Rot.js tutorial, part 1

This is the first part of a rot.js tutorial.

Contents

Setting up the development environment

Our game is played within a web page; a rudimentary HTML file should be sufficient.

```
<!doctype html>
<html>
    <head>
        <title>Ananas aus Caracas: rot.js tutorial game</title>
        <script src="https://cdn.jsdelivr.net/npm/rot-js@2/dist/rot.js"></script>
        <script src="/path/to/the/game.js"></script>
    </head>
    <body onload="Game.init()">
        <h1>Ananas aus Caracas</h1>
    </body>
</html>
```

We are going to put all the game code in one file, to maintain simplicity (when making larger games, it is far more useful to split the code across several files). We do not want to pollute the global name space with our variables; that's why we wrap all our code in an object named "Game".

```
var Game = {
    init: function() {}
}
```

Console output: ROT.Display

Being a JS app, our game can modify the HTML page in many ways. However, rot.js encourages only one kind of output: printing to its "tty console", which is represented by a HTML <canvas> tag. In order to draw anything, we first need to create this console and store it for later usage.

```
var Game = {
    display: null,

    init: function() {
        this.display = new ROT.Display();
        document.body.appendChild(this.display.getContainer());
    }
}
```

Note that this console has a default size of 80x25 cells; if we wanted different default dimensions, we would configure them via `ROT.DEFAULT_WIDTH` and `ROT.DEFAULT_HEIGHT`.

Generating a dungeon map

We will use one of rot.js's built-in map generators to create the game level. One of the design paradigms of rot.js is that people should not be forced to use some pre-defined data structures; this is why the generator is *callback-based*. We will pass our custom function to the generator; it will get called repeatedly during the process.

This might be a good time to check out the rot.js manual, which contains useful code samples and usage overview.

How should we store the resulting map data? We will use a very basic method of storage: an ordinary JS object ("hashmap"), indexed by strings (having the format "x,y"), values representing floor tiles. We are not going to store wall / solid cells.

NOTE: We are passing `digCallback.bind(this)` instead of just `digCallback` to the Digger. This is necessary to ensure that our callback is called within a correct context (*activation object* in ECMA parlance).

```
Game.map = {};
Game._generateMap = function() {
    var digger = new ROT.Map.Digger();

    var digCallback = function(x, y, value) {
        if (value) { return; } /* do not store walls */

        var key = x+","+y;
        this.map[key] = ".";
    }
    digger.create(digCallback.bind(this));
}
```

We still cannot see anything, because we have not written a single character to the display yet. Time to fix this: iterate through all the floor tiles and draw their visual representation.

```
Game._drawWholeMap = function() {
    for (var key in this.map) {
        var parts = key.split(",");
        var x = parseInt(parts[0]);
        var y = parseInt(parts[1]);
        this.display.draw(x, y, this.map[key]);
    }
}
```

Now make some changes so that this code is used: call this._drawWholeMap() at the end of the function Game._generateMap, and call this._generateMap(); from the end of your init function. If it's still not working, have a look at the same code linked to from the bottom of this page.

Randomly generated boxes

Finally, let's create some boxes - potential ananas storage. We will hide the ananas in one of them in later parts of this tutorial. To place 10 random boxes around, we will leverage rot.js's Random number generator.

`ROT.RNG` can do a lot of stuff, but we need something simple: a random, evenly distributed number between zero (inclusive) and one (exclusive), just like the `Math.random` does. The proper way of doing this is calling `ROT.RNG.getUniform()`.

We will store the empty cells in an array; for each box placed, we will pick a random empty cell, remove it from a list and mark that place as a box (asterisk) in our storage structure.

```
Game._generateMap = function() {
    var digger = new ROT.Map.Digger();
    var freeCells = [];

    var digCallback = function(x, y, value) {
        if (value) { return; } /* do not store walls */

        var key = x+","+y;
        freeCells.push(key);
        this.map[key] = ".";
    }
    digger.create(digCallback.bind(this));

    this._generateBoxes(freeCells);

    this._drawWholeMap();
};

Game._generateBoxes = function(freeCells) {
    for (var i=0;i<10;i++) {
        var index = Math.floor(ROT.RNG.getUniform() * freeCells.length);
        var key = freeCells.splice(index, 1)[0];
        this.map[key] = "*";
    }
};
```

And that's all for part 1. The whole working code is available at jsfiddle.net. Feel free to continue in Rot.js tutorial, part 2.

Rot.js tutorial, part 2

This is the second part of a rot.js tutorial.

Contents

The Player Character

Time to make some interesting interactive shinies! First, the player needs a decent representation. It would be sufficient to use a plain JS object to represent the player, but it is generally more robust to define the player via its constructor function and instantialize it.

By this time, you probably got used to the fact that some variable names start with an underscore. This is a relatively common technique of marking them *private*. JavaScript does not offer true private variables, so this underscore-based nomenclature is just our useful way of marking stuff as "internal".

We would like to place the player on some spare floor tile: let's use exactly the same technique we used in Part 1 of this tutorial to place the boxes: just pick one free location from our list.

```
var Player = function(x, y) {
    this._x = x;
    this._y = y;
    this._draw();
}

Player.prototype._draw = function() {
    Game.display.draw(this._x, this._y, "@", "#ff0");
}
```

```
Game.player = null;

Game._generateMap = function() {
    /* ...previous stuff... */
    this._createPlayer(freeCells);
};

Game._createPlayer = function(freeCells) {
    var index = Math.floor(ROT.RNG.getUniform() * freeCells.length);
    var key = freeCells.splice(index, 1)[0];
    var parts = key.split(",");
    var x = parseInt(parts[0]);
    var y = parseInt(parts[1]);
    this.player = new Player(x, y);
};
```

Preparing the game turn engine

There will be two entities taking turns in our game: the Player Character and Pedro (The Enemy). To make things simple, let's just alternate their turns evenly. But even in this simple case, we can use the `ROT.Engine` timing framework to our advantage.

How does this work? First, the `ROT.Scheduler.Simple` will be fed with all available *actors* - this component will take care about fair turn scheduling.
The `ROT.Engine` will then use this scheduler to automatically call relevant actors in a loop.
(Note: we pass `true` as a second argument to `scheduler.add` - this means that our actor is not a one-shot event, but rather a recurring item.)

It is very important to embrace the fact that everything is asynchronous in the world of client-side JavaScript: there are basically no blocking calls. This eliminates the possibility of having a simple *while* loop as our main timing/scheduling instrument. Fortunately, the `ROT.Engine` is well prepared for this.

Creating the scheduler and the engine is just a matter of adding a few lines to our code:

```
Game.engine = null;

Game.init = function() {
    var scheduler = new ROT.Scheduler.Simple();
```

```
    scheduler.add(this.player, true);
    this.engine = new ROT.Engine(scheduler);
    this.engine.start();
}
```

Interaction between actors and the engine

There is a tight symbiotic relationship between the engine and its actors. When running, the engine repeatedly picks a proper actor (using the scheduler) and calls the actor's `act()` method. Actors are allowed to interrupt this loop (when waiting asynchronously, for example) by calling `ROT.Engine::lock` and resume it (`ROT.Engine::unlock`).

It is possible to have multiple lock levels (the lock is recursive); this allows for complex chaining of asynchronous calls. Fortunately, this won't be needed in our simple game.

So, what is an actor? Any JS object with the **`act`** method.

```
Player.prototype.act = function() {
    Game.engine.lock();
    /* wait for user input; do stuff when user hits a key */
    window.addEventListener("keydown", this);
}

Player.prototype.handleEvent = function(e) {
    /* process user input */
}
```

We are using somewhat uncommon (but very useful!) technique of assigning event handlers: we pass a JS object as a second argument to the `addEventListener` call. Such object (`this`, in this case) must have the `handleEvent` method, which will be called once the event ("keydown") occurs.

Working with the keyboard and moving the player around

There is one last bit remaining to implement: detect the pressed key, decide whether it is valid and move the player accordingly.

Our event handler (`handleEvent`) gets executed with one argument: the Event object.

Its `keyCode` property is a number code of the key being pressed. Let's create a mapping of allowed key codes (this code sample uses numpad keys, but it is trivial to extend it to other layouts as well):

```
var keyMap = {};
keyMap[38] = 0;
keyMap[33] = 1;
keyMap[39] = 2;
keyMap[34] = 3;
keyMap[40] = 4;
keyMap[35] = 5;
keyMap[37] = 6;
keyMap[36] = 7;
```

Numeric values are not chosen randomly: they correspond to directional constants in `rot.js` (8-topology, clockwise, starting at the top - the same as CSS does).

We need to perform a two-step validation of user input:

1. If the key code is not present in `keyMap`, the user pressed a key which we cannot handle
2. If the key code **is** present, we need to check whether the PC can move in that direction

To convert a directional constant (0..7) to a map coordinates, we can use the `ROT.DIRS` set of topological diffs:

```
Player.prototype.handleEvent = function(e) {
    var keyMap = {};
    keyMap[38] = 0;
    keyMap[33] = 1;
    keyMap[39] = 2;
    keyMap[34] = 3;
    keyMap[40] = 4;
    keyMap[35] = 5;
    keyMap[37] = 6;
    keyMap[36] = 7;

    var code = e.keyCode;

    if (!(code in keyMap)) { return; }

    var diff = ROT.DIRS[8][keyMap[code]];
```

```
    var newX = this._x + diff[0];
    var newY = this._y + diff[1];

    var newKey = newX + "," + newY;
    if (!(newKey in Game.map)) { return; } /* cannot move in this direction */
}
```

The actual move is performed in two steps - redrawing the old position and redrawing the new position. After that, we remove our keyboard listener (the turn has ended!) and - **importantly** - resume the game engine (`unlock()`).

```
Player.prototype.handleEvent = function(e) {
    /* ...previous stuff... */

    Game.display.draw(this._x, this._y, Game.map[this._x+","+this._y]);
    this._x = newX;
    this._y = newY;
    this._draw();
    window.removeEventListener("keydown", this);
    Game.engine.unlock();
}
```

And that's all for part 2. The whole working code is available at jsfiddle.net. Feel free to continue in Rot.js tutorial, part 3.

Jump to navigation

Rot.js tutorial, part 3

This is the third part of a rot.js tutorial.

Looking inside the box

The game generates several boxes, but so far, none of them contains the prized ananas. Let us store the ananas in the first box generated:

```
Game.ananas = null;

Game._generateBoxes = function(freeCells) {
    for (var i=0;i<10;i++) {
        /* ...previous stuff... */
        if (!i) { this.ananas = key; } /* first box contains an ananas */
    }
}
```

Apart from moving, there is one more interaction a player must perform: looking into boxes. We will allow both Enter (keyCode 13) and Spacebar (keyCode 32) for this action:

```
Player.prototype.handleEvent = function(e) {
    var code = e.keyCode;
    if (code == 13 || code == 32) {
        this._checkBox();
        return;
    }
}
```

Opening a box to verify its contents is as simple as comparing the player's current position with our list of boxes and the stored ananas position:

```
Player.prototype._checkBox = function() {
    var key = this._x + "," + this._y;
    if (Game.map[key] != "*") {
        alert("There is no box here!");
    } else if (key == Game.ananas) {
        alert("Hooray! You found an ananas and won this game.");
        Game.engine.lock();
        window.removeEventListener("keydown", this);
    } else {
        alert("This box is empty :-(");
    }
}
```

Pedro, the angry owner

The game is now winnable! Let's add a villain as a second actor. We will place him using the same algorithm we used previously. To do this, let's refactor the original `_createPlayer` method into a more useful parametrized *factory* `_createBeing` by passing a constructor function as an argument:

```
var Pedro = function(x, y) {
    this._x = x;
    this._y = y;
    this._draw();
}

Pedro.prototype._draw = function() {
    Game.display.draw(this._x, this._y, "P", "red");
}

Game._createBeing = function(what, freeCells) {
    var index = Math.floor(ROT.RNG.getUniform() * freeCells.length);
    var key = freeCells.splice(index, 1)[0];
    var parts = key.split(",");
    var x = parseInt(parts[0]);
    var y = parseInt(parts[1]);
    return new what(x, y);
}

Game._generateMap = function() {
    /* ...previous stuff... */

    this.player = this._createBeing(Player, freeCells);
    this.pedro = this._createBeing(Pedro, freeCells);
}

Game.init = function() {
    /* ...previous stuff... */

    scheduler.add(this.player, true);
    scheduler.add(this.pedro, true);
}
```

This might be confusing to some, but passing functions around (as function arguments, for instance) is very common in JavaScript.

Pathfinding-based AI

Pedro is missing its `act()` method so far. We are going to use one of rot.js's pathfinding functions to implement Pedro's behavior: `ROT.Path.AStar` (the A* algorithm). Some rudimentary scaffolding is necessary:

1. The player must have public methods to read its position,
2. We need a *passableCallback* function which tells the pathfinder what areas are passable,
3. We need a *pathCallback* function, which will be called from within the pathfinder (to notify us about the shortest path found).

   Moreover, to make Pedro somewhat weaker than player, we will use the pathfinder only in 4-topology.

```
Player.prototype.getX = function() { return this._x; }

Player.prototype.getY = function() { return this._y; }

Pedro.prototype.act = function() {
    var x = Game.player.getX();
    var y = Game.player.getY();
    var passableCallback = function(x, y) {
        return (x+","+y in Game.map);
    }
    var astar = new ROT.Path.AStar(x, y, passableCallback, {topology:4});

    var path = [];
    var pathCallback = function(x, y) {
        path.push([x, y]);
    }
    astar.compute(this._x, this._y, pathCallback);
}
```

We now have the shortest path between Pedro and the player, stored in the `path` variable. Note that Pedro's current position is also part of the path; that's why we first discard the first item of our path. If the resulting path is only one-cell long, Pedro is standing close to the player and the game is over (player lost). Otherwise, we apply the same movement logic we used for the player in Part 2 of this tutorial.

```
Pedro.prototype.act = function() {
```

```
    /* ...previous stuff... */

    path.shift(); /* remove Pedro's position */
    if (path.length == 1) {
        Game.engine.lock();
        alert("Game over - you were captured by Pedro!");
    } else {
        x = path[0][0];
        y = path[0][1];
        Game.display.draw(this._x, this._y, Game.map[this._x+","+this._y]);
        this._x = x;
        this._y = y;
        this._draw();
    }

}
```

Ta-dah! The game is complete now; it is possible to win and lose. Some considerations for possible further improvements:

- Player can crash the game by moving onto Pedro's cell. Not only this is currently allowed, but it also disrupts Pedro's pathfinding (which expects the path to be at least two cells long).
- The `Game.map` structure should probably store positions of beings (player, Pedro) as well.
- It would be comfortable for users to increase the set of allowed navigation keys (number keys, vi keys).
- When a box is inspected, its appearance may change (to make it easier for player to distinguish between visited and unvisited boxes).

And that's all for part 3. The whole working code is available at jsfiddle.net.

Jump to navigation