INF-1400 Object-oriented Programming

Assignment 3

Lars Karlsen (lka055)
Mellet Solbakk (mso036)

1.0 Introduction

In this assignment we will be creating a two player shooting game called mayhem. In the game you control a rocket, and your objective is to shoot down your opponent which is controlling the other rocket.
Some of the requirements for the assignment was to include two rocket ships, which can fly around, some obstacles, and common things like the rocket can crash, it has fuel and so on.

2.0 Technical background

2.1 Sprites

One of the requirements for this assignment was to use sprites for everything displayed on the game screen. To accomplish this, we will be using pygames build in sprite class that we will inherit from. The benefit of using this class is that pygame has some build it functions to group, update, draw and to check collisions between sprites.

2.2 Vector library

The vector library is a pre-code that was included with our assignment that gives us the ability to create vector objects. We will be using this to keep track of all our objects position and speed.

3.0 Design

3.1 Engine

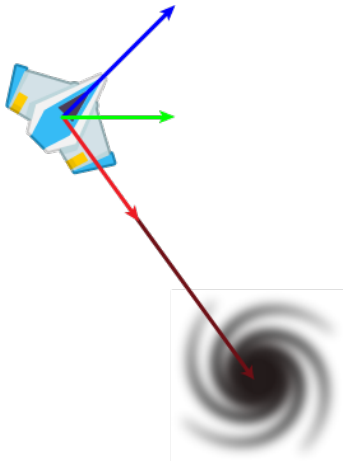
The engine is a class we use to keep track of all game assets and its instance variables. Its main purpose is to have a logic method that we run every frame of the game. The logic method will use all other methods of the engine needed to draw a frame.

The engine also has an event handler that keeps track of key presses and custom user event we created. For this game we created some custom user events for spawning asteroid, spawning ships when they get killed and to disable shields on players after a while.

We also have a method that will go through our ships, read their stats and display it as text on the HUD to show health, score and fuel levels.

3.1.1 Gravity

Our game is a top-down space shooter. So for our gravity we use a simplified version of Newton's law of universal gravity. By doing this objects are affected by gravity depending on their distant to big objects.



First we have to calculate the direction of the gravity vector, we also normalize it because we want it to be a unit vector that we later multiply our force with.

$$directionVector = [planetX - rocketX, planetY - rcoketY]$$

We can now calculate the strength of the vector by using newton's gravity equation. To do this we need the objects to have mass, so we gave our objects fictional masses that we can use in our calculations.

$$force = \frac{(massObject1 \cdot massObject2)}{distance^2}$$

Now that we have a direction and a force we multiply them and get our gravity vector that we can add to the object's speed. The result is the objects new speed. This will result in a gravity like effect.

We see objects flying close to the black hole getting sucked into it and objects with the right incidence angle and speed can orbit it.

3.1.2 Collision detection

Since we are using sprites in this assignment collision detection is trivial. We first check if two rects of sprites has collided and then check for collision between masks. Since the mask collision detection is more resource intensive we first check for rects. If they have collided, we use the mask since we know for sure it is a collision.

3.2 Sprite sheet

Instead of having separate images for every asset in the game we have one image that contains all sprites. This class will hold on to this image and has a method for us to get a specific image out of the bigger image. The method simply creates pygame surface object and we blit the part of the big image onto it.

3.3 Moving Object

Moving object is our base class for all moving objects on the screen, it will inherit from pygame.sprite.Sprite also since all our objects are sprites.

The trivial methods is has is to wrap objects around the screen if they exit it, to rotate a sprite with the given angle attribute and to resize a sprite with a given width and height.

It also has some default attributes; speed, position, sprite sheet, spawn, angle, uid and maxspeed.

3.2.1 Recalculate angle

Since our games gravity will modify the speed vector of our moving objects we have to make sure the angle attribute is in sync with the angle change the gravity will have.
We do this by checking the angle between our base vector (pointing upward) and the speed vector. After calculating this we update the angle attribute to it and the image and speed vector angle will be in sync.

$$base = [0, -1]$$
$$dot = (speedX + baseY) + (speedY \cdot baseY)$$
$$\det = (speedX \cdot baseY) - (speedY \cdot baseX)$$
$$angle = \tan^{-1} \frac{det}{dot}$$

3.4 The Rocket Ship

Our rocket class inherits from our Moving Object class. It has a lot of attributes that controls the state of the ship. Like engineOn, turnLeft/Right, dead, refueling, invisible, speedBreak, shield. These are all booleans that we switch on/off depending on what's happening to the ship.

3.4.1 Ship state

Our rocket can have many states. In our sprite sheet we have 4 sprites representing each state of the ship. Rocket on, rocket off, rocket on with shield and rocket off with shield. We have a separate method for the ship to check what state it is in and to update its sprite with the current state. Here we use the method we inherit from moving object class to update and rotate a sprite.

3.4.2 Rotation and handling angle change by gravity

To rotate the ship, we rotate the speed vector with a math formula.

$$x = speedX \cdot \cos(radians) - speedY \cdot \sin(radians)$$
$$y = speedY \cdot \sin(radians) + speedY \cdot \cos(radians)$$

Pygame has a build it functions to rotate images, but they are rotated around the top left corner of the image. To solve this problem, we simply stored the center position of the image before rotation, and then moved the new center position to the old center position. We then get the effect that the image is rotating around its center.

Our gravity algorithms will also be changing the speed vectors angle every frame. So here we have to recalculate the new angle so the image and speed vectors align.

3.4.3 Movement

The movement of the ship is trivial. We check if we are accelerating and increase the speed vector by multiplying it. If we are breaking we decrease it, making sure it doesn't go to low.

Then we just add the speed vector to the position vector. Only doing it if the magnitude of the speed vector is bigger than 1. Allowing us to stand still if the speed vector is very small.

3.4.5 Handling death

When the ship's health goes to 0 we despawn it by changing its position to the spawn position. We also switch out the sprite with an empty sprite so it can't collide with anything, since we are using mask collision detection.

After 1.5 seconds a user event is triggered to re-spawn it and we change its angle, speed and the sprite back to its default. We also make sure we can't shoot or use other movement mechanism when the ship is despawned.

3.5 Asteroids

Asteroid is a class that inherits from Moving Object. This class can create three different type of asteroids with different mass, size and life based on which rect argument it gets. If an asteroid with bigger mass collide with another asteroid with smaller mass, it will destroy the other. The collision logic is handled by the engine.

3.5.1 Spawning asteroids

To spawn the asteroids, we have set up a timer in the main game loop that runs a custom user event. Every time the event triggers we run a method that tries to spawn a asteroid unless the limit of asteroids on the screen is exceeded.

To make sure the asteroids look natural when spawning we modified the screen wrapping method that it inherits. It now wraps 100 pixels outside the screen. By doing this the asteroids are allowed to move a bit out of the screen and we also give them spawn points outside the screen so they don't appear out of nowhere.

3.6 Explosions

Explosion is a simple class that contains a list of images. When it is spawned it will loop through the images and it has a boolean attribute called kill that will be set to true when the animation is finished. The game engine will go through all explosions and remove them if this attribute is true.

3.7 Bullets

A bullet is a class that inherit from the Moving Object class. A bullet takes the ship's userid, position, speed and witch wing it should spawn on as arguments.

The user id we use to make sure bullets only hit the enemy and not yourself. We also use it to decide who gets score when a player is killed by a bullet.

To make sure the bullet is traveling in the same direction as the ship we normalize the speed we got as a parameter. This is the direction the bullet should travel. We then multiply it by a constant so the bullet always has the same speed no matter the speed of the ship that shot.



Our rockets ships will spawn 2 bullets when shooting, one coming out of each wing. To do this we have to calculate where the wing is after the ship has rotated. We do this by using a mathematical formula to calculate a points position after it has rotated by an angle around a center point.

$$x_1 = x_0 - centerx$$
$$y_1 = y_0 - centery$$
$$x_2 = x_1 \cdot \cos(radians) - y_1 \cdot \sin(radians)$$
$$y_2 = x_1 \cdot \sin(radians) + y_1 \cdot \cos(radians)$$
$$x = x_2 + centerx$$
$$y = y_2 + centery$$

3.8 Static object

A base class for all static object. Simply has a position and some methods to rotate and resize sprites.

3.9 Black Hole

The black hole is a object from the class called Planet. The image of this planet is supposed to be a black hole which is placed in the centre of the screen, and rotates around. It inherit from the Static Object class. The only thing it doesn't have in common with its superclass is that it has a mass that we use for calculating the gravitational effect.

3.10 Powerups

The powerup class inherit from Static Object, and this class creates three different types of powerups based on what powerup type you set it to be when created. There is a health, shield and missile powerup which all three has different icon and a different effect on the rocket which picks it up by colliding with it. This class is controlled by the eventhandler how often a powerup should spawn, and what type is random.

3.10.1 Health

The powerup which has a green health icon gives the rocket who collides with the powerup +100 health.

3.10.2 Shield

The shield powerup puts a defensive shield around the rocket who picks it up. This shield can parry bullets, destroy asteroids when collided with, and even destroy your opponent if he crashes with you.

3.10.3 Missile

The missile powerup is a very special powerup which replaces your normal bullets with heat seeking missiles which is an own class, that inherit from the bullet class. This missiles will get affected by the position of the other rocket, and seek to move towards it. Here we use very similar calculations as we did with the gravity, we just increase it to make the missile go faster. If hit by a missile, it will deal the normal damage that a bullet would do.

4.0 Implementation

This programs was written in Python 3.4.3 and used the Pygame library. We also got provided a library for creating and handling vectors.

5.0 Discussion

In the early stages of development, we noticed some frame dropping and noticed it was when we where shooting a lot. We figured out that this was because we where adding a new image to every instance of the bullet object. We solved this by using sprite sheets instead. We did not do this for the explosion, so there is still a bit of lag when there are a lot of explosions.

We where a bit unsure if our collision detecting might give some performance hit. Especially when we are doing a check like: *if rect_collision and mask_collision*. Here we want it to skip if the rect collision is false, but we are not sure if python checks the second statement as well even if the first is false.

6.0 Conclusion

In this assignment we created a game called mayhem with python and pygame. All of the basic requirements was implementation very early in the process, so that meant we had a lot of time including extra features we wanted for the game. With this assignment we learned much about the features of the sprite class, and some other useful stuff like using docstrings and testing with cProfiler.

5.0 Resources

5.1 Rotating point around another:
http://www.gamefromscratch.com/post/2012/11/24/GameDev-math-recipes-Rotating-one-point-around-another-point.aspx

5.2 Newton's Law of universal gravity:
https://en.wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation#Modern_form