CPL
JavaScript Project

19 december 2013

Abstract

This document (v20131219) contains the necessary background, guidelines and project information for the JavaScript project of the CPL course of prof. Frank Piessens.

Contents

1 Introduction

The goal of this project, in general, is to learn the JavaScript programming language and its ecosystem. This also involves becoming familiar with the typical tool chain, like debuggers and code analysis tools. Of course, you don't have to become an expert of specific APIs.

The subject of this project is about a multi-tier, network game. The game involves tanks (i.e., the clients) that move and perform actions within a battlefield (i.e., controlled by the server).

The rest of this document will cover the game details and API, information on how to develop and general project information.

1.1 Project

The project requires you to implement both the server and client code of the game. The server code will run on the Node.js platform. The client code must be flexible so that it can work both as a stand-alone console application (again on top of Node.js) or a browser application.

You will be given skeleton code and references to API manuals to get you as soon as possible on the rails.

2 Game

2.1 Setup

The application consists of a server and one or more clients or tanks. Each client must implement some logic to allow a tank to act within a battlefield of other tanks. A tank can move, shoot rockets to destroy opponents, and fire radar beams to discover the battlefield. A tank has only a limited view within the battlefield (called its scope) and rockets can only be fired at tanks within a specific radius. The server will control the meta data of the battlefield and will allow all clients to communicate with each other.

The logic of the tank must be able to respond to messages from the server. In the beginning, the logic can be instrumented with human help, e.g., via some control buttons or the mouse. In a later stage, the logic must be made more intelligent so that it can act on its own and automatically decide on its actions.

In a first stage, the server will boot and initiate a new battlefield. After this stage, tanks can join the game. By moving around wisely (e.g., to hide behind walls) and using their radar and scoped view, tanks can discover opponents and try to destroy them. At the same time, each tank must be careful, e.g., by hiding behind walls, not to be taken down by any other tank.

2.2 Battlefield

The battlefield is represented as a matrix of size $n \times m$, assigned to a `field` property of a Battlefield instance. Each cell in the matrix (called a `Tile`) can be 0 (or `Tile.FREE`) meaning that is empty, 1 (or `Tile.WALL`) representing a wall, or 3 (`Tile.TANK`) representing a tank. Each cell has size 1×1.

A battlefield constructor receives a reference to a configuration object and, optionally, a list of field mutators. Each field mutator is a function receiving a reference to the configuration and the current field.

```
function add_wall (config, field) {
    // do something with the field
    field[config.height-1][config.width-1] = Tile.WALL
}
```

A mutator function can modify the current layout of the battlefield. If no list is given, the default mutator should be applied (see later).

The configuration uses terms like `width` and `height`. The height of the battlefield matrix is the amount of rows and the width is the amount of columns (see Scope).

2.3 Scope

Tanks have a limited scope on the battlefield. The next subsection explains how the current scope of a tank can be requested from the server. A tank will also receive a scope response whenever a new tank enters or leaves its scope.

The response of such a request is a sub-matrix of the battlefield, centered around the tank's own position. `Configuration.scope` defines the amount of rows and columns that will be added around this center. Parts of the scope that fall outside of the battlefield are simply cut of.

For example, in a 5x5 battlefield, with a tank T1 in the center and a defined scope of 2, the scope of T1 is exactly the battlefield. If tank T1 would be in position [3,3]

W	W	W	W	W
W				W
W				W
W			T_1	W
W	W	W	W	W

then the scope is

			W
			W
		T_1	W
W	W	W	W

Based on this sub-matrix and its current position, the tank should be able to build up a local view on the battlefield. Part of this view will be 100% accurate (because it falls within the live scope of the tank or because it is polled via a radar beam) while other parts are simply not yet explored or partially accurate (e.g., there was a tank on a specific position, but it may have moved).

2.4 Network API

The API is centralized around a simple message sending/receiving protocol.

Whenever a tank is ready (i.e., bootstrapped), it should connect to the server and send a `register` message with its name as parameter.

```
var socket = io.connect("server:port");
socket.emit("register", {
    name: "the name of your robot"
});
```

Message Type	Sender/Receiver/Direction
register(name)	client → server
registered(configuration, position)	client ← server
new-player(name)	everyone except client ← server
move(direction)	client → server
scope(position,subfield,scope)	client ← server
scope(position,subfield,scope)	some others ← server
shoot(position)	client → server
game-over(killer)	victim ← server
beam(degree)	client → server
beam(result)	client ← server
lost-player(name)	everyone ← server

In case of a successful registration, the server will send back a `registered` message, containing a copy of the server's configuration and a randomly assigned (defined by the server) position of the tank. The server will also notify all other players the join of this new tank via a `new-player` message and a `name` parameter.

To move your tank, you must send a `move` message to the server, with a direction parameter. The server will reply with a `scope` message, containing your current `position`, the view of your `subfield` (explained in the previous subsection) and the `scope` (i.e. `Configuration.scope`). The server will also send a specific, individual scope update to all tanks that have:

1. The tank, on its new position, in scope
2. Lost track of the moving tank (e.g., because it got out of the scope)

Shooting or sending radar beams can be done via a `shoot` or `beam` message. The `shoot` message requires a `position` parameter of a possible victim. If there is a tank on this position (see later for the exact details), the tank will be destroyed.

If a rocket destroys a tank, a `game-over` message will be send from the server to the victim tank (with a `killer` parameter containing the location of the killer). Next, the victim will be disconnected from the server.

The `beam` message requires a `degree` parameter, indicating a direction on where to shoot the beam. The server will reply with a `beam` message and a `result` parameter containing the type (integer) of the tile on which the beam bounced.

If a player decides to stop, it can simply close the connection. On a disconnect of any tank, the server will notify all other players via a `lost-player` message and a `name` parameter to indicate who stopped.

3 Developing

3.1 Requirements

The project relies on a recent web browser, a working installation of Node.js [1] (at least version 0.10.9), and on several third-party libraries for Node.js. The exact specifications of these libraries can be found in the `package.json` file. Read the next section on how to automate their installation.

You will also need the `npm` and a `zip` tool. All tools are available as free- and open-source software. In doubt about the correct version, please don't hesitate to contact.

[1] http://nodejs.org/download/

3.2 Bootstrapping

In order to bootstrap your project, you'll have to download some libraries before you can start any development. Start by execution `npm install` within your project directory. This command will automatically download and install all the necessary libraries in the local folder `/node_modules/` of your project, based on the `package.json` file.

socket.io. Socket.IO[2] aims to make realtime apps possible in every browser and mobile device, blurring the differences between the different transport mechanisms. Section 4 shows some examples on how to use this library.

jshint. To help you write good, qualitative code, you can use the `jshint` tool. JSHint[3] is a community-driven tool to detect errors and potential problems in JavaScript code. Using this tool is not mandatory.

mocha. Mocha[4] is a feature-rich JavaScript test framework running on node.js and the browser, allowing asynchronous testing. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases. This library is used to express the server-side test cases.

node-inspector. Node Inspector[5] is a debugger interface for Node.js. Node Inspector supports almost all of the debugging features and work from within a browser. Section 3.4 explains how to use this tool to debug your own project.

The project directory has a local copy of the Node.js API documentation[6] within the `help/` directory.

The use of each tool will be explained in the following subsections.

3.3 Testing

The `test/` directory contains tests to verify the correct functionality of the server-side part of your application. Don't try to run the test directly via `node`. These tests should be verified using the mocha framework:

```
$ ./node_modules/.bin/mocha --reporter spec
```

Make sure that your code doesn't falsify any provided test, as the mechanized verification tool will rely on this specific behavior!

If you want, you may add extra test cases for all your code. They will be used as part of the manual verification and can be used to quickly overview the more specific parts of implementation and its correctness.

Another form of testing is helping the programmer to write complex programs without worrying about typos and language gotchas. This is where static code analysis tools come into play and help developers to spot such problems. JSHint scans your program and reports about commonly made mistakes and potential bugs. The potential problem could be a syntax error, a bug due to implicit type conversion, a leaking variable or something else.

```
$ ./node_modules/.bin/jshint ./src/
```

Use it as a guiding hand before handing in code to make sure that your code is somehow compliant with the JSHint standard configuration (i.e., the settings that are used when no specific options are present). Of course, you are not forced to behave according to the JSHint specs.

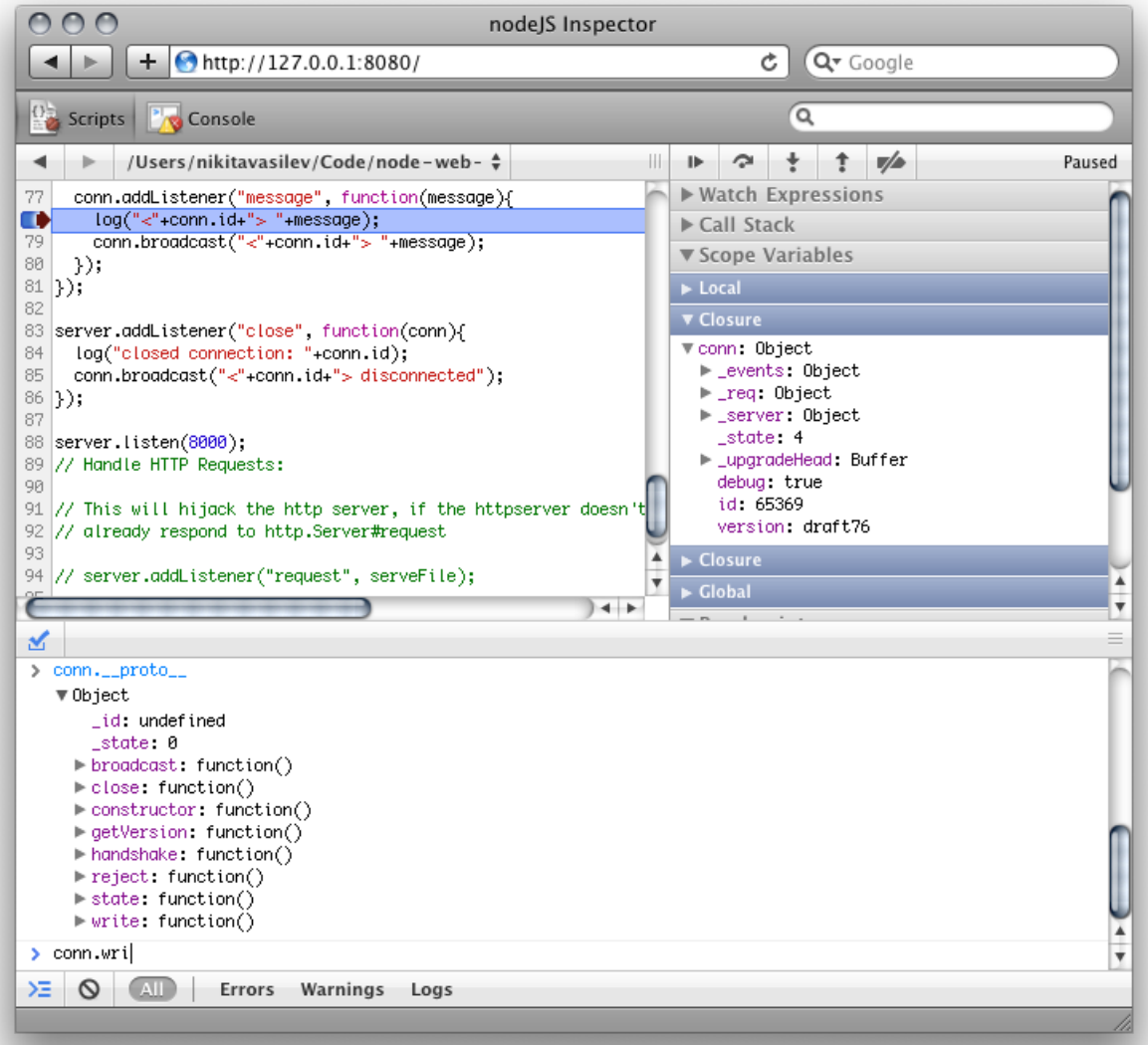

Figure 1: The node-inspector debugging interface works from within a web browser. It is possible to navigate source files, add breakpoints, perform controlled execution, and inspect scopes, variables, and object properties.

3.4 Debugging

Debugging via node-inspector involves two steps. First, one must start the inspection framework via `node-inspector &` (putting it in the background). This will instantiate a debugger and a web server. In standard situations, the debugger will listen on `http://127.0.0.1:8080/debug?port=5858`. The look and feel of the application shouldn't be to different from other debuggers.

The second step is to launch an instantiation of the application you want to debug. Important is to add the debug flag `--debug`. It is also possible to enable debugging on a Node.js instance that is already running (e.g., to use node-inspector on Windows [7]). To pause node immediatly after the start-up phase, use the `--debug-brk` flag.

```
$ ./node_modules/.bin/node-inspector &
$ node --debug ./server.js
```

4 Tasks

4.1 Configuration

As a first step, create a `Configuration` class in `src/configuration.js` that behaves according to the first tests within `test/server.js`. Make sure that the file can be loaded via `require`[8]. The first two parameters for Configuration are `width` and `height`. The third and fourth parameters are `rocketRadius` and `scope` and are optional.

4.2 Battlefield

The Battlefield class is the main class that forms the heart of the game. It keeps state of, and tracks, all connected client/tanks and must calculate scopes. This class is not aware of any networking.

The Battlefield class should live within `src/battlefield.js` and should implement minimally the methods that are used within the tests.

The Battlefield constructor receives references to a configuration and a list of field processors (as explained in the second section). If no explicit list is given, it is assumed that a default mutator is used. This default mutator should simply add a wall around the battlefield.

The method `Battlefield.tanksInScope(t) should return a list of all visible tanks of a tankt`', based on its position within the battlefield.

4.3 Tank

The Tank class should live within `src/tank.js` and should implement minimally the methods that are used within the tests.

A Tank constructor receives references to a socket and a battlefield, making the battlefield network aware. The socket is received from a call to the `socket.io` library and offers two important functions. Via the `on` function, one can install handlers for specific messages:

[2] http://socket.io/
[3] http://jshint.com/
[4] http://visionmedia.github.io/mocha/
[5] https://github.com/node-inspector/node-inspector
[6] http://nodejs.org/api/all.html
[7] https://github.com/node-inspector/node-inspector#windows
[8] http://nodejs.org/api/modules.html

```
socket.on("type", function(data) {
    // function will be called whenever this specific message is received
    // data contains the parameters of the message, e.g. data.position
});
```

To reply to clients, a socket has the `emit` function:

```
socket.emit("registered", {
    configuration: ...,
    position: ...

});
```

To reply to all other tanks, except for this client tank, use the `broadcast.emit` function:

```
socket.broadcast.emit("new-player", {
    name: ""
});
```

A tank should be able to move around the battlefield, shoot rockets, and fire radar beams.

4.4 Rockets

A tank can shoot a rocket towards another tank. If a tank gets hit by a rocket, it is destroyed and should receive a `game-over` message, and be disconnected. A rocket have a predefined radius (`Configuration.rocketRadius`), and can't be shot through or over walls.

The rocket radius defines a circle, starting at the center of the cell of the shooting tank. Each cell within or on this circle (i.e., if the center of a cell is within the circle or on its boundary) can be attacked, unless the straight line between the center of the cell of the shooting tank and the center of the attacked cell *crosses or touches* the boundaries of a wall cell.

Each cell in the battlefield matrix is of size 1×1. The upper left cell with position [0,0] has thus its center at (0.5, 0.5). The center of cell [1,1] is at (1.5, 1.5). The top left and bottom right boundaries of cell [0,0] are (0,0) and (1,1). The boundaries for cell [1,1] are (1,1) and (2,2).

T_1	W
W	T_2

In a battlefield with `rocketRadius` being e.g., 3, tank T1 can't shoot T1 because the straight line between their centers touches both walls. This also means that tanks can't shoot 'around the corner':

T_1	
W	T_2

Again the straight line touches the lower left wall. A tank can shoot over another tank:

T_1	T_2	T_4
T_3	W	T_5

Tank T1 can shoot both tank T2 and tank T4 (and also T3 because all of them are within the rocket scope of tank T1 and thus belong to the list of attackable tanks) because their straight lines don't cross or touch any wall tile. In fact, only tank T5 is secure from tank T1 (but not from T4).

The method `Battlefield.tanksInRocketScope(t)` should return a list of all attackable tanks, based on the position of the given tank `t`. The method `Battlefield.shootTank(t1, t2)` removes tank `t2` from the battlefield and returns `true`, if and only if tank `t2` is within the rocket scope of the shooting tank `t1`. In all other cases it returns `false` and nothing happens with `t1` or `t2`.

As a reminder: a tank's scope is defined as a block and a tank's rocket scope as a circle.

4.5 Radar Beams

A radar beam acts more or less like a rocket without a fixed radius and only interacts with a cell if it crosses its boundaries. The method `Battlefield.shootRadarBeam(t, degree)` shoots a beam starting from the center of the cell of tank `t` with the direction specified by any `degree` (e.g., 45°, 60°, 77°, ...)[9]. A degree of 0° means sending the beam to the south. If the radar beam, starting from the center of the cell of the shooting tank, *crosses* a wall, it should return `Tile.WALL`. If radar beam *crosses* a cell tank, it should return `Tile.TANK`. In all other cases, it simply returns `Tile.FREE`.

For example:

T_1	W
W	

If tank T1 shoots a beam with degree 45°, the result must be `Tile.FREE`.

T_1	W
W	T_2

In this example, if tank T1 shoots a beam with degree 45°, the result will be `Tile.TANK`. Note that tank T2 can't be destroyed by tank T1, because the straight line between their centers touches both wall cells.

4.6 Client

The client part of the game is a tank that connects to a remote battlefield and tries to stay alive as long as possible. Therefor, it can move around, shoot rockets and use a radar. Your task is to create both the tank's logic (i.e., the code that responds on, and sends messages to the server) and some visualization code. All client code must be added to the `src/client.js` skeleton file.

The idea is that your tank's logic is intelligent enough to acts on its own, without human intervention. The visualization code should only visualize what is currently happening on the battlefield.

The client code should be both loadable via a plain Node.js instance:

```
$ node ./src/client.js
```

or via a web page (skeleton code in `src/client.html`) in a browser:

```
<html>
...
<script src="/client.js"></script>
...
```

As a first step, you probably want to start by creating a simple console GUI to control a client tank. If you want (it is not obligatory), you can also create a real GUI based on an HTML5 canvas element. Code to interact with a canvas is already provided. The axis of the canvas are so that `(0,0)` is the top left corner and that `(x, y)` is the bottom right corner. Therefor the Canvas class has a method `transform` that maps the `[row, column]` notation (or `field[row][colum]`) to a position on a canvas. You don't need to change the Canvas class implementation.

```
//fetch a reference to the canvas within the document and build
//an object around it
var c = new Canvas(window.document.getElementById("battlefield"));
```

[9] The actual degree within the network message should be a number, without the '°'.

```
// Calculate the start (left top corner) position of the image
// e.g., the size of each matrix cell is 100px on 50px
c.drawImage(ImageStore.Tank, 320, 240, 100, 50);
```

The browser GUI should map the battlefield matrix (or at least the local view on the battlefield) on that canvas. You should divide the rows and columns of the matrix evenly over the height (`canvas.element.height`) and width (`canvas.element.width`) of the canvas. The ImageStore class has some preloaded images of a tank and a wall. You can add more if you want.

In practice you will probably use three classes. You will have a View part, that instruments a (browser) GUI, and a Tank class (the logic) that implements the 'brain' of your tank by reacting on events. At last, you probably want some kind of Controller class that controls both the View and the Tank and that takes care of the network layer.

If your server code is compliant with the protocol, clients must be able to connect to different servers, e.g., the one from your neighbor, with different configurations and with different other players.

To connect to a server, use the `io.connect` function:

```
// io should live in the global namespace, either because this code is loaded
// within a browser window, or because of require('socket.io')
var socket = io.connect("server:port");
```

To communicate over this socket, use the same techniques as explained earlier.

5 Project Specifics

5.1 Questions

General questions about JavaScript or the project can be asked via the specific Toledo forum. The main purpose of this forum is to let students help each other. However, the forum will be monitored actively in order to correct or improve (wrong) answers. Don't hesitate to ask for help or clarification if something turns out to be unclear.

Specific questions or remarks can be mailed to `Willem.DeGroef@cs.kuleuven.be`.

5.2 Submitting

Your final files should be submitted via e-mail. Create an encrypted ZIP file from your local `src/` (and `test/`, in case you have new tests) directory. Name your submission to `[student-id].zip` and use your student ID as the password. The reason the ZIP file must be encrypted, is to circumvent any defense mechanism from your/our mail servers. On a typical console, the command would look like (with s0111111 as the example student ID):

```
zip -r -e s0111111.zip src/ test/
Enter password:
Verify password:
updating: src/ (stored 0%)
...
```

E-mail the final ZIP-file as an attachment of an empty message to `Willem.DeGroef@cs.kuleuven.be`. As subject, use `[CPL-JavaScript-2013] student-id`.

5.3 Grading

Part of the grading of your project will be done via automated testing. Part of the tests reside within the `test/` directory. A score on these tests gives you an indication about the correctness of (some parts) of your solution. However, keep in mind that your solution will need to pass a more in-depth human code review and that your code can't rely on any third-party library (except those defined (recursively) by packages specified within the original `package.json`)!

5.4 Dates

The strict submission deadline is **Monday 6th of January**, 23h59.