CPL

JavaScript

2016

Abstract

This document, about learning basic programming in JavaScript, is an integral part of the course on ”Comparative Programming Languages” of prof. Frank Piessens.

Contents

1 FightCodeGame

The practical session for CPL on JavaScript is about programming your own robot tank and battle against others. In this session, we will rely on the programming platform of the web site `http://beta.fightcodegame.com`.

The platform consists of a local fight game engine that simulates fights between robots. The server-side of part of the platform, which keeps track of challenges and rankings, will not be used.

FightCodeGame is a free to play game, centered around the creation of so-called robots (or tanks) that have to fight on a battlefield against other robot tanks. You – as a developer – have to think about the AI of the robot and implement it in JavaScript on top of a given API.

You don’t have to hand in anything for this project.

1.1 Setup

For this exercise session you will only need a working web browser. Under normal circumstances you can rely on the online application on the web site `http://beta.fightcodegame.com/`. To work with the application, you will need a Github account. If you don't want this, please download the offline variant from Toledo.

The documentation of the API of the robots can be found on http://beta.fightcodegame.com/docs/.

1.2 Debugging

To help you while programming your robots, you can rely on your browser's built-in functionality for debugging. If you want more help on this topic, please consult one of the following links:

- https://developer.chrome.com/devtools/docs/javascript-debugging
- https://developer.mozilla.org/en/docs/Debugging_JavaScript

You can make calls to `console.log` to print to the debugger console. Don't hesitate to ask for help on this as the debugger will make your life a lot easier while working on this project.

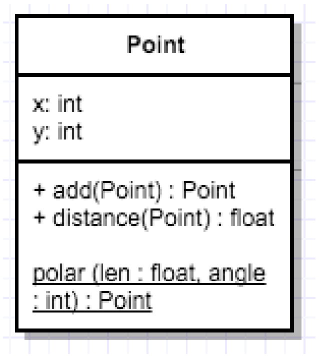
2 Implement your robot

This section will give you five tasks to fullfil in order to become a bit familiar with JavaScript and some of its concepts. There is no standard right or wrong solution for this project. Make sure that you clearly understand what you have programmed and what happens.

```javascript
function Robot() {};
Robot.prototype.onIdle = function(ev) {};
```

Listing 1: Skeleton code for a very minimal robot without any interaction.

Listing 1 shows the skeleton code of an extremely simple robot. The robot consists of a plain Javascript constructor function, named `Robot`. The engine behind FightCodeGame, called the fight engine, will use this constructor function to create an instance that will fight against other robots.


Point
x: int
y: int
+ add(Point) : Point
+ distance(Point) : float
polar (len : float, angle : int) : Point

```
var Robot = function(robot) {
};

Robot.prototype.onIdle = function(ev) {
    var robot = ev.robot;
    robot.ahead(10);
    robot.back(10);
    robot.turn(90);
};
```

Listing 2: Example code of a simple robot.

The fight engine will simulate a battle between robots. To do so, it will call event handlers that can act on specific events that happened during the fight. These event handlers can be implemented by your robot. For example, when your robot runs out of actions during the game loop, the fight engine will call the `Robot.onIdle` event handler. A simple robot that moves around a bit, is shown in Listing 2.

Task 1: Enhance the robot from Listing 2 so that it can win from all three basic scenarios. You can choose a different opponent strategy in the top right dropdown list on the battle-field. Rely on the documentation to learn what event handlers can be installed and what information event objects contain. Don't make it too hard for yourself. This first task is to get familiar with general JavaScript.

Task 2: Make clever use of the ability to clone your robot via `robot.clone()`. This will have a certain impact on your code as you can't be sure anymore that e.g., every scanned robot is an enemy as it could be your own clone. Carefully read the documentation about the `clone()` function.

Task 3: Convert the UML class diagram for `Point` to plain JavaScript, but don't use the ES6 `class` construct as that would be too easy. The arguments `x` and `y` in the constructor are optional. The `polar` method is a static method. Think about the effect of `new` when applied on a function constructor.

```
var Robot = new RobotBuilder()
    .on("idle", /* handler */)
    .on("wallCollision", /* handler */);
```

Listing 3: The use of a ‘RobotBuilder‘ object in practice.

Task 4: Use the builder pattern to implement a builder object `RobotBuilder`. Transform your original robot code so that it uses the custom `RobotBuilder` API as exemplified in Listing 3. Automatically transform the RobotBuilder `.on(...)` handlers to their counterpart within the fight engine. For example `.on("idle", h)` would become `.onIdle = h`.

Task 5: Each event handler receives a robot event `ev`. Optimize your builder object so that you can simplify your handlers by giving them direct access to `ev.robot`. Do this by wrapping the original handler with a custom closure in the `on` function of the builder object. To invoke functions, it might be handy to rely on `Function.call()` or `Function.apply()`. As an example, check the function definition of `directRobotHandler`.

```
function directRobotHandler (robot) {
    /* directly access robot instead of via ev.robot */
};
var Robot = new RobotBuilder().on("idle", directRobotHandler)
```