Добавление усилений (бустеров)

Интерфейс для взаимодействия бустеров с игрой и увеличение скорострельности

Чтобы игровой процесс был более интересным мы можем добавить различные усиления, которые будут облегчать игру. Их реализация может быть довольно сложна, так как обычно разные бустеры затрагивают разные части игры, что требует очень хорошей организации кода. Мы реализуем один из самых простых бустеров, он будет увеличивать темп огня. Но реализованные интерфейсы помогут нам более легко подключать новые усиления.

Сначала нам нужно описать класс на основе которого будут создаваться бустеры.

```
class Booster:
    """Класс представляющий картинку бустера на игровом поле. Все эффекты
от бустеров не будут непосредственно связаны с объектами данного класса и
будут описаны внутри класса Game"""
    # Типы возможных бустеров и их картинки
    booster_types = {"Rapid_fire":
pygame.image.load(os.path.join("images", "Rapid_fire.png"))}
    def __init__(self):
        self.type = random.choice(list(self.booster_types.keys()))    #
Выбор случайного типа для бустера
        self.image = self.booster_types[self.type]                    #
Выбор картинки бустера
        self.pos = np.array([random.randint(100, SCREEN_SIZE[0] - 100),
# Генерация случайных координат для бустера
                             random.randint(100, SCREEN_SIZE[1] - 100)])
        self.rect = self.image.get_rect(center=self.pos)
 # Получение хитбокса бустера
```

В качестве **атрибута класса** мы задаём переменную `booster_types`, содержащую типы возможных бустеров, существующих в нашей игре и их картинки. Для начала мы реализуем бустер *Rapid_fire*, который будет увеличивать скорость стрельбы.

В конструкторе класса мы выбираем случайный тип бустера (`self.type`) из ключей атрибута `booster_types` . Он пока что всего один, но позже мы добавим новые. Затем мы получаем его картинку из того же словаря `booster_types`, генерируем случайную позицию на игровом поле, игнорируя места рядом с краями. Это нужно для того, чтобы бустеры было проще заметить, а также чтобы нас не мог убить астероид, резко вылетающий из-за края игрового поля. Но вы всё равно можете использовать всё свободное пространство для генерации координат, если хотите. В конце концов мы получаем хитбокс уже привычным способом.

На самом деле на этом наша работа с данным классом заканчивается. Он нужен исключительно для того, чтобы отрисовывать бустеры на игровом поле, в то время как вся логика усилений будет описана в классе Game, поскольку из него гораздо удобнее контролировать ход игрового процесса. Мы вернёмся к данному классу только когда нам понадобится добавить новые типы бустеров в словарь `booster_types`.

Теперь мы реализуем удобный интерфейс для взаимодействия бустеров с игрой. Давайте сначала опишем, что мы хотим сделать:

- Случайные бустеры будут появляться в случайных местах на игровом поле (уже реализовано в классе *Booster*).
- Бустеры должны появляться через равные промежутки времени.
- Одновременно активными могут быть 2 и более бустеров.
- Бустеры должны быть активны на протяжении какого-то фиксированного промежутка времени.
- При повторной активации бустера время его действия прибавляется к текущему остатку.

Реализовать данные возможности нам помогут новые методы и атрибуты класса *Game*.

```
class Game:
    def  __init__(self, screen):
        self.screen = screen
        self.starship = Starship()
        self.bullets = []
        self.asteroids = []
        self.boosters = []                                  # <---- Список
бустеров для отрисовки
        self.boosters_timeouts = {"Rapid_fire": 0}      # <---- Хранение
времени деактивации бустеров
        self.booster_handlers = {"Rapid_fire": self.rapid_fire}  # <----
Словарь с функциями-активаторами бустеров
        self.mouse_pressed = False
        self.fire_rate = 0.2
```

`self.boosters` — список бустеров для отрисовки. Здесь будут храниться объекты класса *Booster*. Больше ни для чего они использоваться не будут.

`self.boosters_timeouts` — словарь, хранящий время деактивации для каждого типа бустеров. Мы постоянно будем сверяться с значениями из этого словаря. Это позволит нам отключить бустер, когда срок его действия закончится.

`self.booster_handlers` — словарь, содержаший функции, который будут активировать и деактивирвать бустеры. В этих функциях будет непосредственно реализована некоторая логика работы бустеров (например, увеличение скорости стрельбы). Мы реализуем эти функции чуть позже, а пока для того, чтобы наш код нормально работал, напишем "пустой" метод `rapid_fire`.

```
def rapid_fire(self):  # Метод rapid_fire класса Game (заготовка)
    """Данная функция будет увеличивать скорость стрельбы"""
    pass
```

Практически всесь необходимый интерфейс для взаимодействия с игрой мы сможем реализовать в одном методе, давайте создадим его.

```
def boosters_manager(self, frame):        # Метод boosters_manager класса
Game
    """Метод - обработчик всех событий с бустерами в т.ч. столкновений"""
    if frame % 400 == 0 and frame != 0:     # Каждые 400 кадров (кроме
самого первого!)...
        self.boosters.append(Booster())     # ...размещаем новый бустер на
игровом поле
```

Пока что этот метод может только создавать новые бустеры каждые 400 игровых кадров. Работать с кадрами гораздо удобнее, так как они наиболее точнее описывают состояние нашей игры, чем абсолютное время. Работа с временем потребовала бы от нас гораздо больше вспомогательных переменных и вычислений. В нашем случае, если принять FPS равным 60, то бустеры будут появляться каждые **6.66** секунд (400 / 60).

Теперь добавим вызов этого метода в главный цикл игры, а также добавим бустеры (`self.boosters`) в очередь на отрисовку.

```
def run(self):              # Метод run класса Game
    frame = 0
    clock = pygame.time.Clock()
    while True:
        clock.tick(60)
        self.handle_events(frame)
        self.boosters_manager(frame)     # <-----
        self.check_collisions()
        self.move_objects([[self.starship], self.bullets, self.asteroids])
        self.draw([self.boosters, [self.starship], self.bullets,
self.asteroids])  # <-----
        frame += 1
```

Заметьте, что мы добавили бустеры **первыми** в очередь на отрисовку. На самом деле это действительно имеет некоторое значение. Как думаете, почему мы сделали именно так?

Теперь в нашей игре каждые 400 кадров появляются бустеры на игровом поле, но мы пока что никак не можем с ними взаимодействовать.

Теперь мы вернёмся к методу `rapid_fire` и опишем в нём логику работы бустера.

```
def rapid_fire(self, mode):      # Метод rapid_fire класса Game
    if mode == "activate":       # Если бустер нужно активировать...
        self.fire_rate /= 2      # ...уменьшаем паузу между пулями в 2 раза
    elif mode == "deactivate":   # Если бустер нужно отключить...
        self.fire_rate *= 2      # ...увеличиваем паузу между пулями в 2
раза (возвращаем к исходной)
```

Как мы видим, этот метод крайне прост. Ведь действительно, чтобы увеличить скорострельность нужно всего лишь изменить одну переменную. В дальнейшем мы напишем ещё 2 подобных метода-активатора. Все они должны будут принимать на вход строку, содержащую команду (*activate* или *deactivate*) и производить определённые действия в соответствии с этой командой. Подобное единообразие позволит нам сделать код чище и удобнее. Однако только с данным методом нельзя сделать полноценное взаимодействие с бустерами. Нам также понадобится обрабатывать столкновения бустеров с игроком (подбор усиления) и время их работы.

Давайте полностью допишем метод `boosters_manager`, который будет управлять логикой взаимодействия с бустерами.

```
def boosters_manager(self, frame):        # Метод boosters_manager класса
Game
    """Метод - обработчик всех событий с бустерами в т.ч. столкновений"""
    boosters_rects = [boost.rect for boost in self.boosters]       #
Хитбоксы бустеров для расчёта столкновений
```

```
    hit = self.starship.rect.collidelist(boosters_rects)        # Расчёт
столкновений игрока и бустеров
    if hit != -1:                                               # Если
столкновение есть...
        booster_type = self.boosters[hit].type                  # получаем
тип бустера из атрибута type
        # Если бустер НЕ активирован (время деактивации равно 0)...
        if self.boosters_timeouts[booster_type] == 0:
            self.boosters_timeouts[booster_type] = time.time() + 10 #
задаём время деактивации через 10 секунд
            self.booster_handlers[booster_type]("activate")         #
активируем бустер
        # Если бустер активирован (время деактивации больше 0)...
        elif self.boosters_timeouts[booster_type] > 0:
            self.boosters_timeouts[booster_type] += 10              #
увеличиваем время действия на 10 секунд
        # Удаляем бустер после столкновения (картинка исчезнет, но его
эффект будет активен)
        del self.boosters[hit]
    # Для каждого типа бустеров и времени его деактивации...
    for booster_type, timeout in self.boosters_timeouts.items():
        # Если бустер активен (timeout > 0), но активное время закончилось
(time.time() > timeout)...
        if time.time() > timeout > 0:
            self.boosters_timeouts[booster_type] = 0     # задаём время
деактивации равным 0 (отключённое состояние)
            self.booster_handlers[booster_type]("deactivate")  #
деактивируем бустер (возвращаем исходное поведение)
    if frame % 400 == 0 and frame != 0:
        self.boosters.append(Booster())
```

Здесь появилось много кода, но тут всё довольно просто.

Сначала мы расчитываем столкновения. Здесь нет ничего нового, всё это мы уже реализовывали при расчёте столкновений между игроком и астероидами, но вместо астероидов у нас здесь бустеры.

Итак мы получили индекс бустера с которым произошёл контакт (`hit`), но нам интересен не сам объект бустера, а его тип, поэтому мы получаем его и сохраняем в переменную (`booster_type`).

Далее идёт проверка состояний бустеров. Состояния у нас хранятся в атрибуте `boosters_timeouts`. На самом деле значениями в этом словаре является время, когда мы должны будем деактивировать бустеры. Но зная текущее время и заданное в этом атрибуте мы может понять в каком состоянии находится усиление:

- `self.boosters_timeouts[booster_type] == 0` — деактивированное состояние. В случае подбора усиления при данном состоянии нам нужно задать новое время деактивации бустера (текущее время + 10 секунд (или любой другой интервал)) и непосредственно активировать эффект через метод-активатор (например, `self.rapid_fire`). Так как время деактивации становится больше 0, бустер переходит в следующее состояние.
- `self.boosters_timeouts[booster_type] > 0` — активированное состояние (нам известно время отключения, а значит бустер в данный момент работает). В случае подбора усиления при данном состоянии мы увеличим время до деактивации ещё на 10 секунд. На самом деле вместо этого можно было сделать усиление эффекта (например, увеличение

скорости стрельбы в 4 раза (вместо 2) при подборе двух бустеров). Такая логика намного интереснее, но ещё сложнее написать.

Не забываем удалять объект бустера после регистрации столкновения, чтобы он исчез с игрового поля (`del self.boosters[hit]`).

- `self.boosters_timeouts[booster_type] < time.time()` — сигнал для деактивации (текущее время больше, чем время, в которое мы должны отключить бустер). Проверку на данное состояние мы проводим после обработки столкновений (но можно было и перед ней). Здесь мы проверяем, не пора ли нам деактивировать каждый и бустеров? Если пора, то мы переводим бустер в деактивированное состояние (время до деактивации равно 0) и отключаем эффекты от бустера через функцию-активатор. Не забываем, что на данном этапе нас интересуют только активные бустеры, поэтому у нас есть условие `if time.time() > timeout > 0:`.

Всё, наш интерфейс для работы с бустерами готов, а заодно мы ещё и реализовали усиление скорострельности! Теперь нам не нужно заботиться о этой реализации и мы можем быть уверены, что наши бустеры будут активироваться и деактивироваться в нужный момент, вне зависимости от логики их работы.

Запустим игру и подберём бустер.

Щит, защищающий от астероидов

Как мы сейчас увидим, сделать подобный бустер очень несложно, однако нам придётся добавлять код, связанный с работой этого бустера в другие методы. Это немного ухудшит читаемость нашего кода, поэтому всегда нужно стараться делать так, чтобы какой-то функционал был изолирован, а не раскидан по всему коду программы. В данном случае мы пренебрежём этой рекомендацией для экономии времени, но в будущем старайтесь всегда продумывать структуру сложных проектов заранее таким образом, чтобы код был модульным.

Давайте разместим все данные о новом бустере в нужные места:

```
class Booster:
    # Типы возможных бустеров и их картинки
    booster_types = {"Rapid_fire":
pygame.image.load(os.path.join("images", "Rapid_fire.png")),
                     "Shield": pygame.image.load(os.path.join("images",
"Shield.png"))}  # <----
    def __init__(self):
        ...
```

Сперва поместим информацию о типе нового бустера и его картинке в атрибут `booster_types` класса *Booster*. Добавление этой строчки позволит нам генерировать картинки бустера на игровом поле (но пока что без взаимодействия с ним).

```
def shield(self, mode):     # Метод shield класса Game (метод-активатор)
    if mode == "activate":
        # При активации подменяем оригинальную картинку звездолёта на
картинку с щитком
        self.starship.original_image =
pygame.image.load(os.path.join("images", "Starship_with_shield.png"))
    elif mode == "deactivate":
        # При деактивации возвращаем исходную картинку на место
        self.starship.original_image =
pygame.image.load(os.path.join("images", "starship.png"))
```

Далее мы описали метод-активатор для нового бустера. Поскольку мы не можем обрабатывать столкновения с астероидами внутри этого метода (вся логика усиления "Щит" построена на изменении логики столкновений), здесь мы только лишь подменим картинку у звездолёта на изображение звездолёта с щитом.

```
class Game:
    def  __init__(self, screen):
        self.screen = screen
        self.starship = Starship()
        self.bullets = []
        self.asteroids = []
        self.boosters = []
        self.boosters_timeouts = {"Rapid_fire": 0, "Shield": 0}  # <-----
        self.booster_handlers = {"Rapid_fire": self.rapid_fire, "Shield":
self.shield}  # <-----
        self.mouse_pressed = False
        self.fire_rate = 0.2
```

Также мы добавили запись о новом бустере в атрибут `boosters_timeouts` для его корректной активации и деактивации, а также в атрибут `booster_handlers` мы добавили метод-активатор данного усиления.

Теперь наш бустер будет активироваться при контакте с ним, а звездолёт будет окружён щитом. Однако на данный момент этот щит - всего лишь картинка и он не будет защищать нас от астероидов. Для реализации самой логики защиты мы переместимся в метод `check_collisions` класса *Game*.

```
def check_collisions(self):
    asteroids_rects = [ast.rect for ast in self.asteroids]
    for idx, bullet in enumerate(self.bullets):
        if bullet.pos[0] > SCREEN_SIZE[0] or bullet.pos[1] >
SCREEN_SIZE[1] or (bullet.pos < 0).any():
            del self.bullets[idx]      # удалить объект пули
    for idx, bullet in enumerate(self.bullets):
        hit = bullet.rect.collidelist(asteroids_rects)
        if hit != -1:
            fragments = self.asteroids[hit].explode()
            del self.asteroids[hit]
            self.asteroids += fragments
            del self.bullets[idx]
```

```
        hit = self.starship.rect.collidelist(asteroids_rects)
        if hit != -1:
            # Если бустер "Щит" активен
            if self.boosters_timeouts["Shield"] > 0:  # <----
                del self.asteroids[hit]               # <---- Удалим
астероид с которым столкнулись
            else:
                sys.exit()
```

До этого момента при столкновении с астероидом мы сразу вызывали функцию `sys.exit` для выхода из игры. Однако теперь мы не хотим этого делать при условии, что у нас есть щит. Мы проверяем условие `if self.boosters_timeouts["Shield"] > 0:`, как вы помните оно позволяет нам определить активен ли наш бустер. Если он активен, то мы просто удалим астероид с которым произошло столкновение (`del self.asteroids[hit]`), а если бустер неактивен, то выйдем из игры как обычно (`sys.exit()`).

И всё, наш щит готов! Всего за несколько строчек кода мы добавили новое усиление. Проверим его на практике.

Теперь щит можно активировать и он защищает нас от астероидов. При этом мы можем одновременно использовать несколько бустеров благодаря тому, что храним их время деактивации отдельно. Никаких конфликтов между бустерами нет, потому что мы довольно хорошо описали метод `booster_manager`.

Выстрелы по несколько пуль

Это будет один из самых сложных бустеров в нашей игре. Чтобы всё работало максимально устойчиво к изменениям и масштабировалось (например, если мы заходим поменять количество пуль, которые выстреливает звездолёт) нам снова немного придётся залезть в математику, а также немного изменить несколько методов.

Для начала мы зададим параметр, который будет определять число пуль, которые звездолёт выстреливает за 1 раз, а заодно занесём информацию о новом методе в атрибуты `boosters_timeouts` и `booster_handlers`

```
class Game:
    def  __init__(self, screen):
        self.screen = screen
        self.starship = Starship()
        self.bullets = []
        self.asteroids = []
        self.boosters = []
        self.boosters_timeouts = {"rapid_fire": 0, "Shield": 0,
"Triple_bullets": 0}   # <----
        self.booster_handlers = {"rapid_fire": self.rapid_fire, "Shield":
self.shield, "Triple_bullets": self.triple_bullets}     # <----
        self.mouse_pressed = False
        self.fire_rate = 0.2
        self.max_bullets = 1    # <---- Кол-во пуль, которые звездолёт
выстреливает за 1 раз
```

Сейчас атрибут `max_bullets` равен 1, так как по-умолчанию бустер не активирован и звездолёт стреляет по 1 пуле. При активации бустера мы будем изменять этот параметр.

Не забываем подгрузить картинку бустера для класса *Booster*.

```
class Booster:
    # Типы возможных бустеров и их картинки
    booster_types = {"Rapid_fire":
pygame.image.load(os.path.join("images", "Rapid_fire.png")),
                     "Shield": pygame.image.load(os.path.join("images",
"Shield.png")),
                     "Triple_bullets":
pygame.image.load(os.path.join("images", "Triple_bullets.png"))}  # <----

    def __init__(self):
        ...
```

А также написать метод-активатор.

```
def triple_bullets(self, mode): # Метод triple_bullets класса Game
    if mode == "activate":      # При активации...
        self.max_bullets = 3    # изменить кол-во пуль, которые звездолёт
выстреливает за 1 раз на 3
    elif mode == "deactivate":  # При деактивации...
        self.max_bullets = 1    # вернуть значение обратно равным 1
```

При активации мы изменим количество выстреливаемых пуль на 3, но можно будет задать любое другое число.

Как и раньше мы сделали 4 необходимых действия:

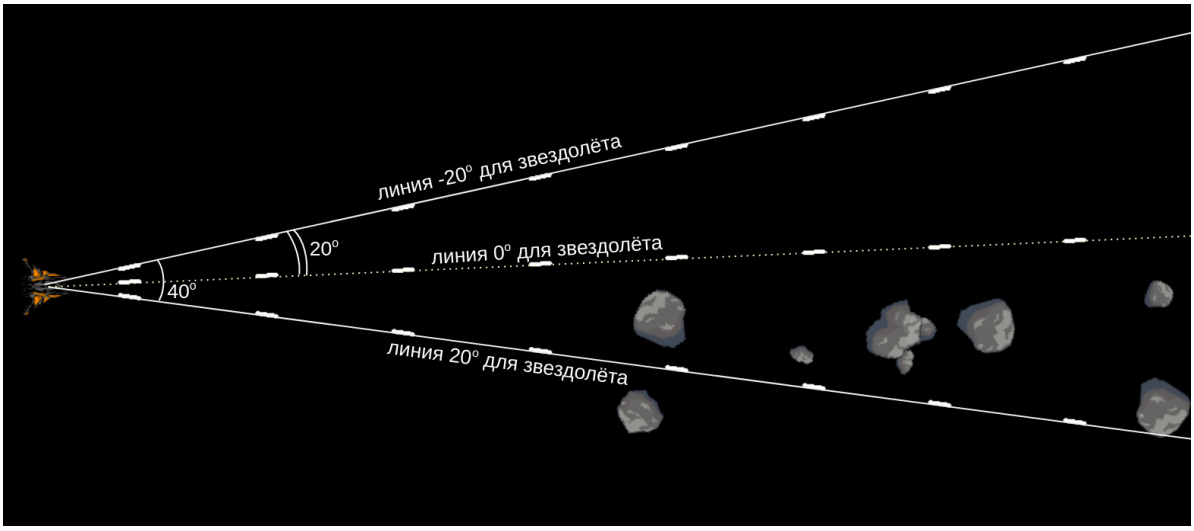
- Загрузили картинку бустера и задали его тип в классе *Bullet*.
- Сделали запись о бустере нового типа в атрибуте `boosters_timeouts` класса *Game*.
- Создали метод активатор для нового бустера.
- Добавили данный метод-активатор в атрибут `booster_handlers` класса *Game*.

После этих действий над любым бустером мы сможем его подбирать и у него появится время действия. Однако логику каждого из них, кроме самых простых (например, *Rapid_fire*) придётся задавать в различных местах нашего кода.

В данном случае мы изменили атрибут `max_bullets` класса *Game*, но сам по себе он ничего не делает. Мы будем использовать его в другом месте кода, но в каком же? Логично использовать его в момент создания пуль, создавая несколько вместо одной. В нашей игре пули создаются в методе `fire` класса *Starship*. Давайте изменим его для создания заданного количества пуль:

```
def fire(self, bullet_num):         # Метод fire класса Starship
    self.last_bullet_time = time.time()     # Сохраняем время последнего выстрела
    new_bullets = []                        # Создаём список, куда поместим все новые пули
    for i in range(bullet_num):             # Заданное количество раз...
        if bullet_num == 1:                 # В случае одной пули...
            angle_offset = 0                # ...зададим смещение относительно направления взгляда равным 0
        else:                               # Иначе...
            angle_offset = -15 + 30 / (bullet_num - 1) * i    # ..расчитаем смещения для всех пуль
        new_bullets.append(Bullet(self.pos.copy(), angle_offset))    # Создаём пулю с заданным смещением
    return new_bullets      # Возвращаем список из созданных пуль (список из 1 элемента по-умолчанию)
```

Теперь наш метод принимает 1 аргумент — `bullet_num`, обозначающий количество пуль, которые надо создать. Метод по прежнему сохраняет время последнего выстрела. Поскольку теперь мы создаём несколько пуль, нам понадобится список для их хранения (`new_bullets`). Но нам недостаточно просто создать несколько одинаковых пуль, они просто наложатся друг на друга и будут неразличимы. Мы же хотим, чтобы они летели немного в разные стороны, отклоняясь от направления стрельбы. Поэтому нам предстоит расчитать это самое отклонение (`angle_offset`). Для 1 пули это отклонение будет равно 0 градусов, так как 1 пуля должна лететь по прямой. Для того чтобы расчитать отклонения для нескольких пуль нам нужно будет задать 1 важный параметр — максимальный угол разброса пуль.



На картинке этот угол равен 40 градусам. Это значит, что каждая пуля не может отклониться от направления стрельбы больше, чем на половину от этой величины (то есть на 20 градусов). Учитывая это, нам нужно разместить заданное количество пуль (`bullet_num`) через равные интервалы в данной области. Расчёт мы производим по следующей формуле: `angle_offset = -15 + 30 / (bullet_num - 1) * i`, где **-15** это начальный угол от которого мы начинаем создавать пули, **30** это полный угол на который будут разлетаться пули, **bullet_num** это общее количество пуль, а **i** - порядковый номер пули (начиная с 0). Попробуйте понять, что происходит в данной формуле самостоятельно.

После расчёта отклонения от направления выстрела мы можем создать пулю с заданным параметром и добавить её в список новых пуль.

Теперь нам осталось лишь учесть данный параметр при создании пуль, поэтому давайте посмотрим на изменённый конструктор класса *Bullet*.

```
class Bullet:
    original_image = pygame.image.load(os.path.join("images",
"bullet.png"))
    # Теперь мы принимаем смещение как аргумент и задаём его по-умолчанию
равным 0
    def __init__(self, pos, angle_offset=0):
        self.pos = pos
        mouse_pos = pygame.mouse.get_pos()
        self.direction = (mouse_pos - pos)
        self.speed = self.direction / max(abs(self.direction)) * 10
        self.angle = self.calculate_angle(mouse_pos) - angle_offset    #
<---- Изменим угол на величину смещения
        self.image = pygame.transform.rotate(self.original_image,
int(self.angle))
        self.rect = self.image.get_rect(center=self.pos)
```

Для того чтобы всё заработало нам нужно передавать число пуль, которые нужно выстрелить в метод `fire` класса *Starship*. Но число пуль нам уже известно! Оно хранится в атрибуте `max_bullets` класса *Game*, а метод `booster_manager` заботится о том, чтобы оно всегда было актуальным. Так что просто будем передавать это значение как аргумент в метод `fire`.

```
def handle_events(self, frame):       # Метод handle_events класса Game
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        if event.type == pygame.MOUSEBUTTONDOWN:
            self.mouse_pressed = True
        if event.type == pygame.MOUSEBUTTONUP:
            self.mouse_pressed = False
    if self.mouse_pressed and (time.time() -
self.starship.last_bullet_time) > self.fire_rate:
        new_bullets = self.starship.fire(self.max_bullets)  # <----
Создаём новые объекты пуль
        self.bullets += new_bullets  # <----  И добавляем их в общий
список
    if frame % 25 == 0:
        self.cast_asteroid()
```

Мы передали количество пуль в метод `fire` и теперь мы получаем **список** пуль (раньше была одна пуля). Соответственно теперь мы **прибавляем** этот список к общему списку пуль (раньше был метод *append*). Давайте запустим игру и подберём новый бустер.

Мы видим очень необычную картину. При подборе бустеров пули довольно странно изменяют свою форму. В чём же может быть дело? На самом деле пули не изменяют свою форму. Просто это 3 пули, повёрнутые на разные углы и наложенные друг на друга. Действительно, мы развернули картинки пуль, но не изменили направление их полёта, давайте исправим это.

```
class Bullet:
    original_image = pygame.image.load(os.path.join("images",
"bullet.png"))
    def __init__(self, pos, angle_offset=0):
        self.pos = pos
        mouse_pos = pygame.mouse.get_pos()
        self.direction = (mouse_pos - pos)
        asr = math.pi / 180 * angle_offset    # <---- Переводим величину
смещения в радианы из градусов
        # "Поворачиваем" первую координату направления
        self.direction[0] = self.direction[0] * math.cos(asr) -
self.direction[1] * math.sin(asr)  # <----
        # "Поворачиваем" первую координату направления
        self.direction[1] = self.direction[0] * math.sin(asr) +
self.direction[1] * math.cos(asr)  # <----
        self.speed = self.direction / max(abs(self.direction)) * 10
        self.angle = self.calculate_angle(mouse_pos) - angle_offset
        self.image = pygame.transform.rotate(self.original_image,
int(self.angle))
```

```
            self.rect = self.image.get_rect(center=self.pos)
```

Направление полёта пули представлено двумя координатами по осям *x* и *y* (`direction = np.array([x, y])`). Для изменения направления на определённый угол мы можем воспользоваться следующими формулами:

new_x = **x** * cos(**asr**) - **y** * sin(**asr**)

new_y = **y** * sin(**asr**) + **y** * cos(**asr**)

new_direction = np.array([**new_x**, **new_y**]),

где **asr** это угол поворота в радианах.

В нашем случае соответственно **y** это `self.direction[1]`, а **x** это `self.direction[0]`.

Перед поворотом мы переводим смещение из градусов в радианы по формуле из 2 урока, поскольку функции *cos* и *sin* из модуля *math* не умеют работать с градусами.

Далее мы поворачиваем направление (`self.direction`) согласно формулам и всё готово!

Теперь всё работает как надо.

Создание файла с настройками

В нашей игре было очень много параметров, которые мы задавали вручную. Например, размер окна, скорость корабля, частота появления астероидов и т.д. Мы можем создать отдельный модуль (файл с расширением *.py*), который будет содержать все настраиваемые параметры, а затем импортировать его из нашей программы. Так мы сможем настраивать нашу игру, не тратя время на поиск нужных параметров в коде. Всё что нам нужно будет сделать это изменить эти параметры в файле с настройками. Давайте создадим такой файл **config.py**:

```
### Общие настройки игры
MAX_FPS = 60                # Максимальное кол-во кадров в секунду
SCREEN_SIZE = (0, 0)        # Размер окна игры в пикселях (0, 0) - полный
экран

### Настройки игрового процесса

## Настройки перемещения
SHIP_SPEED = 40             # Скорость звездолёта (чем больше, тем
медленнее)

## Настройки стрельбы
FIRE_RATE = 0.2             # Темп стрельбы (чем меньше, тем быстрее)
BULLET_SPEED = 10           # Скорость пули (чем больше, тем быстрее)

## Настройки астероидов
ASTEROIDS_SPAWN_RATE = 25   # Частота появления астероидов (чем меньше, тем
чаще)
ASTEROID_SPEED = 300        # Скорость астероида (чем больше, тем
медленнее)
MEDIUM_FRAGMENTS = 2        # Количество фрагментов при разрушении большого
астероида
SMALL_FRAGMENTS = 3         # Количество фрагментов при разрушении среднего
астероида
FRAGMENTS_SPEED = 0.5       # Скорость осколков астероидов (чем больше, тем
быстрее)

## Настройки бустеров
BOOSTER_SPAWN_RATE = 400    # Частота появления бустеров (чем меньше, тем
чаще)
BOOSTER_DURATION = 10       # Длительность действия бустеров (в секундах)
MAX_BULLETS = 3             # Количество пуль при подборе бустера
"Triple_bullets"
RAPID_FIRE_MULTIPLIER = 2   # Во сколько раз изменяется скорострельность
при подборе бустера "Rapid_fire"
```

В нём мы просто задаём нужные нам параметры в виде переменных.

Теперь в основной программе нам необходимо выполнить импорт данного модуля:

```
from config import *
```

Это позволит нам напрямую использовать все переменные из **config.py** в нашей игре. Остаётся только заменить все перечисленные параметры в коде программы на данные переменные. Я сделал это за вас. В архиве с кодом для данного урока содержится файл **config.py** и скрипт **game_with_config.py**, который уже умеет работать с модулем настроек. При желании вы можете скопировать код оттуда в *Jupyter notebook*.

Теперь вы можете поиграться с настройками и при желании довольно сильно изменить игровой процесс.

Домашнее задание

Попробуйте реализовать **хотя бы 1** новую механику или бустер. Придумайте что-нибудь сами, но если никак не получается придумать, то вот вам несколько идей от меня:

- **Бомба**. Появляется в случайном месте. Если попасть в неё пулей, она уничтожит все астероиды в некотором радиусе (на ваш выбор) при этом не будет никак влиять на игрока.
- **Магнит**. Появляется в случайном месте как бустер. Вы можете его подобрать и разместить в любом месте игрового поля, нажав на **пробел**. При этом магнит активируется и начинает притягивать к себе астероиды в течение **10** секунд.
- **Стенка**. Это будет бустер. После подбора бустера вы можете разместить длинную стенку (просто прямая линия) по направлению взгляда игрока, нажав на **пробел**. Астероиды будут полностью разрушаться при контакте со стенкой, звездолёт может свободно через неё проходить. Стенка неподвижна.
- **Повышение уровня**. Реализуйте повышение уровня звездолёта за каждые 30 или 60 секунд в игре или за каждые 100 разрушенных астероидов. При повышении уровня у вас увеличивается скорость стрельбы или/и количество пуль по-умолчанию или/и скорость перемещения.
- **Противники**. Добавьте примитивных противников со следующей моделью поведения:
  - Постоянно движутся в сторону игрока с небольшой скоростью.
  - Иногда могут стрелять пулями, которые тоже имеют небольшую скорость.
  - Вражеские пули так же как и наши могут разбивать астероиды.
  - Для того чтобы убить врага нужно попасть в него 3 раза.

Все перечисленные механики имеют разную сложность реализации. Выберите ту, которая кажется вам оптимальной или самой интересной. Если вам кажется, что эти задачки слишком сложные, постарайтесь придумать что-нибудь своё)

В любом случае вам могут понадобиться следующие ресурсы (ищите информацию там или спрашивайте у меня в телеграме):

- Официальная документация pygame — содержит всю возможную информацию (на английском языке). Относительно нормально получается если перевести через *Google translate*, но осторожно, могут перевестись названия функций и переменных.
- Шпаргалка по основным моментам в pygame — если нужно узнать, что делает какой-то атрибут или как обработать то или иное событие, то вам сюда (на русском языке).
- Или просто гуглите интересующий вас вопрос (например, "Обработка нажатий на клавиатуру в pygame"). Зачастую такой способ является наиболее эффективным.