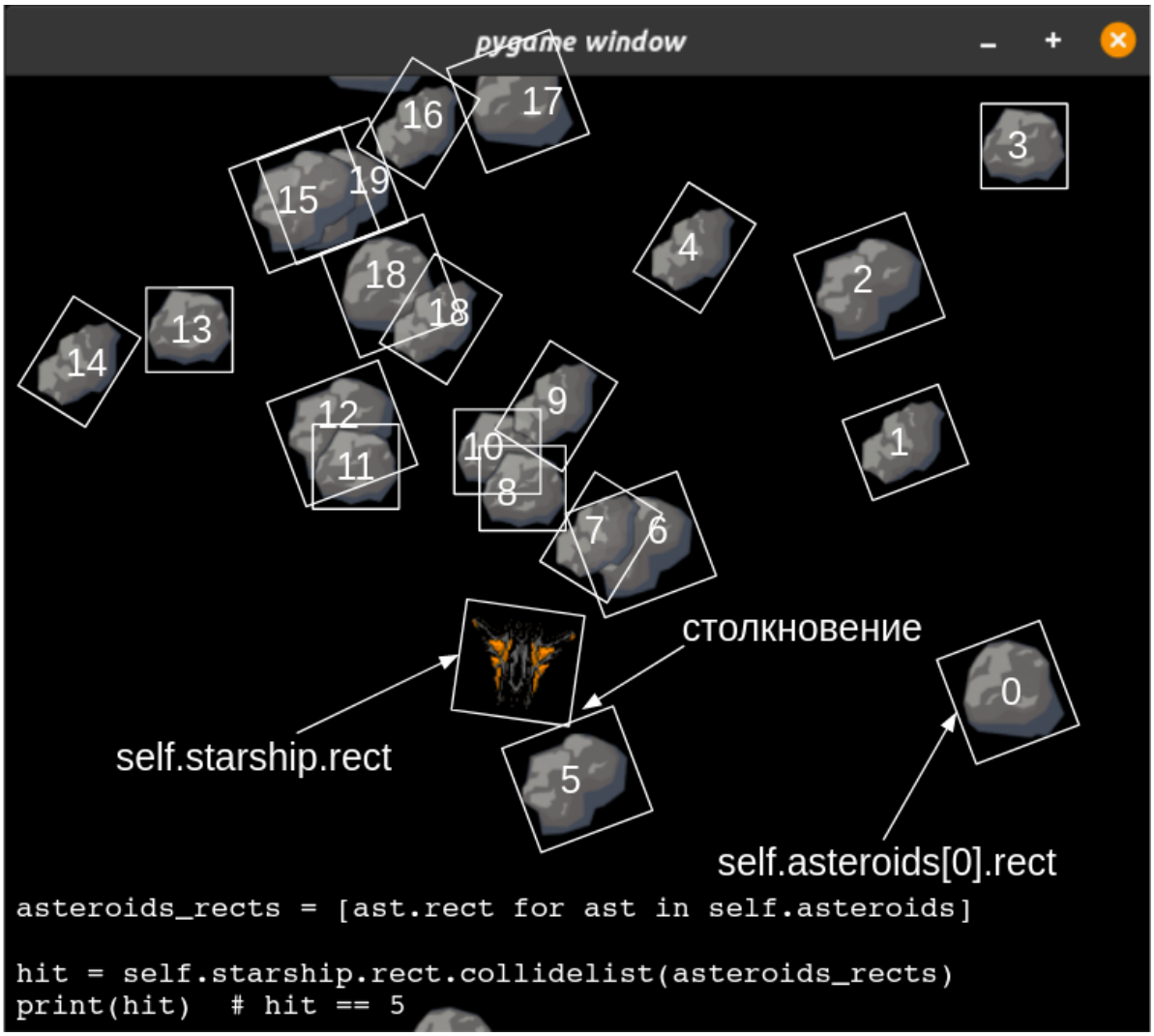
Взаимодействие с астероидами

На данный момент у нас имеются астероиды, но с ними никак нельзя взаимодействовать.

Во-первых, мы можем добавить возможность уничтожения астероидов. Но для начала мы узнаем как расчитывать **столкновения** между объектами. **Столкновение** — это пересечение хитбоксов двух объектов. Реализовать любые взаимодействия между объектами без расчёта столкновений очень сложно и сами расчёты столкновений также довольно сложны. Нам повезло, что в *pygame* есть методы, специально созданные для подобных задач. Нас интересует метод *Rect.collidelist(rect_list)*, который принимает на вход контейнер, содержащий хитбоксы (*Rect*) и возвращает индекс хитбокса с которым происходит столкновение (или -1, если столкновений нет). Рассмотрите пример на картинке для лучшего понимания:



Вы можете заметить, что хотя хитбоксы игрока и астероида с индексом 5 сталкиваются, картинки всё ещё находятся на расстоянии друг от друга. Со стороны игрока это может выглядеть как ложные столкновения. Как вы думаете, что можно сделать для исправления такого поведения?

Теперь напишем код, который будет обрабатывать столкновения. Для этого мы обновим метод *Game.check_collisions*.

```
def check_collisions(self):  # Метод check_collisions класса Game
    # Список хитбоксов всех астероидов
    asteroids_rects = [ast.rect for ast in self.asteroids]   # <-----
    # Удаление пуль при вылете с игрового поля
```

```
    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]
    # Разрушение астероидов при контакте с пулями
    for idx, bullet in enumerate(self.bullets):        # <----
        # С каким из астероидов имеется столкновение?
        hit = bullet.rect.collidelist(asteroids_rects) # <---- С каким
астероидом сталкивается данная пуля
        if hit != -1:                                  # <---- Если
столкновение есть...
            del self.asteroids[hit]                    # <---- ...удалить
астероид с индексом hit
            del self.bullets[idx]                      # <---- ...удалить
пулю с индексом idx
```

Каждый раз при проверке столкновений мы создаём список хитбоксов астероидов (*asteroids_rects*). Это довольно негативно сказывается на производительности. Мы могли бы хранить отдельно списки хитбоксов, но это не очень удобно с точки зрения читаемости кода. Далее для каждой пули мы берём её хитбокс (*bullet.rect*) и проверяем столкновения с астероидами (*bullet.rect.collidelist(asteroids_rects)*). В случае, если столкновения обнаружились (*hit != -1*), мы удаляем соответствующие пулю и астероид.

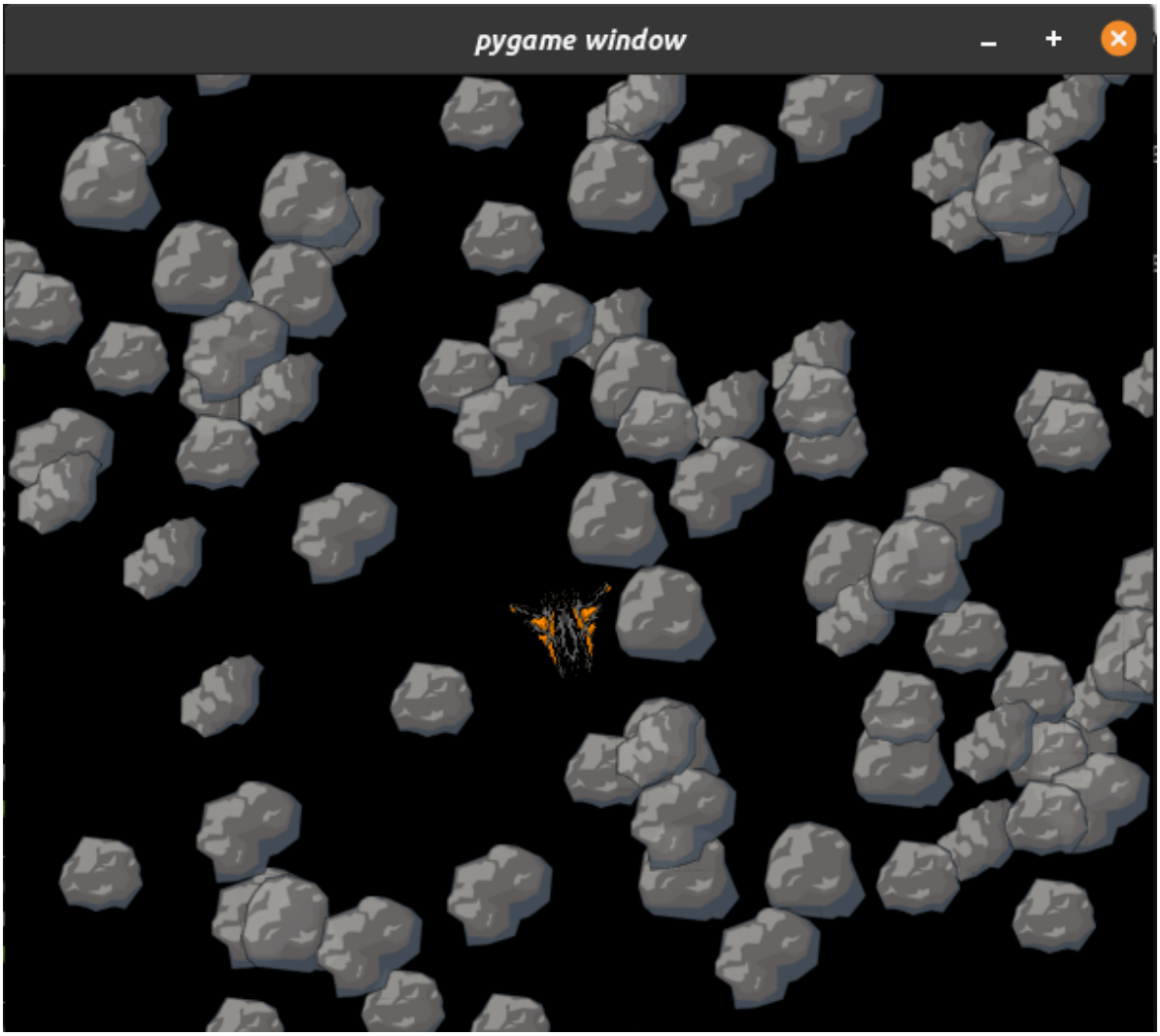
Отлично, теперь мы умеем разрушать астероиды. Однако, что происходит с астероидами, когда они выходят из области игрового поля? Правильно, ничего. Здесь у нас та же самая ситуация, что когда-то была с пулями. Мы можем аналогично описать условия для удаления астероидов, но давайте попробуем оставить их в игре. Сделаем так, чтобы при выходе за границы игрового поля, они появлялись с противоположной стороны. В конце-концов это будет приводить к накоплению астероидов и увеличению сложности игры. Для обработки выхода за границы мы опишем метод *Asteroid.check_borders*.

```
def check_borders(self):      # Метод check_borders класса Asteroid
    # Если центр астероида за нижней границей...
    if self.pos[0] > (SCREEN_SIZE[0] + 100):
        self.pos[0] = -100       # ...перемещаем его на верхнюю
    # Если центр астероида за верхней границей...
    elif (self.pos[0] + 100) < 0:
        self.pos[0] = SCREEN_SIZE[0] + 100  # ...перемещаем его на нижнюю
    # Если центр астероида за правой границей...
    if (self.pos[1] - 100) > SCREEN_SIZE[1]:
        self.pos[1] = -100     # ...перемещаем его на левую
    # Если центр астероида за левой границей...
    elif (self.pos[1] + 100) < 0:
        self.pos[1] = SCREEN_SIZE[1] + 100    # ...перемещаем его на
правую
    # Обновляем хитбокс, так как произошло перемещение
    self.rect = self.image.get_rect(center=self.pos)
```

Код тут довольно прост. Когда одна из координат достигает границы игрового поля +- 100 пикселей (так как мы по прежнему не хотим резких появлений астероидов) мы перемещаем астероид на противоположную сторону. Так как этот метод по-сути производит перемещение, его вызов можно поместить в метод *Asteroid.move*.

```
def move(self):        # Метод move класса Asteroid
    self.check_borders()     # <----- Обработка выхода за границы
    self.pos += self.speed
    self.rect = self.image.get_rect(center=self.pos)
```

Теперь, если открыть игру и не уничтожать астероиды, мы увидим, что их число начинает быстро возрастать.



В нашу игру уже практически можно играть. Осталось лишь добавить возможность проиграть (то есть обработать столкновение игрока с астероидами). И это очень просто сделать!

```
def check_collisions(self):  # Метод check_collisions класса Game
    asteroids_rects = [ast.rect for ast in self.asteroids]
    # Удаление пуль при вылете с игрового поля
    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]
    # Разрушение астероидов при контакте с пулями
    for idx, bullet in enumerate(self.bullets):
        hit = bullet.rect.collidelist(asteroids_rects)
        if hit != -1:
            del self.asteroids[hit]
            del self.bullets[idx]
    # Столкновение астероидов и игрока
    # Индекс астероида, столкнувшегося с игроком
    hit = self.starship.rect.collidelist(asteroids_rects) # <---
```

```
    if hit != -1:    # Если столкновение было...            # <---
        sys.exit()   # ...выйти из игры                     # <---
```

Как и остальные обработчики столкновений, мы поместим данный в метод *Game.check_collisions*. Список хитбоксов астероидов уже создан нами в начале выполнения метода, нам нет нужды делать это второй раз. При помощи *Rect.collidelist* мы проверяем, есть ли столкновения между игроком и астероидами. Теперь нас даже не интересует индекс, участвующего в этом астероида. Нам достаточно самого факта наличия столкновения.

Добавление осколков астероидов

На данный момент астероиды в нашей игре исчезают после попадания пулей, но мы хотим, чтобы поведение было более интересное. Мы сделаем 3 типа астероидов: большие, средние и маленькие. При попадании в большой астероид он будет разлетаться на несколько средних, каждый из которых развалится на несколько маленьких при попадании, маленькие же будут исчезать.

Для этого нам понадобятся картинки для всех типов астероидов. Изначально мы имели только 4 картинки с астероидами среднего размера. Для того чтобы не искать в интернете и не рисовать новые, мы просто сделаем копии существующих картинок с уменьшенным и увеличенным размером (вручную в графическом редакторе). Таким образом мы получим 12 картинок (по 4 на каждый тип астероида). Также нам нужно учесть, что астероиды различного размера будут иметь различные хитбоксы. Учитывая всё это, обновим атрибут класса *Asteroid* в который загружаются картинки.

```
# Атрибут ast_variants класса Asteroid
ast_variants = [("small", pygame.image.load(os.path.join("images",
"ast1_small.png")), (10, 10)),
                ("small", pygame.image.load(os.path.join("images",
"ast2_small.png")), (10, 10)),
                ("small", pygame.image.load(os.path.join("images",
"ast3_small.png")), (10, 10)),
                ("small", pygame.image.load(os.path.join("images",
"ast4_small.png")), (10, 10)),
                ("medium", pygame.image.load(os.path.join("images",
"ast1_medium.png")), (40, 40)),
                ("medium", pygame.image.load(os.path.join("images",
"ast2_medium.png")), (40, 40)),
                ("medium", pygame.image.load(os.path.join("images",
"ast3_medium.png")), (40, 40)),
                ("medium", pygame.image.load(os.path.join("images",
"ast4_medium.png")), (40, 40)),
                ("large", pygame.image.load(os.path.join("images",
"ast1_large.png")), (105, 105)),
                ("large", pygame.image.load(os.path.join("images",
"ast2_large.png")), (105, 105)),
                ("large", pygame.image.load(os.path.join("images",
"ast3_large.png")), (105, 105)),
                ("large", pygame.image.load(os.path.join("images",
"ast4_large.png")), (105, 105))]
```

Запись выглядит довольно сложно, но это всего лишь списокк, где каждый элемент это кортеж из *типа астероида* (small, medium, large), *картинки* и *ширины и высоты хитбокса*. Мы изменили название атрибута на *ast_variants*, так как изменился тип данных хранимых в нём (`original_images` могло ввести в заблуждение).

Теперь, поскольку мы обновили атрибут, хранящий параметры астероидов, нам нужно внести правки в конструктор класса *Asteroids*, чтобы при создании объекта он сохранял его тип и хитбокс.

```
def __init__(self):      # Конструктор класса Asteroid
    # Случайный выбор типа и картинки астероида
    self.type, self.original_image, hitbox_shape =
random.choice(self.ast_variants)  # <----
    self.image = self.original_image   # <----
    self.w, self.h = hitbox_shape # <---- Сохраняем длину и ширину
хитбокса
    self.pos = random.choice([self.left_pos, self.top_pos,
                              self.right_pos, self.bottom_pos])()
    # Получаем хитбокс с заданными размерами
    self.rect = self.image.get_rect(center=self.pos, width=self.w,
height=self.h)  # <-----
    self.direction = pygame.mouse.get_pos() - self.pos
    self.speed = self.direction / 300
```

Здесь мы не видим ничего необычного. Сначала мы выбираем случайный элемент из атрибута *ast_variants*, этот элемент содержит тип астероида, картинку и размеры хитбокса, которые мы сохраняем в атрибуты объекта. Заметьте, что вероятности выбрать астероиды каждого класса равны, так как для каждого из них существует по 4 элемента списка.

Далее мы создаём атрибут `self.image`, так как оригинальное изображение может пригодится нам в будущем.

Сохраняем в отдельные атрибуты ширину (`self.w`) и высоту (`self.h`) хитбокса для их более короткой записи.

Поскольку каждый тип астероида должен иметь свой размер хитбокса, нам необходимо обозначить данные параметры при его получении, передав его ширину и высоту в качестве аргументов `width` и `height` при вызове `self.image.get_rect`. Размеры хитбокса можно было бы и не задавать, но в таком случае размеры хитбокса стали бы равны размеру изображения, что, как мы знаем, слишком много и может приводить к "ложным" столкновениям.

Аналогично конструктору изменим параметры получения хитбокса в методе `move`.

```
def move(self):     # Метод move класса Asteroid
    self.check_borders()
    self.pos += self.speed
    self.rect = self.image.get_rect(center=self.pos, width=self.w,
height=self.h)  # <----
```

Отлично, теперь при запуске игры мы увидим разные типы астероидов с правильными хитбоксами, при этом они пока всё ещё разрушаются одним выстрелом.

Давайте опишем сам механизм разрушения астероидов на маленькие.

```
def explode(self):    # Метод explode класса Asteroid  (черновая версия)
    fragments = []          # Список с осколками астероидов
    if self.type == "large":              # Если астероид большой...
        for _ in range(2):
            fragments.append(Asteroid())  # ..добавить 2 астероида в
список
    if self.type == "medium":             # Если астероид средний...
        for _ in range(3):
            fragments.append(Asteroid())  # ...добавить 3 астероида в
список
    if self.type == "small":              # Если астероид маленький...
        pass                              # ...не делать ничего
    return fragments
```

Метод `explode` является ключевым в данном механизме. Он будет вызываться при столкновении астероида и пули, а элементы списка осколков будут добавляться в основной список астероидов.

```
def check_collisions(self):           # Метод check_collisions класса
Game
    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]
        for idx, bullet in enumerate(self.bullets):
            hit = bullet.rect.collidelist(asteroids_rects)
            if hit != -1:
                fragments = self.asteroids[hit].explode() # <---- Разбиваем
астероид на осколки
                del self.asteroids[hit]
                self.asteroids += fragments               # <---- Добавляем
эти осколки в общий список
                del self.bullets[idx]
        hit = self.starship.rect.collidelist(asteroids_rects)
        if hit != -1:
            sys.exit()
```

Проблема состоит в том, что новые астероиды (осколки) будут создаваться по тому же принципу, что и обычные астероиды (за пределами экрана и двигаться в сторону курсора). К тому же, астероиды теперь нельзя полностью уничтожить, так как на каждый уничтоженный появляется несколько новых. Это приводит к тому, что количество астероидов начинает очень быстро увеличиваться.

Однако мы хотели бы, чтобы осколки сохраняли направление начального астероида и появлялись на его месте. Для этого нам необходимо передать в конструктор объекта класса *Asteroid* все необходимые параметры: его тип, координаты и направление. Поскольку данные параметры не являются случайными (как в случае обычных астероидов), нам понадобится совершенно другая логика создания объекта. Всё это можно описать внутри конструктора через

набор условий, однако такой код будет перегружен и крайне сложен для чтения. Вместо этого мы можем создать 2 метода **имитирующих** конструктор. Каждый из них будет создавать необходимые атрибуты в зависимости от типа астероида (обычный или осколок). Затем мы вызовем их из основного конструктора в зависимости от типа астероида.

```
def init_rand_asteroid(self):     # Метод init_rand_asteroid класса
Asteroid (конструктор для случайных астероидов)
    """Метод инициализирует обычный астероид случайными типом и
координатам"""
    self.type, self.original_image, hitbox_shape =
random.choice(self.ast_variants)
    self.image = self.original_image
    self.w, self.h = hitbox_shape
    self.pos = random.choice([self._left_pos, self._top_pos,
                              self._right_pos, self._bottom_pos])()
    self.rect = self.image.get_rect(center=self.pos, width=self.w,
height=self.h)
    self.direction = pygame.mouse.get_pos() - self.pos
    self.speed = self.direction / 300
```

Метод `init_rand_asteroid` будет инициализировать обычный астероид. На самом деле мы просто скопировали текущий код конструктора в этот метод, так как логика создания обычных астероидов не изменится.

```
def init_asteroid_fragment(self, pos, speed, ast_type):  # Метод
init_asteroid_fragment класса Asteroid
    """Метод инициализирует осколок астероида, принимая на вход его
координаты (pos), скорость (speed) и тип (ast_type)"""
    # Отбираем только картинки с астероидами заданного типа
    ast_type_variants = list(filter(lambda x: x[0] == ast_type,
self.variants))
    # Берём случайную картинку астероида заданного типа
    self.type, self.original_image, hitbox_shape =
random.choice(ast_type_variants)
    self.pos = pos      # Координаты осколка задаём НЕ случайно
    self.w, self.h = hitbox_shape
    self.image = self.original_image
    self.rect = self.image.get_rect(center=self.pos, width=self.w,
height=self.h)
    self.speed = speed             # Задаём направление (скорость) НЕ
случайно
```

Метод `init_asteroid_fragment` будет инициализировать осколок астероида. На вход он будет принимать позицию (`pos`) и направление (`direction`) родительского астероида (т. е. астероида в который попала пуля и он развалился на осколки, которые мы сейчас инициализируем). Тип осколка (`ast_type`) мы будем задавать в методе `explode`.

`ast_type_variants` — здесь мы отбираем картинки астероидов имеющие заданный класс (`ast_type`) из атрибута класса `ast_variants` при помощи функции `filter`.

`self.pos = pos` — задаём координаты осколка равными координатам родительского астероида (так они будут появляться в том же месте).

`self.speed = speed` — задаём скорость осколка равным скорости родительского астероида (скорость хранит внутри себя направление, так что нам не нужно заботиться об этом отдельно).

```
def __init__(self, pos=None, speed=None, ast_type=None):    # <----
Конструктор класса Asteroid
    # Если координаты, скорость и тип астероида были переданы...
    if (pos is None) and (speed is not None) and (ast_type is not None):
        self.init_asteroid_fragment(pos, speed, ast_type)  #
...инициализировать осколок
    else:                                                  # Иначе
(координаты, скорость и тип НЕ заданы)...
        self.init_rand_asteroid()                          #
...инициализировать обычный астероид
```

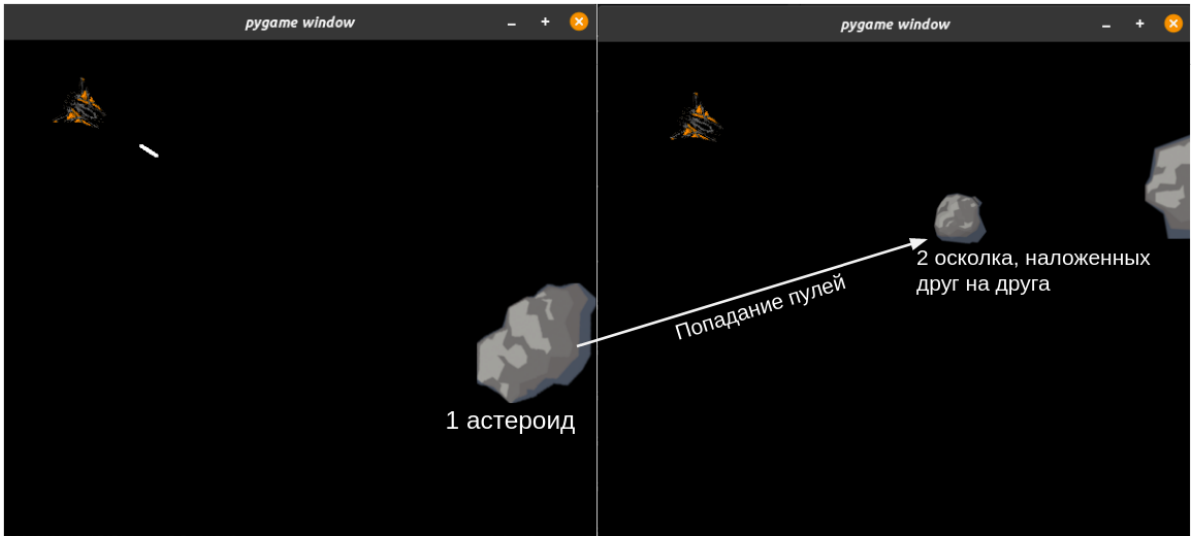
Имея эти два метода для инициализации астероидов мы полностью переписываем конструктор, а также добавляем ему возможность принимать в качестве аргументов координаты, скорость и тип астероида.

Теперь, когда конструктор полностью готов к работе с осколками и обычными астероидами, мы можем вернуться к методу `explode`.

```
def explode(self):    # Метод explode класса Asteroid
    fragments = []                  # Список с осколками астероидов
    if self.type == "large":        # Если астероид большой...
        for _ in range(2):
            # ...добавить 2 средних астероида в список
            fragments.append(Asteroid(self.pos.copy(), self.speed.copy(),
"medium"))
    if self.type == "medium":      # Если астероид средний...
        for _ in range(3):
            # ...добавить 3 маленьких астероида в список
            fragments.append(Asteroid(self.pos.copy(), self.speed.copy(),
"small"))
    if self.type == "small":       # Если астероид маленький...
        pass                        # ...не делать ничего
    return fragments
```

Здесь мы инициализируем осколки с координатами (`self.pos`), скоростью (`self.speed`) и типом на 1 меньше, чем родитель.

Казалось бы всё должно работать. Однако проблема состоит в том, что все осколки имеют одинаковую траекторию, они появляются в одном месте и все летят в одну сторону.



Однако мы хотим, чтобы осколки в целом сохраняли направление родителя, но случайно отклонялись на небольшой угол. Также мы хотим, чтобы осколки немного замедлялись (из-за столкновения с пулей). Если мы внесём данные изменения поведение астероидов будет станет очень правдоподобным.

Для этих целей мы добавим два метода, которые будут генерировать небольшое случайное отклонение для скорости и координат.

```
def speed_offset(self):             # Метод speed_offset класса Asteroid
    # Уменьшаем скорость в 2 раза и изменяем её составляющие по x и y на
небольшую величину
    offset = self.speed * 0.5 + np.random.uniform(-0.5, 0.5, 2)
    return offset

def pos_offset(self):               # Метод pos_offset класса Asteroid
    # Возвращаем немного смещённые координаты центра астероида
    offset = self.pos + np.random.uniform(-10, 10, 2)
    return offset
```

`self.speed \* 0.5` — уменьшение скорости в 2 раза. `np.random.uniform(-0.5, 0.5, 2)` — случайный сдвиг направления вектора скорости (так как 0.5 это довольно немного, направление осколков в целом будет совпадать с направлением родителя).

`np.random.uniform(-10, 10, 2)` — случайный сдвиг координат центра осколков относительно родительского цента, это предотвратит их появление в одной точке.

Теперь перепишем метод `explode` с учётом сдвигов.

```
def explode(self):    # Метод explode класса Asteroid
    fragments = []
    if self.type == "large":
        for _ in range(2):
            # Генерируем новые смещённые параметры для осколков
            pos, speed = self._pos_offset(), self._speed_offset()
            fragments.append(Asteroid(pos, speed, "medium"))    # <---
Передаём их в конструктор астероида
    if self.type == "medium":
        for _ in range(3):
            # Генерируем новые смещённые параметры для осколков
            pos, speed = self._pos_offset(), self._speed_offset()
            fragments.append(Asteroid(pos, speed, "small"))     # <---
Передаём их в конструктор астероида
```

```
        if self.type == "small":
            pass
        return fragments
```

Замечательно, всё работает как мы хотели! При желании мы можем варьировать количество осколков, образующихся при попадании пули.

Домашнее задание

Теперь у нас есть работающая игра. Это очень удобно, так как при внесении изменений мы сразу увидим как это отражается на игровом процессе. В этом домашнем задании вам нужно будет модифицировать имеющийся код. Вам нужно будет реализовать следующие механики:

Поворот астероидов вокруг своей оси — астероиды должны вращаться вокруг своей оси с постоянной случайной скоростью (от -2 до 2 градусов/кадр) (**Обязательно**).

Столкновение с астероидом начинает игру заново — на данный момент при столкновении с астероидом игра завершает свою работу, но мы хотим, чтобы вместо этого сразу запускалась новая игра (**Обязательно**). Вы также можете сделать экран с надписью "GAME OVER!", как в примере (**Опционально**).

Реалистичные столкновения — на данный момент у нас есть проблема того, что при столкновениях объектов их изображения могут находится довольно далеко друг от друга (см. предпоследнюю картинку). Мы хотели бы, чтобы при столкновениях изображения объектов также соприкасались (или хотя бы были очень близко к друг-другу). (**Опционально**). *Подсказка: возможно стоит как-то изменить свойства хитбоксов? (Изменение аргументов метода .get_rect)*

Рассмотрите картинку, чтобы понять как это должно выглядеть.