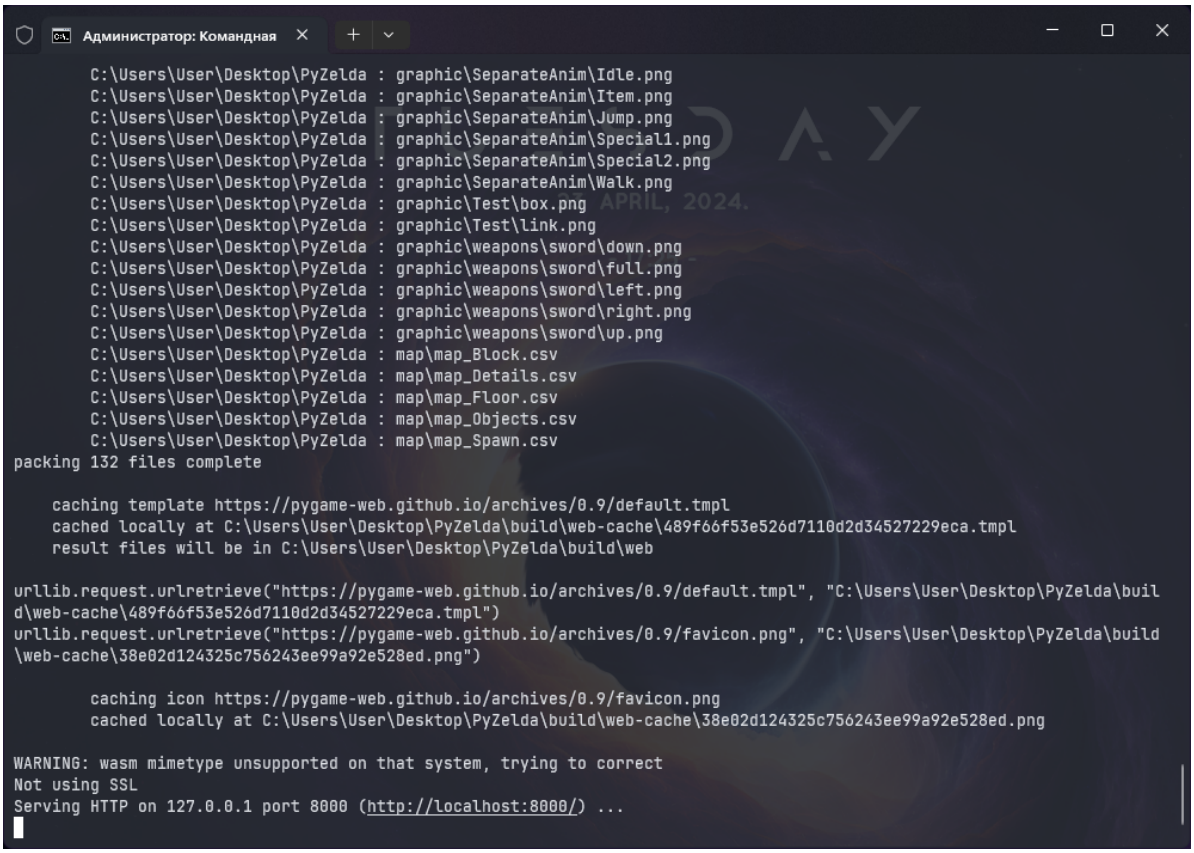
Библиотека `asyncio` позволяет увидеть потоки данных. В вебе, большинство (можно сказать "все") потоков асинхронны, а отслеживать их нужно по PID'ам (Process IDentificator). Вот этим мы и будем заниматься. Функция отслеживания — `run` в классе `Game`, там же мы будем ждать завершения операции. Сами процесс мы не должны запускать, как было с if-ом, а передаём эту задачу потоковому процессу (`asyncio.run(Game().run())`). При запуске ничего не должно поменяться.

Теперь в командной строке (или терминале) переходим на директорию выше и запускаем `pygbag` директории (в моём случае, это "PyZelda"). Таким образом, из директории мы делаем хостинг сервер, который в процессе инициализации будет собирать все потоки в псевдобиты и кешировать всё, что возможно.

В моём проекте получилось 132 файла:

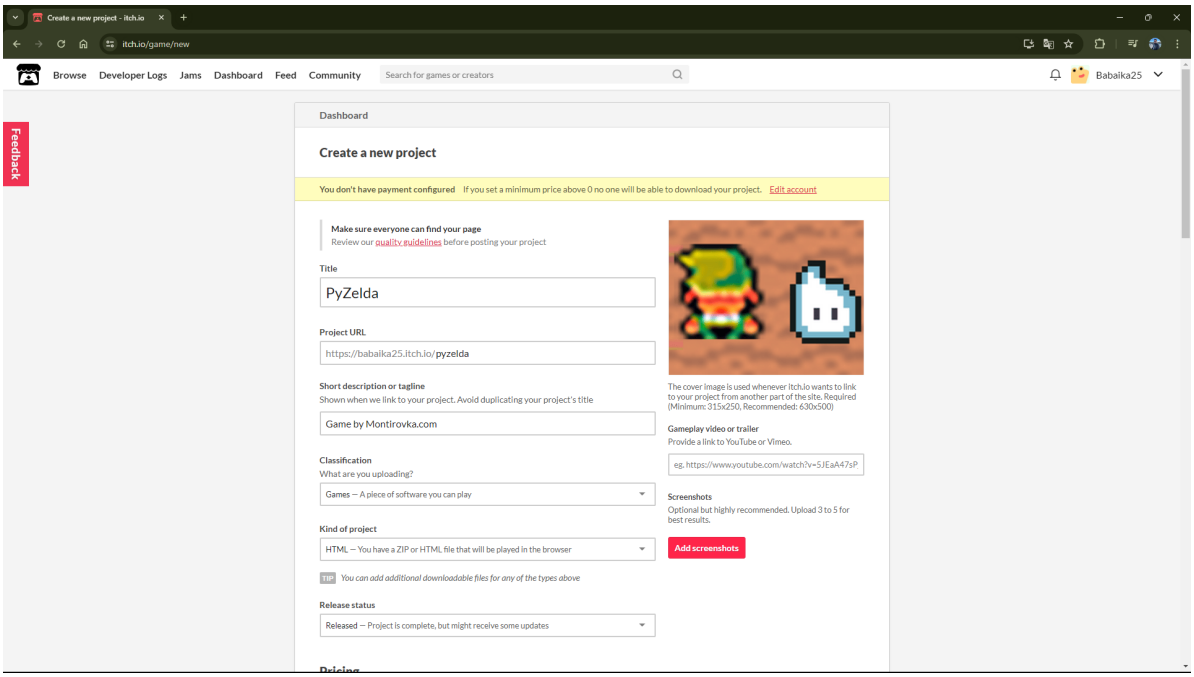


Чуть подождав (примерно, 2 минуты), терминал предложит перейти на localhost:8000. Это созданный сервер.



Перейдите и там будет пустой экран, далее пойдёт сборка всех функций. Теперь можно пойти попить чай, кофе или что покрепче, ведь ждать предстоит минут 5, пока все варианты событий будут проиграны. Дождитесь когда на экране начнётся игра. Процесс можно завершить закрытием терминала или сочетанием Ctrl+C.

Теперь, в папке с проектом, вы обнаружите новую директорию `build`. Это как раз тот билд, который мы собирали из процессов. Внутри директории ещё две: `web` и `web-cache`, а также файл с версией сборщика. В папке `web` лежит всё для деплоя проекта. Я же решил выгрузить его на itch.io. Чтобы это делать, я создал `.zip`-архив папки `web` и выгрузил проект с такими настройками:



Теперь вы сможете насладиться данным Величием!

Итоги

В результате работы мы создали игру на голом Python только с библиотекой PyGame. Были использованы ещё парочка для обхода директорий, но можно было и без них. Выводы данной статьи, которые я хотел бы сделать не заключаются в подчёркивании того, что Python скоро заменит C# с Unity и Unreal. Цель статьи — продемонстрировать возможность работы с Python как с движком для программирования игр. В России, я думаю, этот вариант написания игр будет популярен. Не из-за World Of Tacks, который написан на Python, а из-за мультиплатформенности интерпретируемого языка программирования. Данный код я могу спокойно запускать из любой Unix-системы и это круто. Но встаёт пара вопросов:

1. **Насколько сложно написать игру самому с нуля?** Если вы только входите в геймдев и вообще не понимаете что происходит с играми — не используйте питон. Данный сектор сейчас очень узок и вряд ли вы сможете быстро найти работу (как и вообще в геймдеве). Выбирайте Unity. Он проще Unreal Engine, но при этом весьма работоспособен. Если же вы понимаете язык программирования Python и хотите сделать игру — делайте её на Python. Нет смысла выучивать новый язык только для создания игры. Python на многое способен. В качестве доказательств моих слов, могу сослаться на игры написанные на Python, а могу и на 3D-движок, который набирает обороты под название HARFANG. Он ничем не хуже Unity.
2. **Ну а как быть с опытом разработчиков?** На Unity и Unreal много пресетов. Это правда. Но и на Python не меньше. Я не геймдевер, как можно было понять из введения в статью. Я системный администратор со знанием Python на +/- Middle-уровне (да простят меня разрабы). Я писал код для оптимизации работы в сетях, а вот с играми знаком только как игрок со стажем или программист крестиков-ноликов. Всё что я сделал — я начал изучать вопрос. И каково было моё удивление, что есть куча людей, которые помогают написать игру на Python. Лично я пользовался гайдами от Clear Code (чувак очень подробно рассказывает на какие библиотеки в PyGame ссылаться), собственно, официальным мануалом по PyGame и книгой Making Games with Python & Pygame (внутри базовые математические концепции игр). Эти три элемента позволили мне написать игру меньше чем за месяц.

Конечно, мой проект по ПуЗельде можно долго дорабатывать. Можно сделать фон из деревьев, чтобы экран не оставался чёрным, добавить оружия, подбираемых предметов. Можно сделать больших боссов. Можно создать меню с перезапуском и выходом из игры. В общем, есть много точек роста. Весь проект я оставляю вам на обсуждение и возможные доработки. Надеюсь было интересно! Закончу как один известный выпускник 13 школы.

Иииииииииииииииииииииииии помните! Питон — игривая змея!