Problem Set 8: CS50 Shuttle

due by 7:00pm on Fri 11/12

This problem set's form, which will be posted after lecture on Wed 11/10, will ask that you write a few quiz-review questions (and answers), which should take a few minutes, so do set aside time before the problem set's deadline.

Goals.

- Prepare you for final projects.
- Introduce you to JavaScript and third-party APIs.
- Teach you how to teach yourself new languages.

Recommended Reading.

- `http://www.w3schools.com/js/`
- `https://developer.mozilla.org/en/JavaScript/Guide`
- `http://code.google.com/apis/maps/documentation/javascript/tutorial.html`
- `http://code.google.com/apis/earth/documentation/`

NOTICE.

For this problem set, you are welcome and encouraged to consult "outside resources," including books, the Web, strangers, and friends, as you teach yourself more about HTML, CSS, and JavaScript, so long as your work overall is ultimately your own. In other words, there remains a line, even if not precisely defined, between learning from others and presenting the work of others as your own.

You may adopt or adapt snippets of code written by others (whether found in some book, online, or elsewhere), so long as you cite (in the form of CSS, HTML, or JavaScript comments) the origins thereof.

And you may learn from your classmates, so long as moments of counsel do not devolve into "show me your code" or "write this for me." You may not, to be clear, examine the source code of classmates. If in doubt as to the appropriateness of some discussion, contact the staff.

Academic Honesty

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed in writing by the course's instructor. Collaboration in the completion of problem sets is not permitted unless otherwise stated by some problem set's specification.

Viewing or copying another individual's work (even if left by a printer, stored in an executable directory, or accidentally shared in the course's virtual terminal room) or lifting material from a book, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student. Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this course that you have submitted or will submit to another. Nor may you provide or make available solutions to problem sets to individuals who take or may take this course in the future. Moreover, submission of any work that you intend to use outside of the course (*e.g.*, for a job) must be approved by the staff.

You are welcome to discuss the course's material with others in order to better understand it. You may even discuss problem sets with classmates, but you may not share code. In other words, you may communicate with classmates in English, but you may not communicate in, say, C. If in doubt as to the appropriateness of some discussion, contact the course's instructor.

You may turn to the Web for instruction beyond the course's lectures and sections, for references, and for solutions to technical difficulties, but not for outright solutions to problems on problem sets or your own final project. However, failure to cite (as with comments) the origin of any code or technique that you do discover outside of the course's lectures and sections (even while respecting these constraints) and then integrate into your own work may be considered academic dishonesty.

All forms of academic dishonesty are dealt with harshly. If the course refers some matter to the Administrative Board and the outcome for some student is *Warn*, *Admonish*, or *Disciplinary Probation*, the course reserves the right to impose local sanctions on top of that outcome for that student that may include, but not be limited to, a failing grade for work submitted or for the course itself.

Grades.

Your work on this problem set will be evaluated along three primary axes.

Correctness. To what extent is your code consistent with our specifications and free of bugs?
Design. To what extent is your code written well (i.e., clearly, efficiently, elegantly, and/or logically)?
Style. To what extent is your code readable (i.e., commented and indented with variables aptly named)?

Getting Started.

☐ OMG, last problem set.

☐ If you haven't already, consider downloading and installing **Firefox** and **Firebug** (in that order), both of which are available on the course's website under **Software**. Once installed, Firebug will appear as an option in Firefox's **Tools** menu. Anytime you'd like to print some debugging information to your browser's window, a la `printf` in C, you can include a line like

```
console.log("hello, world");
```

in your JavaScript code. So long as Firebug's window is open, you'll see that text in its console. Be sure to remove any such lines before submitting your work. Alternatively, you can include a line like

```
alert("hello, world");
```

in your JavaScript code instead, but the pop-up that results tends to be more annoying, especially if you accidentally call `alert` in a loop!

☐ For this problem set, your work must ultimately behave the same on at least two major browsers:

- ☐ Chrome 5.x
- ☐ Firefox 3.x
- ☐ Internet Explorer 8.x
- ☐ Opera 10.x
- ☐ Safari 5.x

Be sure, then, to test your work thoroughly with at least two browsers. It is fine, though, to rely on just one operating system. And it's fine if you notice slight aesthetic differences between the two browsers. Make sure that your teaching fellow knows which browsers to use whilst evaluating your work.

☐ Head to the URL below.

```
http://earth.google.com/plugin/
```

If prompted to download the Google Earth Plugin, do go ahead and proceed to install.[1] Once the plugin's installed, you may need to reload that page or restart your browser, but you should ultimately see a 3D Earth embedded in the page.[2] Close the page once you do.

[1] Note that the Google Earth Plugin and, in turn, this problem set require that you meet the system and browser requirements documented at `http://maps.google.com/support/bin/answer.py?hl=en&answer=166094`. If you do not, email `sysadmins@cs50.net`. You may be able to run a supported version of Windows in a "virtual machine" on your computer.

[2] If informed that "you do not have permission to use this service over SSL," odds are you have Force-TLS or HTTPS Everywhere installed; you'll want to disable any such plugin temporarily, at least for `www.google.com`, which is where the Google Earth API lives.

Incidentally, best to minimize the number of programs and windows you have open while working on this problem set; the Google Earth Plugin likes to consume CPU cycles and RAM. In fact, if you ever feel your computer slowing down, quit some programs or even reboot (after saving your work).

☐ SSH to `cloud.cs50.net` and recursively copy `~cs50/pub/src/psets/pset8/` into your own `~/public_html/` directory. (Remember how?) Then `cd` to `~/public_html/pset8/`. (Remember how?) Then run `ls`. You should see the below.

```
buildings.js  math3d.js   passengers.js  service.js
index.html    passengers  service.css    shuttle.js
```

All of the work that you do for this problem set will reside in `~/public_html/pset8/`. Go ahead and `chmod` your files as follows. (Remember how?)

- ☐ Your `~/public_html/pset8/` directory should be readable, writable, and executable by you but only executable by everyone else (*i.e.*, 711).
- ☐ All CSS (`.css`), HTML (`.html`), and JavaScript (`.js`) files should be readable and writable by you but only readable by everyone else (*i.e.*, 644).

As for `~/public_html/pset8/passengers/`, no need to `chmod` it, as it's actually a symlink (so that we don't waste space by making copies of all the JPEGs within).

Now create a Mercurial repository for your code so that you can back up your code as you work by executing the following commands:

```
cd ~/public_html/pset8/
hg init
hg add
hg commit
```

Provide a commit message with `nano` when prompted, then save and quit. Anytime you make non-trivial changes to your code, re-run

```
hg commit
```

to commit the revision so that, just in case, you can roll back in time. See Problem Set 6 for other handy commands!

Now head to the URL below, where `username` is your own username.[3]

```
http://cloud.cs50.net/~username/pset8/
```

You should see University Hall.

[3] Note that this URL is equivalent to `http://cloud.cs50.net/~username/pset8/index.html`.

Shuttletime.

☐ Your mission for this problem set is to implement your own shuttle service that picks up passengers all over campus and drops them off at their houses. Your shuttle is already equipped with an engine, a steering wheel, and 35 seats. Shall we go for a spin? Here's how to drive with your keyboard:[4]

Move Forward: W
Move Backward: S
Turn Left: left arrow
Turn Right: right arrow
Slide Left: A
Slide Right: D
Look Downward: down arrow
Look Upward: up arrow

Note that your mouse is not used for driving in this 3D world. In fact, don't even click on it, else you'll take away "focus" from our (and, soon, your) JavaScript code, the result of which is that the shuttle will no longer respond to your keystrokes. If that does happen, simply click the 2D map or anything above it (within the same window) to give focus back to the code; the shuttle should then respond to your keystrokes again.

Anyhow, go for a ride! (No bikes in the Yard, but shuttles are okay.) As you approach buildings, you may find that they get prettier and prettier as more satellite imagery is automatically downloaded. See if you can make your way to your own house. (Try not to take any shortcuts through buildings.) Along the way, you'll likely see some familiar faces. Those will soon be your passengers!

In fact, go ahead and click a few times the minus (–) sign in the top-left corner of the application's 2D map. You should see more and more red markers as you zoom out, each of which represents a passenger waiting for pickup. If you hover over each marker with your mouse, you'll see each passenger's name and origin. The blue bus, of course, represents you! You're welcome to click and drag the 2D map to see more of campus; as soon as you start driving again, the map will be re-centered around you.

Above the 2D map is a list of your 35 seats, each of which is currently empty. Above that, in the application's top-right corner, are two buttons: **Pick Up** and **Drop Off**. Neither works yet, but both will before long!

If you happen to get lost, just reload the page, and you'll be returned to University Hall. Your soon-to-be passengers will also be pseudorandomly repositioned throughout campus.

[4] Indeed, your shuttle can slide left and right, as though all four wheels can turn 90 degrees. As for looking downward and upward, know that you can look straight ahead down to your shuttle's toes, then back up; you can't looker higher than straight ahead. (Sun visor's in the way.)

☐ Alright, let's take a stroll through this problem set's distro. Open up `index.html` first and read over the lines of HTML within. Notice first how this file references several others in its head, namely `services.css`, `math3d.js`, `buildings.js`, `passengers.js`, `shuttle.js`, and `service.js`, as well as the URL of Google's JavaScript API. Notice next how the `body` tag has a few event handlers, each of whose implementations lives, as we'll soon see, in `service.js`. And notice how each of the `div` elements has an `id` so that we can stylize it with CSS or access it via JavaScript.

At the moment, the HTML within two of those `div` elements (`#announcements` and `#seats`) is meant to be temporary. Eventually, there'll be some announcements, and there will be passengers in seats!

Next open up `service.css`, which stylizes that HTML. You don't need to understand all of the CSS within, but do know in general that `div#foo`, refers to the `div` element whose `id` is `foo`.

Now take a peek at `buildings.js`. Declared in that file is `BUILDINGS`, which is an array of objects, each of which represents a building on campus. Associated with each building is a "root" (Harvardspeak for a building's ID), a name and address, and a pair of coordinates that represents an edge of that building. We could have declared this array in `service.js`, but because it's so big, we decided to isolate it in its own file.

Similarly do passengers have their own file. In `passengers.js` is `PASSENGERS`, another array of objects, each of which this time represents a passenger on campus. Associated with each passenger is a username (*i.e.*, a unique identifier), a name, and a house. Incidentally, each of those usernames maps to a similarly named JPEG in `pset8/passengers/`. Note, though, that some houses comprise multiple buildings, and so even though "Mather House" appears in both `passengers.js` and `buildings.js`, "Quincy House" appears only in `passengers.js`. In `buildings.js`, meanwhile, are objects for "Quincy House New Residence Hall," "Quincy House Library," and (nice and confusingly) "Mather Hall, Quincy House." And so you will find in `houses.js` a third (and final!) data structure, this one an object whose keys are houses' names and whose values are objects representing houses' coordinates. And so you can access the best house's coordinates with something like:

```
var lat = HOUSES["Mather House"].lat;
var lng = HOUSES["Mather House"].lng;
```

You may assume that the coordinates in `houses.js` are the official coordinates for passengers' destinations. In fact, we've already planted yellow markers at those very coordinates on the 2D map to help out the driver. We've not bothered planting placemarks at those coordinates on the 3D Earth, since they're not as easy to spot.

Incidentally, if, during the course of this problem set, you'd like to look up where some building is on campus, feel free to search for it via CS50's own app:

`http://maps.cs50.net/`

Now take a peek at `shuttle.js`. This file's a bit fancy, inasmuch as it effectively implements a data structure called `Shuttle`.[5] You don't need to understand this file's entirety, but know that inside of a `Shuttle` are two properties: `position`, which encapsulates a shuttle's position (including its longitude and latitude), and `states`, which encapsulates a shuttle's various states.

Okay, now turn your attention to `service.js`, where you'll do most of your work, though you're welcome to modify any of this problem set's files, except for `math3d.js` (which is just a library), `buildings.js`, `houses.js`, and `passengers.js`. Atop `service.js` are a bunch of constants and globals. Just below those lines are two calls to `google.load`, which loads the two APIs on which this application relies:

Google Maps JavaScript API V3
`http://code.google.com/apis/maps/documentation/javascript/tutorial.html`

Google Earth API
`http://code.google.com/apis/earth/documentation/`

You won't need to read through the entirety of that documentation; odds are you'll explore it as needed. But do read the first page or so of each to get a sense of what each API does.

Implemented in `service.js` are all of those event handlers you first saw in `index.html`. Scroll down to the implementation of `load` first. It's this function that embeds the 2D map and 3D Earth map in your browser. Next scroll back up to the implementation of `initCB`; this function gets called as soon as the Google Earth Plugin has loaded, and so it's in this function that we finish initializing the app. (If something goes wrong, it's `failureCB` that's instead called.) Notice, in particular, that `initCB` "instantiates" an object, `shuttle`, of type `Shuttle`.

Next take a look at the function called `keystroke`. It's this function that handles your keystrokes. Odds are this function will bring back memories of `sudoku`, though this time we used `if` and `else` instead of a `switch`. In response to your keystrokes, this function changes the state of the shuttle simply by updating the appropriate property in `shuttle.states` with a value of `true` or `false`.

Finally, take a peek at the function called `populate`. It's this function that plants friendly faces all over campus. Each face is implemented as a "placemark" on the 3D earth and as a "marker" on the 2D map.

No need for any PHP or SQL for this problem set, just CSS, HTML, and JavaScript. Be sure, then, to reload your browser (or clear your cache) often so that you always have the latest versions of your code.[6]

[5] Although Shuttle behaves like a "class," JavaScript is actually a "prototype-based" language:
`http://en.wikipedia.org/wiki/Prototype-based`
[6] In particular, you may want to force-reload your browser often, as by holding Shift when you reload.

☐ Alright, it's time start picking passengers up. Recall from `index.html` that, when the button labeled **Pick Up** is clicked, the function called `pickup` in `service.js` is called. Implement pick-ups as follows.

If the button is clicked when the shuttle is within 15.0 meters of a passenger and at least one seat is empty, that passenger should be given a seat on the shuttle and removed from both the 2D map and 3D earth. (If multiple passengers are within 15.0 meters, you should pick up as many of them as there are empty seats.) If the shuttle is not within 15.0 meters of any passenger, make an announcement to that effect. If the shuttle is within range of some passenger but no seats are free, make an announcement to that effect (and don't pick up the passenger). Any such announcements should be cleared (or replaced with some default text) as soon as the shuttle starts moving.

Not sure how to proceed? That's part of the challenge! Odds are you'll encounter similar uncertainty when you dive into your final project. Whenever in doubt, peruse the APIs' documentation, and turn to Google (the search engine) for tips. And you are encouraged to turn to `help.cs50.net` for guidance from classmates and staff. But we'll give you some hints to get you started on pick-ups.

To calculate the shuttle's distance, `d`, from some point (`lat`, `lng`), you'll probably want something like:

```
var d = shuttle.distance(lat, lng);
```

To remove a placemark, `p`, from the 3D Earth, you'll probably want something like:

```
var features = earth.getFeatures();
features.removeChild(p);
```

To remove a marker, `m`, from the 2D map, you'll probably want something like:

```
m.setMap(null);
```

Unfortunately, `populate`, at the moment, doesn't remember the placemarks or markers that it plants, so you may need to tweak populate as well. Perhaps you could store them in some array(s) or object(s)?

How, now, to give a picked-up passenger a seat? Odds are you'll want an array of size 35 to represent all those seats. And odds are you'll want to update `div#seats` anytime that array is updated. Recall that you can update the HTML inside an element with something like:

```
document.getElementById("seats").innerHTML = "hello, world";
```

Although `index.html` currently uses an ordered list for seats, you're welcome to present them however you'd like. Be sure, though, to list passengers' names and houses, so that you know who and where to drop off.

As for making announcements, similar code will get the job done:

```
document.getElementById("announcements").innerHTML = "hello, world";
```

☐ Alright, it's now time to implement drop-offs! Recall from `buildings.js` that each passenger lives in a house. And recall from `index.html` that, when the button labeled **Drop Off** is clicked, the function called `dropoff` in `service.js` is called. Implement drop-offs as follows.

If the button is clicked when the shuttle is within 30.0 meters of an on-board passenger's house, that passenger should be dropped of by emptying their seat. (If there are multiple passengers on board that live in that house, all should be dropped off in this manner.) No need to re-plant a placemark or marker for dropped-off passengers; assume they're going to head straight inside. If the shuttle is not within 30.0 meters of any passenger's house, make an announcement to that effect. Any such announcement should be cleared (or replaced with some default text) as soon as the shuttle starts moving.

☐ Nice work so far! Suffice it to say this shuttle service is starting to feel like a game. It's time to allow you some creativity. (Think back to your Scratch days!) Implement any one (1) of the features below.

- ☐ Implement points, whereby the driver earns one point for each passenger dropped off. Be sure to announce the driver's score anytime it changes. And be sure to announce when every single passenger has been picked up and dropped off.
- ☐ Implement a timer, whereby the driver only has some number of minutes or seconds in order to pick up and drop off some number of passengers. Odds are you'll find `window.setInterval` and/or `window.setTimeout` of interest.[7,8]
- ☐ Rather than just list seated passengers' names and houses, group them somehow by house so that it's obvious how many of your 35 or fewer seated passengers are destined for a particular house.
- ☐ Implement the Konami Code in such a way that, if inputted, the shuttle acquires the ability to "fly." But it can only pick up and drop off passengers with its wheels on the ground. We leave it to you to decide which keystrokes to use for flight.
- ☐ Replace the blue bus on the 2D map with some kind of arrow that points in the direction that the shuttle is actually facing (in the 3D world). Odds are you'll want to rotate a `canvas` element or use as many as 360 different icons (one for each degree). If you take the latter approach, odds are you can get away with fewer icons, one every few degrees.
- ☐ Implement the ability to teleport the shuttle instantly to any building on campus, as by selecting that building's name from a menu.
- ☐ Implement the ability to increase or decrease the shuttle's velocity.

[7] `https://developer.mozilla.org/en/DOM/window.setInterval`
[8] `https://developer.mozilla.org/en/DOM/window.setTimeout`

- ☐ Ensure that passengers do not get pseudorandomly positioned by `populate` at their own houses, lest the driver get annoyed that they only want to travel a few feet.
- ☐ Implement the ability to take joy rides across the campuses of friends' universities and your hometown (then invite your friends and family to get behind your wheel!).
- ☐ Implement the ability to inform passengers, as via an announcement, of the shuttle's current location. Odds are you'll find `http://code.google.com/apis/maps/documentation/geocoding/#ReverseGeocoding` of interest.
- ☐ Implement auto-pilot whereby, when some button or link is clicked, the shuttle drives (without teleporting) itself to a passenger or house. It's fine if auto-pilot likes to drive through buildings.
- ☐ Implement fuel, whereby the shuttle only starts with a finite amount and can only travel some number of meters before it runs out. Build one or more gas stations on campus at which the shuttle can refuel (as by clicking a button or link when nearby or driving through the station).
- ☐ Make-your-own feature. If you would like to implement a feature not listed here but that involves similar effort, you may do so long as your teaching fellow approves in advance.

Although you are only required to implement one of the features above, you are welcome to impress us (and your friends) with more just for fun! And you are welcome to alter your user interface's aesthetics, including your 2D map's icons.[9]

Sanity Checks.

Before you consider this problem set done, best to ask yourself these questions and then go back and improve your code as needed! Do not consider the below an exhaustive list of expectations, though, just some helpful reminders. The checkboxes that have come before these represent the exhaustive list! To be clear, consider the questions below rhetorical. No need to answer them in writing for us, since all of your answers should be "yes!"

- ☐ Do you only pick up passengers if they're within 15.0 meters of the shuttle?
- ☐ Do you only drop off passengers if they're within 30.0 meters of their house?
- ☐ If multiple passengers are within a prescribed radius, do you handle them properly?
- ☐ Did you implement at least one additional feature?

As always, if you can't answer "yes" to one or more of the above because you're having some trouble, do turn to `help.cs50.net`!

[9] See `http://code.google.com/p/google-maps-icons/` for some fun icons.

How to Submit.

In order to submit this problem set, you must first execute a command on `cloud.cs50.net` and then submit a (brief) form online; the latter will be posted after lecture on Wed 11/10.

☐ SSH to `cloud.cs50.net`, if not already there, and ensure that your implementation of CS50 Shuttle is in `~/public_html/pset8/`. Then submit your work by executing the command below.

```
~cs50/pub/bin/submit pset8
```

You'll know that the command worked if you are informed that your "work HAS been submitted." If you instead encounter an error that doesn't appear to be a mistake on your part, do try running the command one or more additional times. You may re-submit as many times as you'd like; each resubmission will overwrite any previous submission. But take care not to re-submit after the problem set's deadline, as only your latest submission's timestamp is retained.

For simplicity, your TF may want to examine your code in situ, so don't modify your work even after you submit without first checking with your TF.

☐ Anytime after lecture on Wed 11/10 but before this problem set's deadline, head to the URL below where a short form awaits:

```
http://www.cs50.net/psets/8/
```

If not already logged in, you'll be prompted to log into the course's website.

Once you have submitted that form (as well as your source code), you are done!

This was Problem Set 8. Your last!