CHAPTER P-4

Too Much Twine

If we return for inspiration to their provocative 2014 Twine hypertext *Quing's Quest VII: The Death of Videogames*, D. Squinkifer takes on the specter of Gamergate by placing the player in a retro, animated-GIF-background-bearing space adventure where, driven from their home planet of Video Games, the hero expresses a longing to "build something again": "You were born to build things. As a member of the royal family of Videogames, building things is in your blood. You grew up apprenticing under your elders, whose queer, beautiful, complex structures you could only dream of coming close to emulating someday. You listened non-stop to their stories of the Golden Age of Videogames, envying the creative freedom and abundance they enjoyed" (Squinkifer).

D. Squinkifer's words are an invitation to learn from the elders, and in that mind-set, we look to several glorious, powerful, campy Twine works for inspiration throughout this and other practical sections. In this chapter, of all chapters, we cannot begin to tell you what to "build." If anything, camp Twine is the invitation to excess, and aesthetic play, which we will explore here as we explore existing techniques.

◊ As in other practical chapters, action items are boxed and set off with the symbol you see at left, in case you want to skip the contextual discussion.

Supporting materials for this chapter can be found online at https://github.com/AMSUCF/Twining. See our discussion at the beginning of chapter P-1 about using the .html and .txt files to follow along or adapt our code to your own purposes.

Example 4.1: End Times

In this making chapter, we will primarily be working with SugarCube, to continue our tour of Twine story formats. SugarCube builds on the most popular of Twine's first iteration story formats, Sugarcane, and thus has a legacy of useful macros built in to allow for versatile design. Start by creating a new story and selecting SugarCube 2.X.

The timed mechanics of *Queers in Love at the End of the World* create a sense of urgency that can lead to more frantic choice—such timers can be used within a work to add pressure to a conversation or pivotal moment, or they can envelop a work, offering the reader a limited amount of time in which to experience it (Anthropy). The Twine Cookbook includes iterations of this fundamental mechanic in every story format, which you can use to compare the complexity and approach of each format (Cox).

SugarCube allows us to embed passages within other passages easily, so we can think of our story as progressing in fragments. Note the characteristics of SugarCube: like HTML, SugarCube relies on < > to designate the beginning and end of tags but adds a second layer to differentiate from HTML itself. Thus HTML and SugarCube markup can comfortably coexist, as in this code:

◊ Let's start by creating a passage entitled "Countdown," where our ticking timer will lurk as the player progresses. Enter the following code into the passage:

```
<span id="countdown">Planetary implosion in
$seconds seconds</span>
<<silently>>
   <<repeat 1s>>
      <<set $seconds to $seconds--1>>
      <<if $seconds gt 0>>
             <<replace "#countdown">>Planetary
             implosion in $seconds seconds<</
             replace>>
      <<else>>
             <<replace "#countdown">><</
             replace>>
             <<goto "Lost">>
             <<stop>>
      <</if>>
   <</repeat>>
<</silently>>
```

In this code, several elements are at work: first, the *$seconds* variable holds the timer itself. Note that we haven't initiated a value for it here—we need to decide when our timer begins and declare it elsewhere. We might also change this variable to add complexity (for instance, if we want the reader to be able to influence or extend the timer through certain choices), so it's best to keep its value separate from the passage that displays its contents.

On its own, this passage doesn't do anything, but it provides the framework for counting time. Note the macros that are similar to those in Harlowe: *set* still assigns a value, and *replace* takes a parameter to replace the content. In place of Harlowe's *(go-to:)*—Chapbook has no equivalent—we use *goto*, which similarly takes the name of a passage and automatically reloads the page. This is similar to the two-second countdowns we used earlier during our hunt for the amulet in chapter P-2, but here we've isolated the timer so we can use it across multiple passages.

◊ Next, let's create the first passage of our narrative. We don't want to launch the timer immediately, so start by altering the first passage (designated, as always, by the rocket) to set up our prelude. Type the following into a first passage labeled "Beginnings":

```
You've come this far. The B Arc was full (too
many telephone sanitizers), first class was
always out of reach, and this hunk of junk
about to take off is the last ship off this rock
before it hits. If only you knew how to fly it.

[[Try.|Controls]]
```

Any need-driven quest will do for this conceit, or if you are feeling existential, a final countdown along the lines of *Queers in Love at the End of the World* is fitting (Anthropy). For now, stick with a single link on this page to avoid the need for multiple timer declarations—you can always add that complexity later by duplicating the code we'll place in the next passage.

◊ Create another passage titled "Controls." In this passage, initialize the timer and include the "Countdown" passage by typing the following code:

```
You're looking at buttons. Like, old school Wing
Commander, you're going to need a manual for
this, unlabeled buttons. You can hear your cell
phone beeping the warning alerts as you look for
something marked throttle.
https://media.giphy.com/media/CKRx4oUu3dzLa/
source.gif
<<set $seconds to 30>>
<<include "Countdown">>
[[Press the green button|Green]]
```

```
[[Press the yellow button|Yellow]]
[[Press the red button|Red]]
[[Look for an index|Search]]
```

Include works much the way *embed passage* does in Chapbook, inserting the passage where we've indicated. It's important to declare the variable sometime before we embed the countdown timer for the first time, as otherwise, you'll see an odd error. We don't need much text for any of these options—the player is going to need to make some fast, frantic decisions. We'll come back to fleshing out the paths in a moment, but first, let's set up a default ending for when the timer goes off.

◊ Create a passage entitled "Lost" and type the following:

```
The rumbling lets you know it's too late--that,
and the sinking feeling.

The flames are the last thing you see.
```

We'll also need to remove the user interface bar from the side of the screen to avoid players backtracking from this ending. You'll notice that many Twine works remove these user controls, particularly when it's important to eliminate backtracking.

◊ Open the story JavaScript file and type the following:

```
UIBar.destroy();
```

Reload and you'll see that the entire user interface bar has been removed—this takes away some of the story format's built-in functionality but also eliminates lots of design problems. This gives us a foundation, but it certainly isn't camp, and there's a lot of room to expand even within our short timer. We'll use this base for the next several exercises to start adding aesthetic enhancements and thinking about the role of audiovisual elements in how we work with Twine.

Example 4.2: Changing Styles

We've focused thus far on the text and functionality of Twine, not the look. However, Twine can integrate anything from the web experience—it's just a matter of figuring out how to mesh your desired elements with the story format you are using. Each story format has different strengths and weaknesses when it comes to bringing in visuals, styling, audio, and even video or animated elements.

Practically speaking, the more you know about CSS, the more control you'll have over Twine's aesthetics. In SugarCube 2, there are a few built-in tools to be aware of, including a foundational set of style sheet rules that give us more detailed control.

Let's start with something simple. We'll set each button link to display in the appropriate color, then change the background of the corresponding linked page to match by setting up passage style sheets.

◊ Tag each of the three colored-button passages with a corresponding color: red, green, and yellow. Open the story style sheet and type the following:

```
body {
   background-color: #A9A9A9;
   color: white;
   font-size: 200%;
}
a {
   color: purple;
}
body.red {
   background-color: red;
   color: white;
   font-size: 250%;
}
body.green {
   background-color: green;
```

```
    color: white;
    font-size: 250%;
}
body.yellow {
    background-color: yellow;
    color: black;
    font-size: 250%;
}
```

Note a few oddities in this style sheet: the first two selectors control what you'll see on any passage that doesn't have a style tag. Most of this is straightforward CSS, using familiar elements of HTML—the body and *a*, or links. Even though we don't write links using <a> in SugarCube, the markup we do write is translated forward to the standard HTML element, and all the usual properties apply. Note that we can use any color data format supported by CSS here—in this example, there are both hexadecimal codes and color names.

The tags we added to each passage are translated into class tags accessible as modifications to the body—thus, *body.red*, *body.green*, and *body.yellow* will each control the corresponding tag's styling. Test out your new design (be warned, it's a little garish). You'll notice that passages that are inserted into a tagged passage inherit the style of the page they are inserted into by default. This means we can use the countdown timer on any page, even as we add more complexity to the style sheets.

◊ Now let's add a bit more drama to the final page by adding CSS animations to fade out the text. Open the style sheet and add the following code:

```
.disappear {
    opacity: 0;
    animation-name: fadeOutOpacity;
    animation-iteration-count: 1;
    animation-timing-function: ease-in;
    animation-duration: 5s;
```

```
}
@keyframes fadeOutOpacity {
   0% {
      opacity: 1;
   }
   100% {
      opacity: 0;
   }
}
```

Currently, we have defined a new class, but we haven't applied it anywhere in the code, so you won't see any changes in the text. Let's break down this animation frame by frame: this is a simple fade-out that diminishes the *opacity* of the element from 1 (fully visible) to 0 (transparent) over time. It will only occur once (the *animation-iteration-count*) and will last for five seconds (the *animation-duration*). Depending on the impact you want, you can change the pacing by specifying an *animation-timing-function*. In this case, "ease-in" means it will start slowly and speed up as it disappears, while "ease-out" would do the opposite—try it and compare later. Importantly, the code *opacity: 0;* specifies the default state when the animation is not occurring—if we set this to 1, the text will abruptly reappear after the animation ends.

◊ Now we need to apply our animation to an element. Open the "Lost" passage and alter the text to match:

```
@@.disappear;The rumbling lets you know it's too
late--that, and the sinking feeling.

The flames are the last thing you see.@@
```

The @@ symbol is SugarCube's way of marking the beginning of in-line CSS. The second iteration of the symbol indicates that this is where the in-line CSS ends. Anything that you can apply in CSS can be added using these properties, so it's a simple way to create emphasis—let's try

it with font color directly by going back and altering the passage where the buttons are first introduced.

◊ Now open the story style sheet and add a new set of classes:

```
.greenLink a { color: green; }
.redLink a { color: red; }
.yellowLink a { color: yellow; }
```

We'll need to make corresponding changes in the "Controls" passage:

```
You're looking at buttons. Like, old school Wing
Commander, you're going to need a manual for
this, unlabeled buttons. You can hear your cell
phone beeping the warning alerts as you look for
something marked throttle.
<<set $seconds to 30>>
<<include "Countdown">>
Press the @@.greenLink;[[green|Green]]@@ button
Press the @@.yellowLink;[[yellow|Yellow]]@@
button
Press the @@.redLink;[[red|Red]]@@ button
[[Look for an index|Search]]
```

Note that there are several other ways we could approach this that just wouldn't work. It's important to reassign the link to surround just the word we want to impact (this also improves readability) and to use a class rather than trying to change the color directly with *.color*. Using *.color* directly creates the equivalent of a span with that color, which is great for changing nonlink text but is ineffectual within a link.

This is a highly visually motivated instance of using link color classes for impact, but changing link colors can also be a way to communicate meaning to the user that is commonly used in Twine. For instance, Porpentine's *With Those We Love Alive* opens with a message to the user:

> Before living this life, have a pen or sharpie nearby, something that can write on skin.
>
> Purple links change. Pink links move forward. The colorblind version is here. (Porpentine)

Importantly, this gives insight into the mechanics that assist in navigation. The monochromatic version, which is optimized to not rely on color, instead uses italics to assist in differentiating between links.

Example 4.3: Sound It Out

We've created the potential for drama with these changes, and you might imagine layering them further to enhance the impact. However, this is only the beginning of what we can do with Twine. Some of the most effective games incorporate audio, including *With Those We Love Alive*, which addresses the audio in the next line of the introduction: "There is music, so headphones are good. But it's okay if you can't" (Porpentine; text formatting preserved for clarity).

These disclaimers are also valuable reminders for our own design philosophies: while using Twine's full audiovisual potential is exciting, we can keep the work accessible by always providing other means of entry into any important information the user needs to progress in or understand the experience.

Some of the earliest examples of Twine audio integration made use of existing music—for instance, it's hard to imagine *Cry$stal Warrior Ke$ha* without the titular artist's track blaring in the background. Looping music of that type is typically declared outside of any particular passage, as it is intended to play uninterrupted as the experience progresses:

Audio is difficult to some extent in Twine for the same reason audio is difficult on the web—logistically, you'll need to host your own audio files to ensure that they will remain accessible. You may have the ability to record your own audio for an experience, but if you do not, Creative Commons licensed sound effects and music can give you a palette of sounds and atmosphere to play with.

The Creative Commons search engine (search.creativecommons.org) is currently optimized for images but is in the process of expanding to incorporate audio; in the meantime, their legacy search portal (available through the same page) links out to several searchable archives for media, although it is incumbent on the user to verify that the results are truly Creative Commons licensed. Freesound.org and Soundbible.com also have a large database of options, although again, it's important to verify the contents.

We'll play with two types of audio: atmospheric, which loops in the background throughout play, and effect audio, which is typically triggered when the player reaches a particular passage or moment in the narrative. First, we'll do a simple audio file embedded in a passage that plays as the end passage is triggered. We'll use a free sound effect called "Fire Burning Sound" recorded by JaBa and shared under a Creative Commons Attribution 3.0 license (which means we'll need to list it accordingly in the credits section of our game).

◊ For this exercise, it's important to use the offline version of Twine 2. You can create the audio macros in the online editor, but you won't be able to test them using the local sound files on your system. First, we'll need to load them in using a "StoryInit" passage. Create a passage with this name and type the following:

```
<<cacheaudio "fireburning" "Fire_Burning-JaBa.
mp3" "Fire_Burning-JaBa.wav">>
```

HTML5 audio is tricky—not every browser supports every audio file type, so when you have multiple versions available, it is best to preload them all by using *cacheaudio* to start the browser loading the audio files before you try to play a sound. The browser chooses which file type to load based on its preferences. If none of the file types are compatible, your sound won't play. The audio files must be right next to your .html file in the folder for the path structure in this example to work.

◊ Next, we'll actually play the audio. Open and edit the "Lost" passage. Add the following above the first line:

```
<<audio "fireburning" play>>
```

The *audio* element takes a reference to a cached audio file and reaches back to find the source file preloaded by the browser. The audio command to play will run as soon as the page loads, but it can also be embedded in a link. There are lots of modifiers available in SugarCube's robust audio macro library, which is one of the best-supported of any of the story formats. The closest functionality currently for Harlowe requires an external library, the Harlowe Audio Library (Chapel, "Harlowe Audio Library").

Let's try one of the modifiers to fade out our sound as we fade out our text. Modify the audio call to instead say the following:

```
<<audio "fireburning" volume 0.5 fadeoverto 5 0>>
```

The *volume* argument takes a number from 1 to 0 and plays the audio at the specified level relative to the source. Obviously, it can be difficult to modulate this without knowing how loudly your reader has set their speakers, so think about using the volume modulation for balancing different effects, such as ambient noises versus dramatic interruptions. The *fadeoverto* argument takes two numbers: The first is the number of seconds, which we've matched here to the number of seconds on the text animation. The second number is the level of the final volume.

Next, we'll add looping background music. We'll use a file of "Creepy Background" sound effects, recorded by Daniel Simon and shared under an Attribution 3.0 license.

◊ Start by caching the background audio files in the "StoryInit" passage:

```
<<cacheaudio "background" "background.mp3"
"background.wav">>
```

Next, we'll call the audio from the controls page of our story but loop the audio.

```
<<audio "background" loop play>>
```

To avoid collision, we'll also want to stop the audio loop before we start the fire effect:

```
<<audio "background" stop>>
```

Incorporating images into a text-driven platform isn't necessary, but it can be powerful and provocative. Let's finish out the audiovisual exercise of this Twine by adding some *image* elements and using SugarCube's markup to make the image active. Images can have three components—a *title*, which provides a text caption; a *link*, which points to another passage; and a *setter*, which activates when clicked and alters the state of a variable.

◊ We haven't done anything with the "find an index" page in our pathways, so let's start there. Open "Search" and change the passage text to include the following:

```
<<include "Countdown">>
You look for any text you can recognize,
but there's nothing helpful.
There are three well-worn buttons on the side of
the console, but instead of words, they only bear
symbols that you don't recognize. If they were
letters, they'd be been lost to oils and waste.

[img[shape.gif][one]]
[img[shape2.gif][two]]
[img[shape3.gif][three]]

You might as well press one.
```

As with audio, all the image files referenced need to be in the folder with the .html file for this structure to work. It's also possible to use the image tag as an alternative to traditional image markup within the CSS. This can be more convenient when you're already comfortable with SugarCube's markup.

◊ Let's try changing the background color on our main passages to a background image instead. Open the story style sheet and add the following code:

```
body {
   background-image: [img[stars.gif]];
   color: white;
   font-size: 200%;
}
```

Notice that this works with animated GIFs (as in this example, a set of animated stars), and by default, it tiles the image. This approach works best with retro background effects, which tended to use repeated patterns rather than stretching a single image to fit. Animated GIFs like this one are used in many Twine works and are usually most effective with a strong contrast to any text colors chosen.

Example 4.4: Tracery and External Libraries

External libraries for Twine extend the capabilities of Twine and frequently provide bridges for more easily pulling in traditional JavaScript to the engine. There are extensive options for SugarCube 2 macros online, including Chapel's Custom Macro collection, which includes particularly useful tools such as pronoun templants and mouseover macros (Chapel, "Custom Macros"). Macros frequently emerge when more scripting-inclined Twine creators want to solve a problem for one of their own works and choose to repackage and share their solution with the community. Frequently, those macros are integrated back into the core of story format projects when they prove particularly useful.

In this exercise, we'll pick one well-loved library to try. Note that combining external libraries can be difficult, as they may include conflict syntax or requirements. Let's take a look at Trice, a library that combines many elements into one (incobalt). Created by Michael Thomét and inspired by Matthew R. F. Balousek's earlier library Twincery, Trice is a 2.X SugarCube-specific wrapper for Tracery that allows us to play with generative techniques like those we explored in our previous chapter. It's easier to break down how Tracery's logic can be integrated with Twine by first approaching the two separately.

Tracery works with *grammars* that are constructed of *symbols* and *rules*: symbols are essentially arrays containing a set of possible values, and once inserted, they choose a particular value for that instance. Rules are more complex patterns that combine fixed works and symbols to generate text or images (more on that later).

◊ Let's break this down in a simple example before we combine Tracery and Twine. To work with this code on its own, try the online visual editor at https://beaugunderson.com/tracery-writer/ and type in the following:

```
{
"origin": ["#codeVerbs# the #craftNouns#,
#codeNouns# #craftVerbs#"],
"craftNouns":["album","backing","bargello","bark
cloth","basting","batik","batting","bearding","b
eading","betweens","bias","binding","stitch","ne
sting","bobbin","tension","chainstitch","emblem"
,"embroidery","frame","sash","gap","gapping","ho
op","hooping","lettering","mirror","monogram","n
eedle","nippers","pantograph","tape","puckering
","punching","density","design","thread","broadc
loth","block","border","calico","charm","die","f
lannel","feeddogs","paper","sleeve","foot","fabr
ic","loft","long arm","medallion","memory","moti
f","quilt","fiber","panel","patch","value","unit"
```

```
,"seam","fill","facing","hook","scale","satin","
punching"],
"craftVerbs":["appliqué","bind","sew","hem","bri
dge","fill","press","back","repeat","rotate","sta
bilize","thread","break","cut","tie","trim","ver
ify","glaze","label","layer","piece","corner"],
"codeNouns":["algorithm","application","bootstra
p","code","structure","data","framework","stack"
,"query","object","function","variable","binary"
,"bug","command","conditional","statement","patt
ern","server","parameter","grid","pixel","resolu
tion","user","flow","element"],
"codeVerbs":["decompose","debug","iterate","cont
rol","program","run","embed","influence","bounce"
,"optimize","mine","declare","edit","design"]
}
```

Syntax-wise, Tracery is built on JSON: curly braces indicate objects and thus surround the complete grammar, or all the rules and symbols that make up a particular iteration of Tracery logic. The hash marks indicate a substitution and must surround a string of characters that matches one of the symbols. Each symbol lists the name first (in quotation marks, which delineates a string), followed by the list of possibilities as an array. As with the generation we discussed in chapter P-2, this allows for emergent play and can result in combinations we don't anticipate. The output to this first iteration is fairly banal but demonstrates the simple generative potential:

```
optimize the lettering, pixel trim
edit the puckering, pattern bind
design the paper, function glaze
influence the barkcloth, structure repeat
```

When we bring Tracery into Twine, the base elements remain the same, but the syntax changes.

◊ The easiest way to work with Trice is to use their bundled starter code, which draws in all the required scripts. Download the repository from GitHub, and open trice.html through the "Input file" option in the Twine main menu. You'll see the Tracery code in the story JavaScript file, but that's not everything—in the example folder, you'll see a project set up with the Tracery libraries in a folder. Make sure you duplicate that same file structure when working on this exercise, or your code will break due to unmet dependencies.

Importantly, the full library won't be loaded in time for our first passage, so we'll need a title screen. Create a "Begin" passage and add text and a link:

```
Cut and Trace

[[Begin]]
```

Let's start by generating some individual symbols: each passage we create and tag with "grammar" will be part of Tracery's grammar. Create a new passage titled "gCraftNouns," tag it, and add one word per line to the appropriate wordset:

```
basting
mirror
lettering
foot
fabric
calico
loft
tape
hooping
bargello
sleeve
block
border
```

Note that you don't need any special characters; the paragraph break alone sets up the conversion. This eliminates the need for extensive syntax and, in doing so, cuts back on some of the more frustrating aspects of generative texts. Each passage can become a freewriting exercise, and elements can be added easily as you expand your code.

◊ Let's test that our grammar is connected by going to the "Begin" passage and generating a word:

```
<<trace "gCraftNouns">>
```

Remember, you can't test through the Twine browser. Each time you test a trace, you'll need to export your file to .html—otherwise, the browser won't allow the files to access the necessary libraries, so Tracery won't run.

Note that whenever you invoke the library, you'll use *trace* to structure the command. This first simple stage just takes one symbol as input, using the string for the passage name, so make sure you type them exactly the same way. If you neglected to tag the passage as grammar, it won't import properly. Trace is a macro, so it provides a SugarCube syntax integration for the Tracery library. It's very flexible: it can be used to build entire rules and output full passages of generative text. Let's build a grammar with enough complexity that we can give this a try.

◊ Following the same pattern as the "gCraftNouns" passage, create a passage for "gCraftVerbs," "gCodeNouns," and "gCodeVerbs." Don't forget to tag each passage as a grammar. Next, we'll test by going to "Begin" and using a more complex trace:

```
<<trace "#gCodeVerbs# the #gCraftNouns#,
#gCodeNouns# #gCraftVerbs#">>
```

Notice a few changes from our simple call: if we are using more than one symbol, we need to integrate the full Tracery syntax with hash marks around each symbol's name. This also means we can start to

integrate some of Tracery's more complex features, such as functions to make certain words plural or change the tense.

◊ Let's give this a try by extending our passage poetry further. First, we'll need some additional grammar passages. Add a grammar passage titled "gColor" and tagged grammar, and input the following text:

```
orange
blue
violet
yellow
pink
apricot
indigo
green
gray
black
white
```

◊ Next, edit the "Begin" passage to include the following:

```
<<trace "#gCodeVerbs# the #gCraftNouns#,
#gCodeNouns# #gCraftVerbs#">>
<<trace "#gCraftNouns.capitalize# is
#gCodeNouns.a# . . .">>

<<trace "#gColor.a.capitalize# #gCodeNouns#
#gCraftVerbs.s#">>
```

Output on the "Begin" passage will vary every time you revisit:

```
program the fabric, function back
Basting is a query . . .
An indigo stack threads
```

Note the modifiers from Tracery's library in play here: *.a* adds the appropriate article based on the starting letter of the modified word, *capitalize* changes the first letter of the modified word to capital, and *.s* pluralizes the word (usually—but not always—correctly, so watch this one). The trace macro isn't the only way to integrate Tracery: sometimes we want to generate text and save it for later. For that, we can use the trace function.

```
◊ Let's pick a noun and save it to a variable, and then we can
use it repeatedly in a generated text.

<<set $myNoun to trace("gCraftNouns")>>

$myNoun is $myNoun is $myNoun.
```

Now we can generate extensively within Tracery in text, keeping in mind examples such as Kate Compton's own stylistic play—for more ideas, take a look at her site. Generative content is also the emphasis of several communities of play, and GitHub is home to a growing "NaNoGenMo" collection of generated novels, made for National Novel Generation Month, which includes Tracery and other tools powering a range of works for inspiration.

Example 4.5: Mood Imagery

While generative text is endlessly playful, we can also use Tracery to assist in generative components for imagery. (This can also be accomplished without Tracery, but less efficiently!) Tracery is popular with the bot-making community for aesthetic bots, which are a Twine-adjacent form of computational creativity that can result in endless content. A great example is Kate Compton's Tiny Space Adventure bot (hosted @TinyAdv and powered by Cheap Bots Done Quick, an easy-to-use hosting service for Tracery bots), which results in content like the following:

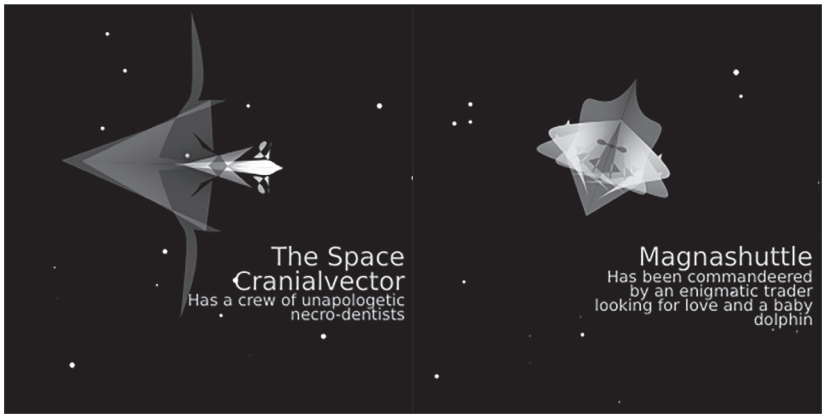

Figure 19: Examples of Kate Compton's Tiny Space Adventure bot

While the code for this is quite complex (and lives here: https://pastebin.com/YYtZnzZ0), looking at a fragment of it reveals how Tracery can be used to make substitutions in code:

```
"ship" : ["[gradID:#id#][bladeID:#id#]
[sideID:#id#]#gradient#<g
transform='translate(120, 100) rotate(#digit##
digit#)'>#shipSide#<g transform='scale(-1, 1)'>
<use xlink:href='\\##sideID#'></g></g>"],
"label" : "<text text-anchor=\"end\"
fill=\"\\#FFFFFF\" fill-opacity=\"0.4\" font-
size=\"12\" font-family=\"Verdana\" x=\"225\"
y=\"250\">#shipName#</text>",
"bg" : ["<rect fill='\\#000000' x='0' y='0'
width='300' height='300'/>#starField#"],
"star" : ["<circle fill='\\#FFFFFF' cx='#r255#'
cy='#r255#' r='#zeroone#.#digit#'/>"],
"starField" : ["#star##star##star##star##star#
#star##star##star##star##star##star##star##sta
r##star##star##star##star##star##star##star##s
tar#"],
```

```
"svgImg" : ["<svg viewBox=\"0 0 256 256\"
width=\"256\" height=\"256\">#bg##label##ship#</
svg>"],
"origin" : "{svg #svgImg#}"
```

Note particularly the SVG (Scalable Vector Graphics) element, defined in "svgImg" and including elements defined in succession: the ship is the product of much more complex code but is then added to the mix along with a label on top of a star field, which is generated from randomly placed and sized circles defined in the "star" symbol. SVGs are a web-friendly format defined mathematically and thus friendly to resizing and responsive design. Most people work with SVGs through graphical interfaces such as Illustrator, but it's also possible to work with simple or even complex SVGs directly through the markup of their code. SVGs are the heart of Tracery graphics. We'll continue with our code from exercise 4.4, since we're already set up for Tracery integration, and the language of code and crafting certainly lends itself to visual accompaniment.

◊ First, let's look at how SVGs work in SugarCube with some basic code. Add the following to your "Begin" passage:

```
<svg><line x1="0" y1="0" x2="200" y2="200"
stroke-width="1" stroke="white"/></svg>
```

You should see a diagonal white line beneath your text. SugarCube supports in-line SVG in this format natively, but only in some of its latest iterations—earlier examples and other story formats cause more problems with SVGs. *Line* designates the shape intended, and the two pairs of x- and y-coordinates indicate the beginning and endpoints of the line within the SVG. *Stroke-width* governs the width in pixels, while *stroke* sets the color.

◊ Now let's make it dynamic by first generating a color to save in a variable and then using a variable as an attribute

within the SVG. To do this, we'll use another SugarCube macro, *print*, which will allow us to combine the HTML markup with a variable:

```
<<set $myColor to trace("gColor")>>
<<print '<svg><line x1="0" y1="0" x2="200"
y2="200" stroke-width="1" stroke="' + $myColor +
'"/></svg>'>>
```

Note that the code is mostly the same, but we need to use $ to designate the variable—and watch the placement of the single quotation marks to indicate when we're asking SugarCube to process the input versus just printing the HTML directly.

Let's build something more complicated. We're going to create graphics that might be used in a number of ways—they might be art that changes without warning when the player revisits a room or atmosphere background imagery to set the mood for a scene. Let's start by restricting our color palette to create a more unified aesthetic. For this, it's easiest to work with color-safe HTML codes if you also want to use the words in text, but you can also use hexadecimal colors.

◊ Create a new passage titled "gNum," and make sure to tag it as grammar. Add a set of numbers—the range will impact the size of the final line:

```
80
100
120
140
160
180
200
220
240
260
```

```
280
300
```

◊ Change the SVG code in “Begin” to the following:

```
<<set $myColor to trace("gColor")>>
<<set $myNum to trace("gNum")>>

<<print '<svg><line x1="0" y1="0"
x2="'+$myNum+'" y2="'+$myNum+'" stroke-width="1"
stroke="' + $myColor + '"/></svg>'>>
```

◊ Modify the contents of “gColor” to:

```
purple
blue
gainsboro
silver
gray
teal
navy
cyan
indigo
orchid
lavender
plum
```

◊ Now we’ll need to create a more complex SVG to add dynamic design. This is going to require several symbols to control our shape’s attributes, so create new passages for each of the following. The bolded lines should be the title of the passage, and every passage must be tagged as grammar:

gY

```
50
```

```
100
150
200
250
300
350
400
450

gX
100
200
300
400
500
600
700
800
900
1000

gOpacity
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
```

Each of these will be used to control the corresponding elements of a generative ellipse—assuming a canvas that is 1024 × 512 pixels, these numbers will let us distribute shapes along the x and y space of the SVG. The opacity will give us more dynamic colors through overlaps:

note that we're not using an opacity of 0, as that would disappear, or an opacity of 1, as that might lead to too abrupt of a layer. Our existing "gNum" and "gColor" will complete the shape's features.

◊ Now we're ready to add a Tracery rule as a passage. This time, let's use the SVG code for an ellipse and create a passage titled "gEllipse" tagged as grammar. Include the following code:

```
<ellipse cx=\"#gX#\" cy=\"#gY#\" rx=\"#gNum#\"
ry=\"#gNum#\" style=\"fill:#gColor#;stroke:#gColo
r#;stroke-width:2;opacity:#gOpacity#\" />
```

Creating a passage to hold the ellipse properties will make it easy to use Tracery to generate as many of these shapes as we want, so we can potentially fill the screen with a variety of overlapping ellipses. Notice how each property is inserted using the Tracery symbol markup, and all the quotation marks are escaped with \ to ensure the code syntax is preserved when we run our trace.

◊ We have all the components now to create ellipses from a passage. Instead of adding it to "Start," let's create a new passage to hold this SVG. Let's start by using Tracery to create a dynamic link to the new passage. In "Start," add the following code:

```
<<set $myLink to trace("gCodeNouns.
capitalize.a")>>
[[$myLink|Screen]]
```

Note that this reversal of the *capitalize* and *a* will result in capitalization on the word, not the *a*, which can be useful when you want a different rhythm to your style. It's also possible to use Trice directly to create rotating links that use the keywords of the grammar—just include the name of the grammar symbol in the link directly, like

[[gCodeNouns]]—but this structure works best when you have a reason for using this dynamic link style, so we won't include it here.

◊ In "Screen," include the following code:

```
<<trace "<svg width='1024' height='512'>#gEllips
e##gEllipse##gEllipse##gEllipse##gEllipse##gElli
pse##gEllipse##gEllipse##gEllipse##gEllipse##gEl
lipse##gEllipse##gEllipse##gEllipse##gEllipse##g
Ellipse##gEllipse##gEllipse##gEllipse#</svg>">>
```

This can look a little overwhelming. Like Kate Compton's star field in the background of her spaceship bot shown earlier, we are using repetition of the generative element to create a layered effect. Note that the SVG size needs to be defined to ensure all our shapes display, so we declare the width and height first, again escaping the quotation marks to avoid errors. Watch the hash marks on repeating symbols, as it is easy to end up with an error in this section of the code. Note how we don't need to use print in this instance, as we are running the trace directly, eliminating the need to store every one of these properties as a variable prior to use.

The output should look something like this:

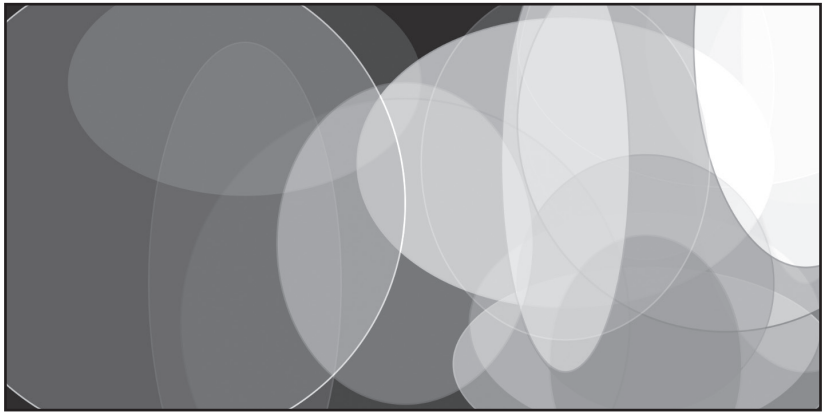

Figure 20: Sample output from our Tracery SVG generation

Note that this is only the beginning of what we can do with Tracery in Twine. SVGs can be scaled to fill the screen and can incorporate generative text as an overlay. They can be placed with CSS behind other elements to create a sense of mood-specific backgrounds. You can even invite the player to indicate a mood or color palette early in a play experience or shift the colors gradually to reflect changes in tone.

Consider how the colors set the tone of the opening decisions in Porpentine's *With Those We Love Alive*: the opening scene lives against a backdrop of turquoise fading into dark blue, reminiscent of the sea abstracted, with vivid pink links against it. This palette, not coincidentally, inspired some of the choices in this example. Note how every element is thoughtfully integrated, from the choice of link colors and fonts to the feel of the gradient across the page. It is harmonious with some of the later palettes in the same narrative, but those can be more immediately unsettling: a deep magenta fading toward a violet-black accompanies the words "Dead brown leaves cover inky black lake. / Something is rising from the lake" (Porpentine).

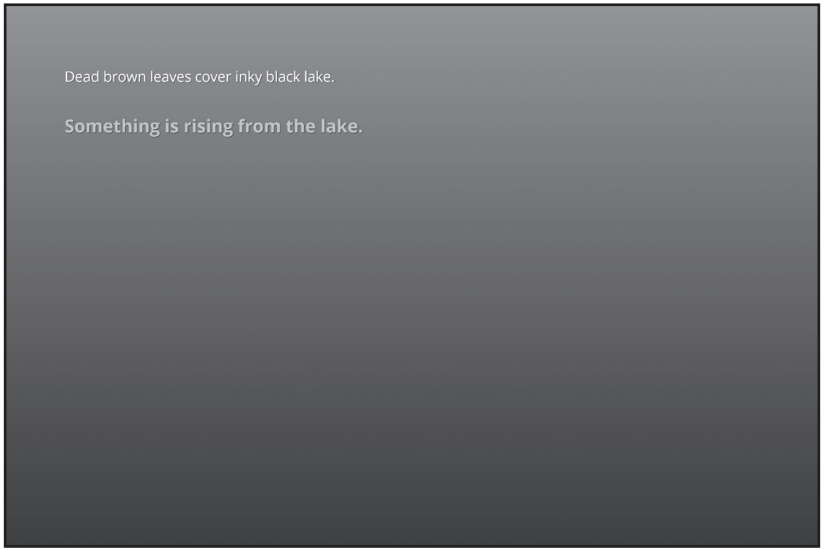

Figure 21: Screenshot from *With Those We Love Alive*

These visual choices can further be layered with the other elements we've discussed so far in this exercise. Consider how unsettling a deep color palette might be alongside a cheerful theme or how the alarming, unsettling aesthetics of Michael Lutz's more minimalist *My Father's Long, Long Legs* gradually alert the player to the wrongness of the family dynamics (Lutz). Whether constrained or over the top, atmospheric or full-on camp, the audiovisual elements of Twine are not necessarily secondary to the text.

Works Cited

Anthropy, Anna. "Queers in Love at the End of the World." itch.io, 2013. https://w.itch.io/end-of-the-world.

Chapel. "Custom Macros." TwineLab, 2019. https://twinelab.net/custom-macros-for-sugarcube-2/#/.

———. "Harlowe Audio Library." TwineLab, 2019, https://twinelab.net/harlowe-audio/#/.

Cox, Dan, ed. "iftechfoundation / twine-cookbook." 2017. GitHub, 2019. https://github.com/iftechfoundation/twine-cookbook.

incobalt. "incobalt /T rice." 2018. GitHub, 2019. https://github.com/incobalt/Trice.

Lutz, Michael. *My Father's Long, Long Legs*. Correlated Contents, September 23, 2013. http://correlatedcontents.com/misc/Father.html.

Porpentine. "With Those We Love Alive." Cambridge, MA: Electronic Literature Organization, 2014. http://collection.eliterature.org/3/work.html?work=with-those-we-love-alive.

Squinkifer, D. *Quing's Quest VII: The Death of Videogames*. Self-published, September 1, 2014. https://games.squinky.me/quing/.