A brief introdution to SpaceBaseRobots

What is SpaceBaseRobots?

SpaceBaseRobots is a blog entry based project, that shows how to use the PyQt5 library to implement a small robot game in python. Its main function is to provide a simple, yet powerful **game engine** for a 2D topdown robot shooter **in SPACE**!
In addition, the SpaceBaseRobots package contains an **example game implementation**, as well as various **different robot AIs**. But don't worry, you can **take control of robots** as human players, too and **fight** the AI or each other.
For interested developers, we support an **easy-to-extend** project structure as well as provide some **suggestions for additional features**.

Base. Play. Extend.

This file will guide you thrugh the three main parts of the project, for additional information of its project history, you can look up our blog on GitHub Pages.
In the *BASE* part, we explain the game engine features that come with SpaceBaseRobots.
In the *PLAY* part, we show you how to use the config file to play our example game implementation. And in the *EXTEND* part, we try to inspire the developers of you with some nice extension ideas and show you the easy extension interfaces of our project structure.

Requirements and starting information

Since the project is a multi-week-blog, the project history is also rcorded on GitHub.
The final project state is in the marked project folder **day9_Finalization**.
ANY OTHER FOLDER CONTAINS A HISTORIC VERSION.
This project is based on features of python 3.7.X, so make sure to have a proper version of python installed.
Besides PyQt5, the project requires other 3rd party libraries, which are listed in **requirements.txt** file.
Using pythons package manager pip, you can make sure, these libraries are correctly installed:
On windows, execute the **start.bat**, to install the modules and start the game.
If this doesn't work for some reason, you can start the game by executing server.py with python 3.
On POSIX compliant operating Systems, you may use the following commands:

```
#!/bin/sh
python3 -m ensurepip
python3 -m pip install -r requirements.txt
python3 server.py
```

Base.

SpaceBaseRobots provides a game developer with a set of powerful core features, that are arbitrarily reusable. In the mean time, it shows how to use these features via handy example implementations.

Game Loop

SpaceBaseRobots implements an accurate self-correcting two-part game loop.
Since in frame based gameloops the graphic rendering power of your device can have side effects on the physics and gameplay mechanics of the game (in older games often exist so called "frame-jumps", that are possible on one machine but impossible on other machines), we **decouple our frame rate** from our update tick rate.
We opt for an accumulation based approach, that constantly checks whether the passed time since the last server tick surpassed the intended tick time to trigger the next iteration of the game loop.

```
def server_clock():

    lag = 0.0

    while 1:
        lag = lag + elapsed_time

        while lag >= SECONDS_PER_TICK:
            # blocking
            update()
            lag -= SECONDS_PER_TICK

        # non-blocking
        Qt_render()`
```

Since we want to keep PyQt in charge of the main thread event scheduling, we move the server clock into another thread to perform blocking calls for the update ticks and non-blocking calls for the decoupled render ticks.

Physics Engine

Movement

This part of the pysics engine simply takes in the velocity of the robot and the angle it is looking (and going) towards and determines the destination every tick. It then calls the collision to determine, whether the robot can actually go to its destination.

Collision

The main focus of the physics engine is the collision detection. It prevents robots from running through walls and detects when they touch one another. For walls and other obstacles we use a grid of blocks. In each of these Blocks there can be a variety of different Obstacles.

```
class Hazard:

    Empty = 0
    Wall = 1
```

```
    Border = 2
    Hole = 3
```

Each with it's own name and desgnated number for it to be saved in an array that covers every block of the grid. Manually filling an array with such numbers is how you create a level. There is a better way though (more on that later).
The first thing that is done with this array, when the game is started, is to simplify it. Every block that contains an obstacle is connected to all the adjacent blocks in its row or collumn. The resulting rectangle is then added to the list of all rectangles as a touple as such:

```
rects = []
# rows
if not tile_x == last_x:
    rects.append((rect_x, rect_y, row_len, TILE_SIZE, tile_type))
# collumns
if not tile_y == last_y:
    rects.append((rect_x, rect_y, TILE_SIZE, col_len, tile_type))
# single tiles
if not in_rects[tile_x][tile_y]:
    rects.append((rect_x, rect_y, TILE_SIZE, TILE_SIZE, tile_type))
```

This simplified version of all the obstacles on the board is then used to do the actual collision detection. Here we take the x- and the y-coordinate seperatley, in order for the robots to be able to slide along walls while preventing them form ever glitvhing through one at reasonable speeds (meaning speeds slower than the entire board size per tick).

```
for _ in range(10):

    if not x_collided:
        dx += dx_step
        robot_center = QPointF(dx + robot.x, dy + robot.y - dy_step)
        x_collided = Utils.check_collision_circle_rect(
            robot_center, robot.radius, tile_origin, rect[2], rect[3])

    if not y_collided:
        dy += dy_step
        robot_center = QPointF(dx + robot.x - dx_step, dy + robot.y)
        y_collided = Utils.check_collision_circle_rect(
            robot_center, robot.radius, tile_origin, rect[2], rect[3])

    if x_collided and y_collided:
        break
```

The collision then compares the collision check of all the obstacles and moves the robot no further, then just before hitting the first one (on both axis). It then saves the type of the obstacle it hit, in order to be able to

react differently upon hitting different types of blocks. For example it deals loads of damage to the unlucky robot that hits a "hole"-type block.

```
if final_tile_type == 3:
    robot.deal_damage(1000)
```

Bullets

As the robots can shoot, there is also a need for the handling of the bullets. Whenever a robot shoots the bullet(s) are added to the set of all bullets that are currently on their way across the board. Whenever they hit anything, they get deleted. In order to know if they hit anything, a set of functions tests if there are any obstacles along their way. They stop when the bullets hits the first one so you can for example take cover behind a wall. Whenever a robot is hit by a bullet it receives damage similarly to the one who ran into the "hole"-block.
Whenever a robot receives damage he loses some of his **life**. When this **life**-attribute drops below 0, the robot dies, after a short time respawns and is briefly put in a state of immunity.

AI controller

Since it is our main goal to have **encapsulated robot AIs** fight each other, it is necessary to provide an interface that forwards selected information, while preventing any other access by the AI to the servers mechanics or data.
Since an AI's calcuations can be quite massive, we move each AI into a separate thread. For this purpose we create an AI controller, that supports a thread-safe two-part communication gateway with the central core calculation server:
Each tick, the server will poll calculation results of the AI that will show their next actions: The robot AI has control over the acceleration and angle acceleration of the respective robot unit as well as the gun.

After validating, the AIs actions, the server will calculate the next board state and send selected information to the AI controller. For example, each robot only sees objects in its **field of view**.

You can host contests for the superior SpaceRobot AI!

This information is wrapped in a message of a certain message type that will be unpacked by the AI controller and sent to the AI.
Since any other access is omitted, no cheating on the side of the AI is possible!

Keyboard input unit

Since the standard key events of the OS are not suited for gaming, we implemented a key input state supervision unit like most of todays games use.
We keep track of hold down keys and can destinguish keys actions that should be triggered on key press from key actions that are performed each server tick depending on the key's state.
In addition, we provide a solution for certain (entwined) keys that normally should not be pressed simultaniously.

With this measurement we ensure tight keyboard controls, destined for local keyboard multiplayer like in the good old days. 😃

Config parser

You can easily configure the board's state as well as the deployed robots by yourself using the config reader unit.

Change the robot's parameters at will!

The config reader features five core functionalities:

```
config_reader = config_provider.ConfigReader()
# 1. Read a map design from file:
config_reader.read_level('filename.ini')
# 2. Create an obstacle map from currently read data:
map_ = config_reader.create_level()
# 3. Read configuration for robots to deploy  from robots.ini:
config_reader.read_robots()
# 4. Create robot units for the core server from currently read data.
robot_list = config_reader.create_robots()
# 5. Provide fallback values if config file or folder are missing:
#    In this case, the config reader will autmatically create a fallback config
file.
```

Note, that creating a new map after creating the robots might result in trouble since their starting positions might be invalid on the new map.

Robot unit access right management

We allow players to take control over robots that would be controlled by an AI per default!
To perform this at any given time in the games flow, we provide an access right and encapsulation management system for robot units.

Switch controls over a robot unit from human player to AI and back!

These functions help toggle who is in charge of the robot unit:

```
# The player is in charge, the AI is decoupled.
robot_unit.hand_control_to_player()
# The AI is in charge, the player is decoupled, but can regain control on key
press.
robot_unit.hand_control_to_robot()
# All gun access is permittet, for example if the robot is dead.
robot_unit.disable_gun_control()
```

And many more functions to enable or disable input/output control over parts of the robot unit.

BASE AIs

These AIs are built in the SpaceBaseRobots package!

An AIs main purpose is to set acceleration values. An AI is a **class** that defines **response functions** to different server messages. The message data may contain visual-, positional- or global information which is then used by the corresponding response function.

The most basic form of Movement:

Setting acceleration values based on current movement. This is only enought to implement simple static movements.

Implemented Examples:

- RandomMovement
- NussschneckeMovement
- SpiralMovement
- SpinMovement

Movements with a target:

As a response to alert messages destinations are set and the AI tries to move towards that direction. It does so by calculating the directions and angles of the destination and then setting the acceleration values accordingly.

- Implemented Examples:
- FollowMovement
- RunMovement
- RandomTargetMovement

Utilising vision data:

Using visual data it is now possible to see and react to obstacles. Current AIs try to avoid obstacles all together, however it is entirely possible to implement different reactions for the different object types.

- Implemented Examples:
- SimpleAvoidMovement
- ChaseMovement
- ChaseAvoidMovement

Shooting:

Determining, whether to shoot or not is also a core aspect of creating AIs. A function to enqueue a shot if the target is straight ahead is utilised by the following movements. But other functions that use other techniques like extrapolation or take other aspects like reloading times into account, can be inserted at will.

Implemented Examples:

- PermanentGunMovement
- ChaseMovementGun
- SimpleAvoidMovementGun
- ChaseAvoidMovementGun

Play.

You can call a game a game, as soon as you can play it! Therefore SpaceBaseRobots comes with an example game implementation and example configurations that you can immediately start having fun!
To **start the game on windows**, execute the start.bat file!
If this doesn't work, execute server.py with python 3.

Current configurations

The current configuration supports two players as well as two NPC-AIs:
One player has default key bindings, the other has controls for player two:

Action	Player 1	Player 2
accelerate forwards	W	Up Arrow
accelerate backwards	S	Down Arrow
accelerate rotation left	-	Left Arrow
accelerate rotation right	-	Right Arrow
turn left	A	-
turn right	D	-
shoot	J	Enter
toggle autopilot	P	End

Since all robots are in **SPACE**, we can normally only control our acceleration in certain directions. That means, if an acceleration key is pressed one, the robot will briefly accelerate in the given direction and **keeps moving** after the key is released. For longer acceleration period, hold the key down.
Since this **default controls** (implemented controls of player 2) are quite notorious to deal with, we also added something like a **flight assistant**: Invasive controls. They allow you to turn left or right normally without minding the momentum or acceleration. Player 1 features these controls.

Every robot can shoot with its respective gun in direction of view. Pressing the key will normally immediately result in firing a bullet with the gun. However, after firing, the gun will be unable to fire for a short time period, since it need to reload. Player 2's gun has been *illegaly modified* with a **gun option** and can shoot three bullets at once, so be careful!
All these keys can be hold down to keep triggering the respective action repeatedly.

Using *toggle autopilot*, bith players can sit back and hand control over their robot units to their respective AIs. At any time after a short cooldown, they can regain control by pressing the key again.

The colored identificator circle around a robot signalises its current **health** status: Green means a high percentage of health while red means danger!
All bullets that hit you will deal damage to you and deplete your current health. If your health falls below zero, your robot **gets destroyed** and you can't move it for a shot period of time.
After that, it will respawn with full health at another location and enter a short **immune** state in which you can't be damaged. The robot will appear in transparent blue.

Create a custom map

If the default map is too boring for you, you can create a custom map!
For this, you need to specify your map in a map file.
A map consist of 100 x 100 Tiles, of which every tile should be represented by a single number. No other characters should be placed.
Order the tiles in 100 lines of 100 characters each.
An invalid config will result in an error.

In the end, all tiles at the border will automatically replaced by a border tile.

List of valid tile numbers:

Number	Tile
0	Empty
1	Wall
2	Border
3	Hole

Be careful with map design, since the robots respawn in the corners!

Deploy robots

You can deploy zero to six robots at the current state by changing the options in the config file **robots.ini**.

Fight as up to 6 players against each other!

The file contains different sections: Each section stands for a robot, exept the default section **BASE**:

```
# comments start with # and are not be evaluated by the config parser
[BASE]
# default section
# this sections doesn't define a robot's parameters, instead it provides default
values for each robot.

# leaving an option empty in a robot section will result with the robot having the
default option instead (in most cases).

# a robot section defining the first robot
```

```
[robo1]
# each robot MUST be given a position.
position = 400, 200

# another robot section defining the second robot
[robo2]
radius = 4
position = 100, 50
# give a robot a non-default movement AI
# if a movement function needs certain parameters, note them down in order
separated by comma
movement = Follow, robo1
```

If invalid configurations are given, the parser will first check the default configurations of the BASE section. If configurations of the base saction are invalid, the config reader will use fallback options from code.
Each robot must be given a **valid starting position**! A robot with invalid starting position will not spawn and robots referencing it will be set on default/fallback movement.

Config parser options

- radius: the size of robot. given parameter is multiplied by the tile size of the board. Accepts float values.
- a_max: Maximum for acceleration in pxels per tick^2 both forward or backwards. Accepts float values.
- a_alpha_max: Maximum for angle acceleration in degrees per tick^2. Accepts float values.
- v_max: Maximum speed of the robot. Accepts float values.
- v_alpha_max: Maximum rotation speed of the robot. Accepts float values.
- fov_angle: The fov angle of the robot given in degrees. Accepts float values.
- max_life: Maximum health pool of the robot. Accepts int values.
- respawn_timer: Duration in seconds, how long the robot will remain dead. Accepts float values.
- immunity_timer: Duration in seconds, how long the robot will remain immune after respawning. Accepts float values.
- auto_resync: Boolean, if True, robot uses auto resync feature.
- position: Starting position of the robot. x, y separated by comma on the board. Values between 0 and 1000. If a robot overlaps with an obstacle, its starting position is invalid. There is no default position!
- alpha: Starting direction of sight of the robot in degrees.
- movement: movement AI for the robot. If a movement needs additions parameters, separate them by comma. For avalable movements look at list of available movements.
- gun: Boolean, if True, the robot has a gun with given gun parameters.
- gun_bullet_speed: Bullet speed of gun, if it exists. Accepts floats.
- gun_reoad_speed: Time in seconds, how long it takes the gun to reload, if it exists. Accepts floats.
- gun_options: String list of additional gun options, separated by comma: Look at list of gun options for available gun options.
- player_control: Boolean, if True, a player can control the robot.
- invasive_controls: Boolean, if True, use invasive controls.
- invasive_controls_turn_rate: If invasive controls are active, determines the turn rate with each tick. Accepts floats.
- keys: Key binding for player control, if it exists. For available key bindings, look at list of availabel key bindings.

- alert_flag: Boolean, determines if a robot receives additional information messages. Different behaviour if used in default section: If True in default section, determine value automatically from AI options. If false, deny alert message to every robot. This value can be manually overwritten in robot sections in both directions: If value in robot section is given, the robot will receive messages if given True and will not receive messages if given False.

List of available gun options:

- trigun: gun will fire two additional bullets

List of available movements:

- Movement: Static default movement.
- RandomMovement
- NussschneckeMovement
- SpiralMovement
- SpinMovement
- FollowMovement, needs a target robot given by section name of target
- RandomTargetMovement
- SimpleAvoidMovement
- RunMovement
- ChaseMovement, needs a target robot given by section name of target
- ChaseMovementGun, needs a target robot given by section name of target
- SimpleAvoidMovementGun
- PermanentGunMovement
- ChaseAvoidMovement, needs a target robot given by section name of target
- ChaseAvoidMovementGun, needs a target robot given by section name of target

List of available key bindings:

- default_scheme
- player_one_scheme
- player_two_scheme
- player_four_scheme
- num_block_scheme

Extend.

This project provides a developer with a kit of useful tools while also allowing him/her to extend the game's features easily:
The project is divided in different modules:

- server: the main module of the game that contains the game loop and the physics engine
- model: contains the robot units - they represent the current state of the robot in the server as well as functionalities like damage and death management.
- movement: contains included robot AIs
- ai_control: contains the control structures for thread safe and encapsulted communication with AIs.

- player_control: contains functionalities, that allow human players to control robot units and keyboard input.
- robogun: includes the gun functionalities
- config_provider: includes config reader and default values
- utils: utils for handy helper functions

Extension interfaces

For certain extension processes, the project supports default proceedures.

Add new AIs

Add a new AI in movement module: An AI should inherit from a Movement class, thus implement responses to each message type.

```
class NewMovement(Movement):
    # if an additional parameter is needed for the movement,
    # add it to the OPTIONS list, that the parser will know.
    OPTIONS = [OPTION_NAME_STRING]

    # if the AI should normally receive additional information messages,
    # state that here
    ALERT = True

    def __init__(self, option):
        # [...]

    # data will be the message payload
    # robot is a robot_control
    # return the new calculated a and a_alpha
    def response_name(self, data, robot):
        # [...]
        # use all helper functionalities of the robot_controller like
        # robot.shoot()
        return a, a_alpha
```

Don't forget to add your new movement AI to the list of available movements in the config_provider module. If you add a new additional option, you also need to write a new validator for it and add the validator mapping in config_provider module for the config parser to notice.

Add new key bindings

Add new key bindings as class attribute of the ControlScheme class in player_control module. A key binding is a dictionary that maps a PyQt5 Key name to a key action string (also defines in ControlScheme):

```
new_mapping = {Qt.Key_X: ACC_STRING,
               Qt.Key_Y: ACC_REV_STRING,
               Qt.Key_Z; SHOOT_STRING
```

```
                # You dont need to map all actions!
                }
```

Don't forget to add your new mapping to the list of available key bindings in config_provider module.

Add new key events

Add new key responses in player_control module. A key response is either stateless or requires a state. So make sure, that you determine the kind of response first.
Add the new key response as a function to PlayerControl class:

```
# If it is stateless, it doesn't need additional parameters.
def new_stateless_response(self):
    # you have access to the robot unit representation of the server
    self.data_robot.do_stuff()

# If it has a state, the state is an additional parameter:
def new_response_with_state(self, state):
    if state:
        self.data_robot.do_stuff()
```

If you want to map your response to a key, add it to the respective set of keys with state or stateless key in ControlScheme class.
Then add a string name, that maps to a string containing the name of the new response:

```
class ControlScheme:
    # make sure to have no typos in the string!
    NEW_RESP_STRING = 'new_stateless_response'
```

Now you can use this string in a key binding!

Add new data traffic

Adding new data traffic means adding a new message type. All AIs need to implement responses to this message type. We look at the ai_control module:
First, give your message type a name. For this go to class SensorData and add a new name string:

```
class SensorData:
    # this name will be used to call responses of AIs
    NEW_MESSAGE_STRING = 'new_message'
```

Add a mapping of the string to the function name in process_data function of the RobotControl class.

Then, all AIs need to implement the new message type:

```
class Movement:
    # add this empty response to default movement
    def new_message(self, data, robot)
        return robot.a, robot.a_alpha
```

Now every AI can perform a more complicated response to the new message type.

Finally, the server needs to be told how to create the new messages.
In server module Board class implement a new creation function and add it to the game loop at the appropriate place:

```
    def create_new_message(self):
        data = # create your data

        # return correct message data.
        return SensorData(SensorData.ALERT_STRING, data, self.time_stamp)
```

Add new hazard types

The adding of a new hazard (a new type of obstacle) is also possible. Simply think of a name and then give it an unoccupied number between 1 and 10. Then add it to the Hazard class.
For example:

```
class Hazard:
    Empty = 0
    Wall = 1
    Border = 2
    Hole = 3

    Asteroids = 5
    BoostUp = 8
    BoostDown = 9
```

You then have to think about how you want the robots to react, when they come in contact with the new obstacle. These Asteroids for example don't stop a robot but rather slow it down and do damage to it when it advances in them:

```
if final_tile_type == 5:
    robot.deal_damage(v/200)
    min_dx = max_dx/2
    min_dy = max_dy/2
```

These boosters accelerate the robot in one direction but dont let him through in the opposite:

```
if final_tile_type == Hazard.BoostUp:
    min_dx = max_dx
    if max_dy < 0:
        min_dy = abs(max_dy) * -2

if final_tile_type == Hazard.BoostDown:
    min_dx = max_dx
    if max_dy > 0:
        min_dy = abs(max_dy) * 2
```

These if clause have to be put in at the and of the collision function where the following comment indicates it.

```
# TODO: Insert conditions for addiditial Hazards here
```

If you want the new obstacle to let Bullets through you also have to add them to the "Whitelist" in the bullet collision as such:

```
# TODO: If you want your Hazard to leave Bullets through
#   insert Hazard.YourNewHazard
can_pass = {Hazard.Empty, Hazard.BoostUp, Hazard.BoostDown}
```

Default configs

Change default configurations in config_provider module. Default robot configs are found in ROBOT_FALLBACK dictionary. Other config options like tick rate and maximum number of robots can be modified here as well. Be careful while doing so, we do not take any liability for malfunctions if these parameters are changed!

Extension proposals.

Of course, you can extend the functionalities in even more creative ways. We have a few suggestions for you:

Add an event handler

There are other things like special abilities or simply special game states (event trigger), that require something (triggered function) to happen. For this, you need to implement an event handler.
In the SpaceBaseRobots package, we have implemented an unused example event handler for collision events: First, we implement the event handler that will look up a triggered function if the event trigger (in this case a special collision between two robots) occures:
Then, create a function that can create these recipes (mapping from event trigger to triggered function).
You will find the code in the server module:

```
def add_catch_recipe(self, fugitive, hunters):
    """Adds a new recipe type for collision events.
```

```
    If fugitive is caught by any hunter, perform recipe action.
    """

    def recipe_action(hunter, board):
        fugitive_bot = board.robots[fugitive]
        fugitive_pos = (fugitive_bot.x, fugitive_bot.y)
        hunter_bot = board.robots[hunter]
        Board.teleport_furthest_corner(fugitive_pos, hunter_bot)

    for h in hunters:
        f = partial(recipe_action, h)
        self.collision_scenarios[(fugitive, h)] = f

def handle_collision_event(self, col_tuple):
    """Collision event handler.
    Perform all given recipes for current collision event.
    """
    if col_tuple in self.collision_scenarios:
        self.collision_scenarios[col_tuple](self)
```

Add different gun types

You might want to have different gun types performing different actions.
In robogun module in the RoboGun class, we left intentional empty space to enqueue different types of data in the fire_queue:

```
def prepare_fire_robot(self, data=True):
    if self.gun_access_robot:
        self._prepare_fire(data)

def trigger_fire(self):
    # in trigger_fire this data can be used
    # to distinguish different gun types!

    # [...]

    task = self._get_fire_task()
    if not task:
        return False

    # get data by: _, task_data = task now.
    # forward that data to the data_robot via return
```

The data_robot can now create the respective projectile for the new gun type!

```
class DataRobot(BaseRobot):
    def perform_shoot_action(self):

        # get gun data to create bullet.
```

```
        maybe_data = self.gun.trigger_fire()
        if maybe_data:
            # do stuff with the data!
```

Add hybrid models of player/Ai control

Using the DataRobots right management system in model module, you might as well create hybrid models between player control an robot conrol:
For example the AI could determine the acceleration values while the player controls the gun or vice versa!

Have fun building, playing, extending!