INTRODUCTION TO TWINE MANUAL
(Harlowe 2.1.0)

By Conor Walsh
Table of Contents

What is Twine?
Usage Examples
Basics of Twine
Formatting in Twine
Implementing Visuals
Interactive Mechanics
Interactive Visuals
Other Twine Formats
Packaging Twine Projects
Sharing Twine Projects
Troubleshooting
Conclusion/Further Learning
What is Twine?

Twine is an open-source, free program that allows users to create interactive stories, games, and other media through the use of interconnected slides, which we will refer to as “passages”, of text. Its main strength is in the versatility of its design; a complete beginner can get the hang of using Twine at the simplest level in a matter of minutes.

Twine is useful for media development and as a teaching tool. You could make an interactive story with multiple endings without delving into the coding aspects of Twine at all, or you could make a more involved project with statistics, moving images, and CSS-adjusted backgrounds. A savvy teacher could build a Twine with a variety of interactive elements, such as text that can be clicked to reveal pictures or further information about certain topics. This would allow students a greater degree of agency in what they’re learning, and would help to keep them interested.

This guide will not tackle all avenues or possibilities that Twine is capable of, but rather the basic-to-intermediate range of functionality in Twine’s Harlowe format, or language, as well as how to package your projects and share them online. Initially I will discuss how to create a Twine project at its most basic level, giving you the tools to create a simple Twine project. Following this I will include information on a variety of topics, including, but not limited to, making your Twine more visually diverse, more reactive to user input, and ways that you can share your project with others. My hope is that this will reflect Twine’s versatility; you can use the sections that seem interesting or otherwise useful, while foregoing those that don’t seem as appealing to you.

I should note that I, as the author of this guide, come from no coding background and still find coding actively intimidating in many respects. However, with enough perseverance and a large amount of help I’ve managed to gain a better understanding of Twine and the confidence to make my own projects and help others with theirs as well. The concepts I’ll discuss may or may not come naturally to you and certainly didn’t come across naturally to me, for the most part, but there’s no shame in repeatedly trying and failing something so long as you continue to have the confidence to try. Ask questions, look around online, and above all, don’t get discouraged! If I can do it, you can do it!

Usage Examples

Twine has a broad array of applications, and as such a variety of different kinds of projects have been made in its framework. What follows will be a number of examples of different kinds of Twine projects.

Many users use Twine to make interactive fiction or text-based adventure games, but even these can vary broadly in content and mechanics. One example is Crow Crow Crow’s “Temple of No”, a comedic Twine game with a reliance on images and sounds. It can be found and played here: https://crowscrowscrows.itch.io/the-temple-of-no.
Figure 1: A screenshot of the title screen of “The Temple of No”.

A personal example of a text-based game is “Six Shots”, a western-themed, choice-focused game that I put together over the course of about nine months. This game contains less in the way of images and sounds but more in the way of meaningful decisions and freedom of gameplay. It can be found here: https://banzaibonsai.itch.io/six-shots

Figure 2: A screenshot of the title screen of “Six Shots”.

A number of different Twine projects also have been created for non-game reasons, whether that may be to teach or as an application. Chapel’s D&D Spellbook is a Twine-built application that can be used to store player spells for the role playing game Dungeons and Dragons. Users can build whatever combination of spells their D&D character uses in the program, making them easy for access at will. It can be found here: https://twinelab.net/spellbook/

![Figure 3: A screenshot of Chapel’s D&D Spellbook.](image1)

On yet another personal note, for Cal Poly’s ENGL-459 class a few of my classmates and I put together a Twine version of a selection from Joseph Conrad’s *The Secret Agent*. Our focus was to create a number of potential situations for the story’s events to unfold differently, and in doing so to expand on some of the themes and concepts explored in the novel. It can be found here: http://philome.la/Poor_Penmanship/conrads-the-secret-agent---the-game

![Figure 4: A passage from “Conrad’s The Secret Agent: The Game”.](image2)

**Basics of Twine**

In this section, you’ll learn the most basic elements of how to use Twine, including how to use the Twine interface, how to create passages of text, how to make it so that users can
travel between these passages, and how to publish your project to an HTML file so that it can be played.

Foremost, before the program can be used, it must either be used online through the Twine website or be downloaded. This guide is primarily for the downloadable version of Twine, as the browser version has a tendency to act strangely and delete user data. Twine 2, the version of Twine we’ll be using, can be downloaded at [http://twinery.org/](http://twinery.org/).

Upon being opened, the user will be brought to a blank screen with several buttons on the side. For now we’ll only worry about the button in the top right. Click this button to create your first Twine document.

![Figure 6: The initial screen that appears when Twine is first opened. Option that will create a project is circled.](image)

It bears mentioning that if you’ve just downloaded Twine, you will automatically be set to the default format, Harlowe 2.1.0. This is the format that this guide primarily discusses, and is the most user-friendly, though it is less functional than the other formats and to those who are more familiar with coding. If you are interested in learning about different formats, or otherwise want to learn how to change formats, more information can be found [here](http://twinery.org/).

If you’d rather continue to use Harlowe 2.1.0, you don’t need to do anything, and can continue reading this document.

After clicking this and naming your document, you will be brought to a grid with a single square labeled “Untitled Passage” There will also be a series of buttons at the bottom of the screen.
Figure 7: A picture of the Twine workspace in an example game.

**Passages**

Twine projects are made up of passages. Each passage is an individual slide with text on it; each passage exists on its own, acting as a single “page” of your project.

Double-clicking the initial passage, which will by default be labeled “Untitled Passage,” will bring you to the following screen:
This first passage will be the opening passage of your Twine, and the first thing your players see after opening your project. You can change the “Untitled Passage” text at the top to anything that might remind you of what the passage is actually about. I, for example, chose to re-title it “Opening” so that I could remember that this is the first passage in my Twine. Players will not see the title of any given passage, but it’s important for you to remember that these titles are the key to creating links between passages. Any errors in typing out a passage name as part of a link will result in the creation of a new passage or will otherwise fail to properly create links as you want them to function.

Your Twine’s visible text will be composed of everything in the field labeled “Double-click this passage to edit it.” You can begin your story here, writing whatever it is you might want your story to start with. Here is an example of an opening passage:
You can also create new passages by clicking the button at the bottom right.

Each passage is self-contained, and users can move between passages while playing through your project through the use of links.
Links

In order to transition to other passages, and to give players options, you must make use of links. Links are lines of text that, when clicked, allow the user to move between passages. In order to create a link, you only need to write a line of text in between two sets of brackets. This will bring the player to whichever passage is within the brackets. For example, a link written as \[\text{Opening}\] in the passage would bring the player to a passage with the title “Opening” when clicked. If a passage named “Opening” does not exist, a new passage will be created. If you want the display text to be different than the passage name, write the link as follows: [Visible Text->Actual Passage Name]. Links can be seen in action in the following images. It should be noted that the boxes that can be seen in the second picture are each passages, and each of these passages bear their titles as the text that can currently be seen on each box.

Figure 11: A passage that has been completed with a number of different examples of working links.
Figure 12: A visualization of how links connect between passages in Twine’s main workspace.

Pressing the “Play” button at the bottom of the screen will bring you into a testable version of your Twine, which will allow you to look at your text as it will appear to users. Doing this will also allow you to see how any coding or visual adjustments will look to your players. As such, testing your project every so often is important to creating a cohesive, well-functioning end product.

Figure 13: Twine workspace with the “Play Project” button circled.
If you have variables or other information that you want to see while you’re playing you can click the “Debug” button next to it. We will discuss variables, and how they can be applied, here.

Publishing

When your Twine is finished, you’ll probably want a way to show it to others. In order to do this (at its most basic level) you must publish your Twine as an HTML file. To do this, click the triangular button on the bottom left of the main Twine screen. A number of options will come up; the one at the bottom will say “Publish to File”.

Figure 14: Example of how a passage of a Twine project may look to a user.
Click “Publish to File”. From here you can save your Twine wherever you want on your computer. This file, when double-clicked, will open your Twine in your browser in “playing” format, and can be freely played by anyone you send it to so long as their browser supports HTML files. If you’ve followed the steps up to this point, congratulations--you have a working Twine project!

Saving

There is no manual “save” function in Twine 2; instead, work progress is automatically saved on your computer. Though I haven’t had any problems with it myself, it’s been reported that Twine’s autosaving is finicky, and can corrupt your files. In order to make sure none of your data is lost, you can periodically create Archives of your files. To do this, go back to the main menu and click the “Archive” button. This will create a copy of your stories somewhere on your computer that can be re-loaded should something happen to your data. You can re-load an archive by clicking “Import From File” in Twine’s main menu and selecting your archive file.
Formatting in Twine

With just the information above, you would be able to create your own working Twine project. The rest of this document will contain tips and explanations of how to use Twine’s functionality—that is, the tools coded into Twine itself, including code and interface—to change certain facets of your projects.

The first Twine functionality that we will discuss is text formatting. Text formatting, such as changing your font size, style, or italicizing certain text blocks, functions a little differently in Twine than it does in applications such as Microsoft Word or Google Docs.

Before we begin, it serves to mention that it’s possible to format text in Twine through the use of CSS, a web-based coding language, rather than only through in-program codes and functions. The use of CSS is often important to larger Twine projects, as certain functions, such as changing all font in a Twine project, can be done much more easily through CSS than on each individual passage page or in a header or footer. If you’d like to learn about some of the elements of CSS-specific coding, you can find more information on w3schools.com.

You can enter CSS code into your Twine project’s Stylesheet, which can be accessed here:
Be forewarned, however, that some of the characteristics of CSS-formatting in Twine are Twine-specific. For example, in regular CSS the style markup, that being the specific way that a line of code is typed out in order to work, for changing code related to links is `a:link{}`. However, in Twine the markup for this code is `tw-link{}`. If you don’t know any CSS, and this doesn’t mean anything to you, don’t worry; we’ll go through the simplest CSS used in Twine, and the associated style markup, below.

If you have no knowledge of CSS and don’t wish to learn it, information on non-CSS coding will be included below in processes where CSS isn’t mandatory. Be warned, however, that learning to use CSS instead of macros can save you time, energy, and will clean up your code, making it less likely that you’ll make mistakes.

**Headers and Footers**

Much like in a text program, Twine projects can include headers and footers. Headers may be used for a variety of purposes, such as improving the aesthetics of your project, giving your piece a running title, or to have a piece of code that runs on every page.
Headers are created by making a separate passage and using the tag function to designate it as a header. Click the “Add tag” button beneath the title section of your passage and type the word “header”. Then press ok.
This passage will now act as a running header, appearing before each passage throughout the rest of your Twine project.

If you wish to include a footer that appears at the bottom of each passage throughout your project, repeat the above process but write “footer” instead of “header” in the tags section.
It’s also possible to create a “startup” passage, which will pre-load any data that your Twine project might use at the very start of a playthrough. This may be useful if you’re using lots of variables or complicated code that needs to load over a period of time. As we currently haven’t discussed more complicated code or what variables are, more details on what you could choose to include in a “startup” passage can be found on this page. For the most part, though, it’s used to pre-load more code-heavy information for your project.
Twine doesn’t have a toolbar of style formatting options as might be found in Microsoft Word or Google Docs, and as such it might not be abundantly clear how to italicize or bold a block of text. Simple formatting is mainly accomplished through the use of characters such as backslashes or asterisks on either side of the block of text you hope to affect. Some elements of this formatting technique, also known as Harlowe’s “markup”, can be found as follows:

- **Italics**: //Text// or *Text*
- **Bold**: ‘Text’ or **Text**
- **Underline**: <u>Text</u>
- **Strike-through**: ~~Text~~
- **Superscript**: ^^Text^^

In order for any of these formatting changes to take effect, there must be a special character before and after the text in question. If your character placement isn’t symmetrical, the characters will simply show up normally in the text.

Twine contains a number of other “text-style” options that can blur text, reverse it, or make it shake, to name a few options. These options can be accessed through the use of a “macro”, which is a line of code that has a set function. In order to activate a macro, the user only has to type said macro, follow it with a colon, add any additional, macro-specific information and surround the entire declaration with parentheses. In this case we’ll be using the (text-style:) macro, which is written as follows:

```
(text-style: “style name”)[text]
```

The various options for text-style, and their effects, can be seen on page 1 of the companion Twine.

**Font Modification**

**Font**

**CSS Optional**

If you wish to change the font style of a section of your Twine project, you can simply enclose the text in a (font:) macro. In order to change the font for a particular block of text, surround it in quotation marks and place it after the colon in the macro, such as the following:

```
(font: “Font Name”)[text]
```
Any spelling errors will result in the font not changing properly. You also can only use fonts that are available in your browser.
As you may have noticed, this example only applies to certain sections of Twine projects; it is not possible to change the default font of your project without using CSS. As individually packaging each section of your project within a (font:) macro is unwieldy, messy, and inconvenient, I will walk you through the relatively simple process of changing your story’s default font.

Click the “Stylesheet” button to open your project’s CSS Stylesheet. This is where all CSS coding happens.

Figure 25: A picture of the Twine workspace with the CSS Stylesheet circled.

If you wish for a CSS declaration to affect your entire Twine project, you need to use the tw-story CSS selector. Type “tw-story” on the first line of your Stylesheet, and follow it with an open bracket. An example of this can be seen as follows:
Add an additional line-break between these brackets and type “font-family:”. After this, type out the name of your font, then close the line by adding a semicolon. If you do not type a semicolon your font won’t change and your project will report an error when played, so be sure to remember it. Again, make sure that whatever font you use can be accessed by your browser, and that you are typing it correctly.
Figure 27: The CSS Stylesheet with the font-family: declaration.

All text in your project will now be of your chosen font.

The situation may arise where you need to use multiple fonts throughout your Twine story. In these cases you need to use <div> classes to specify certain sections of text to act a particular way. If that doesn't make any sense, don't worry—we'll talk in some detail about <div> tags later on. If the information later in the document isn’t sufficient.

**Font Size**  
**CSS Optional**

If you wish to increase the size of only a particular section of text in your project, such as for use in a title, you can use the (css:) macro. This macro is a special macro that creates a CSS style rule to be used directly in your project as opposed to being written in your stylesheet. The particular macro for font size is:

(css: "font-size: size%;")[text]

“Size,” in this case, is how big you want the text to be in percent or pixels. For example, if you want your font to be twice as large as it is normally, write (css: "font-size: 200%;")). This will change only the font size of a particular section of your Twine. You can also change your text size in pixels; simply write “20px;” instead of “150%;”.
If you want to change all of the text throughout your project into a uniform size you will again have to use CSS. Open up your Stylesheet as explained above and type “tw-story” followed by two brackets {} if it is not already present. Then type the same text included in your macro: font-size:(size);
In this case “size” is, again, either a number followed by “px” for pixels or by a percentage mark for a percentage change.

![CSS Stylesheet](image)

Figure 30: The CSS Stylesheet with a number of font adjustments, including the font style and the text size.

**Text Color**

*CSS Optional*

There are a number of ways you can go about changing the color of text in your twine project. Foremost is the (color:) macro, which changes the color of any text contained in brackets beside it. The syntax for this macro is:

```
(color: color name)[text]
```

“Color name,” in this context, can be either a written word for the color, i.e. red or gray, or an HTML color code, i.e. #f44242.
If you wish to change the color of all the text in a project you must do so through CSS. Open up your Stylesheet and type “tw-story” followed by two brackets {}, as per usual. Within the brackets type “color:” followed by either the name of or the HTML code for whichever color you want to use. Follow this with a semicolon (;).
Alignment

You can change the alignment of your text through the use of greater than, less than, and equal signs. For example, if you wish to align all text in a passage to the right side of the screen, write `==>` at the start of a passage. All of the following text will be right-aligned. Writing `<=` will align it left again, and writing `==><=` will center all of the following text. Moving to a new passage will change the alignment to the default.

An example of this in action can be seen on page 2 of the companion Twine.

Implementing Visuals

Pictures and Videos

*HTML Required*

In order to implement images or videos in your Twine project, you must make use of HTML coding. We’ve mentioned HTML a few times prior to this point, but only in filetypes and color codes; here we will discuss HTML coding elements.

In this document we will chiefly be using the `<img>` HTML tag. This tag defines an HTML image on a page and effectively turns what would normally be filepaths or urls into static images.

The syntax to include an image in your Twine project is:

```
<img src="http://example.org/image.png">
```
The “src” section of the HTML element stands for source and must be included in order for your HTML element to work. You must include the full URL of whichever image you want to use in the quotes after this, and you must make sure it is only the image in the URL. You cannot, for example, use a single image on a blog page if you enter in the blog page’s URL; your code will throw an error.

You can also change the size of the image by writing the code as follows:
\[
\text{<img src="http://example.org/image.png" height= "number" width= "number">}
\]

Figure 34: The example code of an image.
The code for embedded videos is similar, but the exact method of implementing it deviates. If you have a raw video URL, you should be able to write your code as follows:

<video src="(video URL)"/></video>

However, it’s rather unlikely that you will be able to find, or want to use, a video that is hosted somewhere online without being on a site like Youtube or Vimeo. It’s more likely that you’ll want to use a video hosted on Youtube or Vimeo, and in such cases copying and pasting the URL found at the top of the page won’t result in the video appearing in your project. Instead, navigate to the page of the video you want to use, right click the player, and click “copy embed code.” If you copy-paste this code into a Twine passage, the video will appear.
Figure 36: An example of how to take the embed code from a Youtube video.

```html
<iframe width="560" height="315" src="https://www.youtube.com/embed/sQ1EXkTUPeE" frameborder="0" allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>
```

Figure 37: An example of the video embed code above depicted in a passage.
Information on adjusting the size of this player as well as other useful characteristics, such as autoplay or looping, can be found here: https://www.w3schools.com/html/html5_video.asp

If you want to include an image or video that is on your computer rather than one you’ve found online, you must package these files in a hierarchy alongside your Twine’s HTML file. If that sounds complicated, don’t worry; this process is discussed in detail in the “Packaging Twine Projects” section, which can be found here.

On page 3 of the companion Twine you can see an image file and a video file that have been successfully implemented.

**Backgrounds**

*CSS Required*

The CSS to include a background image is written as follows within the tw-story selector: background-image: url("http://example.org/background.png");
By default your background image will only be its default size, which may mean that it takes up too little of the screen or too much. It will also repeat if it does not take up the whole screen. If you want to adjust these aspects of your background, or others, you can use CSS declarations. Some background-related properties can be found here:

https://www.w3schools.com/css/css_background.asp

Boxes

CSS Required
HTML Required

If you tested out the above and used a complex or detailed image for your background, you may have noticed that the words within your project are difficult to discern from the background image. If this is the case, or if you just want to make your text stand out more, you can create a text box between your words and your project’s background. This can only be done with a combination of CSS and a coding selector called a class. A class is a reusable set of code that can be applied to particular parts of your project with the use of <div> tags. <div> tags are containers that apply your code. These can be applied directly in passages within Twine. For example, we’ll be using a div called “textbox,” so typing “<div class='textbox'>” before whatever text you want to have a box around it will “activate” the code.

When you reach the end of a passage, or if you want the <div> tag to only apply to a certain section of your passage, you can enclose a section of text by typing </div>. While you
don’t necessarily have to do this if there are no other <div> tags on your page, not doing so can potentially cause problems with more complex projects. Getting into the habit of typing </div> to close out your class is a good practice.

Once you have a block of text that is tagged with a <div>, you need to type out the actual CSS code so that Twine will know how to format your text box. In order to do this you actually can’t use the tw-story{} CSS declaration, as we’ve been using before. Trying to do so would apply your text box code to the entire project, as opposed to the textboxes within your story.

Instead, we’ll be using a div.textbox{} declaration. The word “textbox” here can actually be any word so long as it matches the class name in your passage. That is to say, because we wrote <div class='textbox'> in the passage above, the CSS has to also read div.textbox{} rather than div.text{} or something to such an effect. From here you can enter in information as you would in the tw-story{} declaration.
In this example, I’ve already included some property: value pairs within the `div.textbox` selector. By default the “box” around a set of text in Twine is immediate, which is to say that there’s very little space between the edges of the letters and the endpoints of a would-be text.
box. As such, the “padding:” declaration pushes this space out by a number of pixels, in this case 20. I also included a background: declaration and a color. Without this declaration there would be no visible box behind the words within the <div> class.

If you want every passage in your story to have the same box behind it, you have to individually apply the <div> tag to all of the text within each passage. This can be complicated and annoying if you have a large number of passages or if there's a lot of code in your passages and you don’t want to risk overcomplicating things. You can solve this problem by putting your <div class='textbox'> code at the end of a header. As the header automatically appears before each passage of your project, the div will also automatically be applied without you needing to individually copy-paste it into every passage. This is especially useful if something in your <div> changes, as it will mean that you don’t have to find every instance of it in every passage and change it individually.

There are many ways to style a box component so that it looks nice or adheres to your purposes. More information can be found on this W3schools page: https://www.w3schools.com/css/css3_borders.asp.

A functioning example of a textbox can be found on page 4 of the companion Twine.

Interactive Mechanics

Twine includes a variety of coding methods that can be used to make your projects more responsive to user input. These methods typically make use of a macro, or a number of different macros working in tandem, in order to achieve a particular goal. This section will detail a number of important macros that can be used in your project to track certain in-project events.

Formatting Links

CSS Required

You may find that you want to change the default coloring, size, or other characteristics of your project’s links. This can be done through CSS.

The following CSS declarations are able to affect links:

tw-link, .enchantment-link {}
tw-link:hover, .enchantment-link:hover {}
.visited {}
.visited:hover {}

Each of these declarations affects a different part of your links. The first declaration (tw-link, .enchantment-link{}) affects your link’s initial appearance, when the player has not moused over it or clicked it yet. You can also affect these portions of your links by typing tw-link{} without the .enchantment-link part of the declaration.

The second declaration (tw-link:hover, .enchantment-link:hover{}) affects how your links look when the player hovers their mouse over the link.
The third declaration (.visited{}), affects the appearance of links that will lead the player back to passages where they have already been.

The fourth declaration (.visited:hover{}) affects the appearance of links that will lead the player back to passages they have already visited when the player hovers their mouse over said link.

Each of these CSS selectors can be adjusted individually, though formatting in the \tw-link{} or \tw-link, .enchantment-link{} declarations will also affect all of the other link declarations if you do not have any new formatting changes to overwrite the ones for other link declarations.

**Variables**

A variable is a value or line of text, typed out or “set” by the user in certain lines of code, that can change based on input from the program. This value can then be checked by lines of code and used to cause certain outputs. Variables in Twine have a special character that must precede them ($) without a space in order for Twine to properly identify them as a variable. For example, if I wrote (set: variable to 3) Twine would respond with an error. However, if I wrote (set: $variable to 3), Twine would properly set the variable “$variable” to 3.

Variables do not need to be given numerical values; they can also be given text values instead. Below are a few examples of potential ways of declaring, or writing, variables.

![Figure 43: A number of different ways to set variables.](image)
In order for your Twine to operate based on a variable, it has to “check” the variable. What this means is that there are certain code macros that read variable values and, based on these values, perform a certain function that is defined by the coder, e.g. you. Variables are primarily checked, and used to affect your project, through the use of the (if:) macro.

The way an (if:) macro is documented differs slightly from the other macros mentioned so far. An (if:) macro is written as such: (if: $variable is num)[output], where “variable” can be substituted for any variable name and (value) can be substituted for any number or word. An example of a working (if:) statement is as follows:
This example will print the line “Thanks for calling earlier. I’ve decided not to fire you.” if the value of $apology was set to 1 earlier in the Twine. However, if $apology was not set earlier in the Twine, or if it was set to a different value, nothing will be printed when this passage is reached. If you want something different to be displayed for a different variable value, you can simply write another (if:) statement immediately after the first. An example of (if:) code with two possible options based on code is below:
Figure 46: An example of a branching path. Depending on what the player decides here, they will be fired or keep their job later on. Note that “Work” capitalized and “work” not capitalized are two different passages; titles are case-sensitive.

Figure 47: A visual example of one of the branching paths above. The $apology variable, which the (if:) macro will later act upon, is set to 1 here.
Figure 48: Another visual example of one of the branching paths. The $apology variable is set to 0 here instead, which will result in the player being fired on the “work” passage.

```plaintext
NoCall

+ Tag

* Why would you call your boss? You've got places to be:
* (set: $apology to 0)
* [[Better get moving. -> work]]
```

Figure 49: The “end” of the branching paths depicted above. Depending on what the player picks, one of these options will show.

```plaintext
work

+ Tag

* (if: $apology is 1)
* ["Thanks for calling earlier. I've decided not to fire you.
* (if: $apology is 0)
* ["I wouldn't fire you, but you never apologized for being late. You're fired.
```

In this situation, the player could have a value of $apology that is equal to 1 or 0. Each situation will have a different output; both outputs cannot activate at the same time because $apology can only have a single value. If $apology has any other value, nothing will be displayed.
In some cases you may only want an output to fire if you have a specific variable value, and for there to be only one different output for all other cases. This can be done by using the (else:) macro. The (else:) macro functions similarly to the (if:) macro, except you don’t need to specify that it only will affect a specific variable value, though you still do need to include its output in brackets, e.g. (else:)[output]. An example of the (else:) macro in action is below.

![Figure 50: An example of the (else:) macro alongside an (if:) macro.](image)

In this example, a player can only get into an establishment if the “$money” variable has a value of more than 1000. If $money is set to 1000 or less the line will read “Nobody gets in without an appointment.” In this example I also used a greater-than (>) symbol in the (if:) macro. By using this, a user can check for a variable having a value greater or less than the number written in the macro. So, in this example, a player must have 1001 dollars or more if they want to get into the establishment.

An example of variable setting and if/then statements can be found on page 5 of the companion Twine.

**Passage Transitions**

As you work on your Twine project, it may so happen that you need to bring a player to a new passage without using a regular link. It may be that you want to change a variable depending on which choice a player makes on a passage, or that, if a player takes too long on a particular screen, they will automatically be moved to a different passage (this can be done with the use of the (live:) macro, which we will talk about [here](#)). In these circumstances, the (go-to:) macro is used. Below you can find an example of this macro in the context of the apology/firing scenario used to define (if:) scenarios earlier in this document.
Walsh 42

Figure 51: Another example of the “late to work” scenario above. In this example, through the use of the (link:) and (go-to:) macros, $apology is set to a value and the player is moved in one slide.

The (go-to:) macro will automatically take a player to whichever page it is connected to when the macro is activated, which happens either automatically if the (go-to:) macro is by itself on a page or when a set of criteria is fulfilled, such as the player waiting a certain period of time or clicking a button. Its syntax is as follows:

```
(goto: “Passage Name”)
```

The name of the passage you want to transition the player to goes in quotes after the colon.

If the macro in this example is placed as-is on a passage, the player will automatically be taken to the passage titled Passage Name without seeing anything on the passage containing the (go-to:) macro. If you want to avoid this, you must enclose the (go-to:) macro inside a different macro. The most common example of this is the (link:) macro, which effectively turns any section of text into a custom link that will make something occur when the text is clicked. An example of this in action is as follows:

```
(link: “Clicking this will add to a variable and take you to a new screen.”) [(set: $variable to it + 1) (go-to: “New Screen”)]
```

Much like the (if:) macro, the (link:) macro can only affect text or code enclosed in the brackets next to it.

An example of a link macro in action can be seen on page 6 of the companion Twine.

Resetting Player Progress
There are a number of ways that you can reset player progress in your Twine project. The main two macros for this are the (undo:) macro and the (reload:) macro. Both of these macros has the same issue as the (go-to:) macro in that if they are not included within another macro they will be applied instantly, which you likely will not want.

If you want a player to return to the previous screen, whether forcefully or by choice, you can use the (undo:) macro. This macro will return the player to whatever the prior passage was when activated, effectively “undoing” their progress.

If you choose to use this macro, do be aware that any variables or other data that changes upon reaching and returning from a passage containing the (undo:) macro will also be reverted. For example, if you have a variable that increments every time you visit a passage, going to a passage with the (undo:) macro and then back to the original passage will not increment the variable twice. Instead, Twine will treat it as if your player never visited the passage containing the (undo:) macro.

If you wish to completely reset your player’s progress in your Twine project, you can use the (reload:) macro. If a passage contains this macro by itself, your Twine story will instantly restart from the beginning once this passage is reached. In practice this serves much the same purpose as a user reloading their browser. Be warned, however, that all player progress, including variables, will be completely reset, and this process will be instantaneous unless the macro is contained in another macro.

The (link:) macro can be used with the (reload:) macro to create a simple “restart” button for your project. For example, if you were to write: (link: "Start over from the beginning?")[()]((reload:)), a link that says “Start over from the beginning?” would appear in your player’s version of the project, and would reset the project from the first passage when clicked.

A restart link can be found on page 7 of the companion Twine.

Randomness
You can create randomness in your project through the use of the (either:) macro. This macro takes any text or numbers that you may have included and will randomly output one when the user reaches a passage containing this macro. The syntax is as follows:

(either: text, text, #, #)

To give a clear example of how this macro will function, if a user were to type (either: 1, 2, 3, 4) on a passage, one of these numbers will be randomly selected each time a player views a passage. This can also be used with lines of text, description, or dialogue. If you wish to randomize lines of dialogue, however, you must put backticks (‘) on either side of your quotation marks, or the quotation marks will not appear. An example is as follows:
An example of the (either:) macro can also be found on page 8 of the companion Twine.

**Timed Events**

In Twine's Harlowe format the only way to create a timed event is through the use of the (live:) macro. The (live:) macro constantly repeats the code or text that it is acting upon on a certain time interval, repeating indefinitely unless the player navigates to a new page or the (stop:) macro is used. This can be useful to create timed events, such as a link appearing when a player stays on a page for a certain length of time, or to cycle between a series of text blocks on a certain time interval.
The syntax for the (live:) macro is written as follows:

(live: #s)[text]

“#s”, in this case, is how frequently the macro runs; it will run once every time the period of time you’ve specified passes. This period of time can be defined in seconds (s) or milliseconds (ms).

If there is no code in the “text” section, a certain number of seconds will pass and then the text in the box will appear. As there is no code, the macro will just show whatever text you’ve included without changing. This can be used to make text appear over time in a passage; if you have two paragraphs and use a live macro for both, and write the first as (live: 1s)[paragraph] and the second as (live: 5s)[paragraph], the first paragraph will appear after one second, and the second will appear after five.

If you want a (live:) macro to stop counting, you can package it in with a (stop:) macro. Generally this is best done in conjunction with (if:) statements, as a (stop:) macro contained within a (live:) macro will instantly stop the (live:) macro’s update period, resulting in nothing happening. Therefore the (stop:) macro should only be used if you want your (live:) macro to stop after a certain period of time, or if you want to make something happen only once after a period of time has elapsed happened.

What follows is the code for a visually present timer that counts to 30, then stops.

(live: 1s)[(set: $time to it+1)
$time
(if: $time is 30)[(stop:)]]

This macro can be seen in action on page 9 of the companion Twine.

Other Interactive Macros

In addition to those detailed above, Twine offers a wide array of other macros that can make your project more interactive. Some, such as the (mouse-over:) macro, causes text to change on screen when users mouse over certain words. Others, such as the (click-append:) macro, can add text to linked phrases when they are clicked on by the user. As there are many of these macros, and as their usefulness can vary wildly, I encourage you to read through the Harlowe documentation and test out any that look interesting or that I haven’t covered in this document. Doing so will allow you a better understanding of what Harlowe has to offer and will give you the opportunity to test out and play with a wide variety of different macros.

Harlowe documentation: https://twine2.neocities.org/2.html

Interactive Visuals

Image Links

Using an image as a link is as simple as typing out the image src in the place of the visible text for a link. For example, typing the following would create an image that, when clicked, would bring the user to the passage called “PassageMain”:

[[<img src="http://clipart-library.com/images/riLnggy8T.jpg">->PassageMain]]
A functioning image-link can be found on page 10 of the companion Twine.

Clickable Images

HTML Required

Where image links are fairly straightforward both in purpose and in the work that goes into creating them, clickable images that are not links are decidedly more complicated. Creating a clickable image that will correctly remain in place after it is first clicked requires the use of the (click:) macro as well as Twine’s tag functionality, neither of which we’ve discussed up to this point.

First, we’ll discuss the (click:) macro. This macro takes a block of text and turns it into a link. While this may sound similar to the (link:) macro, the (click:) macro is different in that it does not need to be tied to a specific block of text like the (link:) macro does; instead it can be written anywhere on a page, and will affect all text that matches whatever is in the “text” portion of the macro. The syntax for this macro is as follows:

(click: "text")[output]

A passage utilizing the (click:) macro with the following code:

```
• The end.
•
• (click: "The")[Or is it?]
```

Figure 54: An example function of the (click:) macro.

would function as follows after the “The” is clicked:
It should be noted that the (click:) macro can only be used on text, not code. Writing out (click: "<img src="http://clipart-library.com/images/riLnqgy8T.jpg">")((set: $variable to it + 1)) will throw an error. The workaround for this is to apply something called a “nametag”, or “tag”, to your image src code. Doing this is the only way to cause a code reaction when one of your images is clicked.

A tag is effectively a way to rename a block of text. It is primarily used to apply large amounts of code to large blocks of text. The syntax for applying a tag is as follows:

[Passage text]<paragraph]
In-project all of the text within the [Passage text] section will appear normally. If you wanted to or apply a macro to it, you can now write out whatever word is contained in the <tag> written above with a question mark before it. This has a number of broader applications, but for the moment we’ll just consider how it meshes with images. If you have an image, you can apply a tag to this image to make it much more palatable for Twine’s code to handle.


If you take this example and apply it to the earlier example that would have thrown an error, you get the following:

(click: ?image)[(set: $variable to it + 1)]

Moreover, this coding block will work, though only once, when clicked.

Figure 57: An example of an image src blocked within a tag. The tag here would be ?image.
Figure 58: The code above in action. If this checkbox were clicked, $variable would become 2.

An example of a working clickable image can be found on page 11 of the companion Twine.

Other Twine Formats

At the beginning of this guide I mentioned that although there are a number of formats within Twine, the one we’d be using is the Harlowe format. Though at this point you may be more familiar with Harlowe, it bears mentioning that the other, hitherto unnamed formats have more functionality than Harlowe, although these formats usually require more coding knowledge to use effectively and are less user-friendly. If these other formats interest you, changing a story’s format is a relative cinch.

Click the arrow at the bottom-left of the screen to be brought to your project options menu.
Click “Change Story Format” to open the following box:

Harlowe 2.1.0, which you’ve been using, is the most beginner-friendly out of the three formats, with a code style that is easy on the eyes and simple to understand. Sugarcube is essentially an “intermediate” format, bearing more functionality than Harlowe but less ease-of-use. Snowman is the most “minimalistic” of the three formats and is most frequently used by experienced programmers who want a flexible format at the cost of accessibility.

More information and further reading on Snowman and Sugarcube can be found at the end of this guidebook.

Packaging Twine Projects
We’ve already gone through how to convert your Twine project into an HTML file that can be played anywhere. However, it may become imperative for you to use a file that you’ve made and don’t want to web host in one of your projects. For example, if you expect that your Twine will be accessed over a long period of time, referencing someone else’s files can be a bad idea, as if these files are taken down your project will be filled with broken images. Users also would not be able to play it offline without these images breaking. Whatever the case, you’ll need to compile, or package, your entire project, and any supporting files, into a single .zip or .rar file.

Before we get into that, however, it’s also important to note that because you’re using local files instead of web-hosted ones, you have to reference the files differently. Where before an image src could simply be written as `<img src="http://example.org/image.png">`, now you must create a space for your files to “live.”

Start off by creating a new folder on your desktop. Within this, place the image that you want to use. Other files can also be included in your folders, but for now we’ll stick to images. Rename the image to something simple. For my purposes, I’ve taken a picture of a rat, and have named it “Rat.”

![Figure 61: An example filesystem containing a local file.](image)
Next, type out a new HTML image src. Instead of typing out a url in the “url” section of the src, you'll type the name of the file you’re referencing, followed by the file type. For example, the image that I used is titled “Rat”, and the file type is a JPG. So the image src will read `<img src="Rat.jpg">`. The key point here is that the “Rat” image is not contained in any subfolders, and as such the HTML file for my project will go right next to it. If the image was contained in a subfolder, the url would have to include the name of the folder followed by a backslash prior to the image’s name, such as `<img src="images/Rat.jpg">`.

![Figure 62: HTML referencing a file in the folder above rather than a file online.](image)

From here I can simply publish my story to file and save it in the same folder that the image of the rat is in.
Figure 63: The filespace where you will be saving your HTML file.

Figure 64: The final depiction of your filespace including the referenced image file and your HTML file.

If I double-click the HTML file, the following appears in-browser:
A matter of note: If this is tested in the Twine application, as opposed to with a published HTML file in the correct folder, the image of the rat will not show up as it will not be referenced properly. Make sure to publish your file and test it in-folder if you want to properly confirm that your files are being referenced properly.

Once all of your files are properly referenced and working correctly, you can move on to the compiling portion of your project. We compile files in order to make them easier to transport and host on websites. Though the image of the rat used above is only about 26 kilobytes large, other referenced files, such as videos, can be much larger, and the larger a file is the longer it will take for other users to download and play it. Many file sharing sites also have file size limits.

If your computer has a program such as WinZip installed, right-click your folder and mouse over the tab that says “Send To”. In this tab there should be an option that says “Zipped Folder.”
Figure 66: The compression path for your project.

Clicking this button will automatically create a compiled, or zipped, version of your file. If you were able to complete this step then congratulations—you’ve successfully compiled your files and can host them on sites such as itch.io, which we will discuss in more detail here.

If you do not have a compressing program installed, you will need to install one. WinRAR is a safe and free compiling program that will occasionally hassle you to purchase a license, though you can continue using it without ever doing so. I, and many other users, have used it without any problems whatsoever for years.

A link to WinRAR’s website can be found here: https://www.rarlab.com/.

If you’ve downloaded WinRAR all you now have to do is right-click your example program, mouse over where it says “Add to “(project name).rar”, and click. Confirm any pop-ups with the program and find a place for the compiled file to live. Then, again, congratulations—you’re ready to host your files!
Sharing Twine Projects

Should you wish to share your Twine projects with others, the most easily available sites to publish Twine projects are itch.io and Philome.la. Itch.io is less Twine-specific than Philome.la, but offers more publisher reactivity, whereas Philome.la is less versatile but easier to use.

itch.io

itch.io describes itself as “an open marketplace for independent digital creators with a focus on independent video games.” The site offers free web-hosting and is relatively easy to use, both to download and upload projects on. Even if your project is not explicitly a video game, anything interactive is allowed on itch.io.
If you want to upload one of your projects to itch.io, you'll need to create a profile and fill out the requisite information first. After that, mouse over to the top right corner and click “Upload a new project.”

The “new project” page contains a lot of information, but the greater part of it is self-explanatory and I recommend you read through it on your own. The one exception to this is the “Kind of Project” subheader. In this section you will tell itch.io that you are either uploading
an HTML file, which is your Twine project, or a .zip file containing an HTML file, as we went over in the “Packaging Twine Projects” section above.

![Image of itch.io project depiction](image)

**Figure 70:** The section of itch.io where you depict what kind of project your uploaded file will be.

If your project is contained within an HTML file, simply upload it to itch.io, fill out the other requisite information, and click “Save & view page” at the bottom of the screen. Your project will now be playable by anyone who comes across it on itch.io!

If you’ve packaged your file, it’s a little more complicated. First, you’ll want to go back into the folder containing your HTML file—not the packaged folder, but the original. You’ll then want to rename whatever your HTML file’s name is to “index”.

![Folder with playable HTML file renamed to “index”](image)

**Figure 71:** Your folder’s filespace with the playable HTML file renamed to “index”.

You have to do this in order to make your file playable in-browser; if you don’t rename this file, itch.io won’t know which file is playable, and your users will have to download your project and unpack it manually in order to play it. However, now that you’ve renamed your HTML file, you can re-package your folder by turning it into a .zip or .rar file and can upload it to itch just as you would otherwise.

*Philome.la*
Philome.la is more straightforward to upload to than itch.io, but allows the uploader less freedom and has a much more narrow appeal than itch.io does. It also requires the user to have a Twitter page.

Upon navigating to Philome.la, the user will be met with this page:

![Figure 72: The front page of Philome.la.](image)

This page is as simple as it seems; if you log in with your Twitter handle, drag-and-drop your HTML file, and name your project, your project will be published to Philome.la where it can be played freely in-browser. As efficient and simple as this is, not all users have or want Twitter profiles and this approach stymies more complicated Twine projects as Philome.la does not accept .zip or .rar files.

If your Twine project is on the simpler side or doesn’t reference local files, or if you simply don’t want to deal with the more complicated nature of itch.io, Philome.la could be for you.

**Troubleshooting**

Now that you’ve read the majority of this document, it bears mentioning that the program currently has some characteristics that may make working on your projects more difficult. Knowing these characteristics will help you avoid losing your work later on.

**Undoing/Redoing**

Undoing text in Twine is extremely unreliable, and there is no redo feature. A user can press Ctrl+Z or “Undo” in the “Edit” tab to remove whatever character was most recently entered, as with any other word-editing program. The caveat to this is that it only works on a single line; if you start a new line you will no longer be able to undo anything typed on the prior line. There is no way to “redo” an undone change aside from retyping it, and if you accidentally
delete a segment of text you can’t undo the deletion. Also, if you close a slide and re-open it, you won’t be able to undo anything typed within it.

**Coding Syntax**

Twine can be particular with how its functions are coded in order to work correctly. I’ve most frequently had trouble getting code to function on lines of dialogue or other text contained in quotes without removing said quotes or throwing an error. I’ve also had trouble getting code to properly work on links without throwing an error.

Foremost, if you attempt to include text in quotation marks in any macro that uses quotation marks in part of its syntax, you’ll cause an error. For example, if you want to use the (either:) macro to randomize dialogue, and you write your code like this: (either: “”Hi, how are you doing?””, “Hello!”, “Good day!””), your project will display an error instead of any line. Instead, you need to include a single apostrophe mark (’) before and after each quote, rather than the usual quotation marks. This works in most coding cases. However, quotation marks again cause issues when using links. If you try to link to a passage through dialogue, by typing something like [[“I guess it’s a good idea.”]->Bad Idea Passage]], the quotation marks around “I guess it’s a good idea” will not appear when your project is played. In order for them to appear, you need to include back-ticks (`) before the open quotes and after the close quotes in your link. So, instead of the above, you would write [[““I guess it’s a good idea.””]->Bad Idea Passage]].

Speaking of links, the regular syntax of most macros generally requires the use of brackets ([[]]) in order for the program to know what text should be affected by each macro. However, if you want your macros to work on links, using the regular two-bracket link syntax ([[link]]) will instead throw an error. For example, if you wanted to apply the “mirror” style on a link and typed your code out like this: (text-style: “mirror”)[[“This text is mirrored”]->Passage 2]] the player would throw an error. Instead you need to use the (go-to:) and (link:) macro. With the use of the (link:) macro, you could rewrite the above code to the following: (text-style: “mirror”)(link: “This text is mirrored”)(go-to: “Passage2”)].
Figure 73: An example of a link with code acting on it.

```plaintext
untitled passage
+ tag

* {text-style: "mirror"}[(link: "This text is mirrored")[(go-to: "Passage2")]]
```

Figure 74: The above code in action.
Conclusion/Further Learning

I hope this document was informative. Though it is more of an introduction than a complete guide to the more complex elements of Twine, I believe it is sufficient for those who wish to use Twine in a capacity beyond the most basic but who don’t wish to spend hundreds of hours working out all of the finest elements of the program. For those who do wish to broaden their learning, however, I have contained a few links to other web resources.

The Harlowe documentation contains a large number of macros and potential applications and examples for their uses. It is what I used primarily while crafting my Twine and this guide. Users who hope to know how to use more complicated macros to customize their stories can find more information here: https://twine2.neocities.org/

The “Twine Cookbook” gives information on how to do the same thing in each of Twine 2’s different formats (Harlowe, Sugarcube, Snowman). It goes into further detail on many of the concepts mentioned in this resource: https://twinery.org/cookbook/

This page contains a number of links to different video tutorials that can teach users how to use Twine to accomplish more complex functions in detail. These tutorials span all of Twine’s formats and allow users to see Twine coding in action: https://videlais.com/twine-tutorials/

The Twine 2 guide gives some information about the different languages that Twine 2 can make use of but lacks proper development in some places and is unfinished in others. Some information about Twine’s more complex functions, such as using JavaScript or HTML coding, can be found here: http://twinery.org/wiki/twine2:guide

The website w3schools can be used to learn HTML and CSS coding. Though some of Twine’s styling for certain formatting is different from regular CSS, this site can still be used and applied to your Twine projects. https://www.w3schools.com/

With that, I hope this guide has been sufficient to help you learn to use the program. If you have any questions, or feel like you want to give feedback on any part of this document, feel free to send me an email at conorwalsh(at)yahoo(dot)com.

Good luck, and happy Twining!