

PhET Development Overview

Overview

PhET Interactive Simulations creates free, open source educational simulations in science and math, which you can find at [the PhET website](#). This document explains PhET's libraries, practices and patterns for developing interactive simulations in HTML5. This document is also available at <http://bit.ly/phet-development-overview>. For discussion and feedback, please visit the [Developing Interactive Simulations in HTML5 Google Group](#).

[Overview](#)

[Getting Started \(for Windows\)](#)

[Getting Started \(for OS X\)](#)

[Building and Testing \(Windows & OS X\)](#)

[Source code and dependencies](#)

[Checking out the HTML5 code from GitHub](#)

[Master is Unstable: Accessing Rigorously Tested Code](#)

[Original Java/Flash source code](#)

[3rd Party Dependencies](#)

[Licenses](#)

[Coding Style Guidelines](#)

[Platforms](#)

[Modularity & RequireJS](#)

[Layout](#)

[Compiling the Code](#)

[Offline Operation](#)

[Published Versions](#)

[Development Process & Checklist](#)

[Utilities and Instrumentation for Development and Testing](#)

[Working with GitHub Issues](#)

[Embedding a Simulation in your website](#)

Getting Started (on Windows)

In order to get the code for an existing PhET simulation, you will need to follow several steps (also shown in the [Developing with PhET: Getting Started \(on Windows\) screencast](#)):

Checking out the code and running it in development mode

1. Download and install Git from <http://git-scm.com/downloads>
 - a. Choose the option to create a desktop icon for “git bash”, if it is not already selected.
2. Use Git to check out the code for PhET Libraries and Simulations
 - a. Open Git Bash (from the link on your desktop or through the Start menu)

- b. Create a directory for your development:

```
mkdir phetsims
```

- c. Change directory to phetsims:

```
cd phetsims
```

- d. Run these git clone commands:

```
git clone https://github.com/phetsims/example-sim.git
git clone https://github.com/phetsims/assert.git
git clone https://github.com/phetsims/axon.git
git clone https://github.com/phetsims/brand.git
git clone https://github.com/phetsims/chipper.git
git clone https://github.com/phetsims/dot.git
git clone https://github.com/phetsims/joist.git
git clone https://github.com/phetsims/kite.git
git clone https://github.com/phetsims/phet-core.git
git clone https://github.com/phetsims/phetcommon.git
git clone https://github.com/phetsims/scenery.git
git clone https://github.com/phetsims/scenery-phet.git
git clone https://github.com/phetsims/sun.git
git clone https://github.com/phetsims/sherpa.git
```

3. Download & install node+npm from <http://nodejs.org/download>

4. Launch a node server program on your development machine

- a. Install the http-server as a command line program. Use a different command prompt than the one above since the one above will not have the new path for npm

```
npm install http-server -g
```

- b. Change into the phetsims directory (if you were not already there).

```
cd phetsims/
```

- c. Run the http server program

```
http-server
```

5. Open a browser to the path for one of the simulations:

http://localhost:8080/example-sim/example-sim_en.html

Now you can test modifying the simulation code and see the changes by refreshing the browser. You can also use this to test on remote devices after looking up your ip address. Questions should be directed to the Developing Interactive Simulations in HTML5 Google Group

Getting Started (on OS X)

In order to get the code for an existing PhET simulation, you will need to follow these steps:

Checking out the code and running it in development mode

1. Download and install Git from <http://git-scm.com/downloads>
 - a. You may need to allow the system to run apps downloaded from anywhere
 - i. Open the Apple menu
 - ii. System preferences
 - iii. Security & Privacy
 - iv. Click the lock to make changes
 - v. Allow apps downloaded from “anywhere”
 - vi. After you have installed git, you can restore the prior security settings
2. Use Git to check out the code for PhET Libraries and Simulations
 - a. Open the “Terminal” application
 - i. Click the search icon (magnifying glass) in the top right of the desktop
 - ii. Type “Terminal”
 - iii. Click on the terminal icon
 - b. Create a directory for your development:
`mkdir phetsims`
 - c. Change directory to phetsims:
`cd phetsims`
 - d. Run these git clone commands:

```
git clone https://github.com/phetsims/example-sim.git
git clone https://github.com/phetsims/assert.git
git clone https://github.com/phetsims/axon.git
git clone https://github.com/phetsims/brand.git
git clone https://github.com/phetsims/chipper.git
git clone https://github.com/phetsims/dot.git
git clone https://github.com/phetsims/joist.git
git clone https://github.com/phetsims/kite.git
git clone https://github.com/phetsims/phet-core.git
git clone https://github.com/phetsims/phetcommon.git
git clone https://github.com/phetsims/scenery.git
git clone https://github.com/phetsims/scenery-phet.git
git clone https://github.com/phetsims/sun.git
git clone https://github.com/phetsims/sherpa.git
```

When running the first git clone command, it may show a dialog that says: The “git” command requires the command line developer tools. Would you like to install the tools now? In this case, press “Install”.

3. Download & install node+npm from <http://nodejs.org/download>
4. Launch a node server program on your development machine

- a. Install the http-server as a command line program using the Terminal

```
npm install http-server -g
```

* If that yields an error like "Please try running this command again as root/Administrator." then run using the sudo command like so:

```
sudo npm install http-server -g
```

- b. Change into the phetsims directory (if you were not already there).

```
cd phetsims/
```

- c. Run the http server program

```
http-server
```

5. Open a browser to the path for one of the simulations:

http://localhost:8080/example-sim/example-sim_en.html

Now you can test modifying the simulation code and see the changes by refreshing the browser.

You can also use this to test on remote devices after looking up your ip address

Questions should be directed to the Developing Interactive Simulations in HTML5 Google Group

Creating a New Sim

After checking out the dependencies above, you can create your own simulation using our template.

1. Check out the template sim, called 'simula-rasa' using this git clone command:

```
cd phetsims
git clone https://github.com/phetsims/simula-rasa.git
```

2. Install the sim dependencies using npm install.

```
cd simula-rasa
npm install
```

3. Install the chipper dependencies using npm install.

```
cd ../chipper
npm install
```

4. Use the chipper 'grunt' task to create a new sim, like so:

```
cd ../simula-rasa
grunt create-sim --name=NAME --author=AUTHOR
```

For instance, if the simulation is going to be named Acceleration Lab and the author is Sam Reid from PhET Interactive Simulations, then you could put

```
grunt create-sim --name=acceleration-lab --author="Sam Reid (PhET Interactive Simulations)"
```

5. Test the created simulation in the browser and make sure it launches. It should be a blank simulation. Write to the Developing Interactive Simulations in HTML5 Google Group if you run into problems.

Building + Testing (Windows and OS X)

Building the Simulation with chipper

1. Open Git Bash (Windows) or Terminal (OS X)
2. `cd example-sim`
3. `npm install`
4. `npm install grunt-cli -g`
5. `cd ../chipper`
6. `npm install`
7. `cd ../example-sim`
8. `grunt`
9. Open a browser to the path to test it
 - a. http://localhost:8080/example-sim/build/example_sim.html

Working with Git and GitHub

1. Pulling the latest changes
2. Creating an issue
3. Committing
4. Submitting a pull request

Submit an Issue

Using Chrome dev tools to debug a simulation

Source code and Dependencies

Our simulations and dependencies are hosted publicly on GitHub

<https://github.com/phetsims>

We have 40+ repositories for the simulations and their dependencies, all of which can be found here: <https://github.com/phetsims?tab=repositories>.

Below is a list of some of the repositories that contain libraries and frameworks upon which the individual simulations depend.

- [Scenery](#): A general scene graph for rendering the graphics and handling the input.
[Documentation site for scenery](#)
- [Axon](#): for model implementation

- Assert: assertion framework for development testing
- phet-core: inheritance, extension and other utility functions
- phetcommon: higher-level common dependencies
- scenery-phet: PhET-specific scenery graphics and input handlers
- Joist: Framework for application loading, launching, and handling tabs
- Dot: Mathematics framework for model & view
- Kite: Shape library
- Sun: Graphical user interface components, such as buttons, checkboxes, etc
- Sherpa: All of our 3rd party dependencies

Checking out the HTML5 code from GitHub

Our example-sim repository README.md includes a list of git clone commands that will check out the example simulation and all of its dependencies.

<https://github.com/phetsims/example-sim>

And to clone some of our in-development sims:

git clone git://github.com/phetsims/forces-and-motion-basics.git

git clone git://github.com/phetsims/build-an-atom.git

All repositories should be cloned into the same directory so that relative paths will work.

Here is a full list of all phetsims repos. If the sim won't launch due to a missing dependency, you may need to check out some more of these repos:

<https://github.com/phetsims?tab=repositories>

Also note that this will check out the 'master' branch of all of our dependencies, which may create breaking changes intermittently if you remain up-to-date with them. If you run into any breaking changes, please notify us immediately. Also, we recommend developing your code on a public repo such as GitHub to enable us to test and update your simulations as common dependencies are changed.

Master is Potentially Unstable: Accessing Rigorously Tested Code

The master branch of the PhET simulation and library repositories is constantly under development and not guaranteed to be stable. It is our intent that the master branch of simulations + libraries will build and run properly, but sometimes the code goes through intermediate states where errors can be introduced. On the other hand, our published simulations have been rigorously tested across 18+ platforms and are the most stable option. If you are [adapting a PhET simulation](#), or would like to access simulation code that corresponds directly to one of our published versions, then you will need to check out specific SHA revisions in all of the appropriate repositories. Checking out these fixed, tested revisions is also important

when working on a release-candidate branch of a simulation. Here are the instructions:

1. First, identify the version for which you want to check out the source code, for example:
http://phet.colorado.edu/sims/html/area-builder/latest/area-builder_en.html
2. Navigate to a file named dependencies.json at the same path, for example:
<http://phet.colorado.edu/sims/html/area-builder/latest/dependencies.json>
3. Download the dependencies.json file to the root of the simulation directory.
4. Open a command prompt and cd into the root of the simulation directory.
5. Run ``grunt checkout-shas``. This step will read from the dependencies.json file and check out all of the SHAs corresponding to the entries in the file.
6. To checkout the SHA for the simulation itself, first look up the SHA in the dependencies file, move the dependencies file to some other location or delete it, and use ``git checkout`` to check it out the appropriate SHA. (Note future version of ``grunt checkout-shas`` may handle this last step).
7. Inspect the simulation and its dependencies to make sure ``grunt checkout-shas`` had the intended effect of getting the right SHA for each repo

Now you can use the published source code. To restore each branch to master, you can run ``grunt checkout-master``.

Exception and caveats:

1. Running ``grunt checkout-shas`` give errors when the working copy is not committed. These grunt commands are currently only supported for clean git repos. Stashing may be a way around this problem. Also, if you want to use dependencies from a different version than in the SHAs, that will have to be done as an additional manual step.
2. When working in a branch, `grunt checkout-master` will check out the master branch and additional manual steps will be required to get back to the desired branch(es). For instance, this is an issue when working with the “adapted-from-phet” branch of brand.

Original (Java/Flash) Source Code

Follow the directions at this link to get the source code for original Java and Flash version of the simulations:

<http://phet.colorado.edu/en/about/source-code>

After checking it out (could take 30+ minutes), the source code for the simulations are located in (for example):

svn-checkout/trunk/simulations-java/simulations/forces-and-motion-basics

3rd Party Dependencies

PhET Simulations use around 3 open source 3rd party dependencies for the deployed source code, and more for the build phase. They are all included with the source code checkouts in the sherpa repository. The libraries and licenses are described in this [3rd party dependency licensing document](#)

Licensing

New simulations should be published under GPLv3 and reusable library dependencies should be published as MIT.

Coding Style Guidelines

To improve the readability and maintainability of PhET Simulation code, we have identified several style recommendations for writing code and documentation.

We provide three primary mechanisms for ensuring the code is in a good style. (1) this style guide, see below for details (2) the chipper jshint task, which reports linting errors for the sim during the build task and (3) an [IntelliJ IDEA code formatting style](https://github.com/rwldrn/intellij-idea-code-formatting-style), which can be used to automatically indent and format the code. Our example-sim also shows how to use our libraries idiomatically as well as a good example of code commenting + documentation.

For general coding, please follow the guidelines set out in [idiomatic.js](https://github.com/rwldrn/idiomatic.js) (<https://github.com/rwldrn/idiomatic.js/>). Some PhET-specific differences from and additions to this spec are:

- Variables should be declared on one line with a separate 'var' in front of each. This makes moving and duplicating lines much easier. For example,

```
var x;  
var y;  
and not  
var x, y;  
or  
var x,  
y;
```

- Names (types, variables, properties,...) should be sufficiently descriptive and specific, and should avoid non-standard abbreviations. For example:

```
var numPart           // incorrect  
var numberOfParticles // correct  
  
var width             // incorrect  
var beakerWidth       // correct
```

- Use single quotes around all string literals. For example:

```

console.log( 'This is a test.' ); // correct
console.log( "This is a test." ); // incorrect

```

- All JavaScript scripts should be modularized using requirejs, and should invoke strict mode. For example:

```

define( function( require ) {
    'use strict';
    // etc.
} );

```

- Require statements should be organized into blocks, with the code modules first, followed by strings, images, and audio (any order ok for strings/images/audio). For modules, the `var` name should match the file name. Example below.

```

// modules
var inherit = require( 'PHET_CORE/inherit' );
var Rectangle = require( 'SCENERY/nodes/Rectangle' );
var Line = require( 'SCENERY/nodes/Line' );

// strings
var kineticString = require( 'string!ENERGY/energy.kinetic' );
var potentialString = require( 'string!ENERGY/energy.potential' );
var thermalString = require( 'string!ENERGY/energy.thermal' );

// images
var energyImage = require( 'image!ENERGY/energy.png' );

// audio
var kineticAudio = require( 'audio!ENERGY/energy' );

```

- Constructor and function documentation. Parameter types and names should be clearly specified for each function and constructor (if there are any parameters) using `@param` annotations. The description for each parameter should follow a hyphen. Primitive types should use lower case. Constructors should additionally include the `@constructor` annotation. For instance:

```

/**
 * The PhetDeveloper is responsible for creating code for simulations
 * and documenting their code thoroughly.
 *
 * @param {string} name - full name
 * @param {number} age - age, in years
 * @param {boolean} isEmployee - whether this developer is an employee of CU
 * @param {function} callback - called immediate after coffee is consumed

```

```

* @param {Property.<number>} hoursProperty - cumulative hours worked
* @param {string[]} friendNames - names of friends
* @param {Object} [options] - optional configuration, see constructor
* @constructor
*/
function PhetDeveloper( name, age, isEmployee, callback, hoursProperty,
friendNames, options ) {...}

```

- For most functions, the same form as above should be used, with a `@return` annotation which identifies the return type and the meaning of the returned value. Functions should also document any side effects. For extremely simple functions that are just a few lines of simple code, an abbreviated line-comment can be used, for example: `// Computes {Number} distance based on {Foo} foo.`

- Line 1 of all source files should be a copyright comment of the form:

```
// Copyright 2002-2015, University of Colorado Boulder
```

- All source files should have one or more `@author` annotations indicating the authors of the file.
- Annotations, such as “`@public`”, should be used to improve the maintainability of the code. Information about the private and public annotations (as used by Google) can be found [here](#) and [here](#). In PhET simulation and library code, the main things we want to document are:
 - Mark as `@private` anything that would be likely to cause problems if set or invoked from outside the type
 - Mark as `@public` anything that is intended to be within the type's external API
 - Mark as `@protected` anything that is intended for use only in sub-types
 - Also, we often use a comment like “`// read only`” to distinguish a property or value that should only be read outside of the class and never set.
- Comments should not extend beyond 120 columns, and line breaks should be inserted for multi-line comments just before word that would extend beyond 120th column. Code lines should also be broken up if they pass 120 columns.
- Where inheritance is needed, use `phetcore.inherit`. Add prototype and static functions via the appropriate arguments to `inherit`. Spaces should exist between the function names unless the functions are all short and closely related. Example:

```

return inherit( Object, Line, {

  // Convenience method for creating a line with a different color.
  withColor: function( color ) {
    return new Line( this.x1, this.y1, this.x2, this.y2, color );
  },

```

```

toString: function() {
    return 'Line[x1=' + this.x1 +
        ' y1=' + this.y1 + ' x2=' + this.x2 + ' y2=' + this.y2 +
        ' rise=' + this.rise + ' run=' + this.run + ' color=' +
        this.color.toString() + ']';
},

// Returns true if 2 points on specified line are also on this line.
same: function( line ) {
    return ( line !== null ) && this.onLineXY( line.x1, line.y1 ) &&
        this.onLineXY( line.x2, line.y2 );
},

// Returns true if the slope is undefined.
undefinedSlope: function() {
    return this.run === 0;
},

// Gets the slope. Returns NaN if slope is undefined.
getSlope: function() {
    if ( this.undefinedSlope() ) {
        return Number.NaN;
    }
    else {
        return this.rise / this.run;
    }
},

/*
 * Given x, solve  $y = m(x - x1) + y1$ . Returns NaN if the solution is
 * not unique, or there is no solution (x can't possibly be on the
 * line.) This occurs when we have a vertical line, with no run.
 */
solveY: function( x ) {
    if ( this.undefinedSlope() ) {
        return Number.NaN;
    }
    else {
        return ( this.getSlope() * ( x - this.x1 ) ) + this.y1;
    }
}
} );

```

- Methods should be invoked using the dot operator rather than the bracket operator. For more details, please see <https://github.com/phetsims/gravity-and-orbits/issues/9>
 - For instance: instead of :

```

        self[ isFaceSmile ? 'smile' : 'grimace' ]();
Prefer
        isFaceSmile ? self.smile() : self.grimace();
or
        if ( isFaceSmile ) {
            self.smile();
        }
        else {
            self.grimace();
        }

```

- Naming for Property values: All AXON/Property instances should be declared with the suffix `Property`. For instance, if a visible property is added, it should have the name `visibleProperty` instead of simply `visible`. This will guarantee consistency with Properties created by PropertySet, and help to avoid confusion with primitive (non-Property) values.
- Line comments should be preceded by a blank line. For instance:

```

// Randomly choose an existing crystal to possibly bond to
var crystal = this.crystals.get( _.random( this.crystals.length - 1 ) );

// Find a good configuration to have the particles move toward
var targetConfiguration = this.getTargetConfiguration( crystal );

```

- Line comments should have whitespace between the `//` and the first letter of the line comment. See the preceding example.

JSHint should be used to check the validity of the source code. Checking of the source code with JSHint will be an automatic step of the build-and-deploy process, and deployment will be aborted if JSHint is not passed.

Files should be named like CamelCasing.js when returning a class constructor function, or lower-case-style.js when returning a function. When returning a constructor function, the constructor function name should match the filename.

The HTML5/CSS3/JavaScript source code must be reasonably well documented. This is difficult to specify precisely, but the idea is that someone who is moderately experienced with HTML5/CSS5/JavaScript can quickly understand the general function of the source code as well as the overall flow of the code by reading through the comments. For an example of the type of documentation that is required, please see the Example Simulation.

Also, at the top level of each repository there is a file named “changes.txt”. Each time a new feature is added, or a bug is fixed, this file should be updated with a single one-line description of the change at the top of this file. The file itself will be visible to PhET’s users once the simulation is deployed, so comments should be clear, concise, descriptive, and professional.

Please see the changes.txt file in the example simulation for guidance.

Platforms

The simulation should be tested and run on the platforms listed below. In our experience to date, some optimization is often required to obtain acceptable performance on tablets such as the iPad.

OS/Browser List

- Windows XP+: IE9+, Chrome latest, Firefox latest
- Mac OS 10.6: FireFox latest, Chrome latest (Safari 5 not supported)
- Mac OS 10.7+: Safari 6+, FireFox latest, Chrome latest
- iPad 2+ (iOS6+): Safari latest
- Android 4.1+ (JellyBean): Chrome latest

Specific Hardware List

- iPad 2+
- iPad mini
- Google Nexus 7
- MacBook Pro, Air
- Chromebook
- Desktop or laptop PC running Windows 7 and Windows 8
- Windows 8 tablet
- Samsung Galaxy Tab 2+ (where possible, details TBD)

Modularity & RequireJS

The current implementation of JavaScript (ECMAScript 5) does not directly support modular JavaScript development. RequireJS will be used to support modularization of the JavaScript code. Information about RequireJS can be found at <http://requirejs.org/>. Examples of how it will be used by PhET can be seen in the Example Simulation code at <https://github.com/phetsims/example-sim>. Specifically, check out the config.js file, the grunt.js build file, and the source files in the js directory.

PhET will consider the built-in AMD support that will be a part of ECMAScript 6 when it becomes available.

Layout

Design:

Minimum width x height: 768x504 (1.52:1, inside Mobile Safari)

The simulation should scale isometrically such that all portions are visible at any resolution. An example of this type of scaling can be seen in the example simulation.

Compiling the Code

A minification and unification process is implemented in our repo <https://github.com/phetsims/chipper> . This can be used to create a single-file HTML that contains all images and audio, and is suitable for download for offline usage.

Offline Operation

It is a requirement that all PhET simulations can be downloaded and run off line in all identified browsers from the file:// URL. PhET's chipper build process (described above) produces a single file that can be downloaded for offline use. Please make sure you are not using any APIs that prevent launching and running properly when offline using file:// URL., and test that offline operation works properly for your simulation.

Published Versions

Here is a link to some published sims, so that you can see a demonstration of how some things should look and behave:

<http://phet.colorado.edu/en/simulations/category/html>

Development Process & Checklist

The steps to create a fully functional PhET simulation, given an existing Java/Flash version or development specification:

1. The simulation and its code:
 - a. must use the appropriate libraries in the correct fashion
 - b. must be adequately commented
 - c. must contain no dead code (i.e. commented out code that does nothing)
 - d. must be maintainable
 - e. reusable components should be polished and moved to the appropriate libraries
 - f. should pass all jshint tests when running chipper, and should be compiled into a single file HTML file
 - h. original vector artwork for anything appearing in the images/ directory should be checked into the assets/ directory.
 - i. must run with ?ea (assertions) enabled without any assertion errors being triggered

2. Simulation & User Interface Testing

a. Testing on our supported platforms and identification of problems on different browsers

b. Performance must be sufficiently fast on all supported platforms.

i. The simulation should start in <8 seconds on iPad3

ii. We strive for a steady 60fps on iPad3 when possible

c. The simulation should be tested with assertions enabled ?ea

d. The simulation should be tested for touch areas: ?showPointerAreas

3. Code review

a. The code will be reviewed by one or more PhET developers in order to identify possible bugs or maintenance problems

b. Issues raised in the review will be addressed

4. Release candidate testing

a. Before publication, a release candidate branch is created so that the branch can be thoroughly tested and, if no significant bugs are found, published

5. Publication

a. The simulation is made available on the PhET website

6. Maintenance

a. the simulation is published and any bugs reported by students or teachers will be resolved

Utilities and Instrumentation for Development and Testing

Many aspects of a simulation must be developed properly and working well in order for the simulation to behave properly across our many supported platforms. PhET has developed several utilities and instruments to make this development and testing easier. The most up-to-date documentation for the query parameters is available here:

<https://github.com/phetsims/chipper/blob/master/js/initialize-globals.js>

1. **Query parameter: screenIndex** This query parameter may be used to specify the initial screen of the simulation. It can be paired with standalone above to launch just a specific screen of the simulation. For instance:

http://localhost:8080/energy-skate-park-basics/energy-skate-park-basics_en.html?screenIndex=1&standalone launches Energy Skate Park: Basics using only the 2nd screen.

2. **Phet Allocations:** Object instance allocation tracking, so we can cut down on garbage collection. See <https://github.com/phetsims/phet-core/blob/master/js/phetAllocation.js>

Sample usage:

1. Run the sim and set up the scenario that you wish to profile

2. In the JS console, type: window.alloc={}

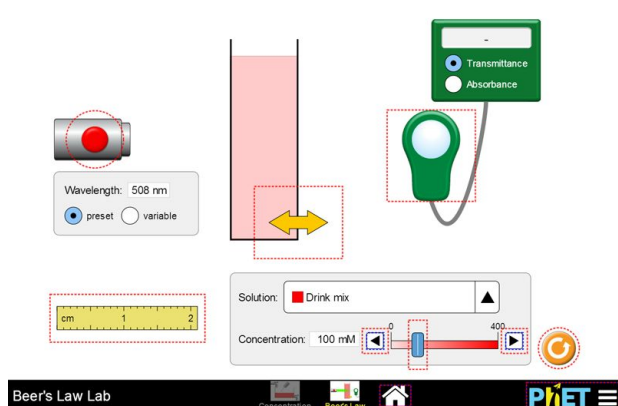
3. Wait until you have taken enough data

4. Type x = window.alloc; delete window.alloc;

Now you can inspect the x variable which contains the allocation information.

3. **Query parameter: ea.** This parameter is used to “enable assertions”. The sim should run without triggering any assertion errors in ?ea mode.

4. **Query parameter: showPointerAreas.** This query parameter shows the areas for mouse and touch input events. On mobile devices (and sometimes for mouse) it is essential to increase the interaction region for a scenery node. Touch areas are shown in red and custom mouse areas are shown in blue.



5. **makeRandomSlowness()** This method can be called after the simulation is started to simulate an intermittently slow system. This can be used to help replicate bugs that only happen intermittently or only on slow platforms. To call this method, launch the sim, show the developer console, and type the command as above.

6. **makeEverythingSlow()** This method can be called after the simulation is started to simulate a slow system. This can be used to help replicate bugs that only happen intermittently or only on slow platforms. To call this method, launch the sim, show the developer console, and type the command as above.

7. **Query parameter: 'profiler'** Launching a sim with ?profiler will print out the time to create each screen, and will show a histogram which updates every 60 frames depicting how long the frames are taking (in ms). Note: just showing the average FPS or ms/frame is not sufficient, since we need to see when garbage collections happen, which are typically a spike in a single frame. Hence, the data is shown as a histogram. After the first 30ms slots, there is a ++= showing the times of longer frames (in ms)

8. **Usage of Unit Tests:** After making changes in one of the repos with unit tests (see if tests/qunit exists), run the unit tests afterwards (tests/qunit/unit-tests.html) to see if anything is broken. We highly recommend checking "Hide passed tests", and wait until all tests are complete (it may pause at 0 tests complete at the start).

9. **Adding Unit Tests:** If you want to add a test, add it to one of the tests/qunit/js/* files that have a QUnit module('...') declaration, and read the QUnit tutorials to understand how it works. You can add new files with more tests by creating the file and referencing it in tests/qunit/js/unit-tests.js (whose purpose is to load those files).

10. **Namespaces for Unit Tests and Playground:** Each unit-tests.html makes certain namespaces global (e.g. Scenery's makes window.scenery/kite/dot/axon/core for Scenery/Kite/Dot/Axon/phet-core respectively). Since unit tests don't use require.js directly (for Scenery/Kite/Dot they need to work with the standalone built-library versions), the namespaces

are used to access types. For instance, "new scenery.Rectangle(...)" is used.

11. **Playground:** If it exists, it will be a tests/playground.html, and allows testing code in the console. To make code available in the console, check the 'main' file used by the playground and add a reference there. For instance, Scenery's playground.html loads 'main' as the first array argument to the inner require() statement, and saves it to window.scenery. In require.js-speak, this sets the returned value of js/main.js to window.scenery, and in Scenery's specific case it returns the library namespace used in all unit tests. In addition, the same namespaces for unit tests are used.

Working with GitHub Issues

When the problem described in a GitHub issue is solved, a description of the solution should be made in the issue and the issue should be reassigned to the original reporter of the GitHub issue for verification and closing. Commits that address the GitHub issue should also reference the issue in the commit message, so the change set can be easily reviewed.

Here are some example issues that show creation of an issue, solving it with a commit message that references the issue, an explanation of the solution and reassignment to the reporter for verification and closing:

<https://github.com/phetsims/color-vision/issues/15>

<https://github.com/phetsims/fraction-matcher/issues/56>

<https://github.com/phetsims/color-vision/issues/37>

Embedding a Simulation in your website

To embed a simulation in your website, use an iframe like so:

```
<iframe  
src="http://phet.colorado.edu/sims/html/forces-and-motion-basics/latest/forces-and-motion-basics_en.html" width="834" height="504"></iframe>
```

The aspect ratio 834x504 is used for new simulations, because it matches the aspect ratio available on popular devices.

To see a full embedding example in context, view the source for this page:

http://www.colorado.edu/physics/phet/dev/html/acid-base-solutions/1.1.0/acid-base-solutions_en-iframe.html