Net-centric introduction to computing

Search & Rescue

m Jenkin + h Roumani

Table of Contents

1. Introduction

In this lab we create a search and rescue simulation. We imagine a scenario in which a ship has sent an emergency distress signal when it lost power and its engines stopped. The rescue operation involves scanning the search area in order to find the target ship as quickly as possible. The rescue vessel has a radar that can detect the target ship when in range. Here is an example of the simulation:

In the above example, the rescue vessel always follows straight paths, and when it reaches any of the four boundaries of the search area, it turns around in a way similar to a light ray reflecting off a surface; i.e. it bounces. But as you can imagine, there are many other strategies to scan the search area, and what we seek is to do is to build an infrastructure that allows one to test various algorithms and explore alternatives. Once built, this infrastructure can also enable us to test other ideas, such as using two rescue vessels or allowing the target ship to drift.

As with all labs, each lab is laid out as an ePub. It is expected that you will have read the lab electronic book prior to attending the lab. In order to encourage this, each lab has an associated set of 'pre lab' assignments that must be completed prior to attending the lab. The lab monitor will not allow you to participate in the lab if you cannot demonstrate that you have completed the pre-lab assignment prior to attending the lab itself. Exercises in the lab are to be documented in your ePortfolio lab book.

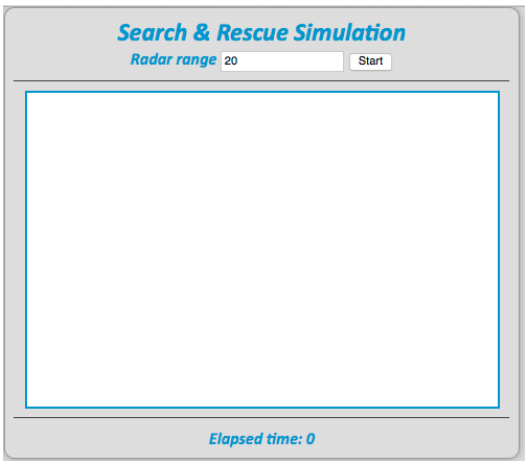
2. Background

Creating the simulation for this lab involves building an interface with HTML, formatting it using CSS, and implementing the logic through Javascript. In this section we highlight the key features needed from each of these technologies.

You may want to skim through the background material in your first review of this lab. After wards, go to the actual exercises associated with the lab and then refer back to the material presented here as needed.

2.1 HTML

The UI (User Interface) in our simulation looks like this:



We can recognize three different sections in the UI:

1. The top part contains a title and an interactive section that allows the user to enter the radar range of the rescue vessel (in pixels) and then click the Start button to start the simulation. They HTML tags needed to build this section are:

```
<h2>Search &amp; Rescue Simulation</h2>
<label>Radar range</label>
<input value="20" type="text" />
<button onclick="start();">Start</button>
<hr />
```

2. The middle part contains a canvas that renders the search area, draws the target and rescue vessels in it, and animates the movement of the rescue ship. We achieve this in HTML as follows:

```
<canvas width="450" height="300"></canvas>
<hr />
```

   Notice that the indicated width and heights are merely defaults. The user will be prompted to optionally change them once the simulation starts.

3. The bottom part is informational in nature. It lets the user monitor the progress of the simulation. Something like this:

```
<label>Elapsed time:</label>
<span>0</span>
```

The tags discussed above will create the needed UI but we need to be able to interact with them dynamically. For example, we must be able to read what the user entered for the radar range; we need to change the value of the elapsed time; and we need to detect that the user has clicked the button. These tags must therefore be tagged (using IDs and classes) so that Javascript and CSS can refer to them. This leads to this revised HTML content:

```
<h2 class="center">Search &amp; Rescue Simulation</h2>
<div class="center">
     <label>Radar range</label>
     <input id="radar" value="20" type="text" />
     <button onclick="start();">Start</button>
```

```
</div>
<hr />
<canvas id="searchArea" width="450" height="300"></canvas>
<hr />
<div class="center">
     <label>Elapsed time:</label>
     <span id="elapsed" class="progress">0</span>
</div>
```

Examine the revision and note the added `div` tags that allows us to divide the HTML document into logical parts and apply tagging and formatting on each.

2.2 CSS

In order to format the UI the way we want, we apply style rules at the tag, ID, or class level. For example, in order to centre contents, we surround it by a `div`, assign that `div` a class, and apply the following rule to all members of that class:

```
.center
{
   margin: auto;
   text-align: center;
   display: block;
}
```

The remaining formats can be implemented using this stylesheet:

```
@CHARSET "UTF-8";

body
{
     width:475px;;
     margin:auto;
     background-color: #dddddd;
```

```
	font-family: Calibri, Ariel, sans-serif;
	font-size: 16px;
	font-style: italic;
	font-weight: bold;
	color: #09C;
	border-radius: 10px;
	padding: 8px;
	border: 1px solid #999;
	box-shadow: 0px 0px 8px rgba(0, 0, 0, 0.3);
}

.center
{
	margin: auto;
	text-align: center;
	display: block;
}

#searchArea
{
	border: 2px solid #09c;
	background-color: #ffffff;
	margin: auto;
	display: block;
}
```

2.3 The Simulation Logic

The simulation produces a series of frames each of which captures the status at a given instant of time. To create a new frame, the program must perform the following operations:

1. Erase the content of the previous frame.
2. Draw the assets of the current frame. This involves drawing two icons (representing the target and rescue vessels) at their correct positions.
3. Update the bottom part of the UI so that the user is informed about progress.

4. Determine if the target is within reach, i.e. that this distance between the two ships is within the radar range. If so, this would be the simulation must stop.
5. If however the target is out of reach, then the position of the rescue ship must be advanced to simulate its movement. This depends on whether it is within the search area (in which case it continues in a straight line) or is at a boundary of the search area (in which case it must bounce back).

These steps are captured by the following logic:

```
function simulate()
{
   clear();
   drawTarget();
   drawRescue();
   updateProgress();
   if (found())
   {
        clearInterval(intervalID);
   }
   else
   {
        if (xBoundary()) direction.dx = -direction.dx;
        if (yBoundary()) direction.dy = -direction.dy;
        rescue.x += direction.dx;
        rescue.y += direction.dy;
   }
}
```

Each time the `simulate` function is called, a new frame appears on the screen, so if we call this function repeatedly, the objects in the frame will appear to be moving. This is known as *animation* and the frequency of the calls (i.e. how many times you call `simulate` or, reciprocally, how long you wait in between calls) determines the speed at which the objects appear to be moving. We do this in Javascript using the `setInterval` function. For example, the statement:

```
setInterval(simulate, 15);
```

will cause the `simulate` function to be called every 15 milliseconds.

2.4 Drawing Shapes

It is very easy to draw shapes on a canvas because the process resembles how we, humans, draw. Rather than filling a raster of pixels, we simply trace lines. Given the graphic context of the canvas, you start your drawing by the statement:

```
context.beginPath();
```

Then, determine the shape (rectangle or arc) that you like to draw, e.g.

```
context.rect(x, y, width, height);
```

This will draw a rectangle with upper-left corner at (x,y) and with the specified dimension. Recall that the canvas coordinate system has the x-axis pointing to the right and the y-axis pointing down, with the origin in the upper-left corner of the canvas.

The last step involves either tracing the shape (an outline) or filling it (solid), e.g.

```
   context.stroke();
```

The full function that draws the taget, for example, can be constructed as follows:

```
function drawTarget()
{
   context.beginPath();
   context.lineWidth = "4";
   context.strokeStyle = "red";
   context.rect(target.x, target.y, TARGET_SIZE, TARGET_SIZE);
   context.stroke();
}
```

3. Exercises

As with other labs in this course, lab exercises are broken down into three sections, A, B and C. Exercises in section A are Pre-Lab exercises. All Pre-lab exercises **must** be completed prior to attending your lab. You will not be allowed to participate in the lab if you have not completed these exercised prior to attending the lab. A backup lab is scheduled each week for students who miss/were unprepared for their normal lab time. You will also get much more out of each laboratory if you spend some time going through the B and C exercises for each laboratory before attending your laboratory session.

3A. Pre-lab

1. Launch a browser and visit the ***JR*** site. Click the HTML5 icon next to the Search & Rescue Lab to see the available files.
2. The three files of interest are ***SR.html***, ***SR.css***, and ***SR.js***. Download them (by right-clicking each and choosing Save) into the SR subdirectory in your server folder.
3. Study the contents of the three files and then complete the Moodle quiz associated with this lab on the course Moodle site. You will not receive course credit for the lab without completing this quiz successfully. You are expected to complete this quiz on your own, but you are free to use other resources (e.g., the web, this ePub, and the provided source code to answer the questions).

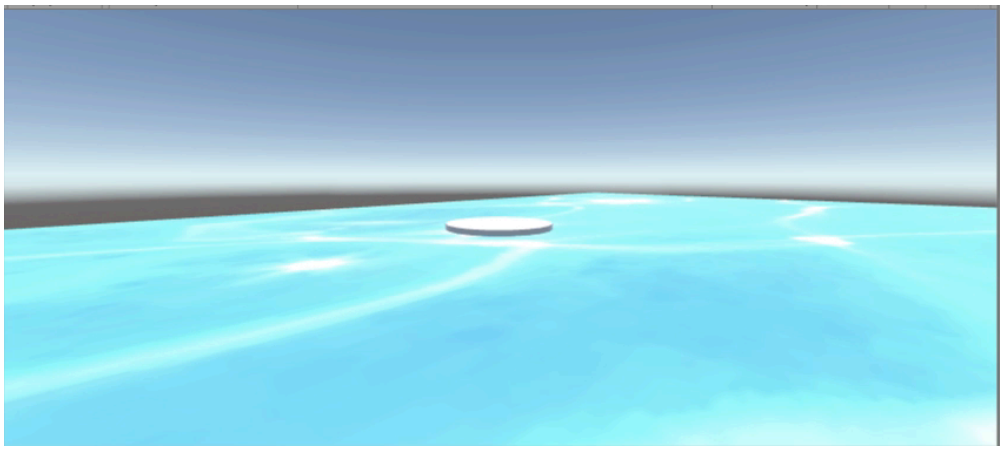
3B. In-lab

1. Obtain a laptop from the lab monitor (if you don't have yours), along with video camera and tripod.
2. Start VBox and launch Atom.

3. Open the SR folder (it was populated with three files as part of the Pre-Lab Exercises).
4. There seems to be a problem in how the rescue vessel detects the boundaries. For example, it seems to *slightly penetrate* the right edge of the search area before bouncing back! Determine which function is responsible, why this is happening, and fix it. The same occurs at the bottom edge.
5. A more serious problem occurs at the left and top edges. In them, the rescue ship keeps going as if the edge is not present. Again, determine for each edge which function is responsible, why this is happening, and fix it.
6. The distance-to-target does not appear in the UI. This informational piece must appear at the bottom of the UI as shown in the opening video of this lab. You will need to revise the HTML file to include it and to revise the Javascript file to update it as the simulation runs. Use *Math.floor()* to strip away the decimal part of the computed distance.
7. The *found()* function always returns *false*! It is supposed to return true when the target is in range. Correct this mistake and verify that the simulation actually ends when the two vessels are close enough.
8. The elapsed time and distance to target values must appear in a distinctive colour. Put them both in the same class (*progress*) and then add a style rule for this class in the CSS so that they appear bold and in a purple shade of your choice. You can specify the shade by specifying its red, green, and blue components.
9. How can you make the animation faster? Modify the code so it re-draws every 5 milliseconds. Capture your simulation in your e-Portfolio.
10. Comment out the *clear()* invocation from the simulate function and re-run the simulation. Explain what you see and capture it in your e-Portfolio.

3C. Advanced

1. The user can override the default height when the simulation starts by entering any desired value. This input, however, is not validated. Add validation so that only a valid entry is accepted. To be valid, the entry has to be positive but not greater than the width of the browser. If not, simply ignore the entry and keep the default width. To that end, you may want to use the *window.innerWidth* property of the window object.
2. Add code to allow the user to also override the height of the canvas.
3. Modify the function that draws the target so that it looks more like a ship. Perhaps a horizontal line and a semicircle below it. You may want to use the *moveTo* and *lineTo* functions.
4. Rather than taking the width (and height) inputs via dialogs, can we do this through an HTML form; i.e. similar to how the radar range was entered? If so, make the necessary changes.
5. The code written here moves an object on the plane and the graphics are not particularly exciting. But with a little bit of work, we can enhance the graphics somewhat. The graphic below shows the sort of imagery that can be accomplished with the simulation developed here



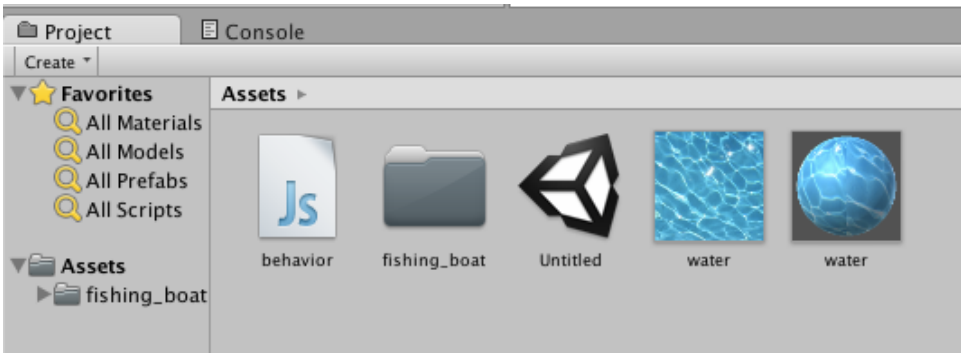
The basic 'trick' here is to use the right tools to make developing this sort of code. Here we are using Unity, a free 3D game engine. Unity provides a simple way of defining 3D graphical entities — here the water, the target (boat) and

the searchlight (a cylinder) — as well as a scripting environment to control things. Unity supports a scripting language that is very similar to JavaScript, so the code written in the lab can be (more or less) be repurposed with few changes to drive the searchlight around the ocean. As an added bonus, Unity can compile to both workstations as well as tablet/phone platforms. In order to do this 'part C', you must be using your own laptop and it must be running an OS supported by Unity (Windows and OSX certainly qualify).

(a) Download and install unity on your computer. This is a reasonably large download.

(b) Download the Unity program from the jr web site. This is a zip file containing the material needed to run the SR code in Unity. We will go through this in detail in a moment, but first open the unzipped material you downloaded using Unity and click on the arrow (run) icon in the middle of the screen. You should get a display similar to the video above.

(c) The code that runs this simulation is in the file behavior.js, which can be found in the 'project' tab. Click on this and it will bring up the JavaScript that runs the simulation



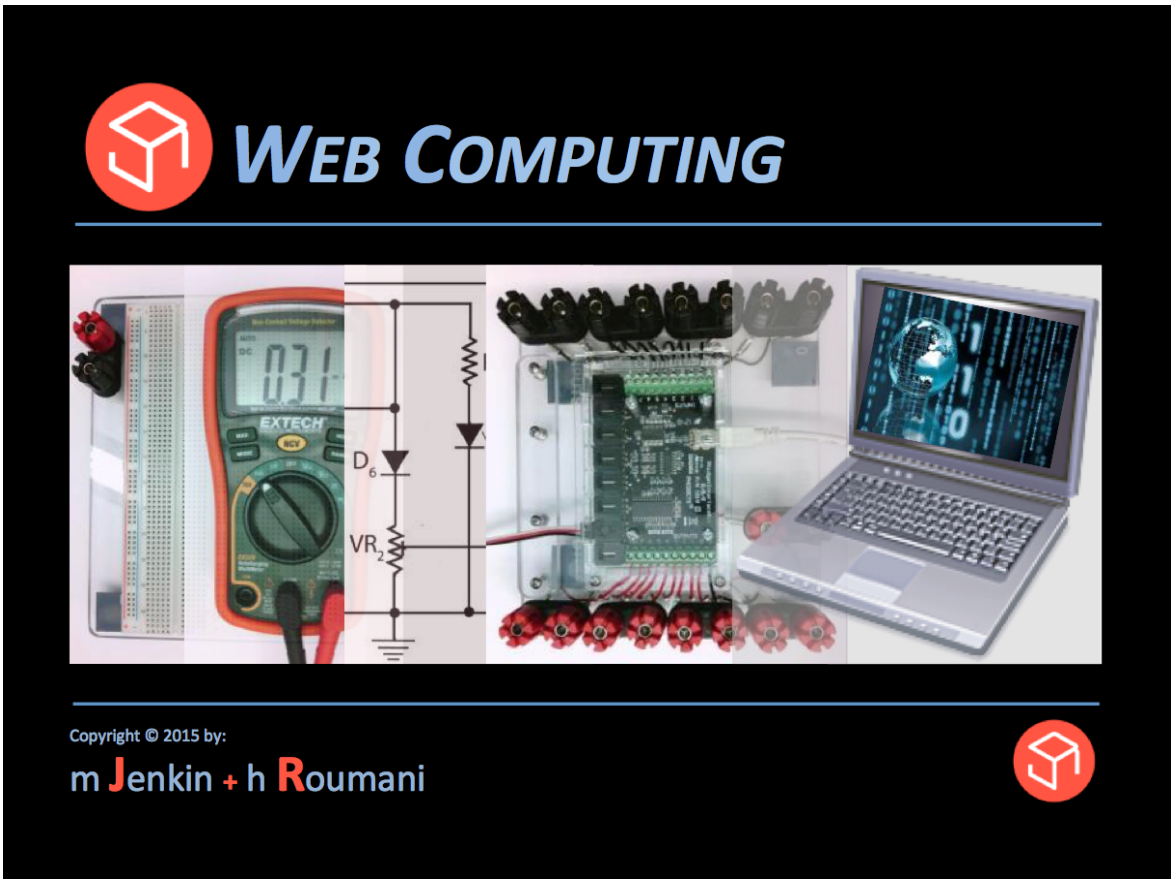
If you double click on behavior.js, this will bring up an editor window within which you can view the code that runs the simulation. The code here is written in UnityScript, which is very similar to JavaScript. Can you identify differences in the code? In UnityScript the start() method is called when your program starts, and the update() method is called every clock tick. Walk through the code and convince yourself that the code works in a manner very similar to that you used in part B above.

Finally, Unity can be used to generate native Android apps. The file sr.apk is a build of the Unity program to run on an Android device. You can find a copy of sr.apk on the jr web page. If you surf to this file from an android device that has 'sideloading' (developer mode) enabled, you will be able to load the program onto your android device. Do this, and run the unity version of the code on your android device.

4. Further Reading

- Tutorials on web technologies.
- Using the HTML5 Canvas.
- A Javascript tutorial.

5. Credits


WEB COMPUTING
EXTECH
D6
VR2
Copyright © 2015 by:
m Jenkin + h Roumani