spaceshooter Documentation

Release 11.10.2025

Martin Schwald

Jan 11, 2026

Modules

1 game

The game module initiates a rendering loop of three phases, as it is classical for video games:

1. Handle user input (i.e., check for keyboard and mouse events and delegate the input to the corresponding modules to process the reactions)
2. Calculate game logic (Calculate the spawning, movement, animation, collision and killing of sprites; handle damage and status effects, update graphical user interface (statusbar and menu))
3. Render the game's current state (Render the background, graphical user interface and all sprites on the screen in the correct order)

It is crucial to keep these three phases strictly separated. Failing to do so can quickly complicate the game's code, making it difficult to understand and develop further.

Pygame provides the necessary methods to measure the time dt passed (in milliseconds) between the rendering of two subsequent frames. This time is passed to all game objects to ensure the game runs at the same speed on every computer.

`class Game`

Initiates the game's modules, starts rendering loop, coordinates the game's interaction with user input, opens menus

`screen`

`player_name = ''`

`level`

`highscores`

`clock`

`run(`*mute: bool = False*`)`

Starts the main loop for the game.

`handle_user_input()`

handles keyboard and mouse events on each frame

`render()`

Blit stats, sprites (and optionally menu) onto the display in the correct order

2 level

The level module contains the design of the game's levels, along with methods to control starting new levels and checking the player's progress to determine when a level's goal is fulfilled and the next level should be loaded. It includes convenience methods to spawn aliens, making it as easy as possible to add new levels to the game.

The update method checks for sprite collisions; upon collision, it delegates the tasks of inflicting damage, spawning items, splitting asteroids, and merging blobs to the corresponding modules.

`class Level(`*number*`)`

Manage game levels, loading enemies, timed events, collisions, and player progress.

number: int - Current level number. goals: list[str] - Descriptions of objectives for each level. ship, crosshairs: Sprites for level objects bullets, asteroids, aliens, ufos, blobs, items, ship_bullets: Pygame sprite groups timer, asteroid_hail, alien_hail: Timers for level events. boundary_behaviour: Default behaviour for enemies in current level.

`number`

`goals = ['Welcome!', 'Destroy all asteroids!', 'Defeat all aliens!', 'Defeat the ufo!', 'Defeat the...`

`max_level = 5`

`boundary_behaviour = None`

`ship`

`crosshairs`

`ship_bullets`

`bullets`

`items`

`asteroids`

`aliens`

`ufos`

`blobs`

`timer`

`asteroid_hail`

`alien_hail`

`timers`

`blit(`*screen: pygame.Surface | None = None*`)`

Blit all level sprites onto the given screen.

`start_current()`

(Re)start current game level.

`start_next()`

Start the next game level.

`restart_game()`

Restart from starting game level.

`alien_spawn(`*alien: src.sprite.Alien*, ***kwargs*`)`

Spawn an alien at a given position and play corresponding sound.

`alien_random_entrance(`*alien: src.sprite.Alien*`)`

Triggers spawning of an alien at a random top position, aiming towards a random bottom position.

`encounter(`*template:* src.templates.AlienTemplate, *amount: int = 1*, *grid: tuple[int, int] | None = None*, *speed_factor: float = 1*, *dir: tuple[int, int] | None = None*, *energy: int | None = None*, *constraints: pygame.Rect | None = Display.screen*, *boundary_behaviour: str | None = None*`)`

Triggers spawning a given amount of aliens. Missing position or direction get set randomly.

`load_level(`*number: int*`)`

Load enemies and start action timers of the current game level.

`property progress: str`

String summarizing the player's progress to be rendered in the status bar.

`property goal_fulfilled: bool`

Bool indicating whether the current level goal is fulfilled

`property status:` *`src.settings.LEVEL_STATUS`*

String indicating why a game level or the entire game ended.

`play_status_sound()`

Play the appropriate sound when a level ends.

`update(`*dt: int*`)`

Update level status according to passed time dt.

`update_sprites(`*dt: int*`)`

Update the status of all level objects.

`update_crosshairs()`

The crosshairs follow the player's mouse position.

`collision_checks()`

Check for collisions of level sprites: inflict damage; kill, split or merge enemies; add points to the player's score; generate items.

`bullets_hit()`

Check if bullets hit enemies or the ship

`enemies_hit_ship()`

If enemies hit the ship, reflect with shield or inflict damage and kill the enemy.

`ship_collects_item()`

If the ship collects an item, trigger its effect.

`blobs_collide()`

Merge colliding blobs (if not too big) preserving total impuls and mass.

3 display

It was necessary to set a fixed resolution for developing the game: I used a 1600×900 window, which I subdivided into a 16×9 square grid for planning the levels and adjusting the sizes and velocities of the game's sprites.

The display module's task is to check the user's screen settings and set up a game surface on their screen. This surface is intended to be as large as possible while maintaining the same $16 : 9$ aspect ratio as the development resolution. If the user's screen has a different aspect ratio, the game surface will be centered with dark grey padding visible on its sides. In any case, the dimensions of the game surface and its square grid are saved as global variables for use by other modules.

`class Display(`*screen_size: tuple[int, int]*, *screen_grid: tuple[int, int]*`)`

Manage fullscreen rendering and grid layout for the game.

This class handles adapting the game surface to the player's screen resolution, keeping the desired aspect ratio, centering the screen, and computing a grid for consistent positioning of sprites.

display: pygame.Surface | None

The actual fullscreen display surface.

screen: pygame.Surface | None

The internal game surface (possibly smaller than *display* to preserve aspect ratio) with automatically set parameters screen_rect, screen_width, screen_height, screen_size. Gets subdivided into a rectangle of square grid cells with parameters grid: tuple[int, int] and grid_width. Gets padded for rendering on the screen with int parameters padding_w, padding_h

```
display: pygame.Surface | None = None

screen: pygame.Surface | None = None

screen_rect: pygame.Rect | None = None

screen_width: int = 0

screen_height: int = 0

screen_size: tuple[int, int] = (0, 0)

grid: tuple[int, int] = (0, 0)

grid_width: int = 0

padding_w: int = 0

padding_h: int = 0
```

`classmethod init(`*screen_size: tuple[int, int]*, *screen_grid: tuple[int, int]*`)` → pygame.Surface

Convenience method to initialize the display and ontain the internal game surface.

`classmethod update(`*padding_color: tuple*`)`

Blit internal game surface centered on the fullscreen display

`classmethod grid_rect(`*x_min: int = 0*, *y_min: int = 0*, *width: int = grid[0]*, *height: int = grid[1]*`)` → pygame.Rect

Return a rectangle aligned to the initialized display grid.

4 settings

This module is a structured collection of all in-game parameters.

`class COLOR`

Bases: `tuple`

Color names

`WHITE = (255, 255, 255)`

`BLUE = (0, 0, 200)`

`YELLOW = (255, 255, 0)`

`LIGHT_GREY = (200, 200, 255)`

`DARK_GREY = (50, 50, 50)`

`GREY = (100, 100, 100)`

`RED = (180, 0, 0)`

`GREEN = (100, 255, 100)`

`BLACK = (0, 0, 0)`

`class KEY`

Key names and settings

`EXIT`

`UP`

`DOWN`

`LEFT`

`RIGHT`

`SHOOT`

`SHIELD`

`START`

`BACK`

`class SCREEN`

Default screen settings for the game's development. (display.py handles rescaling subsequent sizes to user's display settings)

`PADDING_COLOR = (50, 50, 50)`

`BG_COLOR = (0, 0, 0)`

`SIZE`

`GRID = (16, 9)`

`GRID_WIDTH = 100`

`class GAME_MODE`

Possible modes of the game to respond to user's input

`GAME = 'game'`

`MENU = 'menu'`

`ENTER_NAME = 'enter name'`

`EXIT = 'exit'`

`class LEVEL_STATUS`

Possible status properties of the running game level to coordinate the player's progress

`START = 'start'`

`RUNNING = 'running'`

`GAME_OVER = 'game_over'`

`LEVEL_SOLVED = 'level_solved'`

`GAME_WON = 'game_won'`

`class SHIP`

Game and ship settings upon starting the game or depending on the current rank

`SCORE = 0`

`LIVES = 3`

`GAME_LEVEL = 1`

`SHIELD_STARTING_TIMER = 3`

`MAX_SHIELD_DURATION = 15`

`MAX_BULLETS = 3`

`STARTING_MISSILES = 1`

`RANK = 1`

`SPEED`

`ENERGY`

`WIDTH`

`class SHIP_STATUS`

Possible status properties of the ship implemented in ship.py

`NORMAL = 'normal'`

`MAGNETIC = 'magnetic'`

`SHIELD = 'shield'`

`INVERSE_CONTROLS = 'inverse_controls'`

`class PATH`

`BASE`

`DATA`

PREPROCESSED

MEDIA

SOUNDS

FONTS

IMAGES

ITEM

ALIEN

BULLET

SHIP

STATUSBAR

class FONT

Fonts types and sizes in the game

STATS

MENU

TEXT

MENU_SIZE = 30

TEXT_SIZE = 30

class MENU

Menu settings

BOUNDARY_SIZE = 20

TITLE_DISTANCE = 30

LINE_DISTANCE = 12

class HIGHSCORES

High scores settings

DEFAULT = [['Markus', 1000], ['Tobi', 900], ['Nadine', 800], ['Marc', 600], ['Katharina', 400]]

MAX_NUMBER = 5

MAX_NAME_LENGTH = 10

class ANIMATION_TYPE

Animation types of sprites implemented in sprite.py

LOOP = 'loop'

ONCE = 'once'

VANISH = 'vanish'

PINGPONG = 'pingpong'

RANDOM = 'random'

MANUAL = 'manual'

5 templates

This is a collection of all parameters relevant for comfortable comfortable creation of new templates for aliens, bullets, and items.

`class SpriteTemplate`

    name: str

    speed: float

    width: int

    animation_type: str | None = None

    fps: int | None = None

    colorkey: tuple = (0, 0, 0)

    acc: float | None = None

`class AlienTemplate`

Capture the defining properties of an enemy species

    name: str

    speed: float

    energy: int

    points: int

    width: int

    animation_type: str | None = None

    fps: int | None = None

    colorkey: tuple = (0, 0, 0)

    pieces: int | None = None

    acc: float | None = None

    alarm_min: int | None = None

    alarm_max: int | None = None

`class ALIEN`

Default settings of enemiy species in the game

    BIG_ASTEROID

    SMALL_ASTEROID

    PURPLE

    UFO

    BLOB

`class BulletTemplate`

Capture the defining properties of a bullet type `name: str`

`width: int`

`damage: int`

`owner: str`

`speed: float`

`animation_type: str | None = None`

`animation_time: float | None = None`

`class BULLET`

Default settings of bullet types in the game

`BULLET1`

`BULLET2`

`BULLET3`

`MISSILE`

`GREEN`

`BLUBBER`

`class ItemTemplate`

Capture the defining properties of an item type

`name: str`

`size: int = 50`

`speed: int = 0.3`

`effect: float | None = None`

`duration: int | None = None`

`class ITEM`

Default settings of item types in the game

`PROBABILITY = 0.3`

`SIZE_PLUS`

`SIZE_MINUS`

`SCORE_BUFF`

`BULLETS_BUFF`

`HP_PLUS`

`INVERT_CONTROLS`

`LIFE_PLUS`

`LIFE_MINUS`

`MAGNET`

MISSILE

SHIELD

SHIP_BUFF

SPEED_BUFF

SPEED_NERF

LIST

6 image

It is convenient to create a class that handles loading, preprocessing, and further editing of images used for game object sprites. In Pygame, the *Surface* class handles image files, but since it is coded in C, it's apparently not possible to inherit properties directly from it. Therefore, I created a wrapper class, *Image*, whose objects contain a Pygame surface, the file link from which the image was originally loaded, and a Pygame *Mask* object for convenience. In Pygame, masks are bitmaps on surfaces that can be calculated from the surface's background color (frequently called *colorkey* in this module) to enable pixel-perfect collision detection between sprites. Including them in the Image objects is convenient, as it prevents calculating them more often than necessary.

The most complex part of the Image class is its lazy image loader. On one hand, this loader ensures that each original image is loaded and preprocessed only once. (This significantly improves the game's performance. Also, adding new pictures for further development is more comfortable as the preprocessing step is automated, and object sizes can be easily adjusted later.) On the other hand, it loads each image directly in the optimal resolution, adapted to the user's display settings determined in the *display* module.

It would certainly be easier to fix a single scaling for the images (for example, to rescale them all to the 1600×900 development resolution) and then rescale them further if the user's display settings differ. However, I favored the idea that if a user has a much larger screen than mine, the images would also be loaded in a better resolution without quality loss from upscaling. (Specifically, the original asteroid images are in a very high resolution that would be lost by adapting them to a single, arbitrarily fixed development resolution.) To generalize these ideas for animated images, the dataclass *GraphicData* consists either of a single image loaded from a file path or a list of images (called *frames*) loaded from a folder path, together with their mask(s). For animated images, one can provide the description of its animation type (as a string) and its animation speed, either by its *fps* (frames per second), the duration of each frame, or the total duration of its animation. The missing attributes are automatically calculated upon initializing a GraphicData object using the formula $fps = \#frames/animationtime$.

`class Image(`*surface: pygame.Surface*, *mask: pygame.Mask*, *colorkey: tuple = None*, *path: str | None = None*`)`

Manage lazy loading and transforming images and masks to be used for sprites.

`surface`

`mask`

`path = None`

`property rect: pygame.Rect`

`property w: int`

`property h: int`

`scale_by(`*factor: float*`)` $\rightarrow$ *Image*

Rescale image and its mask by a given factor.

`rescale(`*scaling_width: float = None*, *scaling_height: float = None*, *scaling_factor: float = None*`)`

Rescale image either to a given width or height with respect to the default screen resolution, or by a specified factor.

`cache`

`classmethod load(`*path: str*, *colorkey=COLOR.BLACK*, *scaling_width=None*, *scaling_height=None*, *scaling_factor=None*`)` $\rightarrow$ *Image*

Load image with given path lazily and preprocess the image upon first loading. Background of color 'colorkey' gets cropped and transparent. Then rescale either to desired width, height (wrt default screen resolution) or by a given scaling factor. Cache the result and save it to the disk for quick access next time.

`classmethod preprocess(`*path: str*, *colorkey: tuple = COLOR.BLACK*, *scaling_width: float | None = None*, *scaling_height: float | None = None*, *scaling_factor: float | None = None*`)` → *Image*

`classmethod crop_boundary(`*surface: pygame.Surface*, *colorkey: tuple = COLOR.BLACK*`)` → pygame.Surface

Remove boundary of a given color 'colorkey' from a pygame surface.

`reflected_cache`

`classmethod reflect(`*image:* Image, *flip_x: bool*, *flip_y: bool*`)` → *Image*

Reflect an image (including it's mask) along the specified axes.

`blit(`*screen: pygame.Surface*`)`

Blit an image's surface onto the screen.

`class GraphicData`

Capture graphical data allowing for animations. Provide exactly one of path, image or frames. For animated sprites, provide exactly one of fps, animation_time r frame_duration_ms Missing attributes get calculated upon initialization.

`path: str | None = None`

`image: Image | None = None`

`frames: list[Image] | None = None`

`colorkey: tuple = (0, 0, 0)`

`scaling_width: int | None = None`

`scaling_height: int | None = None`

`scaling_factor: float | None = None`

`animation_type: src.settings.ANIMATION_TYPE | None = None`

`fps: int | None = None`

`animation_time: float | None = None`

`frame_duration_ms: int | None = None`

`starting_frame: int = 0`

`reflect(`*flip_x: bool*, *flip_y: bool*`)`

Reflect all images contained in a GraphicData object along specified axes.

7 sprite

Pygame's *Sprite* class is minimally designed to contain image data for in-game objects, along with a Pygame *Rectangle* object as an attribute—a rectangle on the screen with integer coordinates. It includes implemented methods to draw sprites onto their given rectangle, detect collisions (either by checking their entire rectangles or by pixel-perfect collision calculated from their masks and rectangles), and organize sprites into Pygame *SpriteGroup*s, where they can be easily added or removed.

My implemented *Sprite* class inherits these methods, and its objects consist of a *GraphicData* object (see the *image* module), moving data comprising three $2d$ vectors pos, vel and acc describing the object's position, velocity, and acceleration on the screen, as well as information about the area where the object can move and its behavior upon hitting that area's boundary. The vectors are implemented as objects of the *Vector* class from the *physics* module; it would also have been possible to use the *Vector2* class from Pygame. I deliberately avoided using *numpy*, which apparently only offers better performance for calculations involving large arrays. For calculating sprite animations, *ActionTimer*s from the *timer* module are utilized.

The most important game objects (*aliens*, the player's *ship*, *bullets*, *items*) are all realized through classes that inherit from *Sprite*, incorporating more specific attributes and methods. Their initialization functions are each designed so that the full sprite data doesn't need to be provided every time a new object is created. Instead, type-specific data is captured in *AlienData*, *BulletData*, and *ItemData* objects defined in the *settings* module, while other attributes like velocity and boundary behavior are often set automatically via default values or externally by the *level* module. Consequently, the parent class *Sprite* needs to be somewhat complex, while the derived classes *Alien*, *Ship*, *Bullet*, and *Item* become quite convenient to work with.

For further convenience, sprites can also be initialized without a position vector. They will then not be rendered or updated by the game's logic until a position is provided by the *Sprite.spawn* method. This allows for easily spawning multiple instances of an object at various points, as well as implementing other interesting features, such as aliens temporarily becoming invisible while still not counting as defeated.

`class BOUNDARY`

Implemented behaviours of sprites when hitting the boundary of their area of movement

`CLAMP = 'clamp'`

`REFLECT = 'reflect'`

`VANISH = 'vanish'`

`WRAP = 'wrap'`

`class Sprite(`*graphic: src.utils.GraphicData*, *pos: src.utils.Vector | None = None*, *vel: src.utils.Vector = Vector(0, 0)*, *acc: src.utils.Vector = Vector(0, 0)*, *constraints: pygame.Rect | None = None*, *boundary_behaviour: str | None = None*`)`

Bases: `pygame.sprite.Sprite`

Manage movement, boundary collision and animation of ingame objects. graphic: GraphicData, for visual representation of the sprite pos, vel, acc: position, velocity and acceleration vectors (optional) - vel and acc are set to (0, 0) by default. - If no pos is provided, Sprite can be placed later using spawn(). constraints, boundary_behaviour: pygame.Rect, BOUNDARY (optional) Movement area of the sprite and its interaction with its boundary. Implemented boundary behaviours: - None - no boundary restriction / interaction

`graphic`

`animation_timer`

`rect`

`activated = False`

`spawn`(*pos: src.utils.Vector | None = None*, *center: src.utils.Vector | None = None*, *grid: tuple[int, int] | None = None*)

Spawn and activate sprite at a specified location. Exactly one of pos, center or grid must be provided.

`property image: src.utils.Image`

`property surface: pygame.Surface`

`property mask: pygame.Mask`

`property w: int`

`property h: int`

`property diag: src.utils.Vector`

`property center: src.utils.Vector`

`property midbottom: src.utils.Vector`

`property speed: int`

`set_image`(*image: src.utils.Image*)

initializes an image preserving pos of the sprite

`change_image`(*image: src.utils.Image*)

changes the image preserving the center of the sprite

`scale_image_by`(*factor: float*)

rescales the image preserving the center of the sprite

`update_rect_pos()`

`update_rect_size()`

`move_to`(*pos: src.utils.Vector*)

Move the sprite respecting its boundary behaviour.

`update`(*dt: int*)

Calculate the state of the sprite after dt passed ms

`update_vel`(*dt: int*)

`update_pos`(*dt: int*)

`update_frame`(*dt: int*)

Update the Sprite’s image to a new animation frame if needed.

`next_frame()`

Determine the next frame in the animation depending on provided animation type.

`reflect`(*flip_x: bool*, *flip_y: bool*)

Reflects direction of movement and all graphical data along given axes.

`blit`(*screen: pygame.Surface*)

Blit the current state of the sürite onto the screen.

`property ball: src.utils.Ball`

Aproximate the sprite with a 3d ball of the same width.

8 timer

Actiontimers are used in the game for certain events that need to be timed only once (like the duration of an item's effect, one-time animations, or the level timer in the game's fifth level) or with a cyclic repetition (like animation loops, enemy spawning, or enemy actions). Cyclic actions can also incorporate some randomness (used for enemy spawning and actions) by only specifying a lower and an upper bound for the duration of each repetition.

`class Timer`

Simple timer with pause functionality. Measures time in ms.

`reset()`

`pause(`*duration: int | None = None*`)`

Pause timer for a specific duration or indefinitely

`resume()`

Resume measuring time immediately

`update(`*dt: int*`)`

Updates measured time if timer is paused

`class ActionTimer(`*alarm_min: int | None = None*, *alarm_max: int | None = None*, *cyclic: bool = True*`)`

Bases: *Timer*

Timer for single or cyclic actions and animation. Alarm can be randomized.

`set_alarm(`*alarm_min: int | None = None*, *alarm_max: int | None = None*, *cyclic: bool = True*`)`

Change the alarm settings of the timer

`check_alarm()` → bool

Return True (only once) when timer reaches its alarm time. Set a new alarm automatically if cyclic = True was provided.

`property remaining_time: int`

Time that is left until the alarm gets triggered. Can't be negative.

9 alien

All enemies in the game are implemented as objects of an *Alien* class. It includes methods to control enemy actions, inflict damage, and kill them. Enemies can also split or merge (used for large asteroids or blobs, respectively), invoking the realistic physics formulas for elastic or inelastic collisions from the physics module. (To this end, the enemies are modeled as 3D balls.) The blob is the only enemy whose image needs to be updated manually upon changes to its health (its width rescales with its energy such that, modeling blobs as 3D balls, the total blob mass is preserved upon splitting or merging).

`class Alien`(*template:* src.templates.AlienTemplate, *level: src.core.Level*, *energy: int | None = None*, *speed: float | None = None*, *direction: src.utils.Vector | None = None*, *pos: src.utils.Vector | None = None*, *vel: src.utils.Vector | None = None*, *acc: src.utils.Vector = Vector(0, 0)*, *constraints: pygame.Rect | None = None*, *boundary_behaviour: str | None = BOUNDARY.REFLECT*)

Bases: *`src.sprite.sprite.Sprite`*

Manage sprites, spawning and actions of enemies.

`template`

`level`

`energy`

`action_timer`

`update_blob_image()`

A blob's image and size depend on its energy.

`spawn(`***kwargs*`)`

Place an initiated alien and play it's spawning sound

`play_spawing_sound()`

`property mass: int | None`

Mass of an enemy, relevant for collisions between asteroids and blobs.

`update(`*dt: int*`)`

Calculate the state of the alien after dt passed ms

`do_action()`

Triggers an alien's specific action.

`shoot(`*bullet_template:* src.templates.BulletTemplate, *size: int | None = None*`)`

`throw_alien(`*alien_template:* src.templates.AlienTemplate *= ALIEN.PURPLE*`)`

`get_damage(`*damage: int*`)`

`split(`*piece_template:* src.templates.AlienTemplate, *amount: int*`)` → list[*Alien*]

Splits an Alien preserving the total impuls. Used for asteroids and blobs.

`classmethod merge(`*blob1:* Alien, *blob2:* Alien`)`

Merge two blobs. Can also be used for other aliens if their masses are implemented.

`kill()`

Removes an enemy, trigger splitting for asteroids and blobs.

`hard_kill()`

Remove an enemy without triggering further splitting.

`reflect()`

Reflecting enemies with shield sound effects.

10 ship

The Ship class manages controlling the player's ship movement and shield, shooting bullets, and status effects upon collecting items. While some status effects are permanent until the player loses a life, some have a fixed duration, and some also require changing the ship's sprite image. It ensures that the bullet's fire points (i.e., the points from which they are shot from the ship) scale correctly with changes in the ship's size.

`class Ship(`*level: src.core.Level*, *lives: int = SHIP.LIVES*, *rank: int = SHIP.RANK*`)`

Bases: *src.sprite.sprite.Sprite*

Manage the ship's position, status properties and item effects.

`level`

`lives = 3`

`rank = 1`

`score = 0`

`energy = 15`

`shield_timer`

`controls_timer`

`score_buff_timer`

`size_change_timer`

`speed_change_timer`

`timers`

`start_new_game(`*lives=SHIP.LIVES*, *rank=SHIP.RANK*`)`

Reset the ship to it's default starting settings.

`reset_pos()`

Ship starts each level at the midbottom of the screen.

`reset_stats(`*lives=SHIP.LIVES*, *rank=SHIP.RANK*`)`

The ship starts the game with given stats.

`set_rank(`*rank*`)`

Changes the rank and updates dependend properties of the ship.

`property max_energy: int`

The maximal energy of the ship depends on it's current rank

`property shield_time: int`

Currently remaining shield time.

`gain_rank()`

`lose_rank()`

`lose_life()`

`get_damage(`*damage: int*`)`

`property default_fire_points: list[tuple]`

Fire points on the original ship sprites.

`property default_width: int`

`property default_height: int`

`property fire_points: list[src.utils.Vector]`

Rescale where the ship shoots bullets respecting size changes of the ship.

`property bullet_sizes: list[int]`

`shoot_bullets()`

Shoot bullets if there aren't too many ship bullets on the screen yet.

`control(`*keys*`)`

Control the ship's direction and shield according to the current state of the keyboard.

`update_graphic()`

Upon changes of status or size, the ship's graphic must be updated.

`collect_item(`*item:* src.sprite.item.Item`)`

Trigger item effects and sound. Timers are set for temporary effects.

`activate_shield()`

Activate shield if the player has remaining shield time left.

`deactivate_shield()`

Method used to deactivate the shield when player releases the shield key or running out of shield time.

`shoot_missile(`*pos: src.utils.Vector*`)`

Shoot missile to the given position on the screen.

`get_points(`*points: int*`)`

`reset_item_effects()`

`update(`*dt: int*`)`

Update the ship's position, shield and item timers after dt passed ms

11 bullet

*Bullet*s store their owner as an attribute. When reflected by the player's shield, an enemy's bullet becomes the player's bullet, allowing it to subsequently hit enemies. The blob's bullets (called blubber) rescale in size depending on the blob's dimensions.

`class Bullet(`*template: src.templates.BulletTemplate, speed: float | None = None, pos: src.utils.Vector | None = None, vel: src.utils.Vector | None = None, acc: src.utils.Vector = Vector(0, 0), owner: str | None = None, damage: int | None = None, size: int | None = None, constraints: pygame.Rect | None = None, boundary_behaviour: str | None = BOUNDARY.VANISH*`)`

Bases: *`src.sprite.sprite.Sprite`*

Manage creation and properties of the player's and enemies' bullets.

`template`

`owner`

`damage`

`classmethod from_size(`*size: int, **kwargs*`)` → *Bullet*

Creates a ship bullet of a given size 1, 2 or 3

`play_firing_sound()`

`reflect()`

Reflecting bullets with shield sound effects.

12 item

*Item*s are simply sprites with a fixed falling rate. When the player has collected the *magnet*, items are attracted by the ship horizontally. I intentionally did not implement a vertical attraction (which would look more realistic) to make it easier for the player to dodge unwanted bad items.

`class Item(`*template: src.templates.ItemTemplate*, *level: src.core.Level*, *pos: src.utils.Vector | None = None*, *vel: src.utils.Vector | None = None*, *acc: src.utils.Vector = Vector(0, 0)*, *constraints: pygame.Rect | None = None*, *boundary_behaviour: str | None = BOUNDARY.VANISH*`)`

Bases: *`src.sprite.sprite.Sprite`*

Manage sprites, spawning and properties of items.

`template`

`level`

`duration_ms`

`play_collecting_sound()`

`update(`*dt: int*`)`

Calculate the state of the item after dt passed ms. If the magnet effect is active, item's get horizontally accelerated towards the ship.

13 physics

When two point masses m_1 and m_2 with velocity vectors v_1, v_2 collide, their total momentum

$$P = m_1 v_1 + m_2 v_2$$

is preserved, and there are classically two extreme cases for the outcome:

1. *Elastic collision*: We assume that the two masses bounce off each other with new velocity vectors v_1' and v_2' without losing any energy due to friction or deformation. Then the total kinetic energy

   $$E = \frac{1}{2}m_1|v_1|^2 + \frac{1}{2}m_2|v_2|^2 = \frac{1}{2}m_1|v_1'|^2 + \frac{1}{2}m_2|v_2'|^2$$

   of the system must be preserved, and thus

   $$m_1(|v_1|^2 - |v_1'|^2) = m_2(|v_2'|^2 - |v_2|^2).$$

   Preservation of the total momentum implies

   $$m_1(v_1 - v_1') = m_2(v_2' - v_2).$$

   In one dimension, and given that $v_1 \neq v_2$ (otherwise the two masses move parallel, meaning there is no collision), these two equations can be resolved to yield the classical formulas

   $$\begin{aligned} v_1' &= \frac{m_1 - m_2}{m_1 + m_2} v_1 + \frac{2m_2}{m_1 + m_2} \\ v_2' &= \frac{2m_1}{m_1 + m_2} v_1 + \frac{m_2 - m_1}{m_1 + m_2}. \end{aligned}$$

   In two dimensions, the direction of the collision plays a role. Let n be a normal vector at the point of collision. (In particular, for two colliding 2D balls at positions p_1 and p_2, we can simply take $n = \frac{p_1 - p_2}{|p_1 - p_2|}$, which is a normal vector to both balls at the point of collision.) Then the formula for the 1D elastic collision can be applied in the normal direction n, as no collision occurs in the parallel direction.

2. *Totally inelastic collision*: The two masses merge together into a single mass $M = m_1 + m_2$ centered at the center of mass $\dfrac{m_1 + m_2}{m_1 v_1 + m_2 v_2} = \dfrac{P}{M}$ and with some new velocity vector v. Preservation of the total momentum implies $P = Mv$, and hence $v = \dfrac{P}{M} = \dfrac{m_1 v_1 + m_2 v_2}{m_1 + m_2}$.

`class Vector(`*x: int | float*, *y: int | float*`)`

Simple class for 2d vector calculus, could also be replaced by pygames Vector2 class.

`x`

`y`

`change_direction(`*other:* Vector`)` → *Vector*

Changes direction of a vector while preserving its norm

`change_norm(`*n: int | float*`)` → *Vector*

Changes norm of a vector while preserving its direction

`clamp(`*min_v:* Vector, *max_v:* Vector`)` → *Vector*

Clamps x and y coordinates of vector to minimal or maximal possible value when outside the given ranges of x and y values

`randomize_direction(`*phi_min: float = 0*, *phi_max: float = 2 * pi*`)` → *Vector*

Changes direction of a vector randomly while preserving its norm `property norm: int | float`

Euclidean norm of a vector

`property norm2: int | float`

Square of Euclidean norm; avoids computing sqrt function for faster computations

`property normalize: Vector`

Changes norm of a vector to 1 while preserving its direction

`static random_direction(phi_min: float = 0, phi_max: float = 2 * pi) → Vector`

Returns a random unit vector

`turn_by_angle(phi: float) → Vector`

Turns a vector by a given angle (in radians), counterclockwise

`class Ball(pos: Vector, vel: Vector, r: float)`

Simple class for computations of moving 3d balls

`elastic_collision(v1: Vector, v2: Vector, m1: float, m2: float, n: Vector) → tuple[Vector, Vector]`

Calculate new velocity vectors of two masses m1, m2 after they collide elastically in a given normal direction n

`inelastic_collision(p1: Vector, p2: Vector, v1: Vector, v2: Vector, m1: float, m2: float) → tuple[Vector, Vector]`

return center of gravity of two masses m1, m2 and their velocity after they merge by a completely inelastic collision

`ball_collision_data(ball1: Ball, ball2: Ball, m1: float, m2: float) → tuple[float | None, Vector, Vector]`

if two balls collided elastically in the past, return the (negative) time of their collision and their new velocity vectors afterwards

14 text

`class Layout`

Layout utilities to align pygame surfaces and render texts

`classmethod pad_surface(`*surface: pygame.Surface*, *padding_size: int*, *padding_color: tuple | None = None*, *vertical_padding: bool = True*, *horizontal_padding: bool = True*`)` → pygame.Surface

Add padding of given color and size around a Pygame surface. vertical_padding, horizontal_padding : bool - indicate if padding is desired from all sides

`classmethod align_surfaces(`*surfaces: pygame.Surface*, *orientation: str*, *alignment: str = 'center'*, *rescale_to_surface: pygame.Surface | None = None*, *spacing: int = 0*, *padding_size: int = 0*, *padding_color: tuple | None = None*`)` → pygame.Surface

Align list of Pygame surfaces horizontally or vertically. Padding and spacing can be added. orientation: 'horizontal' or 'vertical'; alignment : str alignment: describes alignment along the perpendicular axis ('center', 'left' or 'right' for vertical alignment; 'center', 'top' or 'bottom' for horizontal). rescale_to_surface: If provided, rescale surfaces to match its height or width.

`classmethod render_line(`*line: str | list*, *font: pygame.Font*, *text_color: tuple*, *bg_color: tuple | None = None*`)` → pygame.Surface

Render a single string of text or a list of strings and surfaces as a pygame.Surface.

`class Text(`*lines: list*, *font: pygame.Font*, *text_color: tuple*, *bg_color: tuple | None = None*`)`

Can render a list of strings as a text with specified alignment and color of text and background

`lines`

`font`

`text_color`

`bg_color = None`

`rendered_lines`

`max_line_length`

`title_length`

`render(`*alignment: str*, *line_distance: int*, *padding_size: int = 0*, *padding_color: tuple | None = None*, *center_title: bool = True*, *title_distance: int | None = None*`)` → pygame.Surface

Composes rendered lines to a text with desired alignment 'left', 'right' or 'center', title formating, padding

15 menu

`class Menu(`*header_surface: pygame.Surface*, *options: list[str]*, *current_selection: int = 0*`)`

Manage creation and navigation of menus consisting of a header surface and a list of options. Header surface can be rendered from a given text.

`classmethod init_settings()`

Initiate menu formating scaled according to the user's display settings

`header_surface`

`inactive_options`

`highlighted_options`

`options`

`current_selection = 0`

`rendered_menu`

`w`

`h`

`highlight()`

Used for highlighting the selected option when rendering the menu.

`render()` → pygame.Surface

Renders the the menu as a pygame.Surface

`blit(`*screen: pygame.Surface | None = None*`)`

Blits the menu with the current selection highlighted onto the screen

`move_selection(`*event_key*`)`

Navigate through the menus otion according to user's input

`classmethod create(`*message: list[str]*, *options: list[str]*, *current_selection: int = 0*, *highscores: src.core.Highscores | None = None*`)` → *Menu*

Creates a menu from a given title message and options, allows for showing high sores in between

`classmethod create_main_menu(`*game: src.core.Game*`)` → *Menu*

Return the correct Main / Pause menu in each game situation.

`classmethod create_level_menu(`*level: src.core.Level*`)` → *Menu*

Return an info menu when a level ends according to the given level status.

`classmethod create_enter_name_menu(`*game: src.core.Game*`)` → *Menu*

Create the menu for entering the player's name into the high score list.

`handle_input(`*game: src.core.Game*, *event*`)`

Handles entering the name into the high score list upon user's input

`classmethod init_info_menus()`

Informational Menus in the game get rendered only once

`classmethod choose_current_selection(`*game: src.core.Game*`)`

Triggers the right game effects or opens new Menu when player chooses an option of a menu.

16 statusbar

`class Statusbar`

Format and render a two lined status bar with the relevant game stats.

`initialized = False`

`classmethod init(`*level: src.core.Level*`)`

Load images and initialize fonts once.

`classmethod blit(`*level: src.core.Level*, *screen: pygame.Surface | None = None*, *rescaling: bool = True*`)`

Blits the rendered status bar on the top of the screen. If rescaling is True, the status bar will be stretched to the user's screen width

17 highscores

`class Highscores`

Load, render, update and save highscores

`allowed_chars = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'`

`property players: list[str]`

`property scores: list[str]`

`load_default_highscores()`

`fill_list_with_zeros()`

When the default high score list in the settings is to short, fill the table with zeros.

`save()`

`highscore_rank(`*score: int*`)` → int | None

For a given score, return its rank in the high score table. Return None if the score is too low.

`insert_score(`*name: str*, *score: int*, *rank: int*`)`

`update_name(`*name: str*, *rank: int*`)`

18 sound

The class *Sound* was merely supposed to be a simple sound library. However, I noticed that when working in a docker container on a Windows computer, it is non-trivial to give the game access to the sound hardware of the computer from inside of the container. Hence I introduced a way to mute the game that not only sets the volume to zero, but which prevents the initialization of the pygame.mixer sound module in the first place. When the sounds are supposed to get initialized in muted mode, they become initialized as objects of the a mock class *MuteSound*, which is coded to not do anything when called via the usual play, stop or volume methods of pygame.mixer.Sound. The upshot is that for implementing this functionality, I did not need to modify the game's core code anymore.

`class MuteSound(`**args*, ***kwargs*`)`

A mock class imitating pygame.mixer.Sound without doing anything.

`play(`**args*, ***kwargs*`)`

`set_volume(`**args*, ***kwargs*`)`

`stop(`**args*, ***kwargs*`)`

`class Sound`

Sound library for convenient access from other modules

`SOUND_PATHS = []`

`classmethod init(`*mute: bool = False*`)`

Python Module Index