Problem Set 8: CS50 Shuttle

Table of Contents

- Objectives
- Recommended Reading
- Academic Honesty
  - Reasonable
  - Not Reasonable
- Scores
- Getting Started
- Shuttletime
  - Guided Tour
  - pickup, chart
  - dropoff
  - extra features
- Sanity Checks
- How to Submit
  - Step 1 of 2
  - Step 2 of 2

due ~~Thu 11/14~~ Fri 11/15 at noon

No need to submit any work early (i.e., by Wed) this week in order to get an extra day. Everyone gets an extra day this week; everyone’s deadline is Fri 11/15. No more coupon codes!

Questions? Head to cs50.net/discuss (https://www.cs50.net/discuss) or join classmates at office hours (https://www.cs50.net/ohs)!

Objectives

- Introduce you to JavaScript and third-party APIs.
- Prepare you for final projects.

Recommended Reading

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide)

Academic Honesty

This course’s philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course’s material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy’s other constraints. Collaboration on quizzes is not permitted at all. Collaboration on the course’s final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it until you solicit and receive approval in writing from the course's heads. Acts considered not reasonable by the course are handled harshly. If the course refers some matter to the Administrative Board and the outcome is Admonish, Probation, Requirement to Withdraw, or Recommendation to Dismiss, the course reserves the right to impose local sanctions on top of that outcome that may include an unsatisfactory or failing grade for work submitted or for the course itself.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code at Office Hours, elsewhere, or even online, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Reviewing past semesters' quizzes and solutions thereto.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on CS50 Discuss or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual's work during a quiz.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Redeeming or attempting to redeem someone else's code for a late day.
- Searching for, soliciting, or viewing a quiz's questions or answers prior to taking the quiz.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Submitting work to this course that you intend to use outside of the course (e.g., for a job) without prior approval from the course's heads.
- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Scores

Your work on this problem set will be evaluated along four axes primarily.

Scope

To what extent does your code implement the features required by our specification?

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)?

All students, whether taking the course SAT/UNS or for a letter grade, must ordinarily submit this and all other problem sets to be eligible for a satisfactory grade unless granted an exception in writing by the course's heads.

Getting Started

- OMG, last problem set.
- If you don't have it installed on your computer already (outside of the appliance, not inside of the appliance), download and install the latest version of Chrome from http://ww.google.com/chrome/ (http://ww.google.com/chrome/).
- Then, using Chrome, download and install the Google Earth Plug-in from http://www.google.com/earth/explore/products/plugin.html (http://www.google.com/earth/explore/products/plugin.html). Note that the plug-in requires Mac OS or Windows, unfortunately. If running Linux or some other operating system, email mailto:sysadmins@cs50.net if you don't have access to Mac OS or Windows; you may be able to run Windows inside of a virtual machine.

  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. Close the page once you do.

  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, you might want to quit some programs or even reboot (after saving your work).
- Next, start up your appliance and, upon reaching John Harvard's desktop, open a terminal window (remember how?) and execute

  ```
  update50
  ```

  to ensure that your appliance is up-to-date!
- Like Problem Set 7, this problem set comes with some distribution code that you'll need to download before getting started. Go ahead and execute

  ```
  cd ~/vhosts/localhost/public
  ```

  in order to navigate to your `~/vhosts/localhost/public` directory. Then execute

  ```
  wget http://cdn.cs50.net/2013/fall/psets/8/pset8/pset8.zip
  ```

  in order to download a ZIP (i.e., compressed version) of this problem set's distro. If you then execute

  ```
  ls
  ```

  you should see that you now have a file called `pset8.zip` in your `~/vhosts/localhost/public` directory (in addition to `index.php`, which was already there). Unzip it by executing the below.

  ```
  unzip pset8.zip
  ```

  If you again execute

```
ls
```

you should see that you now also have a directory called `pset8`. You're now welcome to delete the ZIP file with the below.

```
rm -f pset8.zip
```

If you next execute

```
cd pset8
```

followed by

```
ls
```

you should see that `pset8` contains `index.html` and four directories (`css`, `fonts`, and `img`, and `js`). Ensure that all files inside of `pset8` (and its descendents) are world-readable by executing

```
find . -type f | xargs chmod 644
```

so that the appliance's web server (and you, from a browser) will be able to access your work. (You could use `chmod` alone in the usual way, but piping the output of `find` to `xargs` in this way is less tedious!) Then ensure that all directories inside of `pset8` (at any depth) are world-executable by executing the below.

```
find . -type d | xargs chmod 711
```

Finally, ensure that a few other directories are (still) world-executable by executing the below.

```
chmod 711 ~
chmod 711 ~/vhosts
chmod 711 ~/vhosts/localhost
chmod 711 ~/vhosts/localhost/public
```

If you opt to create any other files or directories for this problem set, keep in mind that:

- any files should probably be readable and writable by you but only readable by everyone else (i.e., `644`); and
- any directories should probably be world-executable (i.e., `711`).

Unlike Problem Set 7, your code for Problem Set 8 will indeed live in a virtual host (aka vhost) called `localhost` rather than one called, e.g., `pset8`. Using `localhost` will make it easier to access your code from Chrome on your own computer (outside of the appliance).

- Even though your code for this problem set will indeed live in `~/vhosts/localhost/public`, let's ensure that it's nonetheless backed up via Dropbox, assuming you set up Dropbox inside of the appliance. In a terminal window, execute

```
ln -s ~/vhosts/localhost/public/pset8 ~/Dropbox
```

in order to create a "symbolic link" (i.e., alias or shortcut) to your `~/vhosts/localhost/public/pset8` directory within your `~/Dropbox` directory so that Dropbox knows to start backing it up.

- Now head to the URL below using a browser on your own computer (outside of the CS50 Appliance), where `w.x.y.z` is your appliance's IP address (which should be displayed in the appliance's bottom-right corner), and each of `w`, `x`, `y`, and `z` is thus a number between `0` and `255`:

http://w.x.y.z/pset8/ (http://w.x.y.z/pset8/)

You should see University Hall! If not, do double-check that you've not missed or mistyped a step. In particular, leverage `cd` and `ls -l` to check the permissions of every directory and file. And open up Chrome's Developer Tools, as with shift-control-i, specifically its Console tab, to see if you see any red errors. And take care not to visit http://localhost/pset8/ (http://localhost/pset8/) using Chrome outside of the appliance, since `localhost`, in that context, will refer to your own computer, not the appliance.

- Incidentally, anytime you'd like to print some debugging information to the **Console** tab of Chrome's **Debugging Tools** during this problem set, a la `printf` in C, know that you can include a line like

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

  in your JavaScript code. So long as Chrome's **Console** is open, you'll see that text. Just be sure to remove any such lines before submitting your work. Do keep an eye on that **Console** during development, too, as syntax errors and more tend to appear there in red. Alternatively, you can include a line like

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

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

Shuttletime

Guided Tour

- 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 some seats. Shall we go for a spin? Assuming you're still at http://w.x.y.z/pset8/ (http://w.x.y.z/pset8/) looking at University Hall, here's how to drive with your keyboard once you've clicked **Start Engine**:

Move Forward	W
Move Backward	S
Turn Left	←
Turn Right	→
Slide Left	A
Slide Right	D
Look Downward	↓
Look Upward	↑

  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 by holding ↓ for a few seconds, then back up; you can't look higher than straight ahead. (Sun visor's in the way.)

  Note that your mouse is not used for driving in this 3D world. In fact, don't even click on it, else you might 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 **Start Engine** 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 shuttle's seats, each of which is initially 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.

- Alright, let's take a stroll through this problem set's distribution code. Open up `index.html` first and read over the lines of HTML within. Notice first how this file references some (local) CSS and JavaScript files in its `head` as well as the URL of Google's JavaScript API. Notice how most every one of the `div` elements in the page's `body` is uniquely identified with an `id` attribute (so that we can stylize it with CSS or access it via JavaScript).

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

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

  Now take a peek at `js/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 unique identifier), 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 `js/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 `img`. Note, though, that some houses comprise multiple buildings, and so even though `"Mather House"` appears in both `js/passengers.js` and `js/buildings.js`, `"Quincy House"` appears only in `passengers.js`. In `js/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 `js/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/ (http://maps.cs50.net/)

  Now take a peek at `js/shuttle.js`. This file's a bit fancy, inasmuch as it effectively implements a data structure called `Shuttle`. (Although Shuttle behaves like a "class" (if familiar), JavaScript is actually a "prototype-based" language, per http://en.wikipedia.org/wiki/Prototype-based (http://en.wikipedia.org/wiki/Prototype-based).) 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 `js/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 `js/math3d.js` (which is just a library), `js/buildings.js`, `js/houses.js`, and `js/passengers.js`. Atop `js/service.js` are a bunch of constants and globals followed by two calls to `google.load`, which loads the two APIs on which this application relies:

  - Google Maps JavaScript API v3 (https://developers.google.com/maps/documentation/javascript/tutorial)
  - Google Earth API (https://developers.google.com/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.

  Below those calls to `google.load`, meanwhile, is some code that will "register" some "event listeners" once a browser has loaded `index.html`. Also called is `load`, which is implemented further down in `js/service.js`. Take a look, and you'll see that `load` embeds the 2D map and 3D Earth map in your browser. Next scroll back up to the implementation of `initCB`; this function, a "callback," 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`, another callback, 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. 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`.

Now 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.

Finally, take a look at the function called `chart`. That function renders a seating chart (in the `div` whose `id` is `chart`) as an ordered HTML list (0-indexed for your debugging convenience) for the shuttle's seats. Notice how it iterates over `shuttle.seats`, an array that's initialized to be of size `SEATS` (which is defined atop `js/service.js`) in `js/shuttle.js`. Note that `null` represents an empty seat!

- By the way, 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. And you may want to hold Shift when reloading your browser to ensure it re-downloads files that might have been cached.

pickup, chart

- 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 `js/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 cs50.net/discuss (https://www.cs50.net/discuss) for guidance from classmates and staff. But allow us to offer 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, per its `TODO`. Perhaps you could store those placemarks and markers in some global array(s) and/or object(s) so that you can remove them from the 3D Earth and 2D map, respectively, later?

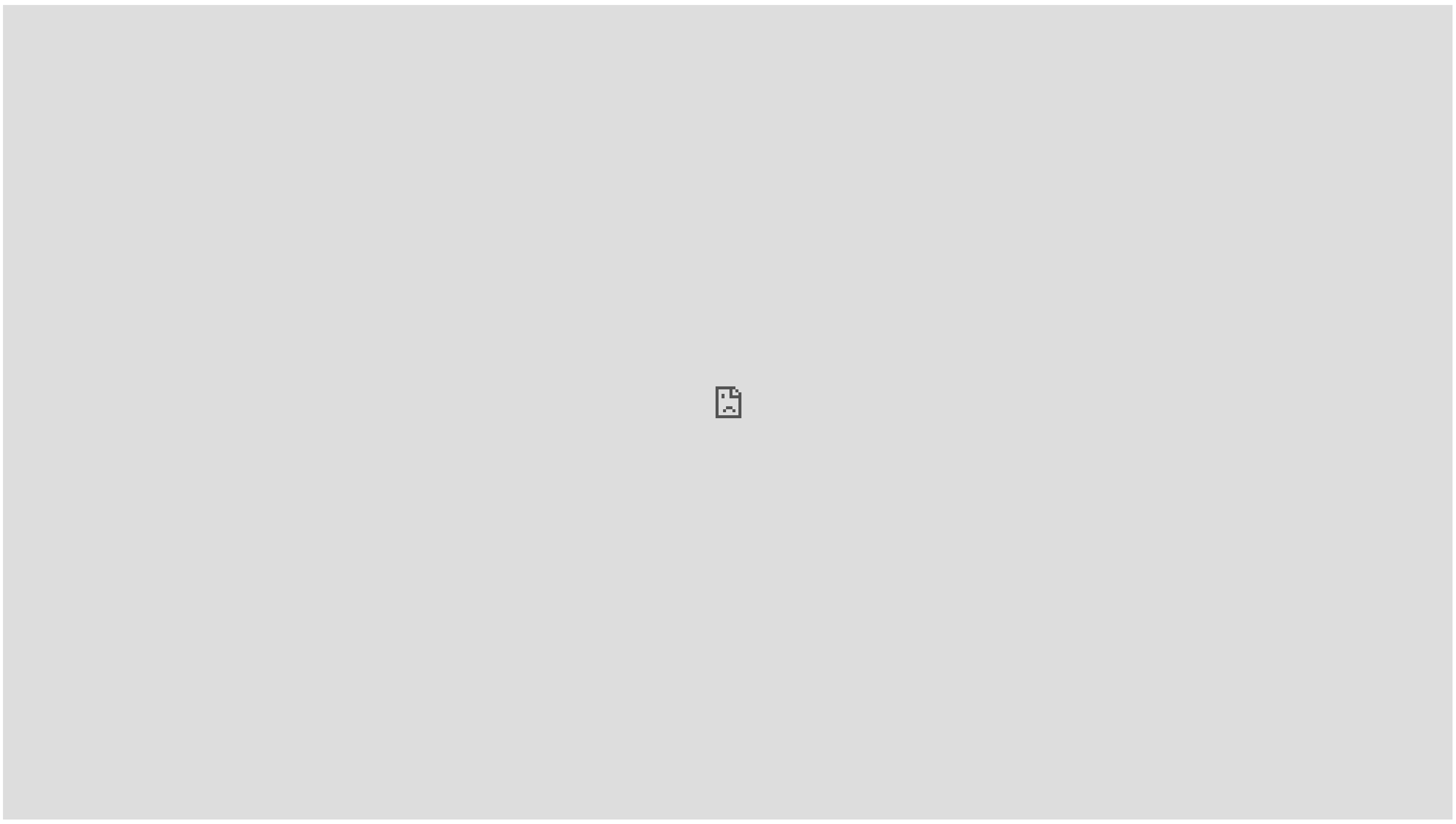
  And how to give a picked-up passenger a seat? Odds are you'll want to update `shuttle.seats`, but be sure not to add extra seats. And odds are you'll want to update `#seats` anytime that array is updated by calling `chart`. But, at the moment, `chart` only knows how to display empty seats; be sure to replace its `TODO` with some code that displays picked-up passengers names and houses, so that you know who and where to drop off.

  As for making announcements, recall that (jQuery) code like the below will get the job done:

  ```
  $("#announcements").html("hello, world");
  ```

  Incidentally, be sure not to let drivers pick up any freshmen (i.e., anyone whose home isn't in `js/houses.js`)!

  Here's Zamyla with some additional advice!



dropoff

- Alright, it’s now time to implement drop-offs! Recall from `js/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 `js/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.)
  - If the shuttle is not within 30.0 meters of any passenger’s house, you should 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.
  - No need to re-plant a placemark or marker for dropped-off passengers; assume they’re going to head straight inside. As for passengers who’re still on board after a drop-off, be sure not to make them change seats (as by changing their indices in some array) just because some other seat is now empty.

  Here’s Zamyla again:

extra features

- 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 three (3) of the features below.

  a. 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.

  b. 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` (https://developer.mozilla.org/en-US/docs/Web/API/window.setInterval) and/or `window.setTimeout` (https://developer.mozilla.org/en-US/docs/Web/API/window.setTimeout) of interest.

  c. Rather than just list seated passengers' names and houses, group them somehow by house so that it’s obvious how many of your seated passengers are destined for a particular house.

  d. Design a (more) beautiful seating chart by associating a color and/or logo with each house (and each passenger in it), as via a CSS class per house.

  e. Implement the Konami Code (http://en.wikipedia.org/wiki/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.

  f. 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.

  g. Implement the ability to teleport the shuttle instantly to any building on campus, as by selecting that building’s name from a menu.

  h. Implement the ability to teleport the shuttle instantly to any location on Earth, as by typing an address into a text field. Odds are you’ll find https://developers.google.com/maps/documentation/geocoding/#GeocodingRequests (https://developers.google.com/maps/documentation/geocoding/#GeocodingRequests) of interest.

  i. Implement the ability to increase or decrease the shuttle’s velocity.

  j. 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.

  k. 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 (http://code.google.com/apis/maps/documentation/geocoding/#ReverseGeocoding) of interest.

l. 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.

m. 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).

n. Make-your-own feature. If you would like to implement a feature not prescribed 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 three 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; see http://mapicons.nicolasmollet.com/ (http://mapicons.nicolasmollet.com/) for some fun icons.

Here’s Zamyla with some final thoughts!

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 pick up passengers if there are seats empty?
- Do you only pick up passengers if they’re not freshmen?
- Does your seating chart display passengers' names and houses for non-empty seats?
- 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 three additional features?

As always, if you can’t answer "yes" to one or more of the above because you’re having some trouble, do turn to cs50.net/discuss (https://www.cs50.net/discuss).

How to Submit

Step 1 of 2

- Create a ZIP (i.e., compressed) file containing your entire `pset8` directory by executing the below. Incidentally, `-r` means "recursive," which in this case means to ZIP up everything inside of `pset8`, including any subdirectories (or even subsubdirectories!).

  ```
  cd ~/vhosts/localhost/public
  zip -r pset8.zip pset8/
  ```

  If you type `ls` thereafter, you should see that you have a new file called `pset8.zip` in `~/vhosts/localhost/public`. (If you realize later that you need to make a change to some file and re-ZIP everything, you can delete the ZIP file you already made with `rm pset8.zip`, then create a new ZIP file as before.)

- Once done creating your ZIP file, open up Chrome *inside* of the appliance (not on your own computer) and visit cs50.net/submit (https://www.cs50.net/submit), logging in if prompted.

- Click **Submit** toward the window's top-left corner.

- Under **pset8** on the screen that appears, click **Upload New Submission**.

- On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.

- Navigate your way to `pset8.zip`, as by clicking **jharvard**, then double-clicking **Dropbox**. Once you find `pset8.zip`, click it once to select it, then click **Open**.

- Click **Start upload** to upload your ZIP file to CS50's servers.

- On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to cs50.net/submit (https://www.cs50.net/submit) and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

- Head to https://forms.cs50.net/2013/fall/psets/8/ (https://forms.cs50.net/2013/fall/psets/8/) where a short form awaits. Once you have submitted that form (as well as your source code), you are done!

  This was Problem Set 8, your last!!!

Last updated 2013-11-13 20:54:22 EST