Игра 8: Пришелец vs зомби



# Цель игры

# 

В этой игре мы будем управлять пришельцем. Его целью будет истреблять зомби и уклоняться от торчащих из-под земли рук зомби. Чем больше зомби пришелец уничтожит, тем больше победных очков мы заработаем.

# Концепция (план)

В этот раз количество строк кода кратно увеличится и вместе с этим хотелось бы добавить более логичную структуру проекта: в структуре должны сочетаться логика и возможность масштабирования, код должен легко читаться. Предлагаю в этот раз отталкиваться от следующих пунктов:

1. Весь код игры теперь - объект, завернем всю логику в отдельный класс.
2. Для того, чтобы не перегружать этот класс, вынесем переменные с настройками в отдельный файл settings.py, изменения этих настроек будет влиять на игру.
3. Также вынесем всю логику спрайтов в отдельный файл sprites.py, в нем будут реализованы классы, связанные с объектами в игре.
4. Файл с главным классом игры Game, назовем main.py, в него мы будем импортировать все необходимые данные из других файлов.

Таким образом у нас получится достаточно серьезный проект с хорошей логикой. К примеру, если мы захотим добавить какой-либо спрайт, то нам не придется искать место в огромном количестве строк, куда можно было бы правильно вставить логику для нового спрайта; в таком случае, мы бы просто добавили еще один класс в файл sprites.py - это и есть наш главный тезис: облегчить нашу структуру кода и сделать ее очень простой для дальнейшей разработки.

## Процесс игры

Процесс игры будет очень простой: пришелец непрерывно бежит и навстречу ему появляются зомби (и зомби-руки). У пришельца есть возможность перепрыгнуть врагов или выстрелить в них, чтобы уничтожить и получить победные очки. В конце игры пользователю показывается количество заработанных победных очков.

Приступим.

# Импорт

В файле main.py нам понадобится импорт из settings и sprites, а также модуль path для правильного обращения к файлам и, конечно же, pygame.

|  |
| --- |
| import pygame as pg from os import path from settings import \* from sprites import \* |

В этот раз сократим имя pygame до двух букв.

В файле sprites.py сделаем идентичный импорт, без импорта sprites (самого себя), но возьмем randrange из модуля random:

|  |
| --- |
| import pygame as pg from settings import \* from os import path from random import randrange |

В файле settings.py импорт не требуется, там мы только обозначим глобальные параметры.

# 

# Файл settings.py

Обозначим уже на этой стадии в нем весьма очевидные вещи:

|  |
| --- |
| TITLE = 'Alien vs Zombies' WIDTH = 1000 HEIGHT = 600 FPS = 60 FONT\_NAME = 'arial'  BACKGROUND = 'background.png'  WHITE = (255, 255, 255) BLACK = (0, 0, 0) RED = (255, 0, 0) GREEN = (0, 255, 0) BLUE = (0, 0, 255) YELLOW = (255, 255, 0) LIGHTBLUE = (135, 206, 250) |

* Название файла.
* Ширина экрана.
* Высота экрана.
* Ограничение по кадрам в секунду.
* Имя шрифта для текста в игре.
* Название файла с фоновым изображением.
* Несколько цветов в RGB коде не будут лишними.

На данный момент это все, что будет в нашем файле для настроек. Позднее мы добавим туда еще пару параметров.

# Файл main.py

Начнем инициализировать наш main. Как решили выше, теперь вся игра - это класс Game, в котором будут собственные методы и инициализация.

## Инициализация

Напишем первую часть инициализации:

|  |
| --- |
| class Game:  def \_\_init\_\_(self):  pg.init()  self.screen = pg.display.set\_mode((WIDTH, HEIGHT))  pg.display.set\_caption(TITLE)  self.clock = pg.time.Clock()  self.running = True  self.font\_name = pg.font.match\_font(FONT\_NAME) |

Первые строчки будут выглядеть таким образом:

* Инициализируем модуль pygame и экран.
* Установим имя программы.
* Запустим счетчик для ограничения кадров в секунду.
* Установим переменную для главного цикла.
* Загрузим и сохраним объект шрифта для текста в игре.

## 

## Загрузка данных

В данной игре все используемые данные - это изображения, в основном - спрайтов.

Назовем метод load\_data().

### Фон

Для начала предлагаю загрузить фоновое изображение.

|  |
| --- |
| def load\_data(self):  file\_dir = path.dirname(\_\_file\_\_)  img\_dir = path.join(file\_dir, 'img')  bg = path.join(img\_dir, BACKGROUND)  self.bg = pg.image.load(bg).convert()  self.bg = pg.transform.scale(self.bg, (1500, 600))  self.bg\_rect = self.bg.get\_rect()  self.bg\_x = 0  self.bg2\_x = self.bg\_rect.width |

Давайте разбираться:

* file\_dir - это строковое представление директории, в которой хранится наш текущий файл. Используем метод dirname модуля path для того, чтобы получить эту информацию.
* После чего перейдем в папку img, теперь в img\_dir хранится полный путь до папки с изображениями. Очень важно использовать модуль path в таких случаях вместо того, чтобы прописывать путь вручную, потому что во втором случае программа не заработает на разных ОС и/или у разных пользователей.
* Как мы уже зафиксировали в settings, в переменной BACKGROUND хранится имя файла фонового изображения, снова используем path.join(), чтобы добраться до полного пути изображения.
* Теперь уменьшим изображение до 1500х600 пикселей, это на 500 пикселей длиннее, чем наш выбранный размер экрана.
* Возьмем прямоугольную область от нашего фона и сохраним две координаты: левую границу и правую границу этой области, эти координаты понадобятся для движения фона, об этом позже.

### Спрайты

С фоном мы разобрались, но как поступить со спрайтами, ведь их намного больше и работать с ними придется по-другому, чем с фоном?

В помощь нам придет файл sprites.py, там мы реализуем первый класс для обработки изображений спрайтов.

Назовем класс Images.

Для начала стоит решить то, как мы будем хранить все спрайты. Было бы очень удобно распределить их по группам: одна группа для изображений бегущего героя, вторая для прыжка, третья для монстров и т.д.

Для этих целей нам подойдет словарь, в ключе будет записано слово, отражающее суть группы изображений в данной ячейке, а в значении - список объектов готовых к работе изображений.

Не буду долго томить и покажу то, как можно реализовать этот класс:

|  |
| --- |
| class Images:  def \_\_init\_\_(self):  self.images\_dict = dict()   def load\_images(self, folder, file\_name, amount, name, scale\_x=1, scale\_y=1):  img\_list = []  self.images\_dict.update({name: img\_list})  for i in range(1, amount + 1):  filename = path.join(folder, file\_name + str(i) + '.png')  img = pg.image.load(filename).convert()  img.set\_colorkey(WHITE)  img\_rect = img.get\_rect()  img = pg.transform.scale(img, (int(img\_rect.width/scale\_x),  int(img\_rect.height/scale\_y)))  img\_list.append(img) |

В инициализации создадим пустой словарь, в который будем добавлять все наши объекты.

Метод load\_images принимает 6 параметров:

1. Путь к папке, где находятся нужные изображения.
2. Имя файла без индекса (к примеру, ‘run’ для нашего пришельца).
3. Количество файлов в группе (6 для ‘run’).
4. Имя группы для записи словарь (чаще будет совпадать с именем файла).
5. Уменьшение (во сколько раз) изображения по оси X.
6. Уменьшение (во сколько раз) изображения по оси Y.

Внутри метода создадим пустой список, в который добавим все изображения группы, после чего отправим этот список как значение в словарь.

Заполним список внутри цикла for:

Все изображения в папках пронумерованы от 1 до n (даже если изображение всего 1).

Каждое изображение имеет формат png.

В каждой итерации:

1. Получаем полный путь до файла с нужным изображением.
2. Конвертируем файл в понятный pygame формат.
3. Удаляем белый фон изображения.
4. Берем прямоугольную область от изображения.
5. Уменьшаем изображение на выбранные параметры по X и Y.
6. Добавляем полученный объект в лист (лист уже лежит значением в словаре).

Итого, останется создать экземпляр класса Images в \_\_init\_\_ и вызвать метод load\_images() в load\_data(), перед этим форматируя путь, где лежат нужные изображения, внутри load\_data() добавим:

|  |
| --- |
| self.images.load\_images(img\_dir, 'bullet', 1, 'bullet', 3, 3) zombie\_dir = path.join(img\_dir, 'zombie') self.images.load\_images(zombie\_dir, 'go', 10, 'zombie', 2.5, 2.5) self.images.load\_images(zombie\_dir, 'zarm', 1, 'zarm', 2, 2) alien\_dir = path.join(img\_dir, 'alien') self.images.load\_images(alien\_dir, 'run', 6, 'run', 3, 3) self.images.load\_images(alien\_dir, 'jump', 2, 'jump', 3, 3) self.images.load\_images(alien\_dir, 'shoot', 7, 'shoot', 3, 3) |

После этого в нашем распоряжении будут готовые к работе файлы всех изображений.

Метод load\_data() будем вызывать в \_\_init\_\_ нашего класса Game, перед вызовом создадим экземпляр класса Images и рекомендуется инициализировать все новые переменные (если не знаем, что по итогу в них окажется, можно присвоить None).

Добавим в \_\_init\_\_:

|  |
| --- |
| self.images = Images() self.bg = None self.bg2 = None self.bg\_rect = None self.bg\_x = 0 self.bg2\_x = 0 self.load\_data() |

На этом загрузка файлов завершена.

## Движущийся фон

Кстати говоря, бесконечно движущийся фон.

Наша задача - сделать из фонового изображения “беговую дорожку” для главного персонажа. Давайте разберемся идейно, как это можно сделать:

1. В первую очередь нужно отметить, что нужно взять одно изображение и продублировать его на экране: прикреплены друг к другу горизонтально.
2. Оба изображения двигаем влево, чтобы создать видимость того, что персонаж бежит вправо.
3. Если левое изображение скрылось за экраном, прикрепляем его справа от правого, это не будет заметно, так как у нас остался запас в 500 пикселей от фона.
4. Получится эффект бесконечно меняющегося изображения. За плавность можем поблагодарить художника фона, он сделал его таким, чтобы левая часть продолжала правую.

Реализуем это в коде таким образом:

Создадим метод draw() в нашем классе Game:

|  |
| --- |
| def draw(self):  self.screen.blit(self.bg, (self.bg\_x, 0))  self.screen.blit(self.bg, (self.bg2\_x, 0))  if self.bg\_x < -self.bg\_rect.width:  self.bg\_x = self.bg\_rect.width  if self.bg2\_x < -self.bg\_rect.width:  self.bg2\_x = self.bg\_rect.width   pg.display.flip() |

Отображаем первое изображение в координатах bg\_x и второе - в координатах bg2\_x.

Если правый край изображения вышел за левый край экрана (координата стала равна минус ширине изображения), то меняем это изображение и закрепляем его справа второго. Сделаем два условия для двух изображений.

В конце метода отобразим все изменения на экране.

## Бесконечный цикл игры

### Метод run

Цикл реализуем в методе класса Game, назовем run().

|  |
| --- |
| def run(self):  self.playing = True  while self.playing:  self.events()  self.draw()  self.clock.tick(FPS) |

В цикле пока что только вызываем метод draw() и ограничиваем количество кадров в секунду, перед циклом обозначим переменную playing для возможного выхода из цикла, а выходить мы будем с помощью захвата ивентов в методе events().

## Метод events

В этом методе будем захватывать происходящие ивенты.

|  |
| --- |
| def events(self):  for event in pg.event.get():  if event.type == pg.QUIT:  if self.playing:  self.playing = False  self.running = False |

Пока что реализуем простейший ивент - выход из программы при нажатии на крестик.

Не забудьте инициализировать playing и running в \_\_init\_\_ нашего класса (изначально принимают значение False и True соответственно). running должно принимать True при создании экземпляра класса, это будет означать, что приложение пока не закрыто пользователем, переменная будет использоваться во внешнем цикле.

## 

## Цикл вне класса

Вне класса у нас будет еще один цикл, который будет запускать каждый раз новую игру.

|  |
| --- |
| game = Game() while game.running:  game.new()  game.end() pg.quit() |

Создадим экземпляр класса Game.

Запустим цикл, пока running (переменная принимает значения False, если пользователь нажимает на крестик) запускаем метод new(), потом метод end() (при проигрыше в игре). После поражения снова начинается новая игра - это очень удобный способ реализовать перезагрузку игры.

## Метод end

Все просто. Для начала проверяем, если пользователь нажал на крестик, то выходим из метода во внешний цикл и впоследствии из условия. Если игра продолжается, то нужно подождать ввода пользователя - так мы будем узнавать о его готовности начать новую игру.

|  |
| --- |
| def end(self):  if not self.running:  return  self.screen.blit(self.bg, (0, 0))  self.draw\_text("GAME OVER", 48, RED, WIDTH / 2, HEIGHT / 4)  self.draw\_text("Score: " + str(self.score), 22, WHITE, WIDTH / 2, HEIGHT \* 3 / 4 - 50)  self.draw\_text("Press a key to play again", 22, WHITE, WIDTH / 2, HEIGHT \* 3 / 4)  pg.display.flip()  self.wait\_for\_key() |

* Проверяем нажат ли крестик (running станет False).
* Снова рисуем фоновое изображение.
* Выводим сообщения о проигрыше и итоговый счет игрока.
* Отображаем все изменения на экране.
* После чего нужно, чтобы метод не заканчивался, для этого реализуем метод wait\_for\_key.

## Метод wait\_for\_key

Все просто. Запускаем бесконечный цикл, который прерывается по нажатию клавиши (это правило написано на экране, если игрок проиграл):

|  |
| --- |
| def wait\_for\_key(self):  waiting = True  while waiting:  self.clock.tick(FPS)  for event in pg.event.get():  if event.type == pg.QUIT:  waiting = False  self.running = False  if event.type == pg.KEYUP:  waiting = False |

## Метод draw\_text

В методе end() мы используем метод draw\_text, он аналогичен тем, которые мы реализовывали в прошлых проектах.

|  |
| --- |
| def draw\_text(self, text, size, color, x, y):  font = pg.font.Font(self.font\_name, size)  text\_surface = font.render(text, True, color)  text\_rect = text\_surface.get\_rect()  text\_rect.midtop = (x, y)  self.screen.blit(text\_surface, text\_rect) |

На вход мы подаем:

* Текст для вывода.
* Размер шрифта.
* Цвет текста.
* Координаты верхней центральной части прямоугольной области, в которой будет расположен текст.

Методы внутри draw\_text() должны быть вам известны из прошлых проектов, здесь все также.

## Метод new

Во внешнем цикле вы, наверное, заметили, что вместо метода run() мы использовали метод new(). Такое решение не случайно. new() должен заново создавать основные инициализировать счет, спрайт игрока и группы спрайтов врагов, запускать таймер на ивенты спавна мобов и только после этого вызывать метод run(), который отвечает за бесконечный цикл игры. Давайте посмотрим, как можно реализовать new():

|  |
| --- |
| def new(self):  self.score = 0  self.all\_sprites = pg.sprite.Group()  self.zombies = pg.sprite.Group()  self.zarms = pg.sprite.Group()  self.bullets = pg.sprite.Group()  self.player = Player(self)  pg.time.set\_timer(self.spawn\_zombie, ZOMBIE\_FREQ)  pg.time.set\_timer(self.spawn\_zarms, ZARMS\_FREQ)  self.run() |

Идентично прошлым проектам, только сейчас все содержится в классе Game, обращаемся к атрибутам с помощью self.

* all\_sprites - группа, содержащая в себе все спрайты в игре, нужна для отрисовки.
* zombies - группа, включающая в себя только зомби.
* zarms - включает в себя только руки зомби, торчащие из-под земли (смотреть папку img).
* bullets - включает в себя пули, которые вылетают из бластера главного персонажа.
* player - экземпляр класса Player - следующим шагом создадим его.
* Устанавливаем таймеры для ивентов spawn\_zomibe и spawn\_zarms, которые отвечают за то, как часто нужно создавать зомби и торчащие руки в игре.

Инициализируем переменный ZOMBIE\_FREQ (частота зомби) и ZARMS\_FREQ (частота зомби-рук) в файле settings.py, предлагаю для зомби сделать значение 4000, а для ZARMS\_FREQ - 7000 (4 и 7 секунд, соответственно).

* Запускаем метод run().

Не забудьте все атрибуты класса инициализировать в \_\_init\_\_ (None, если не знаете, чем будет объект).

Для инициализации ивентов в \_\_init\_\_ добавьте две строчки:

|  |
| --- |
| self.spawn\_zombie = pg.USEREVENT + 1 self.spawn\_zarms = pg.USEREVENT + 2 |

Если закомментируете строчку с созданием экземпляра класса Player (так как такой класс мы еще не реализовали), то сможете убедиться в работе бесконечно движущегося фонового изображения.

# Файл sprites.py

Переходим в файл sprites.py и начинаем реализовывать классы для объектов-спрайтов, в файле у нас уже находится метод Images.

# Класс Player

Этот класс отвечает за спрайт нашего главного персонажа и все действия, которые совершает объект.=

## Инициализация

Главное, что будет в классе Player - это его три состояния, а именно:

* Бег.
* Прыжок.
* Стрельба.

Для каждого состояния нам понадобится переменная, которая будет указывать на это состояние и список соответствующих изображений.

|  |
| --- |
| class Player(pg.sprite.Sprite):  def \_\_init\_\_(self, game):  self.groups = game.all\_sprites  super().\_\_init\_\_(self.groups)  self.running\_imgs = game.images.images\_dict['run']  self.jumping\_imgs = game.images.images\_dict['jump']  self.shooting\_imgs = game.images.images\_dict['shoot']  self.game = game  self.running = True  self.jumping = False  self.shooting = False  self.image = self.running\_imgs[0]  self.rect = self.image.get\_rect()  self.rect.center = (75, HEIGHT - 150)  self.vel\_y = 0  self.current\_frame = 0  self.last\_update = 0 |

* Инициализируем группу спрайта до инициализации как pygame Sprite - это другой способ добавить спрайт в группу спрайтов.
* running, jumping и shooting imgs - списки объектов готовых изображений, которые мы достаем из словаря, созданного ранее.
* Обратите внимание на то, что \_\_init\_\_ в Player принимает экземпляр класса game, запомним это в отдельный атрибут.
* Обозначим 3 переменных состояния, изначально пришелец начинает бежать, поэтому running примет значение True, а image, который отвечает за текущее изображение - первый объект в списке изображений для бегущего состояния.
* Возьмем прямоугольную область от изображения и установим стартовые координаты.
* vel\_y - отвечает за вертикальную скорость пришельца, понадобится нам для прыжка.
* current\_frame - переменная, которая запоминает то, какое по изображение из списка изображений мы сейчас используем.
* last\_update - счётчик того, когда в последний раз меняли фрейм.

## 

## Методы jump и shoot

В этих методах нам почти не придется реализовывать никакой логики. Все просто - если персонаж прыгает или стреляет, запишем это в соответствующую переменную.

|  |
| --- |
| def jump(self):  if self.jumping is False and self.shooting is False:  self.game.speed += 1  self.jumping = True  self.vel\_y = -18  def shoot(self):  if self.shooting is False and self.jumping is False:  self.shooting = True  self.current\_frame = 0  self.game.speed = 0 |

* Сделаем проверки для того, чтобы выполнялось только 1 действие за раз.
* В случае с прыжком присвоим вертикальную скорость -18 и ускорим фоновое изображение. Для этого инициализируйте в main параметр speed:

|  |
| --- |
| self.speed = GAME\_SPEED |

GAME\_SPEED будет последним параметром нашей игры - это то, с какой скоростью изначально двигается фон, добавим его в наш settings.py:

|  |
| --- |
| GAME\_SPEED = 2 |

* В случае shoot будем сбрасывать текущий фрейм на нулевой, так как будем использовать другой список изображений.
* Устанавливаем в shoot скорость игры 0, чтобы фон не двигался, когда пришелец стреляет, так как он стоит на месте в этот момент.

## 

## Метод update

Метод отвечает за обновление положения спрайта в пространстве.

|  |
| --- |
| def update(self):  self.animate()  if self.jumping:  self.rect.centery += self.vel\_y  self.vel\_y += 0.6  if self.rect.centery > HEIGHT - 150:  self.rect.centery = HEIGHT - 150  self.jumping = False  self.game.speed = GAME\_SPEED |

* В начале метода вызываем метод animate() - он будет отвечать за показ изображений в нужный момент.
* Спрайт меняет свое положение в координатах только при прыжке. Мы задаем начальную скорость спрайту (vel\_y), после чего уменьшаем ее на 0.6 с каждым тиком (0.6 в данном случае является показателем гравитации), таким образом прыжок персонажа будет выглядеть более натуральным.
* Если спрайт достиг начальной высоты, возвращаем его на исходную позицию, присваиваем переменной jumping значение False и устанавливаем начальную скорость игры.

## 

## Метод animate

В это методе будет реализована логика анимации персонажа.

В первую очередь, вспомним о трех состояниях. Подумаем о том, как они должны быть реализованы.

|  |
| --- |
| def animate(self):  now = pg.time.get\_ticks()  if self.jumping:  if self.vel\_y < 0:  self.image = self.jumping\_imgs[0]  else:  self.image = self.jumping\_imgs[1]  elif self.shooting:  if self.current\_frame == len(self.shooting\_imgs) - 1:  self.shooting = False  self.current\_frame = 0  self.game.speed = GAME\_SPEED  Bullet(self)  elif now - self.last\_update > 75:  self.last\_update = now  self.current\_frame += 1  self.image = self.shooting\_imgs[self.current\_frame]  else:  if now - self.last\_update > 75:  self.last\_update = now  self.current\_frame = (self.current\_frame + 1) % len(self.running\_imgs)  self.image = self.running\_imgs[self.current\_frame] |

* Переменная now отвечает за счетчик тиков с самого начала игры.
* Если персонаж прыгает, возьмем первое изображение, если он летит вверх (скорость отрицательная); второе - если пришелец летит вниз. Вспомним, что скорость меняется с показателем гравитации за каждый тик.
* Если персонаж стреляет, нам нужен только 1 цикл прохода по списку изображений, поэтому вначале добавим условие, что если мы дошли до конца этого списка, то сбрасываем задействованные параметры в исходное положение и создаем экземпляр класса Bullet (пуля), передадим туда экземпляр текущего класса Player, так мы создадим пулю после того, как персонаж выстрелит с точки зрения анимации. Иначе, если игра не дошла до конца списка с изображениями для выстрела и прошло 75 тиков, то обновляем счетчик тиков и отображаем следующий фрейм.
* Иначе, персонаж просто бежит. В таком случае, нам необходимо бесконечно проходится по списку изображений с бегом с промежутком в 75 тиков, для этого обновляем переменную current\_frame таким образом, чтобы она всегда была в диапазоне от нуля до длины списка.
* Во всех трех случаях присваиваем image наш текущий фрейм.

## Обновим main

В нашем классе Game (в файле main.py) добавим в метод draw вот эти две строки:

|  |
| --- |
| self.all\_sprites.draw(self.screen) self.draw\_text(str(self.score), 22, WHITE, WIDTH - 30, 30) |

Добавьте их в конец перед вызовом метода display.flip().

* Первая из двух строк будет отвечает за отрисовку всех спрайтов, принадлежащих группе all\_sprites (очевидно из названия группы, что туда мы включим абсолютно все спрайты).
* Вторая строка отображает текущий показатель победных очков игрока.

Добавьте метод update() в класс, он будет вызывать все update у всех спрайтов по аналогии с draw(), позже в update() реализуем взаимодействие между спрайтами:

|  |
| --- |
| def update(self):  self.all\_sprites.update() |

Добавим в главный цикл игры (в методе run) вызов функции update, теперь метод будет выглядеть таким образом, менять мы его больше не будем:

|  |
| --- |
| def run(self):  self.playing = True  while self.playing:  self.events()  self.update()  self.draw()  self.clock.tick(FPS) |

В метод events() добавим захват клавиш для применения способностей нашего персонажа:

|  |
| --- |
| if event.type == pg.KEYDOWN:  if event.key == pg.K\_SPACE:  self.player.jump()  if event.key == pg.K\_1:  self.player.shoot() |

Будем прыгать на кнопку пробел и стрелять на кнопку ‘1’, конечно же, можете поменять клавиши на ваше усмотрение.

Закомментируйте создание экземпляра класса Bullet и проверьте, теперь наш персонаж уже должен появиться на экране. Проверьте управление и анимацию. Все должно работать.

Есть одна проблема: персонаж всегда прыгает очень высоко. Конечно, можно уменьшить начальную вертикальную скорость, тогда пришелец не будет взлетать так высоко, однако, такую возможность хотелось бы иметь, к примеру, если бы игрок зажимал кнопку пробел, а не нажимал. Для того, чтобы реализовать прыжок на удержании клавиши, нам всего лишь нужно сбрасывать скорость, если пользователь отжал клавишу, добавим условие в events():

|  |
| --- |
| if event.type == pg.KEYUP:  if event.key == pg.K\_SPACE:  self.player.jump\_cut() |

Дословно, если поднята клавиша пробел, вызываем метод игрока jump\_cut(). Не забудем его реализовать в классе Player:

|  |
| --- |
| def jump\_cut(self):  if self.jumping:  if self.vel\_y < -3:  self.vel\_y = -3 |

Все просто: если сейчас вертикальная скорость пришельца меньше -3, то сбрасываем эту скорость до -3.

Проверьте, работает ли прыжок с зажатием и короткий прыжок на нажатие клавиши пробел.

Класс Player полностью реализован.

# Класс Bullet

В этом классе будет реализована логика для объекта пули, которая вылетает из бластера пришельца.

## Инициализация

|  |
| --- |
| class Bullet(pg.sprite.Sprite):  def \_\_init\_\_(self, player):  self.groups = player.game.all\_sprites, player.game.bullets  super().\_\_init\_\_(self.groups)  self.vel = 30  self.image = player.game.images.images\_dict['bullet'][0]  self.rect = self.image.get\_rect()  self.rect.center = (player.rect.centerx + 50, player.rect.centery + 5) |

Аналогичные атрибуты из прошлого класса:

* Инициализируем группы, в которых будет находиться наш объект пули, после чего вызовем инит класса, от которого наследуемся.
* Назначим скорость пули.
* Выберем изображение (список состоит из одного объекта), изображение меняться не будет.
* Создадим прямоугольную область объекта.
* Выберем начальные координаты относительно координат персонажа.

## Метод update

|  |
| --- |
| def update(self):  if self.rect.x > WIDTH:  self.kill()  self.rect.centerx += self.vel |

Простейшая логика, которую позволяет реализовать архитектура нашего приложения:

* Если объект вылетел за пределы экрана, удаляем его из всех групп.
* Перемещаем объект по оси X на значение скорости.

На этом класс Bullet завершен. Попробуйте запустить игру и посмотреть то, как теперь из бластера пришельца будет вылетать снаряд.

Нам осталось добавить врагов пришельца - зомби.

# Класс Zombie

В этом классе реализуем всю логику для объекта наших главных врагов (но не их рук, торчащих из-под земли).

## Инициализация

Инициализируем класс аналогично предыдущему классу:

|  |
| --- |
| class Zombie(pg.sprite.Sprite):  def \_\_init\_\_(self, game):  self.groups = game.all\_sprites, game.zombies  super().\_\_init\_\_(self.groups)  self.game = game  self.vel = self.game.speed + 1  self.images = game.images.images\_dict['zombie']  self.image = self.images[0]  self.rect = self.image.get\_rect()  self.rect.centerx = randrange(WIDTH + 50, WIDTH + 600)  self.rect.centery = HEIGHT - 140  self.last\_update = 0  self.current\_frame = 0 |

Все по тем же шаблонам для класса спрайта:

* Инициализировали группы и объект как спрайт.
* Сохранили объект игры и установили начальную скорость (зомби движется на 1 пиксель быстрее, чем фон).
* С изображениями поступаем как в классе Player. Берем первое изображение в качестве начального, инициализируем прямоугольную область и устанавливаем начальные координаты.
* last\_update и current\_frame нужны для смены фреймов (аналогично классу Player).

## 

## Метод update

Реализуем механику “за кадром” аналогично предыдущим классам.

|  |
| --- |
| def update(self):  self.vel = self.game.speed + 1  self.animate()  self.rect.x -= self.vel  if self.rect.x < -self.rect.width:  self.game.score += 1  self.kill() |

* Обновляем скорость передвижения зомби. Это необходимо делать, так как скорость меняется, если персонаж прыгает или стреляет из бластера.
* Вызываем метод animate(), в котором пропишем всю анимацию спрайта.
* Передвигаем персонажа на значение скорости.
* Если спрайт зомби вышел за левую границу экрана, добавляем 1 к победным очкам и удаляем спрайт из всех групп.

## Метод animate

Анимация спрайта, все делаем по аналогии с прошлыми классами:

|  |
| --- |
| def animate(self):  now = pg.time.get\_ticks()  if now - self.last\_update > 150:  self.current\_frame = (self.current\_frame + 1) % len(self.images)  self.image = self.images[self.current\_frame]  self.last\_update = now |

Обновляем фрейм каждые 150 тиков, в два раза медленнее, чем персонажа: зомби очень нерасторопные ребята. Остальное точно также как в Player.

## 

## Обновим main

Для того, чтобы наши зомби появлялись, исходя из предустановок с ивентами, добавим соответствующие строки в метод events:

|  |
| --- |
| if event.type == self.spawn\_zombie:  Zombie(game) |

Всего-навсего создаем экземпляр класса Zombie, остальное происходит внутри.

Теперь добавим логику для коллизий зомби с пришельцем и зомби со снарядом из бластера пришельца. Добавим строки в метод update():

|  |
| --- |
| def update(self):  self.all\_sprites.update()  bullet\_hit = pg.sprite.groupcollide(self.bullets, self.zombies, True, True)  if bullet\_hit:  self.score += 1  zombie\_hits = pg.sprite.spritecollide(self.player, self.zombies, False)  if zombie\_hits:  self.playing = False |

* Используем метод groupcollide для проверки, не столкнулись ли какой-либо снаряд и какой-либо зомби. Если да, то метод сразу удалит оба спрайта из групп (последние два передаваемых аргумента), в таком случае также зарабатываем победные очки.
* Для проверки столкновений зомби и игрока, будем использовать метод spritecollide, который проверяет коллизию одного спрайта с группой спрайтов, False - для того, чтобы спрайт из группы не удалился. Если столкновение произошло, присваиваем параметру playing значение False, после этого из внешнего цикла запустится метод end().

# 

# Класс Zarm

Зомби-рука. Тоже в sprites.py.

## Класс

|  |
| --- |
| class Zarm(pg.sprite.Sprite):  def \_\_init\_\_(self, game):  self.groups = game.all\_sprites, game.zarms  super().\_\_init\_\_(self.groups)  self.game = game  self.vel = self.game.speed  self.image = game.images.images\_dict['zarm'][0]  self.rect = self.image.get\_rect()  self.rect.centerx = randrange(WIDTH + 50, WIDTH + 300)  self.rect.bottom = HEIGHT - 50   def update(self):  self.vel = self.game.speed  if not self.game.player.shooting:  self.rect.x -= self.vel  if self.rect.x < -self.rect.width:  self.game.score += 1  self.kill() |

Думаю, вы и сами уже поняли, как его реализовать.

* Инициализируем аналогично предыдущим классам. Выбираем единственное изображение из списка, который достаем из сохраненного словаря.
* Берем прямоугольную область, устанавливаем начальное положение.
* Не забываем про начальную скорость.

В методе update():

* Обновляем скорость по аналогии с классом зомби, чтобы рука оставалась статичной относительно земли.
* Меняем положение руки только если игрок не стреляет (в таком случае фон не двигается).
* Если рука скрылась за левой частью экрана, увеличиваем значение победных очков и удаляем спрайт из игры.

## Обновим main

В events() добавим соответствующие строки кода:

|  |
| --- |
| if event.type == self.spawn\_zarms:  Zarm(self) |

В метод update() добавим проверку коллизии игрока и зомби-руки:

|  |
| --- |
| zarm\_hits = pg.sprite.spritecollide(self.player, self.zarms, False) if zarm\_hits:  self.playing = False |

Сейчас все готово, но есть небольшая проблемка с рукой - коллизия проверяется по прямоугольным областям вокруг спрайтов, а она у зомби-руки очень большая и персонаж будет ударяться, даже не доходя до зомби-руки. Есть программные решения для этой проблемы, их мы рассмотрим в следующем уроке, а пока что просто уменьшим область с использованием парой if’ов:

|  |
| --- |
| zarm\_hits = pg.sprite.spritecollide(self.player, self.zarms, False) if zarm\_hits \  and self.player.rect.right > zarm\_hits[0].rect.centerx - 20 \  and self.player.rect.centerx < zarm\_hits[0].rect.centerx + 20 \  and not self.player.jumping:  self.playing = False |

Область сделаем 20 пикселей вправо и влево от центра руки и коллизию возможной только, если персонаж не в полете.

Сейчас все должно сработать лучше.

На этом все классы реализованы и код программы полностью завершен.

# Заключение

Мы подошли к тому, что написали комплексный проект, в котором все достаточно логично и удобно структурировано и готово к масштабированию: отдельные части кода легко заменять и дополнять. Получилась симпатичная игра с хорошей анимацией, а главное, что реализованы не самые простые игровые и программные механики. Подобные проекты смело можно добавлять в портфолио, если смогли сделать их самостоятельно.

# Ссылка на проект

Для того, чтобы увидеть мой вариант решения, переходите по прикрепленной ссылке:

[Игра 8: Пришелец vs зомби](https://drive.google.com/drive/folders/1Ky6Y9G_8tV2z-QlIKqVLdXpLDWFwjKJp?usp=sharing)