Appendix III

Bonus Practical Chapter: Beyond Twine

One of the major virtues of Twine, especially with the Chapbook story format, is the way it connects smoothly to other kinds of software practice. In this encore practical chapter, we look beyond Twine into the coding world we've briefly glimpsed in previous chapters. The projects in this chapter do not use Twine at all but depend instead on HTML and JavaScript, the associated coding language supported by all modern web browsers. We make this departure from Twine not because we've exhausted its possibilities but to serve two complementary purposes. On one hand, we explore some design techniques Twine does not readily support. At the same time, this excursion into the wilderness may put into welcome relief the things Twine makes easier. You may come away from this chapter with a renewed appreciation for Twine, especially if your interests lie mainly with storytelling and turn-based interaction.

This chapter might not be for everyone. As we said all the way back in chapter P-2, code work can be daunting. Twine spares its users significant drudgery and detail, including things like spelling, capitalization, syntax, order of operations, and some basic math. The projects in this chapter involve only very modest code structures, but they do raise the bar of complexity slightly above even our more code-intensive

Twine practicals. If you are willing to trade design constraints for relative simplicity, skip this wilderness tour and stay in the civilized precincts of Twine. It's an eminently livable environment. If you're tempted but uncertain about the exploring that lies ahead, here are some questions for the boarding ramp:

1. Have you ever built your own web page or site, without using a code-generating tool like Dreamweaver?
2. Are you considering working with more sophisticated game design systems such as Unity?
3. Do you like making things that break conventions?

Answering yes to any of these questions qualifies you for the trip. Of course, you may also proceed if you don't have a choice—maybe this chapter has been assigned for class—or if you're just the kind of person who always does things they're told to avoid. We warned you.

Tools and Procedures

Even though this chapter does not work with Twine, supporting materials can be found online at https://github.com/AMSUCF/Twining. You're as welcome to adapt the code examples here as in the other practical chapters. You won't be able to import our .html pages into Twine, obviously, but you can do something just as useful: use the "View Source" feature of your web browser to see our code. If for some reason your browser makes this difficult, we've provided the code in text files with notes in cases where the code is meant to be placed anywhere other than the HEAD division of a web page (mentioned later).

Instead of the Twine application, you will need two other pieces of freely available software: a web browser and a text editor. Any reasonably current browser will do except Microsoft Edge, which for some ill-considered reason makes opening local web pages very hard. At this writing, Google's Chrome browser is generally preferred by web professionals. When we say *text editor*, we do not mean a word processing

program like Microsoft Word but a simpler program designed to produce plain text or ASCII files. If you have a Windows system, type "Notepad" into your search window. You should have an application by this name. On a Mac OS computer, the equivalent program is *Text Edit*. For Linux users there is *VI*. Because we spend significant time writing code, we use a commercial product called *TextPad*, available for Windows and Mac OS. This program adds many useful features but is by no means required.

Do not attempt to build any of the exercises in this chapter with Word or another word processor, even if you choose the text-only save or export option. Word processors often add unseen formatting information that can cripple a web file. In fact, because the level of detail in code structures of these projects is high, you may not want to type what you see at all. You're better off downloading the digital version of this chapter, or even better the complete finished code, from the *Twining* website (https://github.com/AMSUCF/Twining). Open one of these files in a text editor and you can modify and tinker as you like. Because of this recommendation, we'll modify the visual convention of previous chapters. Code fragments will be boxed, but we've omitted the ◊ prompt for text entry.

The kind of DIY web coding we describe here is a two-window experience. You need your text editor and your browser both running. (Two windows does not mean two monitors; you can toggle back and forth.) You need the same file open in both applications—doing this will not cause a crash. The taskbar is your friend. The typical development process starts with entering some code into a web page file, which is a text-only document with the extension .htm or .html. *The extension is not optional.* If you try to open what you think is a web page in a file having any other extension, such as .txt or .rtf, all you will see is the page markup, not what the markup is supposed to produce. Once you've saved your changes in the text editor—*make sure to save*—switch over to your browser and open the page file locally.

If you wonder what *open the file locally* means, we have some news for you. (Also, consider replacing that Microsoft browser.) Browsers

mainly pull in data from the internet at large by communicating with computers called servers via HTTP. However, all web browsers—even Microsoft bleeding Edge—can obtain data by opening a document on the computer on which they are running—which is to say, locally. Civilized browsers allow you to do this by selecting something like "Open" or "Open Local" from the file menu or equivalent. The keyboard shortcut in Windows and Linux is usually CTRL+O (or on Mac OS, Apple key and O). You may need to traverse your file structure to find your page file, which brings us to another important point: *always know where your files are!*

In these Cloud-y days of remote and virtual storage, this principle may need reinforcing. Before starting any web project, we recommend two things. One, put your phone down, or set it aside because it may come in handy if you need to look for help at some point—but *a smartphone is not a coding tool*. Two, a minimum requirement for coding is access to a main directory or, ideally, a graphical desktop. You'll need the other kind of computer, a desktop or laptop machine. (Netbooks are acceptable.) This brings us to the other basic procedure: create a folder (or directory) on your PC desktop—by which we mean the level of file storage you see when you log into your computer—and save your work there. Now we come to the contents of your web page, which will eventually get us to code—but first, one more preparatory section.

Basics of Page Coding

The most important elements of a web page are three *containers*, one called HTML, the second called *head*, and the last called *body*. In terms of *markup*, which is another word for HTML code, a container is a pair of *tags*. A tag is a statement within a set of angle brackets, or a less-than and a greater-than sign:

```
<HTML>
   <HEAD></HEAD>
   <BODY></BODY>
</HTML>
```

The closing tag of a container always begins with a forward slash. Eventually, we'll introduce some other containers, including the all-important container *<script></script>*, within which almost all our JavaScript code will reside.

With these basics out of the way, let's look at the structure of a blank web page. You can type this into your own blank text-editor document, making sure you've saved your file with the extension .htm or .html. There's enough complexity here that you might want to download from our website rather than transcribe.

Here's the page code before any JavaScript happens:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
<html>
<head>
   <meta charset="utf-8"/>
   <title>A page needs a title</title>
   <style type="text/css">
      div{ font: 18pt Cambria; padding: 20px; }
   </style>
   <script>
      //JavaScript goes here
   </script>
</head>

<body>
   <div id="out"></div>
</body>
</html>
```

If this were a book on HTML and general web coding, we'd go into detail about various features of this markup—the document type definition, the meta declaration, and the CSS style sheet, for instance. We might also point out that though web code ignores white space, it's important for readability to keep tags on separate lines whenever you

can. We should also note that this page has a preconfigured style sheet for the projects we intend to build. That style sheet contains a rule for the page element called *DIV*, specifying that any text found within such a container will be in eighteen-point Cambria with twenty pixels of padding around it. You can see a blank DIV container within the body container. This DIV has an ID attribute (informally speaking, a name), which is the word *out*.

We direct your attention primarily to this pair of tags:

```
<script>
</script>
```

This is a script container. It's where all our JavaScript code will be installed. Though you can put a script container at several points in the markup, we'll always place it within the head container. Putting it there ensures that all the page elements we reference are loaded into memory before we start doing things with them.

With this quick tour of page infrastructure done, we can proceed to our first coding project.

Bonus Example 1: JavaScript Text Selector

If you notice a certain resemblance between our first two projects here and example P-3.3, you're paying attention. We'll base all our projects in this chapter on the same kind of substitution grammar we used in chapter P-3, though we'll move on to other techniques besides text generation. In the JavaScript context, we can streamline text generation in a couple of important ways. The first of these improvements is the use of specialized functions, or as programmers call them, *custom methods*. We'll come to the second major refinement, the use of a switch structure, in bonus example 2. For the moment, consider three small but momentous bits of code. If you were typing along, we'd have you enter them into the script container of our blank web page, yielding this:

```
<script>
   function r(range){
      return Math.floor(Math.random()*range)
   }

   function g(source){
      theArray = source.split(",")
      return theArray[r(theArray.length)]
   }

   function outIt(what){
      document.getElementById("out").innerHTML =
      what
   }
</script>
```

As we've mentioned earlier, a function is a set of code statements that can be put flexibly into action or *invoked* as needed. Functions can be invoked by *handlers* when a page loads or some other event takes place in the browser. They can also be invoked by other functions, which is what puts the *fun* in functions. Our three functions are called *r*, *g*, and *outIt*. The writer of a function can use any name not reserved by JavaScript and the browser. The name *r* refers to random numbers. The name *g* suggests "generate" or possibly "gimme." The name *outIt* refers to output or display. As you can see, in each of our custom methods, the name is followed by a word in parentheses. This word is called a *parameter*. It is a special kind of variable that passes a value from the invoking code to the function.

In *r*, the parameter is called *range*, and as you can see, it is used to set a maximum value for the generation of a random number, using the same call to the JavaScript *Math* object we used (somewhat liberally) in chapter P-3. The *r* function supplies a random number between 0 and (because it rounds down) a value one less than the range parameter. This arrangement is perfect for working with arrays, which are series of

items numbered from 0 to a number that is one less than the number of items—for an array of 6 items, 0 to 5. The *return* statement converts the chosen number into output so that if we write *r(5)*, the expression may translate as 0, 1, 2, 3, or 4.

The parameter for g is called *source*, and unlike the range parameter of r, it will be the type of variable called a string—which is to say, a non-mathematical series of letters and numbers such as "hooey" or "44 A.D." or, closer to our context here, "firefly,omnivore,beauty." That last example is a comma-separated string. The g function accepts a string and does two things with it. First, it uses the built-in *split* method to convert the comma-separated list into an array. So "firefly,omnivore,beauty" yields the following array:

```
firefly
omnivore
beauty
```

"Firefly" is item 0, "omnivore" is item 1, "beauty" is item 2, and the length of the array is 3. Next, the *g* function calls the *r* function to generate a random number between 0 and the length of the array, which it uses to make a random selection from the array via square bracket notation. The result is again turned into output using the *return* statement.

The *outIt* function does not use a *return* statement but instead modifies a page element directly. JavaScript can do that! In fact, this is what JavaScript was invented to do. The *outIt* function changes the *innerHTML* property of the DIV we created with the ID of "out"—our display DIV, in other words. To achieve this, *outIt* traverses the DOM of our web page. All pages automatically have such a model, which is a listing of all the various elements they contain along with their names and, in some cases, numbers. The traversal uses something called *dot notation*, which is a familiar convention in web coding. It uses the *getElementById* method of the *document* object to find a DIV called "out." When found, said object's *innerHTML* property—what it contains—is changed from something (or in this case, nothing) to something else.

The custom methods *g* and *r* create a basic framework for randomly selecting from comma-separated strings of text. We can show how they work by adding a few more pieces to our page markup. First, let's build a string with which to test our random selector. We'll add this line at the top of the script container:

```
testSource = "firefly,omnivore,beauty,greeble,Prov
o,whimsical,flatiron,Mme. Ortega y Bullfrog"
```

The final item of this test string shows our system will work even with multiword phrases and punctuation marks, so long as they don't include commas. Notice the final item is not followed by a comma.

Next, also within the script container, we'll add a fourth function immediately after *outIt*:

```
function writeUp(){
   for(var i=0; i<5; i++){
      outIt(g(testSource))+ " "
   }
}
```

This function contains a *for* loop—we've mentioned this element before. It's a way of repeating an instruction a specified number of times and keeping track of the repetitions. We'll see something like it in our further examples. *For* loops are enormously useful. Notice also we've appended a space to our output, using the + operator, which is smart enough to know that it's dealing with strings (because " " is a string) and not numbers. The line within the loop is the key to the show: it calls the *outIt* function, passing it the result of the *g* function, which is passed the test source string to work with. The result on each iteration of the *for* loop is a randomly chosen word from *testSource*, with a trailing space added.

Finally, we need to invoke this new *writeUp* function. We do this by adding an *onLoad* handler to the initial body tag of our page, which is

outside of both the script container and the head container. The body tag will now look like this:

```
<body width: 800px onLoad="writeUp()">
```

When we load the page in our browser, we may see something like the following:

firefly omnivore Mme. Ortega y Bullfrog greeble beauty

Though much more likely, we'll have something like this:

firefly omnivore flatiron greeble flatiron

Repetitions! Yes, well, remember the orange wheelbarrow. As you'll recall, getting a random number generator to avoid repetition takes some work, and for the sake of simplicity, we won't bother this time. Instead, we'll move on to a more sophisticated application of the substitution grammar technique.

Bonus Example 2A: JavaScript Generator (Sequential)

As we said in the previous section, switching to JavaScript and the DIY web brings two major benefits for our text-generation project. Custom methods are the first. The second is a very powerful programming structure called *switch*. In this example, we'll show how to apply this technique. First, some setup. Assume we've opened a new copy of the template web page introduced in the "Tools and Procedures" section. The script container is blank.

In this container, we'll install our three core functions—*r*, *g*, and *outIt*—exactly as they were in the previous example. (In the next example, we'll see a way to avoid this duplication of text. For clarity, we'll hold off on that for the moment.) At the top of our script container, ahead of our core functions, we'll add the following lines:

```
numOptions = 5;
switcher = 0;

function writeUp(){
      for(var i=0; i<numOptions; i++){
            outIt(generate())
      }
}
```

The first two lines are variable declarations. The *numOptions* variable declares the number of grammar options we will have in our switch statement (we're coming to that). The *switcher* variable is a counter we'll use to cycle through our options in sequence. The *writeUp* function resembles its counterpart in the previous example, with two small variations. First, the limit set for the loop is whatever value we assign to *numOptions*. We could simply have put a 5 here, but referring to the variable makes adding grammar options easier. If we add more options, we can just increase the number assigned to *numOptions*. Second, the function called for generating text is not the *g* function directly but a new function called *generate*, which we'll proceed to define. We'll start by roughing in the bare outlines of this function:

```
function generate(){
t = "";

   switch(switcher){

      case 0:
            //you wake up
            break

      case 1:
            //somewhere nearby is the sound of
            a chainsaw
            break
```

```
        case 2:
                //you can smell woodsmoke
                break

        case 3:
                //you remember there was a bonfire
                break

        case 4:
                //go back to sleep
                break
    }

}
```

Before we come to our switch, structure let's briefly discuss the first line, which defines a variable called *t*. *T* is for text—this variable will hold the text of the sentence we are generating. We'll build it up step by step. When we're done, we'll turn it into output with a *return* statement. All that will be explained later.

First, let's discuss the switch structure. As the name suggests, it's a mechanism for directing the operation of the program to certain lines depending on the value of its parameter, the variable named in the parentheses that follow the word *switch*. For this version of our project, we'll use a sequential counting variable, *switcher*. A switch structure has branching options called *cases*. A case is identified by that keyword followed by a value or an expression and a colon. Here we're using integer values, so you see a series of numbers. Switch cases can also be written on strings or logical expressions.

Following each case line are two indented lines. The indentation is required by JavaScript—one of the few cases where it matters. Right now, there are only two lines indented for each case. We'll come to the first one momentarily. For the moment, let's consider the second, which is the single statement *break*. This very powerful command tells JavaScript to break out of its current operation—marching through a series

of statements in a switch structure—and go to the next line outside of that structure. In other words, *break* breaks the action. Every case in a switch structure must have a break statement. Technically, the final case doesn't need one, but you should put one in just in case you decide to add more cases, as you can do if you like.

The other element in our switch structure is a series of sentences preceded by double slashes, one for each case. The // indicates that the following text is a *comment*, material that will be skipped by the JavaScript interpreter when the script is run. A comment introduced by // continues to the end of its line. (There's another construction for multiline comments, but it's not relevant here.) Our comment sentences are, strictly speaking, optional. Each one indicates a template or grammar we'll use for variation. We put them in as a mnemonic device to remind us of the pattern we are matching. For the human writer, they are not optional. You don't have to set things up this way; it's just one model, though it's served us well on many occasions.

We flesh out the switch structure by adding conditions under each case. We'll discuss the first in detail, then look at the completed structure. Case 0 is our first option—programmers like to count from zero. We'll flesh it out as follows:

```
        case 0:
                //you wake up
                t += "you " + g("think you're
                awake,cease to dream,open your good
                eye")
                break
```

That new line is kind of monstrous, but it's very useful. It starts by adding to the *t* variable: that's what "+=" means. Technically, we could have just said "=" at this point because the *t* variable is empty when this line executes—it was declared that way. Since later cases will have multiple assignments to the *t* variable, when we'll need to add rather than replace, we've used "+=" for the sake of uniformity. The first thing we add to our *t* variable is the word *you* followed by a

space. All variations of our sentence will start this way. This brings us to the variations.

Remember, our *g* function takes in a comma-separated string, splits it into an array, and then chooses from the array at random. We're passing along just such a list, consisting of a series of phrases that could follow *you*. There are only three options in our string. That's strictly for convenient readability. You can add as many variations as you like without any change to the *g* function. The function always knows how many options to choose from, no matter how many or few you throw it. In setting up the string passed to *g*, which contains our variations, some very careful typing is required. We have to remember to add no spaces around the commas, to put the quotation marks around the whole series and not single items (as we'd do for an array), and to make sure there are "+" signs connecting all pieces of the template. Let's just say it's easy to get all this wrong.

After a whole bunch of careful typing—or after much sloppy typing and some grumbly debugging—we end up with this completed version of our switch construction:

```
switch(switcher){
   case 0:
      //you wake up
      t += "you " + g("think you're awake,cease
      to dream,open your good eye")
      break;

   case 1:
      //somewhere nearby is the sound of a
      chainsaw
      t += "somewhere " + g("nearby,far away,not
      here")
      t += " is the " + g("sound of a
      chainsaw,smell of mahogany,country of
      smiles")
      break;
```

```
    case 2:
       //you can smell woodsmoke
       t += "you can smell " +
       g("woodsmoke,begonias,an elephant")
       break;

    case 3:
       //you remember there was a bonfire
       t += "you " + g("forget,remember,imagine")
       t += " there was a " + g("bonfire,search
       party,barn raising")
       break;

    case 4:
       //go back to sleep
       t += "the " + g("chaplain,barista,walrus")
       t += " says " + g("go back to sleep,dream
       more carefully,walk on")
       break;
}
```

Again, there are just three options at each substitution point, mainly to make the example marginally readable. You may add more without making any changes to the script. Just add a comma at the end of any of the sequences and type in your additional text. Be sure to preserve the closing quotation marks. You can also add template options by putting more cases into the switch structure. If you do that, however, be sure to increase the value of *numOptions*. If you forget, though the script will run successfully, you'll never see your new sentences.

Speaking of running the script, we need three more lines to make this possible. They go outside of the curly brace that closes the switch structure—the last character you see in the aforementioned block—but before the curly brace that closes the *generate* function as a whole. Here are those final instructions:

```
switcher ++;
if(switcher == numOptions) switcher = 0;
return t;
```

The first command increases the value of switcher by one. The second checks to see if switcher has reached the number set in *numOptions*—in other words, have we run through all five of our grammar templates? Finally, we return *t*, the string variable in which we've been building our variant sentence. (If a function contains a return statement, it must always be the last statement in the function.)

With these details in place, we can run the example and observe the output, which ought to look something like this:

you cease to dream
somewhere not here is the country of smiles
you can smell woodsmoke
you remember there was a barn raising
the chaplain says go back to sleep

or this:

you think you're awake
somewhere nearby is the smell of mahogany
you can smell begonias
you imagine there was a barn raising
the chaplain says dream more carefully

There's enough structure here—the unvarying sequence of those five sentences, designed to read as a certain kind of narrative—to balance the variations, which are written carefully enough, unlike our free-verse excursions in chapter P-3, for at least an approximation of coherence. For our next trick, we'll make some key changes to the example we've just completed to convert it from a sequential generator to a random-access generator.

Bonus Example 2B: JavaScript Generator (Randomized)

Only a few changes are required to convert the sequential generator to random operation. First, delete two of the lines we added to the bottom of the script at the end of the previous example:

```
switcher ++;
if(switcher == numOptions) switcher = 0;
```

Do *not* delete the third line, containing the return statement! We're just dispensing with that sequential counter, the variable called *switcher*. While we're at it, we can also delete this line from the top of the script:

```
switcher = 0
```

Nothing bad happens if you don't delete this line, but it's good practice to eliminate useless lines, as they can be confusing when you try to understand your code later on.

Next, change the parameter at the beginning of the switch structure so that it looks like this:

```
switch(r(numOptions))
```

Now, instead of marching through the sequence of sentence templates, we're choosing one on each pass, as randomly as we choose any number in this chapter, using our faithful *r* function. After completing these changes, the output looks like this:

the barista says walk on
somewhere not here is the smell of mahogany
you think you're awake
the barista says dream more carefully
you can smell an elephant

or this:

somewhere not here is the smell of mahogany
somewhere not here is the country of smiles
the chaplain says walk on
you can smell woodsmoke
you imagine there was a bonfire

Once again, our little machine seems to hold up pretty well. The repetitions look almost deliberate (which they are, in an indirect way). The narrative scheme, such as it is, is impressionistic enough to survive the imposition of randomness.

Externalizing the Generator

As we move toward our final three examples, we'll need to make one more important change to our text generator: moving it to an external script file. As it happens, JavaScript need not be written into a script container on a single page. We can move JavaScript code to a separate text file with the file extension .js. To set up our last examples, we will do this, copying the complete contents of our script container into a new text document, which we name "generator.js." In that new document, we delete the lines <script> and </script>. Externalized JavaScript doesn't need a script container.

Why do we move our work to an external file? As you may suspect, it's so we can use the same instructions flexibly in multiple projects without having to cut and paste or (mercy!) type them in. Once a set of functions have been moved to an external file, we can invoke them from within JavaScript code on any other page, so long as we include this special script container in our new page:

```
<script src="generator.js"></script>
```

An important detail here: the *src* ("source") attribute added to the initial script tag takes as its argument the location of the external file. As

the tag is written here, that file must be in the same directory as the page that is loading it. Put everything into one folder and you'll be fine. Note, however, that you can access external JavaScript pages from anywhere in your local system or indeed from any point accessible to the web. Our use of an external script demonstrates two important principles: *modularity* and *dependence*. A program is a composite or assemblage of distributed parts. The parts depend on one another; they interoperate. If you plan to use more sophisticated game development tools like Unity, or if you think you might want to learn programming on a more serious basis, you'll need to understand these concepts.

Now back to our example. In effect, this blank container is filled, at least virtually, with the contents of the external file. (We're not sure that's technically accurate, but it feels that way.) Web pages can have more than one script container, as it happens, and in our next examples, we'll build additional containers and scripts that coordinate with our original text generator.

Bonus Example 3: An Everlasting Scroll

When we discussed Montfort's *Taroko Gorge* in chapter P-2, we noted the importance of its limitless operation. Like the gorge, the poem keeps unfolding (or in terms of its code, folding back on itself). What we see is an infinite scroll. There may be no way to achieve such an effect in Twine without slipping into JavaScript. (More on that possibility at the end of this section.) It's certainly not possible within the basic script affordances of Chapbook, which includes no loop structures. There are timed effects in Chapbook and Harlowe, but they are meant to run only once and have generally limited function.[1] Generally speaking, the Twine idiom assumes that changes will follow player action, not occur automatically.

We can break that taboo easily enough with JavaScript. All we have to do is look away politely when a function invokes itself. We've

1 We have not experimented with SugarCube, a story format with robust support for programming.

been using this technique for many years now without problems, so until Skynet sends a robot assassin from the future, we'll assume it's safe.

We start again with a blank version of our template web page. The first thing we do is add, above the existing script container, the reference container for our externalized text generator:

```
<script src="generator.js"></script>
```

We'll be using the feed from the text generator as content for our endless scroll. This is, of course, an arbitrary choice, but it has the virtue of tying our examples together and showing a remote script in operation. Note that the reference container does not do anything in itself because the script we brought over has no activating instructions. If you remember, its operation was triggered by an *onLoad* handler written into the body tag of the page, which is not part of the external JavaScript. So our generator code just sits in memory until we ask some bit of it to do something—which we will, directly.

Before we discuss the fresh code for this example, let's explain what we're trying to do and how we'll go about it. We want text to scroll constantly. We'll decide that the new text should appear at the bottom and disappear at the top of the window, because *Star Wars*. (It's easy enough to reverse the effect if desired.) We're adding to our scroll in discrete units, one sentence at a time. This makes the job a bit easier.

We need a data structure that will let us keep track of items in a numerical sequence, with the ability to add new items to the bottom of the sequence and delete from the top. This is why JavaScript gave us arrays. By now, you're very familiar with arrays in both Twine and JavaScript. We'll be using two built-in functions of the JavaScript array object, *push* and *shift*, which perform the needed addition and trimming. Since the code for this project is refreshingly compact compared to our text generator, we'll just show it complete and then discuss its features. Everything you see here sits inside the main script container.

```
textArray = new Array();

function writeUp(){

   //push on a new line
   textArray.push(generate());

//trim top line
   if(textArray.length==10) textArray.shift()

//output
document.getElementById("out").innerHTML = ""
   for(var i=0; i<textArray.length; i++){
      outIt(textArray[i]);
   }

   //don't stop
   theTimeout = setTimeout(writeUp, 1000);
}
```

First, we declare *textArray* to hold our generated sentences. We use the keyword *new* (technically called a *constructor*) to generate an array. The empty parentheses mean the array has nothing in it and an undefined size or length.

Next, we define the lone function in this example, called *writeUp*. You could call it anything you like. As you can see, we've marked off the four parts of this script with descriptive comments. To add to the bottom of our array, we use the *push* function, and what we push onto the array is the output from *generate*, our randomized, template-based sentence generator that is sitting in the remote file generate.js. (See how this works!) Next, we set an instruction to trim off the top line of the array once the array contains ten elements. That number is an arbitrary design decision, entirely changeable. It determines how many sentences will be visible in your scrolling window. You may want to keep this value low enough to fit

the entire stack onto a typical screen. This could be accomplished mathematically by bringing in some parameters about the browser window and the line height, but we'll rely on guesswork for simplicity.

Next comes output, where we write the updated contents of *textArray* to the screen. Before we can do this, we remove any version that may have been displayed on a previous pass through this script—it's designed for repetition, remember? So we replace the *innerHTML* of our "out" DIV (in effect, the display window) with the null value, signified by two quotation marks without a space between them: "". At this point, you might wonder why we don't use the *outIt* function that is handily sitting in our remote script. We will use it later, but we can't do so here. That's because *outIt* is designed to add to the contents of the display DIV using the "+=" operator. Passing it a null value would just add a null value. We need to replace, not add. If we wanted to be clever, we could either write a second function (say, *blankIt*) or, even better, change *outIt* to accept a second parameter determining whether it adds or replaces. These improvements would have made the example more complicated, so we leave them to your imagination.

With the board erased, we're ready to write. You might think we could just pass *textArray* to our *outIt* function. If you try this, you'll see your sentences all jammed together, separated by commas, which is not what we want. We need to peel each of our sentences off, one at a time. That's what a good old *for* loop is for. It marches through the array from 0 to the last value before its length (which is the last item), referring to the item in question with the loop's built-in counter variable *i*. Notice we don't need to add <*br* /> at the end of our sentences because that's included in *outIt*.

Now we come to the final piece of the code, thoughtfully labeled "don't stop." This function reactivates itself. Generally speaking, programmers do not recommend that practice, but Montfort does much the same thing in *Taroko Gorge*, and he has advanced degrees in computer science and computational linguistics. As we said, this is technically an infinite loop, but it does not crash the browser, destroy the internet, or open any wormholes that we know of. The simplest way for a function to invoke itself is, of course, simply to write, on the last line of *writeUp*,

```
writeUp()
```

We could do that, but only at the expense of reading. Without some delay, the function will simply spew sentences up the screen, iterating several times a second. To avoid this, we wrap the reinvocation in a *setTimeout* function, which formally requires us to create a new variable called *theTimeout* and invoke the delay from there. The number parameter used is a value in milliseconds. One thousand milliseconds equal one second. You can change this value if you like. The effect of *setTimeout* is much like the delay factors in Chapbook and Harlowe: it holds operation until a certain amount of time has elapsed. The difference here is that the function it eventually invokes sets up another timer at the end of its run and so forth ad infinitum, if you can wait that long.

The result is an eternal scroll, filled with a constantly changing (and only occasionally repeating) series of sentences from our now familiar generator. As in *Taroko Gorge*, you won't see the scroll effect until enough lines have appeared to start the trimming process. After that, the business runs as long as you stay on the web page. As we've said, Montfort's poem may tell us something about the infinite complexity of the natural world. What this little example says about anything except coding is probably beside the point.

Finally, a further note on what can and can't be done with Twine. Because Chapbook supports both JavaScript code and HTML elements like DIVs with IDs, we can in fact port almost every piece of this project back to Twine and produce a passage (not page) with an endless scroll. We could mix this feature with other affordances of Twine for a richly hybridized experience. The only thing we can't do in this context is move our key functions to an external JavaScript page. Actually, that might be possible, but it would be necessary to know more about the inner workings of Twine than you probably want to learn right away. The hybridized Twine story is included in our online examples as bonus example 3A. We won't go through the code because it's essentially what you've seen already.

Bonus Example 4: Drifting down the Screen

For our next set of tricks, we'll explore another feature of the computing environment that has no obvious place in the Twine world: animation. From the start, we should point out that HTML and JavaScript are less-than-ideal platforms for motion graphics. Yes, you can watch movies through your browser, but you generally do so in a video window running a specialized resource called a *coder/decoder* (codec) or sometimes a browser enhancement called a *plugin*. Back in the day, before someone decided it should no longer be supported, there was a famous plugin called Shockwave Flash, designed to run content developed by the two Adobe products of those names. You may recall our mention of those programs in chapters T-1 and T-4. Those applications and their plugin handled animation very, very differently than we can or will, depending on just the resources of your web browser, unplugged. Nonetheless, these two simple exercises will at least give you a taste of poetry in motion.

Let's begin with a single falling object. We start, as always, with a fresh copy of our starter web page, to which we add the reference container to link up generator.js. Next, we go to the style sheet—the style container found within the head just before the script containers. The style container is blank. We add the following:

```
div{
   font: 18pt Cambria; padding: 20px; position:
   absolute;
}
```

The font and padding specifications are familiar from earlier examples. Note that last item, though: it declares that the position of any DIV in our document will be mathematically fixed, not determined in relation to other page elements. Animation won't work without this declaration.

Next, we open the main script container of the page and add two functions. The first of these is called *setUp*. It's designed to run once

when the page loads, so we also go down to the *onLoad* handler in the body tag and set it to activate *setUp*. Here's what *setUp* looks like:

```
function setUp(){
      theDIV = document.
      getElementsByTagName("DIV")[0]
      theDIV.innerHTML = generate()
      theLeft = r(600)
      theDIV.style.left = theLeft + "px"
      theTop = 0--r(100)
      theDIV.style.top = theTop + "px"
      animate()
}
```

The first line introduces a variable called *theDIV* and assigns it a value. The construction we use here looks a bit like the one with which you're probably familiar, *getElementById*, though actually, it's the cousin of that method, *getElementsByTagName*. Note it says "Elements," plural. This method of the *document* object can be used to reach out to a single page element, as we do here, but it first situates that element within a set of similar elements—the *collection* of DIV elements on our page. You'll see why we do this when we get to our next example. For the moment, have a look at the arguments we pass to the *getElementsByTagName* method: a tag name in parentheses ("DIV") followed by a number in brackets [0]. This is the same notation used to identify elements of an array—and, indeed, a document object collection is a bit like an array, though it does not have all the features of that object. Why do we say *0* here? Because our page only contains one DIV, and programmers always start with nothing (or count from zero). So item 1 (and only) is item 0.

Why do we attach this laborious identification to a variable? Strictly for convenience, because we are going to operate on our one and only DIV in ways that require us to name it. Our variable *theDIV* acts like a pronoun, saving much bothersome typing. What we're doing, specifically, is placing our DIV at a specific point on the screen. Yes,

JavaScript, the DOM, and HTML can do that. That's what makes animation possible.

Any DIV, or block-level page element, has properties called *top* and *left* that indicate where its respective edges are located within the browser window. To move the element, we reset those properties. They are actually subproperties of a more general *style* property, so we address them in dot notation as *style.top* or *style.left*. There are some further complexities beyond this. First, we can't modify the values with a statement like the following:

```
theDIV.style.top ++
```

For arcane reasons, the values of geometric properties must be expressed with metrics—for instance, *100px*, which means one hundred pixels from the top of the window. We need to append the text string "px" to the number, which means we first extract the number, assign it to a variable—*theTop* and *theLeft*—modify the variable as we wish, append the metric, and then bang the result back in. This is quite baroque, and we've never understood the reasoning behind it, but so be it. You'll see that for the left position of the DIV (x-axis), we're asking good old *r* for a value between 0 and 599, which assumes the browser window is at least six hundred pixels wide. (Here's hoping.) For the top position, we do something that may seem strange: we ask for a random number between 0 and 99, subtracting that number from 0 to make it negative. That's because we want our DIV positioned *above the top edge of the browser window*. And yes, we can do that. This way, we start with a blank screen, and our drifting DIV can make a dramatic appearance.

Let's get to the drifting part, which is the business of our second function, *animate*—which, you'll note, is invoked at the end of *setUp*. Here's the code:

```
function animate(){
      theDIV = document.
      getElementsByTagName("DIV")[0]
```

```
        theTop += 5
        theDIV.style.top = theTop + "px"
        if(theTop > 500){
               setUp()
        }
        else{
        theTimeout = setTimeout(animate, 50);
        }
    }
```

We're repeating that step from *setUp* where we identify our DIV and assign it to a pronoun-like variable. There are ways to avoid this inelegancy, but they would complicate the conversion of this one-DIV example to a multi-DIV example in the next section, so we do it again, somewhat mysteriously. The variable *theTop* comes into play again in the next statement. Until we modify it here, it has whatever value it received in *setUp*. Our modification increases it by five, meaning we move our DIV five pixels down the screen. We assign the modified value in the same way you saw in *setUp*. Notice we don't change the left position of the DIV. We're only animating in one axis, though you could use two (or even three) if you wanted.

Now we come to that *if* test. Once our drifting DIV passes line 500 of the browser window, we want it to go through the *setUp* routine and reposition at the top of the screen. We use an inequality (>) because we're using increments of five, and it's possible for our DIV to exceed 500 without ever having that value—for instance, if its position changes from 499 to 504, which is possible. Remember, the vertical position of the DIV is assigned randomly in *setUp*, so we don't know the exact value (and don't really need to). We're actually using an *if/else* construction here because we want another thing to happen if our DIV has not yet dropped offscreen. In that case, we start a *setTimeout*, just as we did in our eternal scroll, using a delay factor of fifty milliseconds. Higher values slow the animation, lower ones speed it up. Experiment as you like. Once again, we have a function that calls itself. What's a little recursion among friends?

If all the pieces are properly assembled, this example drifts a randomly generated sentence down the screen, followed by another and another, at various horizontal locations. It's about as simple as animations get. In our next and final example, we'll make it just a bit more interesting.

Bonus Example 5: It's Raining Story

For our final example, we'll multiply the floating DIVs to give greater visual (and maybe narrative or poetic) interest to the project. To do this, we'll need a way to animate, track, and reset several page elements independently. Now you'll see why we started referring to our single DIV via its place in the DIV collection. We'll need the whole set in play for this one.

Since we modified our basic page template slightly for the previous example, we'll start this one by making a copy of that page file, renaming it, and erasing the contents of the main script container—the two functions we created in example 5.4. We'll end up rebuilding some of that code, but there are enough differences to reward a fresh start. Before we start on the JavaScript, we'll go down into the body portion of the markup and make two changes. We'll add an *onLoad* handler to the body tag:

```
<BODY width: 600px onLoad = "startUp">
```

Next, we'll replace the single DIV that's sitting in the body container with a stack of five:

```
<BODY>
   <DIV></DIV>
   <DIV></DIV>
   <DIV></DIV>
   <DIV></DIV>
   <DIV></DIV>
</BODY>
```

These DIVs need neither IDs nor contents. There do need to be five of them, however.

Now for the scripting. At the top of our pristine script container, we'll declare and initialize three very important arrays:

```
leftNum = new Array(0,0,0,0,0);
topNum = new Array(0,0,0,0,0);
DIVSpeed = new Array(0,0,0,0,0);
```

You'll remember that arrays can be set up with initial values, as we do here. Those zeroes will be replaced with nonzero numbers when the script starts up. We could have used any number, so long as it's an integer. We need to start with integers here, since that's what we'll be storing in these arrays as we go.

You'll recall that in bonus example 4 we had a function called *setUp*. This time we'll have one called *startUp*. It's a bit different from our previous *setUp* function:

```
function startUp(){
   for(var i=0; i<5; i++){
      reset(i);
   }
   theInterval = setInterval(animate, 50);
}
```

Here's a familiar five-step *for* loop, but all it does is call a function called *reset*, passing it a number from 0 to 4. We'll build reset next. Before we do, have a look at the final line of *startUp*, which uses the first cousin of *setTimeout* called *setInterval*. The *setTimeout* method runs once; that's why we need to keep reinvoking it in our earlier examples. By contrast, *setInterval* repeats automatically as long as the page is loaded and the interval is not canceled by some other instruction.

When we only had one falling object, we could have it restart its animation function every time it passed offscreen. However, this time we'll control five DIVs with one function. Under that scheme, it's easier

to start the animating engine once and let it run. Before we can get to animation, however, we need to create our reset function:

```
function reset(which){
   theDIV = document.getElementsByTagName("DIV")
   [which]
   DIVSpeed[which] = 3 + r(5)
   theDIV.innerHTML = generate()
   leftNum[which] = r(600)
   theDIV.style.left = leftNum[which] + "px"
   topNum[which] = 0--r(100)
   theDIV.style.top = topNum[which] + "px"
}
```

This is the function called five times by *startUp*. Much of it will look very familiar from bonus example 4. There are two main differences. First, this function takes a parameter called *which* (it could be called anything). This parameter is an integer between 0 and 4, inclusive. Notice that in our DIV-identifier (*theDIV*), we use this number to say which DIV we're addressing. Remember, all the DIVs are numbered in the collection. (This scheme assumes our animating DIVs are the first five to appear in the markup. If you change the page in a way that breaks this pattern, the animation won't work.) The second variation here is the reference to that third array we created, *DIVspeed*. This array stores an integer value for each of our five animating DIVs, setting the amount of downward displacement that will occur on each cycle of the animation—in effect, the speed at which they fall. We require a minimum of three pixels but add to that a random selection on a range of five, meaning the maximum amount is seven. You can experiment with different values here. The important thing about this feature is that it can give each DIV a different rate of descent. The effect is very important visually.

The final element of this project is the *animate* function, which goes into operation at the end of the *startUp* function, activated on page load. Here's the code:

```
function animate(){
   for(var i=0; i<5; i++){
      topNum[i] += divSpeed[i];
      theDIV = document.
      getElementsByTagName("DIV")[i]
      theDIV.style.top = topNum[i] + "px";
      if(topNum[i] > 500) reset(i);
   }
}
```

The bones of this function should be familiar from the previous example. Here we have a five-way *for* loop that addresses each of our falling objects in sequence (so quickly it seems instantaneous). We do all the usual business of updating the top location of the DIV, handing it off to the reset routine when the DIV passes the five-hundred-pixel line. But notice that the value passed to reset is just the number of that particular DIV. This animating routine manages each element separately.

The result is a shower of sentences or a variable story crossed with a confetti machine. What that might amount to, beyond an excuse to practice JavaScript coding, is the subject of our very last section.

Conclusion

You'll notice something conspicuously missing from the five examples presented in this chapter: interactivity. All five are focused on display. They presume a reader, or perhaps a viewer, but not really a player. Should we conclude therefore that moving from Twine to JavaScript/HTML means leaving behind interactive fiction and games? Is web coding primarily a replacement for the cinematic aesthetics and poetics of dear, dead Flash?

Beware of hasty conclusions. It's easy enough to see how some or all of these examples could be harnessed for story-centered games. For all three of our final examples (scroll and falling texts), imagine a stack of clickable prompts (words, names, faces, symbols) that call on alternative text generators, allowing the reader/player to steer the unfolding

story in specific directions. Adding features like these to the demonstrations would make the code-crawling unbearably tedious. The drawback of simple examples is simplicity. We invite you to think beyond them.

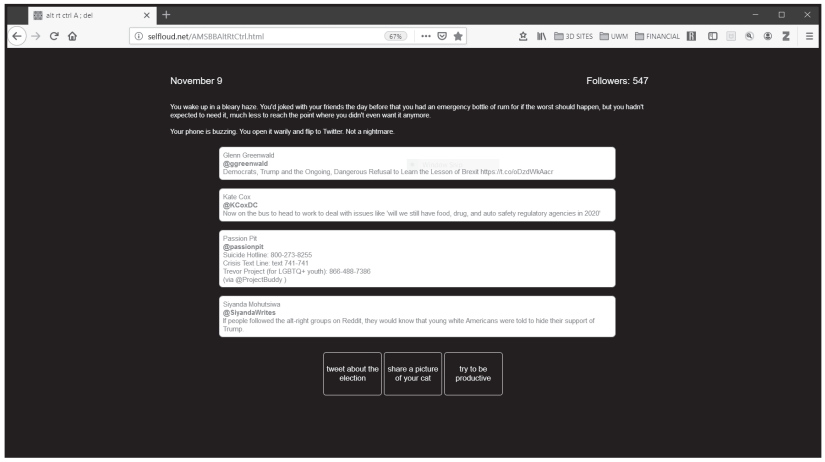

Figure 25: Salter and Blodgett's *ALT-RT: Ctrl+A; DEL* (2017)

Both authors of this book have built creative works that use HTML, JavaScript, and other web resources to design evocative interfaces and tell salient stories. Salter and Blodgett's *ALT-RT: Ctrl+A; DEL* creates a simulated tweetstream drawn from a database of actual and invented material to capture the nightmare of social media (Salter and Blodgett). It combines quasi-randomized text sampling with selectable options for self-preservation. These selections have meaningful consequences, making the work legitimately interactive. Its experience has a distinct ending and alternative outcomes, making it very gamelike. Moulthrop's *Emaji Naratgee Marakka*, born of similar inspiration, renders trollish tweets as a visually accreting mass that the player can suppress or erase completely by doggedly choosing acts of resistance (Moulthrop). Successfully wiping out the troll-storm (if only for a moment) earns the reader an installment of a fable. Once this chunk of story has been read, the tweets return, growing ever more deranged. To reach the end of the fable, the reader must repeatedly fend off the troll, then penetrate a few final mysteries of cryptographic text. Both of these works

are hypertextual, narrative, and gamelike; both depend on affordances (database access; animation) not easily supported in Twine.

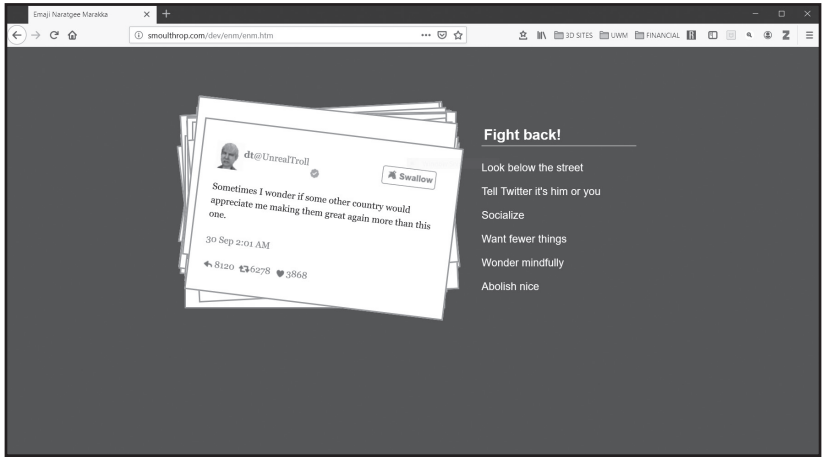

Figure 26: Moulthrop's *Emaji Naratgee Marakka* (2018)

Twine is not the only way to make interactive fictions. However, as the extent and density of even the modest code examples in this chapter will show, there is a significant trade-off between the broad creative scope of hand-built web work and the elegance, stability, and community of Twine. As always, creators and communicators should understand the range of possibilities implicit in these tools and feel empowered on any platform.

Works Cited

Moulthrop, Stuart. *Emaji Naratgee Marakka*. Work in progress, 2018. www.smoulthrop.com/lit/enm.

Salter, Anastasia, and Bridget Blodgett. "ALT-RT: Ctrl+A; DEL." *Persona Studies* 3, no. 1 (2017). https://ojs.deakin.edu.au/index.php/ps/article/view/656.