# Creating Interactive Web Simulations Using HTML5 and JavaScript A workshop and tutorial

#### **Daniel V. Schroeder**

#### **Tutorial lessons**

- Introduction
- Web Page Basics
- Drawing and Animation
- User Interaction
- · Finishing Touches
- Next Steps

#### Reference sheets

- HTML and styling reference
- JavaScript reference
- Canvas graphics reference
- · User interface reference
- HTML/CSS vocabulary and syntax

#### Sample pages (online only)

- · Newton's Cannon, progressive versions:
  - 1. Static web page only
  - 2. With animation but no GUI
  - 3. Functional but lacks finishing touches
  - 4. Final version
- Hello, Canvas!
- Canvas graphics and GUI demo
- <u>Transformations</u> (translate, scale, rotate)
- Canvas animation demo
- Simple Paint (demonstrates mouse and touch interaction)
- Spaceship (demonstrates image drawing and transformations)
- Chaotic Pendulum (demonstrates plotting a graph)
- Two-Source Interference (demonstrates pixel manipulation)
- Star Motion (demonstrates arrays, objects, and a separate data file)
- This page demonstrates fixed positioning and local storage.
- More examples from around the web

#### Other resources (online only)

- W3Schools
- Stack Overflow
- HTML5 Canvas Tutorials
- Learning Web Design by Jennifer Niederst Robbins
- JavaScript: The Definitive Guide by David Flanagan
- Core HTML5 Canvas by David Geary

All original content in this HTML5 Tutorial is copyright © 2014, Daniel V. Schroeder. The tutorial lessons and reference sheets are available for copying and reuse under the <u>Creative Commons Attribution 4.0 International License</u>. The HTML/JavaScript software is released under the <u>MIT License</u>, as indicated in the source files.



For the online version of this tutorial, go to: http://physics.weber.edu/schroeder/html5/

## Introduction

This tutorial is for physics instructors and anyone else who wants to learn how to create interactive web pages with animated graphics.

I'll assume that you have programmed a computer before, so you're familiar with basic programming concepts and you know how to be careful with syntax.

I won't assume that you already know how to create web pages, or that you're familiar with the specific syntax of web programming.

My goal is to introduce you to the web as a programming platform, similar in most ways to the programming environments that are traditionally used by scientists. The "program" in this case is simply a web document; the interpreter that runs the program is the user's browser; and the program's output is the web page as seen by the user.

As a method of developing and deploying software, the web has several advantages:

- Your code can run on virtually any computer with a web browser, including tablets and smart phones; there's no need to create separate versions for each device.
- Users of your program don't need to go through a burdensome installation process.
- Web technologies are well supported and are likely to be around for a long time (by the standards of the computing industry).
- It's easy to get started coding, since you can begin with very simple web pages and you don't need to purchase or install any development tools.
- Your program's web page provides a convenient place to put instructions and other documentation, including links.
- Your code is inherently open-source, visible to any user who can run it.

INSTALLING THINGS HAS
GOTTEN SO FAST AND PAINLESS.

WHY NOT SKIP IT ENTIRELY,
AND MAKE A PHONE THAT HAS
EVERY APP "INSTALLED" ALREADY
AND JUST DOWNLOADS AND RUNS
THEM ON THE FLY?

I FELT PRETTY CLEVER UNTIL I
REALIZED I'D INVENTED WEBPAGES.
http://xkcd.com/1367/

Naturally, web programming also has some disadvantages:

- Before you can write a web *program* you must create a web *page*. This means more coding overhead and syntax for you to learn.
- Your code must (or at least should) accommodate a variety of screen sizes, user input mechanisms, and processor speeds. This variety limits your design options and makes optimization difficult.
- Your code has no direct way of accessing the file system on the user's computer (though you can store data during one session for later retrieval during another).

- Your code and any associated data must be small enough for users to download quickly over their networks.
- Your code won't run as quickly as native code on any device (though it can often come close).
- Your code is inherently open-source, visible to any user who can run it.

These limitations make web programming less than ideal for those who want to build big simulations or just tinker or crunch data on their own. Web programming can be ideal, though, if you want to share your programs with the rest of the world. That's probably the case if you're a teacher writing programs for your students—for instance, a physics instructor sharing simulations of guantum wavefunctions or fluid dynamics.

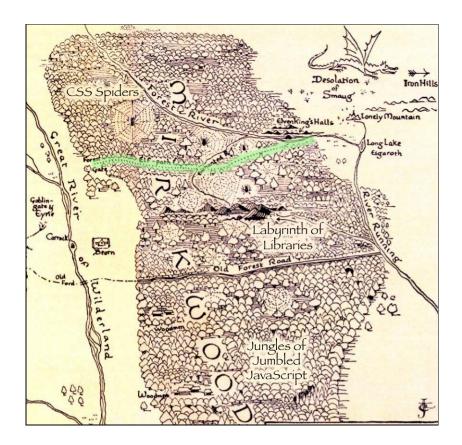
So that's the motivation behind this tutorial. More specifically, this tutorial will teach you the basics of:

- Hyper-Text Markup Language (HTML), for defining the logical structure of a web page;
- Styling ("CSS"), for specifying the appearance of each element on a web page;
- JavaScript, the programming language that brings web pages to life;
- The HTML5 canvas element, which provides a space on a page where your code can draw using JavaScript instructions;
- HTML input controls such as buttons and sliders for user interactivity.

That's a lot to learn! But it's not as much as you might think, because this tutorial covers only a small subset of each of these technologies, and it completely omits several related technologies. In particular, this tutorial will *not* prepare you to:

- use every feature of JavaScript or CSS or HTML (including most of what's been added in HTML5);
- understand most other people's JavaScript code:
- use the many JavaScript libraries that simplify certain high-level tasks;
- program a server to assemble web pages from a database in response to user input;
- design a large corporate web site; or
- get a job as a professional web developer.

Our focus instead will be on creating self-contained web pages that run entirely on the user's computer (the "client"), performing computations and presenting visual results, taking input parameters from the user through graphical controls. With this limited goal, we can avoid many of the more difficult aspects of web programming and take certain shortcuts that wouldn't be available to an aspiring web professional. If you think of the universe of web development as the vast <a href="Forest of Mirkwood">Forest of Mirkwood</a> (see next page), then our goal is not to visit every tree and creature in the forest, but merely to find the magical Elf Path that will get us safely to our destination. Leave the path at your own risk!



Of course, once you have gained some experience and grown stronger, you can always return and explore some of the more challenging aspects of web development.

#### About the author

If you're hoping to learn web programming from an experienced professional, this tutorial is not for you. I'm a full-time <u>physicist and educator</u>, but only a part-time hobbyist-level programmer. My knowledge of web programming doesn't go far at all beyond what's in this tutorial, and I've learned this subject almost entirely from books and online references and trial-and-error, without the benefit of an expert mentor. I'm sure many experts would scoff at some of the ways I do things, but on the other hand, their priorities aren't the same as mine. I do hope that, as an inexperienced newcomer to web programming, I can at least offer a fresh perspective that you won't find elsewhere.

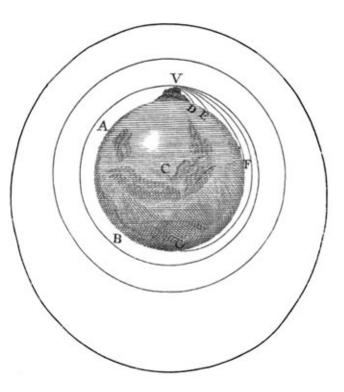
# Web Page Basics

In this tutorial you will create a working HTML5 physics simulation with interactivity and animated graphics.

The example I've chosen is to animate Newton's Cannon, a famous thought experiment and illustration by Isaac Newton, in which he asks his reader to imagine the motions of projectiles launched horizontally from a high mountaintop at various speeds.

You can find many animated versions of Newton's Cannon on the web, but most of them are done in Flash or Java, both of which require plug-ins and don't work on mobile devices. My own inspiration comes from a <u>version</u> by Michael Fowler and Drew Dolgert.

I've broken the creation of this simulation into steps, described in detail below. The initial steps are the same as for creating any web page; the animation and interactivity will come later.



## 1. Create a working folder.

The first thing you need is a folder on your hard drive in which you will work. Go ahead and create one now, and call it "html5" or whatever else you like.

#### 2. Get a text editor.

Next you need a text-editing application where you can type. If you already have a favorite programmer's text editor, just use that. Otherwise, I recommend <a href="Notepad++">Notepad++</a> for Windows or <a href="TextWrangler">TextWrangler</a> for MacOS. Both are free and easy to use, so don't hesitate to download and install one of them right now. You can get away with using Notepad on Windows or TextEdit on MacOS (both of which come pre-installed), but these editors lack some of the features that are most convenient for editing code. (If you use TextEdit, be sure to choose Make Plain Text from the Format menu before you begin typing.)

## 3. Type some text content.

Once your text editor is ready, launch it and open a new blank document. Save it in the folder you just created, with the name "NewtonsCannon.html". Then type the following text into your document:

Newton's Cannon

This simulation is based on Isaac Newton's famous thought

experiment and illustration of firing a projectile from a high mountaintop at various speeds, as he described in A Treatise of the System of the World.

After saving this text, launch your favorite web browser and use it to open the document you just created. You can probably do this by simply double-clicking on the document's icon in a Windows Explorer or Finder window. Alternatively, look for an "Open File" command in your browser's menus. Either way, the document should open in a browser window and you should see the text above (though the formatting will be different).

#### 4. Add basic markup.

Your browser probably just displayed what you typed as one big "paragraph", with no indication that the first two words are actually a title or heading. To remedy this, you need to add *markup tags* to label the logical parts of the document. In HTML, markup tags are enclosed in angle brackets. To mark a major heading on a page you enclose it in h1 tags:

```
<h1>Newton's Cannon</h1>
```

Note the / character, which distinguishes the closing tag from the opening tag. (The numeral 1 denotes a first-level heading; you can use h2 through h6 for lower-level headings.) To mark a paragraph, you enclose it in p tags:

```
This simulation is based on Isaac Newton's...
```

After you've added the h1 and p tags, save your document and reload it in your web browser. The heading should now appear on a line by itself, in a much larger font.

#### 5. Add boilerplate markup.

Although your minimal markup works just fine, it's technically in violation of the rules of HTML because you've omitted a bunch of boilerplate markup that sets out the basic context. Here's what it should look like with the boilerplate:

I've used indentation to make the logical structure more clear to the eye. (This isn't required, but you should use some combination of indentation and blank lines to make your HTML code more readable.) Each pair of opening and closing HTML tags defines what we call an *element*, such as the heading and paragraph elements that you defined previously. But all of the content of your page should be enclosed inside a body element, which in turn is nested inside the all-encompassing

html element. Inside html but before body comes the head element, which sets out some preliminary information about the character encoding of your HTML file and the title of your web page (which your browser will ordinarily display at the top of the window). Before everything comes the DOCTYPE declaration, which technically isn't an HTML element at all even though it looks like one.

Whether all these codes make sense to you or not, please type them verbatim into your document, save the changes, refresh your browser window, and check that the page content appears the same as before.

This is one example of how web browsers tend to be *extremely* forgiving of coding errors and omissions, as long as their algorithms can still come up with some way to display your content. The lack of enforcement of rules makes your job as a programmer *more difficult*, because coding errors can lead to unexpected behavior whose cause is difficult to trace. So you need to develop good habits and pay attention to details, even when you think they don't matter.

#### 6. Change the styling.

So far, your browser is displaying everything on your page using its default *styles*: probably a serif font of a certain size, in black on a white background, left-justified with lines that grow or shrink as you change the size of the window, with a much larger font for the heading and a certain amount of space between it and the paragraph. All of these defaults are easy to change.

The easiest way to change styling is by inserting a style *attribute* into an opening HTML tag. For example, you can change the font throughout the page by modifying the <body> tag as follows:

```
<body style="font-family:sans-serif;">
```

Be careful of the syntax here, paying attention to all the punctuation symbols. After you save this change and reload the page, you'll see that the font has changed.

In this example, font-family is just one of many CSS *properties* that you can specify. Here are some other property settings to try in the body element:

```
font-size:15px;
width:600px;
margin-left:auto;
margin-right:auto;
background-color:#c0c0c0;
```

Put these all, one after another (in any order), inside the quotes along with the font-family setting, and check the results. The margin settings cause your content to be centered in the browser window. The background color (light gray in this case) is specified as three two-digit hexadecimal (0 through f) numbers, for the brightnesses of the red, green, and blue components: #rrggbb. By all means, try some other colors!

Besides changing the style of the whole page through the body element, you can also change the styles of smaller pieces. Here are some styles to try in the <h1> tag:

```
font-size:24px;
text-align:center;
color:#800000;
```

You may have noticed that setting font-size to 15px (15 pixels) in the body affected the nested p (paragraph) element, but not the nested h1 element; we say that the paragraph *inherited* this property from its parent element (body), but h1 didn't. It isn't always easy to guess when properties will be inherited and when they won't, but the phenomenon is something to be aware of.

I should probably warn you that most web developers frown upon this low-level method of specifying styling. The alternatives are either (1) to put styling specifications at the top of the document inside the special <style> element (which goes inside the <head> element), or (2) to put styling specifications in a separate document called a *style sheet*, which gets a .css extension. In these ways you can apply common styling to *all* elements of a certain type (e.g., all paragraphs or all headings), and, in the second case, to multiple web pages (by linking from all of them to the same style sheet). (The acronym CSS stands for *Cascading Style Sheets*, where "cascading" refers to the way that specific style rules can override more general ones.) But for a small one-of-a-kind web page, the "in-line" styling method described here is easiest.

#### 7. Add a link.

Perhaps the most useful HTML element is a *link* to another web page. To create one you use the <a> tag, like this:

```
<a href="http://physics.weber.edu/">Click here!</a>
```

The letter a stands for *anchor*, for reasons that were more relevant in earlier versions of HTML; the linked URL is specified in the href attribute, where href stands for *hypertext reference*.

In your web page, a good place to add a link is on the title of Newton's work, *A Treatise of the System of the World.* Here's the URL for a Google Books scan of the *Treatise*:

```
http://books.google.com/books?id=DXE9AAAACAAJ
```

After inserting the a tags around the title, with this URL for the href attribute, save your page and test the link in your web browser.

#### 8. Add the image.

The last "static" element that you need for your page is an image to serve as a background for the graphical output. I've placed a modified version of the original drawing from Newton's *Treatise* at the following URL:

```
http://physics.weber.edu/schroeder/html5/NewtonDrawing.jpg
```

You could just link your page directly to my copy of the image, but it's safer to keep your own copy, so please use your browser to download the image now and save it in your working folder.

To insert the image in your web page, use the img tag:

```
<img src="NewtonDrawing.jpg" width="500" height="500">
```

The attribute name src is an abbreviation for *source*, while the width and height attributes specify the size of the image as it should be displayed on the page, in units of nominal screen pixels (which should usually be the same as the intrinsic resolution of the image). Place this tag in

your html file after the h1 element and before the p element. The img tag is self-contained, with no text content, so it has no corresponding closing tag (no </img>).

When you test this new code in your browser, the image should appear. But the image is left-justified, not centered, which looks odd because it's only 500 pixels wide while the width of the page body is 600 pixels. You could, of course, change the latter to 500 pixels and then everything would look fine. A more general solution is to separately center the image by giving it some styling. A still more powerful approach (whose advantage you'll see when you add animated graphics in front of the image) is to enclose the image in an invisible container or block, called a div (for division), and apply the styling to the div:

Go ahead and make this change, and check that the image is now centered. (Be sure to note the difference in syntax between the CSS width property in the first line and the HTML width attribute in the second line.)

#### Review

Before going on to add animation and interactivity to your web page, let's pause to assess where we stand.

You've created a simple web page with a heading, a paragraph, a link, and an image, and you've applied some styling to these elements to change the default fonts, colors, and layout. To do these things, you've learned the basics of HTML to specify the logical structure of the page elements, and you've learned a bit of CSS to specify the styling.

To review the fundamental vocabulary and syntax of HTML and CSS, you may want to look at the summary page at the end of this packet. There's also an HTML and CSS reference sheet, which covers all of the elements and styles you've used so far, plus a few more that you may wish to learn in the near future.

Perhaps the best way to learn web development is to study examples. Unfortunately, most of the web pages you see these days are at least partially generated by machines, with no attempt to make the source code readable by humans. However, all of the pages associated with this tutorial were typed by hand and should be relatively easy to figure out (even if some of them use features that aren't fully explained). Learn how to use your browser's View Source command, and use it frequently.

If you would like a more comprehensive introduction to HTML and CSS, then I recommend the W3Schools site, which has a brief explanation of just about every HTML tag and attribute, and just about every CSS property, with simple examples that you can easily study and modify. To get answers to specific questions, I recommend simply googling them; more often than not, the search results will take you to Stack Overflow, an enormous question-and-answer site for programmers. Finally, sometimes what you want is a traditional textbook, organized by a real teacher and perhaps even printed on paper; the best one I've found on HTML and CSS is Learning Web Design, by Jennifer Niederst Robbins.

# **Drawing and Animation**

After finishing the <u>previous section</u> of this tutorial, you should have a static web page displaying a title, some text, and an image; the source code should look something like this:

```
<!DOCTYPE html>
<html>
<head>
 <meta charset="utf-8">
 <title>Newton's Cannon</title>
<body style="font-family:sans-serif; font-size:15px; width:600px;</pre>
   margin-left:auto; margin-right:auto;
   background-color:#e0e0e0;">
 <h1 style="font-size:24px; text-align:center;">
   Newton's Cannon
 </h1>
 <div style="width:500px; margin-left:auto; margin-right:auto;">
   <img src="NewtonDrawing.jpg" width="500" height="500">
 </div>
 This simulation is based on Isaac Newton's famous thought
 experiment and illustration of firing a projectile from a high
 mountaintop at various speeds, as he described in
 <a href="http://books.google.com/books?id=DXE9AAAACAAJ">A
 Treatise of the System of the World</a>.
</body>
</html>
```

In the following steps you will bring this page to life with animated graphics.

#### 1. Add a place to draw.

The HTML5 canvas element provides an initially blank rectangular space where you can draw using low-level graphics instructions. Here's the syntax for inserting a canvas:

```
<canvas id="theCanvas" width="500" height="500">
   Canvas not supported; please update your browser.
</canvas>
```

The content that you put between the opening and closing canvas tags will be displayed only in older browsers (most notably Internet Explorer 8 and earlier) that don't support canvas. As with img, the width and height attributes are given in units of nominal screen pixels. The id attribute is simply a unique name by which your code can identify this canvas.

Please insert this code in your web page, right after the img element (and inside the div). Then

test it, and note that your page now has a 500-pixel-high blank space just below the image. That's the canvas, but we want it in front of the image, not underneath.

To move the canvas in front of the image, you need to use CSS absolute positioning. Simply insert style="position:absolute;" in the img tag, and style="position:relative;" in the opening canvas tag. Although it isn't necessary here, I also recommend adding position:relative; to the style of the enclosing div.

Basically, position:absolute tells the browser to go ahead and place the element as usual, but then back up and place the *next* element as if the absolutely positioned element weren't there. The explanation of position:relative (relative to not setting the position property at all) is more technical, but it works. In any case, please make these changes now and check that the page layout looks the same as it did before the canvas was there.

#### 2. Draw on it using JavaScript.

You now have an invisible space where you can draw. To actually draw on it, you need to add some JavaScript to your page.

JavaScript is a programming language that you can use to manipulate anything and everything on a web page, changing the page from mere static information into a dynamic program (usually with user interactivity). To insert JavaScript code into a page you use the script element, like this:

```
<script>
  // JavaScript code goes here.
</script>
```

The script can be placed inside either head or body, but I usually put it in the latter, after all of the page's visible content. So please do that, and then put the following JavaScript code inside the script:

```
var theCanvas = document.getElementById("theCanvas");
var theContext = theCanvas.getContext("2d");
theContext.beginPath();
theContext.arc(300, 50, 5, 0, 2*Math.PI);
theContext.fillStyle = "red";
theContext.fill();
```

The first line searches your page for an HTML element with id="theCanvas", and gives your JavaScript code access to this element through a variable of the same name. (You *could* use a different name for the variable, and actually most browsers will create a variable with the same name for you by default, but it's a good idea to make it explicit.) The second line is needed for technical reasons: It gives you access to what's called the canvas's *graphics context*, which is what you use for the actual drawing. Once you have that, the next four lines draw a red filled circle on the canvas, centered at pixel coordinates (300, 50), with the second coordinate measured *down from the top*. The radius of the circle is 5 pixels, and the last two arguments of the arc function specify the beginning and ending angles (in case you ever want to fill a pie slice instead of a full circle).

Go ahead and test this JavaScript code, then try changing the location and size and color of the red circle. (Besides common color names like "red", you can use hexadecimal codes of the form

"#rrggbb", just as in CSS.)

These six lines of JavaScript illustrate several important features of the language's syntax:

- All words are case-sensitive.
- The = symbol is used to set the value of a variable.
- Every statement ends with a semicolon.
- Function arguments are in parentheses, which are required even when there are no arguments.
- A period (or "dot") indicates variables and functions that belong to a particular object.

In fact, most of the syntax of JavaScript is ultimately borrowed from C, and from its descendants such as C++ and Java, so if you already know one of these languages, you won't need to learn much new syntax. However, the higher-level features of JavaScript are quite different from these other languages and may take more getting used to. For example, all JavaScript variables are declared using the var keyword, rather than by specifying a particular data type such as int or double. This is because JavaScript variable types are determined automatically as your program runs—a feature that many of us find unsettling.

#### 3. Put it in motion.

Our next goal is to put that red dot (which will represent the launched projectile) into motion.

First, insert two more lines of JavaScript, somewhere before beginPath, to define general coordinates x and y for the center of the dot:

```
var x = 300;

var y = 50;
```

Then, in the line that calls the  $\mathtt{arc}$  function, change the constants 300 and 50 to  $\mathtt{x}$  and  $\mathtt{y}$ . These changes should have no effect by themselves, but now you can move the four graphics instructions into a self-contained function that will work for any  $\mathtt{x}$  and  $\mathtt{y}$  values. To create your own function in JavaScript you use the function keyword (analogous to  $\mathtt{var}$  for variables), like this:

```
function drawProjectile() {
    theContext.beginPath();
    theContext.arc(x, y, 5, 0, 2*Math.PI);
    theContext.fillStyle = "red";
    theContext.fill();
}
```

(Again the syntax is taken from C, using curly braces to group together the statements that are part of the function definition.) In addition, you need another function that will change the values of x and y. Let's call this one moveProjectile:

```
function moveProjectile() {
    x += 1;
    drawProjectile();
    window.setTimeout(moveProjectile, 1000/30);
}
```

Here the first statement adds 1 to the value of x (you could alternatively write x = x + 1); the second statement calls the drawProjectile function; and the third statement tells your browser to call the moveProjectile function again in 1000/30 milliseconds (so it will be called about 30 times per second). This is the "trick" that keeps the animation going.

You still need to get the process started, by calling moveProjectile once from outside of any function:

```
moveProjectile();
```

(Put this statement right after your variable initializations.) When you test this code, you should see the red dot being drawn repeatedly at rightward-moving locations across the window, right up to the edge of the canvas.

To merely *move* the red dot, erasing it from its previous location, you can just erase the whole canvas at the beginning of the drawProjectile function:

```
theContext.clearRect(0, 0, theCanvas.width, theCanvas.height);
```

(The clearRect function can be used more generally to erase any rectangular area, where the parameters are the left, top, width, and height, respectively, all in nominal pixel coordinates. Here you also see that the theCanvas object provides width and height variables that you can use as needed.)

Again, be sure to test this change to your code.

This method of erasing and redrawing the projectile might seem cumbersome, especially compared to higher-level graphics environments in which you could simply change the values of x and y and see the object move automatically in response. That higher-level approach is called *retained-mode graphics*, and it's more convenient for animating the motions of a limited number of objects. The canvas element, instead, implements what's called *immediate-mode* graphics, in which each shape is immediately drawn as pixels and only the pixels themselves are stored in the computer's memory. Immediate-mode graphics provides better performance for complex images that cannot be reduced to a few hundred (or fewer) discrete lines or polygons. Should you ever want a retained-mode environment for your animations, you can learn one of the JavaScript <u>libraries</u> that implements retained-mode graphics on top of canvas.

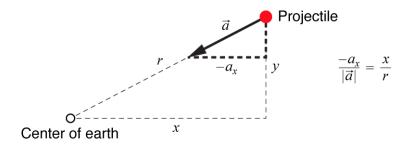
#### 4. Put in the physics.

Of course you don't want the projectile to move in an arbitrary way; you want it to move in response to earth's gravity. So please remove the statement x += 1 from your moveProjectile function and replace it with some physics calculations, starting with these variable definitions:

Here you see how to enter numbers in scientific notation and how to do some simple arithmetic,

including a square root. Note that JavaScript (like C and Java) has no exponentiation operator, so to square a number you just multiply it by itself. As in Java and C++, the double-slash indicates the start of a comment, which continues to the end of the line. I should also explain that by declaring these new variables (with var) inside your moveProjectile function you are making them *local* to that function, so they aren't available to any of the rest of your code, and their values disappear between successive calls to the function. The alternative is to make a variable *global*, which you do by declaring it outside of any function (as you did with x and y). It's a good habit to make variables local whenever possible. (Unfortunately, JavaScript doesn't actually require you to declare variables at all—but any undeclared variables automatically become global, and this can result in buggy code. Please declare all your variables!)

The last line of code above calculates the magnitude of our projectile's acceleration, using Newton's law of universal gravitation. (The projectile's mass cancels out.) Notice how I've assumed that  $\mathbf{x}$  and  $\mathbf{y}$  are now measured in meters, from the center of the earth. We can find the  $\mathbf{x}$  and  $\mathbf{y}$  components of the acceleration vector with the aid of a sketch:



From the similar triangles you can see that the x component of the acceleration vector is minus its magnitude times the ratio x/r, and similarly for the y component. The code to compute these components is therefore:

From these acceleration components you can now update the components of the projectile's velocity, and then use these to update the position:

I've already chosen a dt value of 5 seconds, which is quite short compared to the natural time scale for a low-earth orbit (on the order of an hour). But because dt isn't infinitesimal, this algorithm predicts the projectile's motion only approximately. The particular approximation used here is sometimes called the <u>Euler-Cromer algorithm</u>; it improves upon the standard Euler algorithm by using the *new* velocities, rather than the old ones, to update the positions.

Besides inserting all this code into your moveProjectile function, you also need to declare and initialize the velocity variables. Since their values need to persist through multiple calls to moveProjectile, you should declare them at the global level:

```
var vy = 0;
```

While you're at it, you'd better properly initialize the position:

```
var earthRadius = 6371000;  // meters
var mountainHeight = earthRadius * 0.165;  // chosen to match image
var x = 0;
var y = earthRadius + mountainHeight;
```

Finally, in your drawProjectile function, you need to convert the physical coordinates into canvas pixel coordinates:

```
var metersPerPixel = earthRadius / (0.355 * theCanvas.width);
var pixelX = theCanvas.width/2 + x/metersPerPixel;
var pixelY = theCanvas.height/2 - y/metersPerPixel;
```

The number 0.355 in the first line is chosen to match the image, and the minus sign in the last line is because the pixel coordinate is measured down from the top. Now just use pixelx and pixely instead of x and y in the arc function (there's no need to round these values to integers), and it should all work. Try it!

One piece of physics that's still missing is the earth's hard surface. To stop the simulation when the projectile hits the ground, all you need to do is enclose most of the code in moveProjectile in an if statement:

```
var r = Math.sqrt(x*x + y*y);
if (r > earthRadius) {
   var accel = newtonG * earthMass / (r * r);
   // and so on...
}
```

(Let's continue to neglect the rather dramatic effects that earth's atmosphere would have on a projectile moving several kilometers per second!)

Of course, the whole point of the simulation is that for sufficiently high initial speeds, the projectile will *never* hit the ground. In that case you probably want to stop it after a complete orbit. You can do so by saving the old value of  $\mathbf x$  just before you update it, and then testing whether  $\mathbf x$  has changed from negative to positive:

```
var lastx = x;
x += vx * dt;
y += vy * dt;
drawProjectile();
if (!((lastx < 0) && (x >= 0))) {
    window.setTimeout(moveProjectile, 1000/30);
}
```

(Note the use of the logical operators && for and and ! for not.)

#### 5. Debug JavaScript errors.

It would be astonishing if by this time you haven't made at least one typographical or other error

while entering your JavaScript code. In that case, you may have found that the code simply fails to run, with no apparent explanation.

Fortunately, you can get an explanation in such cases by viewing your browser's JavaScript console. The way to view it varies with different browsers; it may be called something like Web Inspector, perhaps under a special Developer menu that you have to enable. If you haven't already, take the time to find it now, and then check out what it shows when you introduce a fatal error into your code (perhaps misspelling one of the variable names). The console should display an error message, the most important part of which is the line number of the offending code. This would also be a good time to configure your text editor to display line numbers in the left margin, if it isn't already doing so.

Of course, the more troublesome bugs are those that don't generate error messages but still cause your program to produce incorrect results. To diagnose those, you can use the console.log function to send your own diagnostic messages to the console:

```
console.log("Entering drawProjectile function");
console.log("Current position is " + x + ", " + y);
```

The second example also illustrates how the + operator can be used to concatenate text strings, and how JavaScript automatically converts numbers to strings when necessary.

#### **Review**

This was a long section of the tutorial, but it covered a lot of material: the JavaScript programming language, the HTML5 canvas element, and an algorithm for integrating Newton's second law to predict the motion of a projectile.

At the end of this packet are one-page reference sheets on JavaScript and Canvas graphics, which give an overview of the features covered here and a number of other useful features. The example pages that accompany this tutorial also demonstrate several features of these technologies.

For more comprehensive references, you can try <u>W3Schools</u> and also the <u>HTML5 Canvas</u> <u>Tutorials</u> site. <u>Stack Overflow</u> is again an essential resource. The definitive book on JavaScript is David Flanagan's <u>JavaScript: The Definitive Guide</u>, which includes a chapter on canvas graphics. A nice book that's dedicated to the latter is <u>Core HTML5 Canvas</u> by David Geary.

To learn more about solving Newton's second law (and other differential equations) on a computer, some of the best textbooks are <a href="Numerical Methods for Physics">Numerical Methods for Physics</a> by Alejandro Garcia, <a href="Computational Physics">Computational Physics</a> by Giordano and Nakanishi, and <a href="Introduction to Computer Simulation Methods">Introduction to Computer Simulation Methods</a> by Gould, Tobochnik, and Christian.

# **User Interaction**

After finishing the <u>previous section</u> of this tutorial, you should have a working web simulation of Newton's Cannon, with animated graphics. But the simulation runs just once when the page loads, and the projectile's initial velocity is hard-coded to a fixed value. Your next task is to hand the control of the simulation over to the user.

#### 1. Add a "Fire!" button.

First let's add a "Fire!" button to start the simulation. To put a button to your page you can use the HTML input element, like this:

```
<input type="button" value="Fire!">
```

Go ahead and insert this code, below the div that contains the image and the canvas, but above the paragraph of text, and check that the button appears. To center the button, put it inside its own div with a style attribute that sets text-align:center. (You could put the button inside a paragraph instead of a div, but it's best to reserve paragraphs for text.)

Next you need a JavaScript function to call when the button is pressed. This function should initialize the position and velocity variables, then call moveProjectile to get the simulation started:

```
function fireProjectile() {
    x = 0;
    y = earthRadius + mountainHeight;
    vx = 6000;
    vy = 0;
    moveProjectile();
}
```

The position and velocity variables still need to be declared at the global level (so their values persist after this function exits), but those declarations can now be shortened to a single line:

```
var x, y, vx, vy; // position and velocity
```

To call fireProjectile when the button is pressed, you can simply set the button's onclick attribute inside its HTML tag:

```
<input type="button" value="Fire!" onclick="fireProjectile();">
```

Now, when you load the page, you can fire the projectile repeatedly. Try it!

But there's a slight problem: If you impatiently click the Fire! button again, before the projectile has landed, the simulation will restart and will now run twice as fast as before. This is because you've called moveProjectile again while another call to moveProjectile is still pending via the most recent setTimeout, so you end up with two calls to moveProjectile either pending or running at any given time. You can fix this bug by storing the result of setTimeout in a variable:

```
timer = window.setTimeout(moveProjectile, 1000/30);
```

Declare this timer variable at the global level, and then insert the following line at the beginning of fireProjectile:

```
window.clearTimeout(timer);
```

#### 2. Add a slider to set the launch speed.

Finally it's time to let the user adjust the projectile's initial speed. The best way to do this is with a slider control:

```
<input type="range" min="0" max="8000" step="100" value="3000">
```

Insert this tag right after the one for the button, inside the same div, and check that it shows up in your browser. (Support for this HTML5 feature has been slow in coming to certain browsers, most notably Firefox, which didn't implement it until August 2013.) The attribute meanings should be self-explanatory; I've chosen their values based on my personal judgment, which you should feel free to override.

To access the slider's value from your JavaScript code, you need to set its id attribute:

```
... type="range" id="speedSlider" min="0" ...
```

Then declare and initialize a global variable with the same name:

```
var speedSlider = document.getElementById("speedSlider");
```

Now you can simply replace the line in fireProjectile that sets vx with the following:

```
vx = Number(speedSlider.value);
```

(The Number function forces JavaScript to convert the value from a character string to a number immediately, rather than later when you try to do arithmetic with it. This doesn't matter here, but I've found that it can sometimes dramatically affect performance so I consider it a good habit. The reason why the value is a string to begin with is rooted in the conventions used for all input controls.)

Once again, be sure to test your code after making these changes.

#### 3. Add a numerical readout for the slider.

Unfortunately, the slider doesn't automatically come with a numerical readout to show the user its value. But you add one pretty easily.

Start by adding the following line of code to your HTML, in between the button and the slider:

```
Initial speed = <span id="speedReadout">3000</span> m/s
```

The span element is the in-line <u>version</u> of div; it does nothing inherently to its content, but lets you change the styling or, in this case, assign an id. To access this element from JavaScript, put it into a variable as usual:

```
var speedReadout = document.getElementById("speedReadout");
```

Next, define a function that sets the content of the readout to the slider's value whenever it is called:

```
function showSpeed() {
    speedReadout.innerHTML = speedSlider.value;
}
```

Now you can just add calls to this function as attributes inside the slider's HTML tag:

```
... value="3000" oninput="showSpeed();" onchange="showSpeed();">
```

(Why two different attributes? The first, oninput, is supposed to "fire" whenever the user moves the slider's thumb, while the second, onchange, is supposed to fire only when the thumb is released. But there has been some inconsistency among browsers in implementing these features, so I'm in the habit of using both, just to be safe.)

Just as most web developers frown upon the use of in-line styling via the style attribute, they also tend to frown upon the use of in-line JavaScript via attributes like onclick and onchange. Apparently their view is that all JavaScript code should be segregated in its own separate file, rather than being mixed in with the HTML. That's actually pretty easy to do, but I frankly don't see any advantage to it for the types of applications discussed here.

#### **Review**

The User Interface reference sheet at the end of this packet summarizes the syntax not only for buttons and sliders, but also for other common types of graphical user-interface features such as checkboxes, drop-down menus, and direct mouse (or touch) events on a canvas. The examples that accompany this tutorial demonstrate each of these interaction mechanisms.

If you look up user-interface controls in a more general HTML reference site or book, you'll find them in the section on *forms*. That's because these controls were originally intended for use on forms that users fill out in order to send information back to web servers. Fortunately, the technologies are sufficiently powerful that we client-side web programmers have successfully coopted them. Just don't be surprised if the examples that you find elsewhere are cluttered with extraneous <form> tags and references to "submit" buttons.

# **Finishing Touches**

After finishing the <u>previous section</u> of this tutorial, you should have a fully functional simulation of Newton's Cannon. If it's good enough for you, you can now put this simulation aside and <u>move on</u> to another HTML5 project. On the other hand, if you're a perfectionist like me, you might first want to refine the Newton's Cannon simulation in a few ways.

Each of the following refinements can be made independently, so feel free to skip any that don't interest you.

#### **Trails**

The original Newton's Cannon illustration shows the paths of the projectiles, so perhaps yours should too.

The best way to add "trails" to this simulation is to draw them on a separate canvas, sandwiched in between the existing canvas and the underlying image. You can do this with just four additional lines of code:

- 1. A line of HTML, in between your existing img and canvas elements, to add the second canvas. Give it id="trailCanvas", with the same dimensions as the others and position: absolute.
- 2. A declaration/initialization for a JavaScript variable trailCanvas, otherwise identical to the one for theCanvas.
- 3. Similarly, a declaration/initialization for the new graphics context (call it trailContext), analogous to theContext.
- 4. Finally, in your drawProjectile function, a line to draw a small dot at the projectile's current location. This is easiest if you actually make the dot a rectangle, because there's a one-line convenience function for drawing rectangles:

```
trailContext.fillRect(pixelX-0.5, pixelY-0.5, 1, 1);
```

The first two parameters are the coordinates of the rectangle's upper-left corner, so I've subtracted 0.5 to center it precisely; the third and fourth parameters are the rectangle's width and height.

The new canvas's fillStyle defaults to black, but feel free to change this to a different color if you prefer.

When you test these changes, you should find that the dots are close enough together to form a continuous curve. (If they weren't so close, and you still wanted a continuous curve, you could use the moveTo and lineTo functions to draw lines. This would also require a couple of new global variables to store the projectile's previous location, and a bit of code to ensure that you don't connect the end of one path to the beginning of the next.)

Notice that the canvas can accumulate an unlimited number of trails, with thousands upon thousands of individual dots, with absolutely no performance penalty. This is one advantage of immediate-mode graphics over retained-mode graphics.

It's also a nice touch to add a button to clear the trails and start over. Add a line of HTML code for this new button, analogous to that for the "Fire!" button. Set the onclick attribute to call a new function called clearTrails, and have this function call trailContext.clearRect in a way analogous to the first line of your drawProjectile function.

#### 3-D shading

Newton's illustrator used shading to give the planet a three-dimensional appearance. You can do the same for your projectile by filling it with a *radial gradient* instead of a solid color. In your drawProjectile function, just replace the line that sets fillStyle with the following four statements:

The parameters of the createRadialGradient function define two circles, in terms of their center locations and radii (here 1 and 5, respectively). The addColorStop functions then specify the colors on (and beyond) these circles, and the gradient automatically interpolates between them. Try changing the gradient metrics and colors until you're happy with the appearance.

#### **Bigger buttons**

Choosing optimum sizes for things is tricky in a web app, because you don't know the user's screen size. There are several reasons, though, why you might want to make buttons bigger than their default sizes. A quick-and-dirty way to enlarge them is to specify a larger font via the CSS font-size property. But I've found that styling buttons is tricky, producing inconsistent results in different browsers.

A robust, but cumbersome, alternative is to avoid buttons entirely and instead just use links. The HTML code for your Fire! button would then be:

```
<a class="customButton" href="javascript:void(0)"
    onclick="fireProjectile();" ontouchstart="">Fire!</a>
```

Try this out and check that the "button" still works. I don't actually understand the reason for javascript:void(0), but seemingly knowledgable people <u>recommend</u> it. The empty ontouchstart attribute is a detail that improves the behavior on mobile devices.

The class attribute is a convenient way of applying the same styling to multiple elements. To make it work, you define the class inside a style element in the head portion of your source file. After much fiddling, I've settled on the following styling for my custom buttons:

```
<style>
.customButton { /* style a link as a push-button */
    display: inline-block;
    width: 60px;
    height: 24px;
    line-height: 24px;
    font-size: 15px;
```

```
font-family: sans-serif;
    text-align: center;
    color: black;
    background: -webkit-linear-gradient(white, #eeeeee, #eeeee, #e0e0e0);
    background: linear-gradient(white, #eeeeee, #eeeeee, #e0e0e0);
    text-decoration: none;
    border: 1px solid gray;
    border-radius: 5px;
    -webkit-user-select: none;
    -moz-user-select: -moz-none;
    -ms-user-select: none;
    user-select: none;
    cursor: pointer;
    -webkit-tap-highlight-color: rgba(0,0,0,0);
  .customButton:active {
    background: -webkit-linear-gradient(#909090, #808080, #808080, #707070);
    background: linear-gradient(#909090, #808080, #808080, #707070);
  }
</style>
```

Yes, it's cumbersome, and I don't claim that it's perfect. I hope you can guess what most of these CSS properties do. The gradients use various gray levels to give the buttons a nice 3-D appearance. (You can, of course, use brighter colors if you don't think they'll be too distracting.) Some of the property names are nonstandard, specific to particular browsers or groups of browsers (Webkit, Mozilla, and Microsoft). The dot before customButton indicates that we're defining a class, and the curly braces enclose all the property settings that belong to that class. The active pseudo-class changes the button's color while it is being pressed.

#### Styling the slider

Slider controls look a little different in each different browser—with the exception of Internet Explorer, in which they look a *lot* different. In IE the default width is much larger, and the default "padding", or extra space, above and below the slider is absurdly large. IE also puts ugly tick marks along the slider's track, and puts a pop-up numerical readout, rounded to the nearest integer, above the slider while you're adjusting it.

Here's the styling that I use to fix all these issues (formatted for use inside the style element in your page's head):

Even with these changes, sliders will look very different in IE than in other browsers. But the remaining differences are primarily a matter of taste, and there's something to be said for keeping

the appearance consistent within each browser.

#### Fixed-width speed readout

You may have noticed that if you reduce the launch speed below 1000 m/s, all the GUI controls shift as they are re-centered to accommodate the loss of a digit in the speed readout. If you find this behavior annoying, you can easily fix it with a bit of styling:

Specifying the width in em units (each equal to the font size in pixels) is more robust than using pixel units if you later decide to change the font size. But this fix is still a bit of a kludge, because I determined the optimum width by trial and error and the number is somewhat font-dependent.

If you now view the page on a smartphone, however, you'll see that the readout looks funny because the number is in a smaller font than the text around it. That's because smartphones automatically enlarge font sizes for readability when a "block" (such as a paragraph or a div) is wider than a certain amount (typically 320 pixels). Giving the readout a fixed width required making it its own block, so now it doesn't get enlarged. The best fix, at least for the most common mobile browsers, is probably to turn off the automatic enlargement by inserting

```
-webkit-text-size-adjust:100%;
```

into the style of the enclosing div.

#### **Special characters**

As a final tweak to the appearance of the line of GUI controls, you can insert a little extra space to separate logically distinct elements. One way to do this is by adjusting the left and right margins, but a slightly easier method is to insert two or three "non-breaking space" characters,

into the HTML just after the Fire! button (and two or three more just before the Clear button, if you've implemented that).

Another nice aesthetic touch is to change the straight typewriter-style apostrophe in Newton's to a pretty typographer's apostrophe (or right single quote):

You can use similar syntax in your HTML to insert a wide variety of special characters, inluding math symbols, Greek letters, and accented letters. Each code begins with an ampersand and ends with a semicolon, with a unique few-letter sequence in between. Several of the most useful are listed on the accompanying <a href="https://example.com/HTML">HTML</a> and styling reference sheet, which also provides a link to a complete list.

#### More tweaks for mobile

On touch screen devices, touching an element on a web page can sometimes make the browser

think you want to select it for copying. That makes sense for text, but not for most GUI controls. I've found this behavior particularly annoying for sliders, so I generally put the following CSS into the <style> element in my document's header:

```
input {
  -webkit-user-select: none;
  -moz-user-select: -moz-none;
  -ms-user-select: none;
  user-select: none;
}
```

Another issue with mobile devices is that they need to know or assume a value for the full width of your web page in pixels—and on iOS (at least), this number defaults to an annoyingly large value of 980. (I try to keep the content of my pages considerably narrower, so they won't monopolize my laptop screen or exceed the width of an 800-pixel projector.) Fortunately, you can change this default by putting another meta tag into your page's header:

```
<meta name="viewport" content="width=640">
```

As long as users are holding their devices in portrait orientation, the optimum setting for the content width is just a little more than the width of your page's body (or widest block). The down-side is that this makes your page harder to use in landscape orientation, because users probably won't be able to zoom out far enough to see all the important parts at once. Try it, in any case, and decide what setting best suits your layout and the uses you have in mind.

# **Next Steps**

I hope that after <u>finishing</u> the Newton's Cannon simulation you are now eager to go on and create your own web simulations. Of course, every simulation is different, and will present its own challenges. Here, though, is some general advice on how to proceed.

#### Review the reference materials.

Spend some time looking over all of the reference materials that accompany this tutorial. In these materials I've tried to concisely summarize most of the HTML, CSS, and JavaScript features that I've used in creating physics simulations for the web. If you invest some time to learn what's in these materials, then you'll know where to quickly look up most of the vocabulary and syntax you need.

#### Look at all the sample pages.

To complement the dry austerity of the reference sheets, a variety of sample pages also accompany this tutorial. Please try each of these, and then use your browser's View Source command to see how they work.

You may also wish to look at some more elaborate examples of HTML5 physics simulations. Here is a list of links to example simulations, by myself and others, whose source code you should mostly be able to read after completing this tutorial.

#### Modify the sample pages.

I also recommend downloading each of the sample pages to your own computer, so you can use them to tinker and explore. Here are just a few ideas:

- Modify the <u>Canvas Graphics and GUI</u> sample page to add more color choices, a slider to
  control the opacity, and/or one or more additional shapes. If you're feeling ambitious, modify
  it so you can draw a shape by clicking on (or touching) the canvas instead of pressing the
  button.
- Modify the <u>Simple Paint</u> sample page by adding a slider to set the brush size and a dropdown menu to select the color. You could even try adding more sophisticated features such as different brush shapes and airbrush effects.
- Modify the <u>Spaceship</u> sample page to put the spaceship's speed and direction under the user's control.
- Modify the <u>Star Motion</u> sample page to draw star trails as in a long-exposure photo.

#### Choose a project that interests you.

Enough of my ideas! Now that you've explored and practiced for a while, I hope you're ready to create a web simulation of your own design. Now would be a good time to think over your ideas and choose one that seems reasonably straightforward to code. Think about what you want the simulation to accomplish, and how you can implement the needed features using the tools you've learned in this tutorial. Sketch what you want the simulation to look like on the screen, and jot down whatever other thoughts you have about the details.

The rest of this document discusses a few miscellaneous issues that you may need to consider as you develop your own simulations in HTML5 and JavaScript.

#### Think carefully about interface design.

While it's natural for a physicist to focus on the underlying calculations that a simulation performs, your audience will learn more (and will grow to include more users) if you also make your best effort to design a good user interface. Doing so often requires quite a bit of creativity and hard work.

Here are some of the questions you should ask yourself as you design a user interface:

- Which type of GUI control (e.g., checkbox, radio buttons, slider) is the most appropriate and the most intuitive for each simulation setting or action?
- How wide a parameter space do you want your users to explore? Will they be able to find the
  parameters you have in mind without having to read lengthy instructions? Or are you being
  so restrictive that there is no room for creative exploration?
- Who are the intended users of your simulation? Students who need to use it to complete an assignment, or members of the general public who are merely curious or want to be entertained? Will your intended users have the patience to read whatever instructions are needed to successfully use the simulation?
- What is the best visual design for your simulation? Should the graphics be realistic or abstract? How can you make the layout, fonts, and colors aesthetically pleasing without distracting from the content? Should all of the controls be visible at all times, or should some of them be initially hidden?
- Is your interface equally usable on both traditional computers and mobile devices? How wide a range of devices can you feasibly support?

While there is no universal recipe for a good user interface, I still rely heavily on three principles emphasized in the original (1985) edition of <a href="Inside Macintosh">Inside Macintosh</a>, the programmer's manual for the first mass-market computer with a graphical user interface. Those principles are *responsiveness*, *permissiveness*, and *consistency*.

Responsiveness means that user inputs have immediate, rather than delayed, effects. Try to provide some instant visual feedback whenever the user checks a box or drags a slider. Don't make the user choose multiple settings before the simulation even starts.

*Permissiveness* means that your simulation should allow the user to take any reasonable action at any time. Try not to disable controls while the simulation runs. You want the user to feel in control of the software—not the other way around.

Consistency means using input mechanisms that are similar to what users already understand from using other software. Use push-buttons, checkboxes, and menus in standard ways, and give them standard appearances. If you must invent an unusual input mechanism, try to make it mimic some real-world action such as pointing or pulling. Don't force users to read lengthy instructions just to figure out how to control your simulation.

Inevitably, there will be times when you need to violate all of these principles. Just be sure to violate them thoughtfully, not accidentally.

#### **Consider performance limits.**

For some types of simulations, computational speed is critical. Examples would include many-body simulations of 100 or more particles, solving partial differential equations, and anything that needs to quickly generate intricate, detailed images.

The good news is that in most modern browsers, JavaScript runs only a little slower than native code. That's because modern browsers employ just-in-time compilation, rather than interpreting your source code anew during every iteration of a loop. On today's personal computers, this means you can perform about a billion elementary computations per second. That's about 40 million computations per animation frame, at an aesthetically pleasing frame rate.

Of course, some users may be running your simulation on mobile devices that are slower than your personal computer by an order of magnitude or more. For this reason it may be a good idea to let the user adjust whatever parameter determines the required computational horsepower—such as the number of particles or the resolution of a grid.

One quirk of JavaScript is that it provides no true multi-dimensional arrays. You can always create an array of arrays, but I've found that this can impair performance. Instead I recommend using large one-dimensional arrays and indexing into them manually: index = x + xSize\*y. I've also found that it helps to initialize such an array sequentially, nesting the x loop inside the y loop rather than the other way around.

In general, JavaScript just-in-time compilers tend to be temperamental, because they must be able to adapt to bizarre actions like suddenly changing the type of a variable from an integer to a string (or to something even more complex). To optimize performance your code must behave predictably, rather than taking advantage of all of JavaScript's permissiveness.

To generate intricate images, you may need to manipulate the pixel data directly via the createImageData and putImageData functions. The <a href="Two-Source Interference">Two-Source Interference</a> sample page demonstrates this method.

#### Learn about libraries.

If you want to do something complicated for which JavaScript doesn't provide a built-in solution, you may be able to build on the work of other programmers by using one of the many freely available JavaScript code libraries. Using a library can save you a lot of work, and you'll often end up with better results. On the other hand, each library comes with its own learning curve, and using one often means sacrificing some flexibility and performance.

Personally, I try to use libraries only when they'll help me do something that I'm convinced I otherwise wouldn't be able to do at all (given my limited time and background). This means that I avoid using libraries when they would merely make a task slightly easier: experience has taught me that more often than not, the time spent figuring out how to use the library outweighs the time saved coding with it. Of course, your needs and priorities may be different from mine. But my general aversion to libraries means that I know very little about most of them, so I'm not well qualified to make recommendations. Here, in any case, is an inevitably incomplete list of libraries that, from what I've heard, might be useful (or even essential) for certain types of physics simulations:

<u>¡Query</u> is a general-purpose library for manipulating the elements of web pages and

circumventing many incompatibilities between browsers. It's very widely used by professionals, and many other libraries depend on it. One of these is **jQuery UI**, a library of user-interface widgets. A companion library called **jQuery Mobile** is also compatible with touch-screen devices.

- <u>Flot</u> is one of many libraries for plotting two-dimensional charts, built on top of jQuery. It can produce some slick-looking plots with little effort, but it lacks some features you might need for scientific plotting.
- <u>CreateJS</u> is a suite of libraries for retained-mode graphics, animation, and sound. It seems to be aimed at developers who are already familiar with Flash.
- <u>Paper.js</u> is a sophisticated retained-mode graphics library built on top of canvas. <u>Super Planet Crash</u> uses this library.
- GlowScript is a retained-mode 3-D graphics library that is similar in many ways to <u>Visual Python</u>. It can be used as a library in your own projects, but it also comes with its own webbased editing and hosting environment that frees you from having to provide HTML boilerplate. Unfortunately it is not yet compatible with most mobile devices.
- <u>Easy Java/JavaScript Simulations</u> is not a stand-alone library but rather a front-end development tool that simplifies the process of creating many types of physics simulations. The latest version can output JavaScript (in addition to Java) code.
- Numeric Javascript is a library for linear algebra and other numerical tasks.
- MathJax is a library for mathematical typesetting in HTML pages. It can translate LaTeX code into high-quality typeset equations.

To use a JavaScript library, your page imports it just like any other external script, via a (separate) script element with a src attribute and no content:

You can either host a copy of the library on your own server, or provide (in the src attribute) the URL of a copy that is hosted on a so-called content distribution network (CDN). Follow any of the links above for specific instructions on obtaining and using that library.

#### Choose a web hosting service.

Once your web application is complete, how do you share it with the world?

Ideally, your school or institution provides a server to which you can upload web pages of your own creation via FTP or the equivalent. But not everyone is so fortunate. More and more schools are now using content management systems for all their web-based course materials, and these systems won't always play nice with your page design and JavaScript code. Or you may be in a situation where you have no institution-provided web hosting at all.

The best solution in these cases is probably to just pay for your own web hosting service. I can't recommend a particular vendor, but there are many choices and it shouldn't cost more than a few dollars per month. You're looking for "shared" hosting (the inexpensive kind, for sites that won't get huge amounts of traffic), and you should be sure that the service includes FTP access so you can directly upload your finished web pages.

There are also free hosting services out there in the Cloud, but they all seem to involve

compromises. For a web simulation that uses no images or other external files, a good option is <u>CodePen</u>, a site intended for sharing code among web programmers. This site requires that you license your code for unlimited reuse, with attribution (like the examples in this tutorial). You can also try more generic free hosting services such as <u>Weebly</u> or <u>Google Sites</u>, but these typically give you less control over the appearance of your pages, or force you to treat stand-alone pages as attachments (to be downloaded and saved by users, rather than immediately displayed in browsers).

## Join the Physics HTML5 Google Group.

I've set up a Google Group for physicists and physics teachers to discuss web programming in HTML5 and JavaScript. You can view the group discussions at:

https://groups.google.com/forum/#!forum/physicshtml5

To join the group (so you can post your own messages), go to this URL and click "Apply to join group". (You need to have a Google account.) My hope is that, over time, this forum will remedy many of the deficiencies of the tutorial you have just completed.

# **HTML and Styling Reference**

HTML template  Text structure	html <html></html>
Text su dotale	<pre><h2>Sub-heading</h2> <h2>Sub-heading</h2> <h3>Sub-sub-heading</h3> Paragraph       code&gt;Monospaced <sub>Subscript</sub>   <sup>Superscript</sup> </pre>
Special characters	<pre>&lt; &lt; &gt; &gt; &amp; &amp; ° ° • • Δ Δ π π</pre>
HTML comment	This comment won't affect the page display
Link	<a href="http://site.com/path/file.html">Link text goes here</a>
Image	<pre><img height="100" src="imagefile.png" width="200"/></pre>
Lists	<pre><ul> <!--bulleted list--></ul></pre>
Generic elements	<pre><div>Generic block element</div> <span>Generic in-line element</span></pre>
Fonts and colors	<pre><h1 style="font-size:24px; color:red;"> <body style="font-family:sans-serif; background-color:#d0d0d0;"></body></h1></pre>
Centered text	<h1 style="text-align:center;">Centered Heading</h1>
Floating block	<pre><div style="width:200px; float:right; margin:0 0 10px 10px;">   Floating content (often an image) goes here.   (Note the 10-pixel bottom and left margins.) </div> This paragraph will wrap around the floating block.</pre>
Centered layout	<pre><div style="width:600px; margin-left:auto; margin-right:auto;">   This 600-pixel-wide block will be centered in the window. </div></pre>
Two-column layout	<pre><div style="width:610px;">     <div style="width:400px; float:left; margin-right:10px;">         Left column content     </div>     <div style="width:200px; float:left;">         Right column content     </div> </div> </pre>

## **JavaScript Reference**

```
Script tag
                   <script> /* JavaScript code */ </script> (code in same file)
                   <script src="myScript.js"></script>
                                                                (code in external file)
Variable declaration
                                           // declares variable scope
                   var myVariable;
                                           // combines declaration and initialization
                   var myVariable = 42;
Arithmetic
Shortcuts
                                   /=
                        _=
                                         ++
Relations
                   ==
                         ! =
                                <
                                     <=
                                            >
                                                 >=
Logic
                                         true false
                   && (and) | | (or) ! (not)
Math object
                   Math.PI
                                    Math.E
                                                      Math.sqrt(x)
                                                                       Math.pow(x,y)(x^y)
                   Math.cos(t)
                                    Math.acos(x)
                                                      Math.exp(x) (e<sup>x</sup>)
                                                      Math.log(x) (natural log)
                   Math.sin(t)
                                    Math.asin(x)
                   Math.tan(t)
                                    Math.atan(x)
                                                      Math.atan2(y,x) (angle to point x,y)
                   Math.max(x,y)
                                    Math.min(x,y)
                                                      Math.abs(x) (absolute value)
                   Math.round(x)
                                                      Math.ceil(x) (round normally, down, or up)
                                    Math.floor(x)
                   Math.random() (pseudo-random number between 0 and 1)
Control structures
                   if (balance <= 0) {
                                           while (t < 10) {
                                                                for (i=0; i<100; i++) {
                     color = "red";
                                                                  total += i*i;
                                             doStuff();
                   } else {
                                             t += dt;
                                                                }
                     color = "black";
                                           }
Function declaration function hypotenuse(a, b) {
                     return Math.sqrt(a*a + b*b);
Arrays
                   planets = ["Mercury", "Venus", "Earth", "Mars"];
                   thirdPlanet = planets[2];
                                                     // numbering starts at 0
                   planetCount = planets.length;
                                                     // 4
                   planets.push("Jupiter");
                                                     // append an element
                   planets.sort();
                                                     // put into alphabetical order
                   x = \text{new Array(N)}; // create array of N elements, 0 to N-1
Objects
                  particle = {name: "electron", mass:9.1e-31, charge:1.6e-19};
                  eOverM = particle.charge / particle.mass;
Formatting numbers
                  myString = Number(Math.PI).toFixed(2);
                                                               // "3.14"
HTML event
                   onclick = "myFunction();"
                                                   (use for buttons, links)
attributes
                   onchange = "myFunction();"
                                                   (use for sliders, etc.)
HTML element
                   id = "myElement"
                                       (in HTML opening tag)
access
                   var myElement = document.getElementById("myElement");
                                                                               (in JavaScript)
Modifying content
                   myElement.innerHTML = "New content";
                   myElement.innerHTML += "Added content";
                   myButton.value = "New button text";
Modifying style
                   myElement.style.color = "red";
                   myElement.style.backgroundColor = "#ffff80";
                   myElement.style.display = "none";
                                                           (hide an element completely)
                   myElement.style.display = "block";
                                                           (unhide a block element)
                   myElement.style.display = "inline"; (unhide an inline element)
Measuring time
                   theTime = (new Date()).getTime(); // time since 1970 in ms
Delayed execution
                   window.setTimeout(myFunction, 10);
                                                           // call myFunction in 10 ms
Debugging
                   console.log("myVariable = " + myVariable);
```

## **Canvas Graphics Reference**

```
Canvas
               <canvas id="myCanvas" width="300" height="200">
                Your browser doesn't support the canvas element; please update!
               </canvas>
               <script>
                var myCanvas = document.getElementById("myCanvas");
                var myContext = myCanvas.getContext("2d");
                // now draw using myContext as shown below
               </script>
Draw a line
              myContext.beginPath();
or polygon
                                             // y is measured down from top
              myContext.moveTo(x1, y1);
              myContext.lineTo(x2, y2);
                                             // coordinates need not be integers
              myContext.lineTo(x3, y3);
                                             // add as many segments as desired
              myContext.strokeStyle = "blue";
              myContext.lineWidth = 2;
              myContext.stroke();
              myContext.fillStyle = "yellow";
              myContext.fill();
                                             // fill the polygon
Draw a rectangle
              myContext.fillStyle = "green";
              myContext.fillRect(x, y, width, height);
Draw a circle
              myContext.beginPath();
               myContext.arc(x, y, r, 0, 2*Math.PI); // use other angles for arcs
              myContext.fill();
                                    // or myContext.stroke() for open circle
Write text
              myContext.font = "30px sans-serif";
              myContext.fillStyle = "#8000ff";
              myContext.fillText("Hello, Canvas!", x, y); // baseline left x & y
Specify colors
              BlueViolet
                                        color name (full list at en.wikipedia.org/wiki/Web_colors)
               #8000ff
                                        hexadecimal rrggbb
                                        values 0 to 255
               rgb(128,0,255)
               rgba(128,0,255,0.4)
                                        with transparency (1.0 is opaque)
               hsl(270,100%,50%)
                                        hue in degrees, saturation%, lightness%
              hsla(270,100%,50%,0.4) with transparency (1.0 is opaque)
Draw an image
              myImage = new Image();
               myImage.src = "imageFile.png";
                                                     // .jpg and .gif work too
                                                      // function to call when ready
              myImage.onload = myFunction;
              myContext.drawImage(theImage, x, y); // upper-left corner x & y
Clear the canvas
              myContext.clearRect(0, 0, myCanvas.width, myCanvas.height);
Coordinate
              myContext.translate(dx, dy);
transformations
              myContext.scale(xScale, yScale);
              myContext.rotate(angleInRadians);
              myContext.setTransform(1, 0, 0, 1, 0, 0);  // reset
Pixel-by-pixel
               image = myContext.createImageData(myCanvas.width, myCanvas.height);
               for (y=0; y<myCanvas.height; y++) {</pre>
image creation
                 for (x=0; x<myCanvas.width; x++) {</pre>
                   index = (x + y*myCanvas.width) * 4;
                                                           // index into data array
                   image.data[index] = redLevel;
                                                           // 0 to 255
                   image.data[index+1] = greenLevel;
                   image.data[index+2] = blueLevel;
                   image.data[index+3] = transparency;
                                                           // 255 is opaque
                }
              myContext.putImageData(image, 0, 0);
                                                           // blast image to screen
```

## **User Interface Reference**

```
Push-button
           <input type="button" value="Start" onclick="myFunction();">
           <script>
             function myFunction() { /* code to execute when button is pressed */ }
           </script>
           <input type="checkbox" id="myCheck" onchange="myFunction();">Label
Checkbox
           <script>
             var myCheck = document.getElementById("myCheck");
             if (myCheck.checked) { /* code to execute if box is checked */ }
           </script>
Radio
           <input type="radio" id="radio1" name="groupName">Label1
buttons
           <input type="radio" id="radio2" name="groupName">Label2
           <script>
             var radio1 = document.getElementById("radio1");
             if (radiol.checked) { /* code to execute if first button selected */ }
               else { /* code to execute if second button selected */ }
           </script>
Drop-down
           <select id="myMenu" onchange="myFunction()">
menu
             <option>Option 0</option>
             <option>Option 1</option>
             <option>Option 2</option>
           </select>
           <script>
             var myMenu = document.getElementById("myMenu");
             var myIndex = myMenu.selectedIndex; // 0, 1, 2, etc.
           </script>
Slider with
           <input type="range" id="mySlider" min="0" max="10" step="0.1" value="5"</pre>
readout
               onchange="myFunction()" oninput="myFunction()">
           Value = <span id="myReadout">5.0</span>
           <script>
             var mySlider = document.getElementById("mySlider");
             var myReadout = document.getElementById("myReadout");
             var myVariable = Number(mySlider.value);
             function myFunction() {
               myReadout.innerHTML = Number(mySlider.value).toFixed(1);
             }
           </script>
Text area
           <textarea id="dataArea" rows="10" style="width:500px;">
             Content
           </textarea>
           <script>
             var dataArea = document.getElementById("dataArea");
             dataArea.innerHTML = "New content";
           </script>
           myCanvas.addEventListener("mousedown", function1, false);
Mouse/touch
           document.body.addEventListener("mousemove", function2, false);
events on a
canvas
           document.body.addEventListener("mouseup", function3, false);
               // and similarly for touchstart, touchmove, touchend
                                                        // e is an event object
           function function1(e) {
             e.preventDefault();
                                                        // cancel browser response
             canvasX = e.pageX - myCanvas.offsetLeft; // location relative to
             canvasY = e.pageY - myCanvas.offsetTop;
                                                        // canvas upper-left corner
               // (often the calculation needs to be more complicated)
```

# HTML/CSS Vocabulary and Syntax

Here is a summary of the most important vocabulary terms that pertain to the structure and appearance of web pages. The following code snippet illustrates each of the terms:

```
<h2 id="s1" style="font-size:20px; text-align:center;">Section 1</h2>
```

(In the online version of this page, you can click or tap on any of the boldface terms below to highlight the corresponding portion of the example.)

- HTML: Hyper-Text Markup Language, the language used to define the logical structure of web pages, breaking them down into various elements using tags.
- **Element**: A logical unit of an HTML document, such as a header, a paragraph, or a link. Every element begins with an opening tag; most also include content and a closing tag. (The code snippet above consists of a single element.)
- **Opening tag**: A unit of HTML code, enclosed in angle brackets, that marks the beginning of an element. Examples include <h2>, , and <a>. Some opening tags, such as <img> and <br/> and <br/> are self-contained elements that have no content or closing tag.
- Closing tag: A unit of HTML code that marks the end of an element, such as </h2>, , and </a>. Each closing tag is the same as the corresponding opening tag, except for an additional slash after the first angle bracket and the omission of any attributes.
- Content: Everything between the opening and closing tags of an HTML element. The content can consist of text and/or nested HTML elements.
- Attribute: A property or characteristic of an HTML element, such as id or src or style or onclick, that is specified inside its opening tag using the syntax attributename="value". Attribute values should be enclosed in quotes (though browsers don't always enforce this rule), and multiple attribute assignments are separated by spaces. The order in which multiple attributes are specified doesn't matter.
- CSS: Cascading Style Sheets, the language used to specify the appearance (e.g., fonts, colors, borders, and placement) of HTML elements. CSS code can be placed in an external .css file or inside the special HTML <style> element, but for simple web pages it is easiest to assign it using the style attribute.
- **Property**: Any characteristic of the appearance or placement of an HTML element that can be specified using CSS, such as color or font-family or margin. The syntax for specifying a property is *property-name:value;* (note the colon before the value, the semicolon after it, and the absence of quotes). The order in which multiple properties are specified doesn't matter.

So in the code snippet above, there are two CSS properties that are specified in the style attribute, which is one of two attributes specified in the opening tag of the h2 HTML element. Got it?:)