Проект PyGame. Игра в целом

План урока

Аннотация

На этом занятии мы поговорим об игре в целом. Игра — это достаточно большая программа, и важно организовать ее правильно. Сегодня мы будем проектировать заготовку для игры Mario.

1. Заставка и экран конца игры

Практически каждая игра состоит из нескольких экранов, на которых происходит действие. Такими экранами являются **заставка** и **конец игры**.

На отдельном экране (а в небольших играх — прямо на экране) заставки можно расположить правила. Небольшая проблема в том, что Pygame не умеет выводить несколько строк одновременно, приходится делать это построчно вручную.

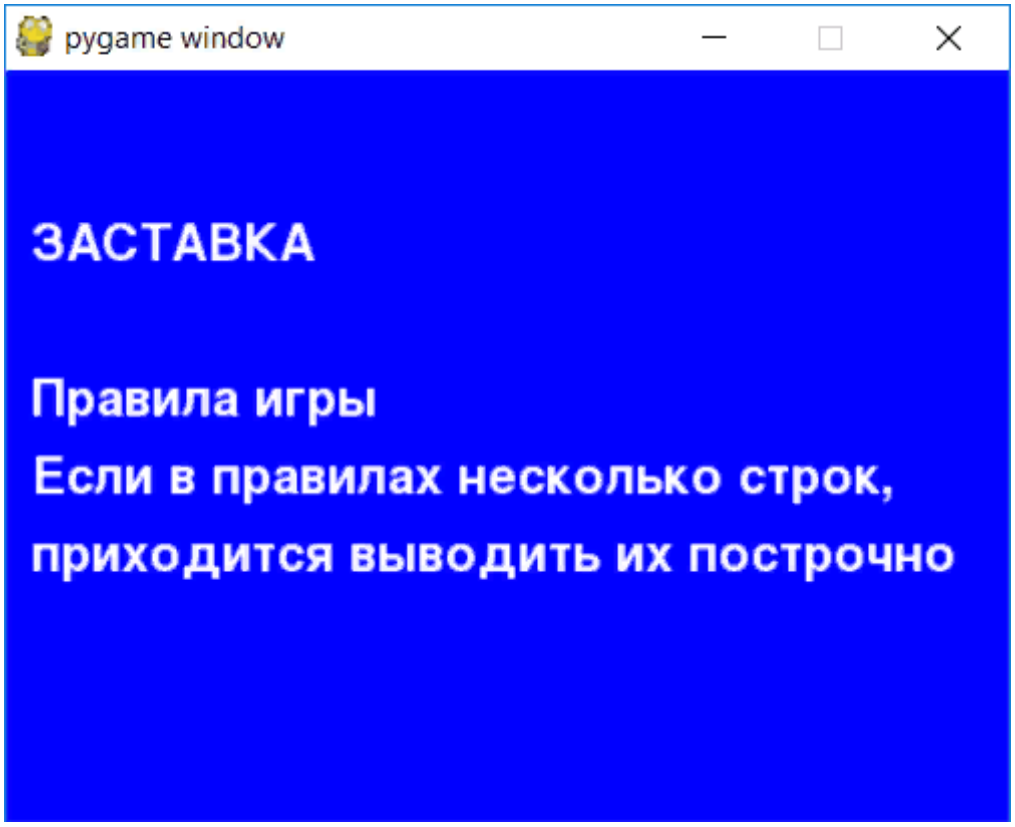
В функции **start_screen()**, приведённой ниже, мы организовали свой мини-игровой цикл, который выполняется до тех пор, пока игрок не нажмет клавишу на клавиатуре или кнопку мыши.

На экране окончания игры обычно выводится итоговый счёт, авторы игры, прочая рекламная информация.

Преждевременный выход из игры возможен на любом экране, поэтому удобно «аварийное завершение» оформить в виде отдельной функции **terminate()**:

```
FPS = 50


def terminate():
    pygame.quit()
    sys.exit()


def start_screen():
    intro_text = ["ЗАСТАВКА", "",
                  "Правила игры",
                  "Если в правилах несколько строк,",
                  "приходится выводить их построчно"]

    fon = pygame.transform.scale(load_image('fon.jpg'), (WIDTH, HEIGHT))
    screen.blit(fon, (0, 0))
    font = pygame.font.Font(None, 30)
    text_coord = 50
    for line in intro_text:
        string_rendered = font.render(line, 1, pygame.Color('black'))
        intro_rect = string_rendered.get_rect()
        text_coord += 10
        intro_rect.top = text_coord
        intro_rect.x = 10
        text_coord += intro_rect.height
        screen.blit(string_rendered, intro_rect)

    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                terminate()
            elif event.type == pygame.KEYDOWN or \
                    event.type == pygame.MOUSEBUTTONDOWN:
                return  # начинаем игру
```

```
        pygame.display.flip()
        clock.tick(FPS)
```



2. Уровни игры и их загрузка

Уровни в игре удобно хранить в текстовых файлах. Так их очень удобно редактировать. В этом примере мы будем использовать всего три символа:

1. # — означает стену;
2. @ — положение игрока;
3. . — пустая клетка.

Например,

```
...###
..##.#.####
.##..###..#
##........#
#...@..#..#
###..###..#
..#..#....#
```

```
.##.##.#.##
.#......##
.#.....##
.#######
```

Чтение карты из файла — это стандартная операция работы с файлами. После чтения удобно дополнить карту до прямоугольника.

```
def load_level(filename):
    filename = "data/" + filename
    # читаем уровень, убирая символы перевода строки
    with open(filename, 'r') as mapFile:
        level_map = [line.strip() for line in mapFile]

    # и подсчитываем максимальную длину
    max_width = max(map(len, level_map))

    # дополняем каждую строку пустыми клетками ('.')
    return list(map(lambda x: x.ljust(max_width, '.'), level_map))
```

3. Отрисовка уровня

С загруженным уровнем можно работать по-разному.

Для больших уровней может потребоваться большая производительность, и тогда стоит делать так:

1. Сначала отрисовать неподвижные элементы карты один раз на отдельном **Surface**-е,
2. А после выводить их как одну общую картинку.

Для «обычной жизни» достаточно сгенерировать соответствующие спрайты. И потом будет очень удобно проверять столкновения. Мы поступим именно так.

Изображения **тайлов** (клеточек) удобно хранить в словаре.

Стоит сразу определить тайлы и подвижные объекты в разные группы спрайтов.

```
tile_images = {
    'wall': load_image('box.png'),
    'empty': load_image('grass.png')
}
player_image = load_image('mario.png')
```

```
tile_width = tile_height = 50


class Tile(pygame.sprite.Sprite):
    def __init__(self, tile_type, pos_x, pos_y):
        super().__init__(tiles_group, all_sprites)
        self.image = tile_images[tile_type]
        self.rect = self.image.get_rect().move(
            tile_width * pos_x, tile_height * pos_y)


class Player(pygame.sprite.Sprite):
    def __init__(self, pos_x, pos_y):
        super().__init__(player_group, all_sprites)
        self.image = player_image
        self.rect = self.image.get_rect().move(
            tile_width * pos_x + 15, tile_height * pos_y + 5)
```

Генерацию спрайтов лучше оформить в виде отдельной функции, которая создаст все элементы игрового поля и вернёт спрайт игрока.

```
# основной персонаж
player = None

# группы спрайтов
all_sprites = pygame.sprite.Group()
tiles_group = pygame.sprite.Group()
player_group = pygame.sprite.Group()


def generate_level(level):
    new_player, x, y = None, None, None
    for y in range(len(level)):
        for x in range(len(level[y])):
            if level[y][x] == '.':
                Tile('empty', x, y)
            elif level[y][x] == '#':
                Tile('wall', x, y)
            elif level[y][x] == '@':
                Tile('empty', x, y)
                new_player = Player(x, y)
    # вернем игрока, а также размер поля в клетках
    return new_player, x, y
```

4. Камера

Если мы немного ошибёмся в размерах экрана, то увидим, что игровое поле целиком на экран не влезает. Нам нужен скроллинг. Обычно в играх для этого добавляют специальный объект — камеру, которая «следит» за персонажем.

В Pygame нет специального класса камеры. Вернее, модуль камеры есть, но он предназначен для работы с видеокамерой, подключённой к компьютеру.

В простом случае камера должна выполнять две функции:

1. Обновлять своё положение (менять **viewpoint**, область просмотра) и
2. Влиять на игровое поле таким образом, чтобы оно менялось в зависимости от положения камеры.

Часто игровые объекты реализуют так, чтобы они **отрисовывались** правильным образом, опираясь на положение камеры. То есть, объекты сохраняют своё положение в игровом мире и только в момент отрисовки смещаются.

В Pygame процесс отрисовки спрайтов не контролируется пользователем. И в простой игре вполне оправданно, чтобы камера изменяла *положение* объектов.

Тогда класс камеры, которая **держит** цель, например, главного персонажа, в центре окна может выглядеть так:

```
class Camera:
    # зададим начальный сдвиг камеры
    def __init__(self):
        self.dx = 0
        self.dy = 0

    # сдвинуть объект obj на смещение камеры
    def apply(self, obj):
        obj.rect.x += self.dx
        obj.rect.y += self.dy

    # позиционировать камеру на объекте target
    def update(self, target):
        self.dx = -(target.rect.x + target.rect.w // 2 - width // 2)
        self.dy = -(target.rect.y + target.rect.h // 2 - height // 2)
```

Перед началом игрового цикла создадим камеру:

```
camera = Camera()
```

и потом на каждой итерации игрового цикла будем обновлять ракурс камеры и корректировать положение всех объектов в группе **all_sprites**:

```
# изменяем ракурс камеры
camera.update(player);
# обновляем положение всех спрайтов
for sprite in all_sprites:
      camera.apply(sprite)
```

Легко сделать, чтобы камера следила за другим объектом или управлялась игроком.

Доведите до логического завершения разбираемую задачу, а затем продолжайте работать над своим проектом.