

Twine Workshop

Interactive Fiction



1

"What?!" cries Batman, nearly crashing the Batmobile. He has been speeding toward the dark streets of Gotham on his way to Police Headquarters, when the familiar Bat-Signal suddenly changed into a grinning skull! *Someone has tampered with the searchlight, thinks Batman. But why turn the bat silhouette into a DEATH'S HEAD?! It must be a warning . . . or a trap.*

Batman's next move could be critical. A wrong decision might mean his DOOM!

If Batman drives straight to Police Headquarters, turn to page 17.

If he radios Commissioner Gordon from the Batmobile, turn to page 26.

If he plays it safe and uses a public phone, turn to page 3.

For more information on the Bat-Signal, turn to page 119.

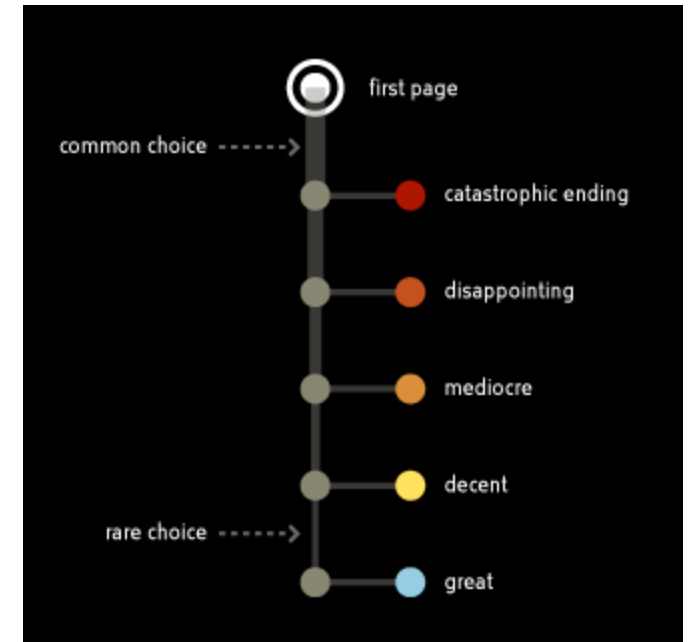
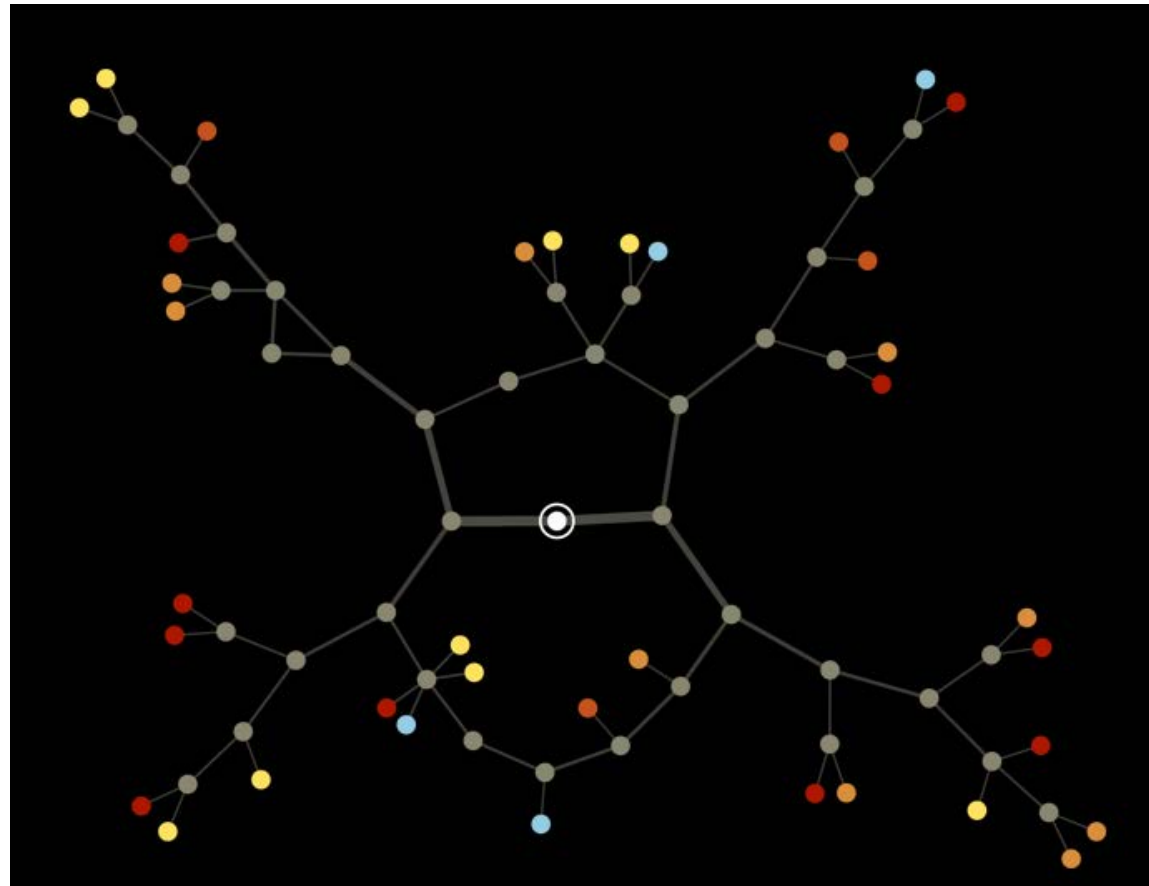
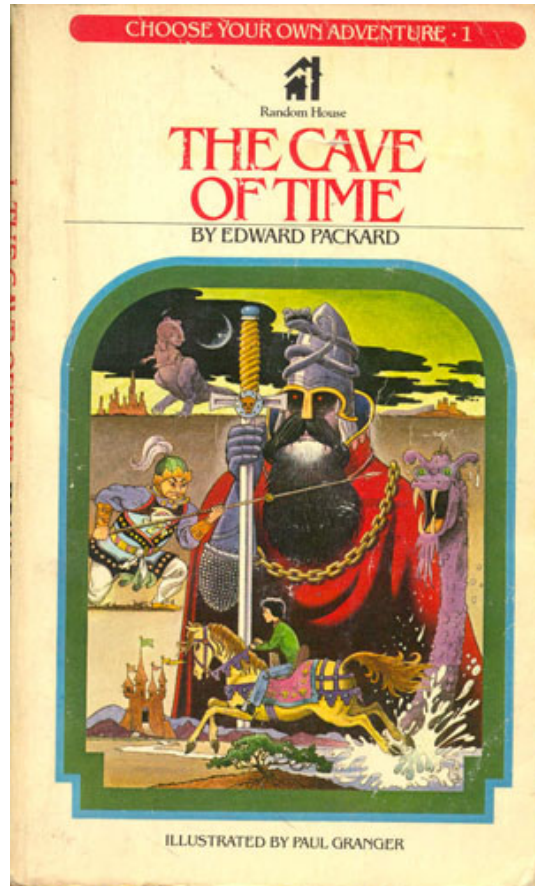
78



As you sit in the warm sunshine deciding about going to Morocco, you catch sight of a small girl—actually a midget—leading a dog. The midget walks up to you, hands you the leash, and before you realize that the dog is a mechanical dog, not a real one, it explodes into a thousand brilliant shards of metal. The explosion finishes you off.

UGH! What a horrible way to go.

The End



<http://samizdat.cc/cyoa/gallery/cave-of-time.html#endings>



Twine

- Open-source tool for building interactive stories
- Outputs to HTML
- Built upon HTML, CSS and JS



Twine is good at:

- Stories
- Poetry
- Text-based RPG
- Hypermedia art
- Prototyping



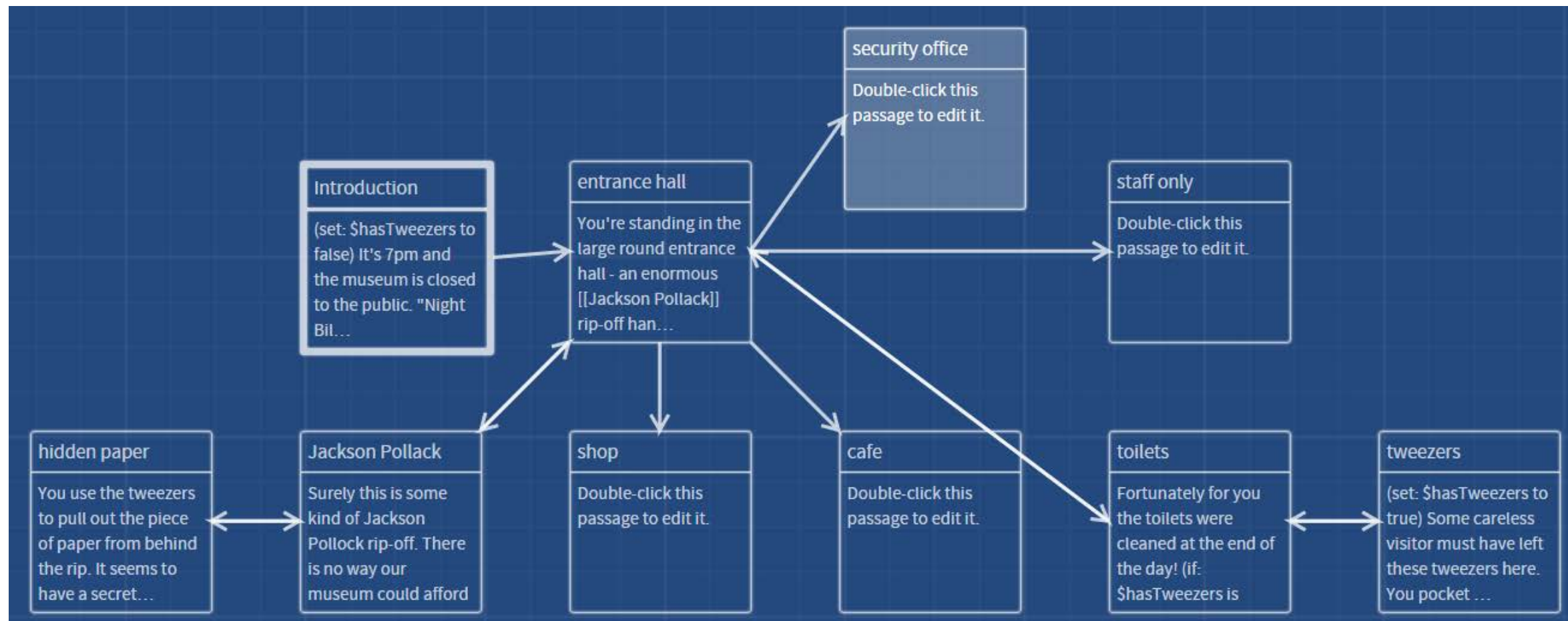
Twine Examples

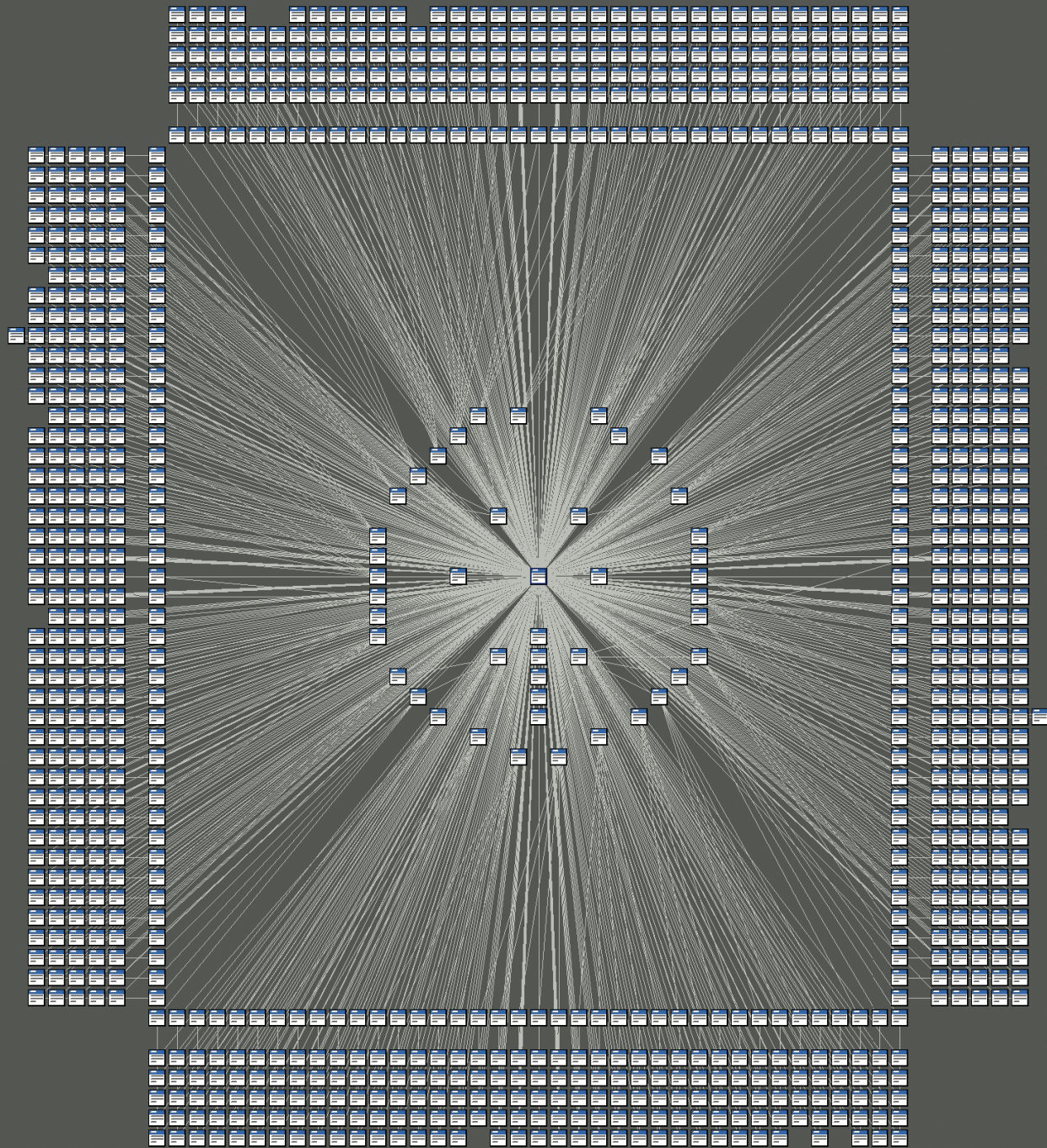
- Story
 - [Lifeline](#) (mobile app based on twine)
 - [Player 2](#)
 - [Queers in Love at the End of the World](#)
 - [Even Cowgirls Bleed](#)
 - [The Uncle Who Works for Nintendo](#)
- RPG/Puzzle
 - [Candy Quest 3: Edge of Sweetness](#)
 - [Live, Run, Die Shop](#)
- Empathy Games
 - [Cis Gaze](#)
- Poetry
 - [A Kiss](#)
 - [Burnt Matches](#)
- Other
 - [Twineplat](#)
 - [HHH.exe](#)



Non-Twine Examples

- [My boyfriend came back from the war](#)
- [Sleep is Death](#)
- [Digital: A Love Story](#)
- [Hatoful Boyfriend](#)





Twine is not so good at:

- 2D/3D graphics-based games
- Platformers, first-person shooters, etc.
- For those, check out: [Phaser](#), [Three.js](#), [Unity](#)





Installing Twine

- Options:
 - Twine 1.4.2 – the old standalone application
 - Browser-based – the latest version of twine (2.01), running in the browser
 - Twine 2.01 – the new standalone application
- Download 2.01 from here: twinery.org

Twine UI Demo



Story Formats

- SugarCube
 - Easy to pick up
 - Flexible, includes save system, widely used
- Harlowe
 - Default in Twine 2
 - A little more restrictive than SugarCube
- Snowman
 - Advanced
 - Allows you to write raw HTML/CSS/JS easily



SugarCube Installation

1. Download the current local version of [SugarCube 2.x for Twine 2](#).
2. Extract the archive to a safe location on your computer and take note of the path to it. I recommend some place like: Documents/Twine/Formats.
3. Click on the Formats link in the Twine 2 sidebar.
4. In the dialog that opens, click on the Add a New Format tab.
5. Finally, paste a [file URL](#) to the format.js file, based on the path from step #2, into the textbox and click the +Add button.
6. Set SugarCube as the default under "Story Formats"



SugarCube



SugarCube Documentation

- Documentation: motoslave.net/sugarcube/2/
- Important sections to start with:
 - Markup – info on formatting
 - TwineScript – info on variables
 - Macros – info on SugarCube's built-in functionality

Headings

An exclamation point which begins a line defines the heading markup. It consists of one to six exclamation points, each additional one beyond the first signifying a lesser heading.

Type	Syntax	Example	Rendered As
Level 1	!Level 1 Heading	Level 1 Heading	<code><h1>Level 1 Heading</h1></code>
Level 2	!!Level 2 Heading	Level 2 Heading	<code><h2>Level 2 Heading</h2></code>
Level 3	!!!Level 3 Heading	Level 3 Heading	<code><h3>Level 3 Heading</h3></code>
Level 4	!!!!Level 4 Heading	Level 4 Heading	<code><h4>Level 4 Heading</h4></code>
Level 5	!!!!!Level 5 Heading	Level 5 Heading	<code><h5>Level 5 Heading</h5></code>
Level 6	!!!!!!Level 6 Heading	Level 6 Heading	<code><h6>Level 6 Heading</h6></code>

Basic Formatting

Type	Syntax	Example	Rendered As
Emphasis	<code>//Emphasis//</code>	<i>Emphasis</i>	<code>Emphasis</code>
Strong	<code>''Strong Emphasis''</code>	Strong Emphasis	<code>Strong Emphasis</code>
Underline	<code>__Underline__</code>	<u>Underline</u>	<code><u>Underline</u></code>
Strikethrough	<code>==Strikethrough==</code>	Strikethrough	<code><s>Strikethrough</s></code>
Superscript	<code>Super^^script^^</code>	Super ^{script}	<code>Super<sup>script</sup></code>
Subscript	<code>Sub~~script~~</code>	Sub _{script}	<code>Sub<sub>script</sub></code>
Code, Inline	<code>{{{Code}}}</code>	Code	<code><code>Code</code></code>
Code, Block	<code>{{{ Code }}}</code>	Code	<code><pre>Code</pre></code>
Em-dash	<code>Em--Dash</code>	Em—Dash	Em–Dash
Avoiding formatting (all markup inside is not transformed and rendered as-is)			
	<code>""Non-formatted""</code>	No <code>'//formatting//'</code>	No <code>'//formatting//'</code>
	<code><nowiki>Non-formatted</nowiki></code>	No <code>'//formatting//'</code>	No <code>'//formatting//'</code>

Images

SugarCube's wiki image syntax consists of a required `Image` component and optional `Title`, `Link`, and `Setter` components. The `Image`, `Title`, and `Link` components may be either plain text or any valid TwineScript expression, which will be evaluated early (i.e. when the link is initially processed). The `Setter` component (which only works with passage links, not external links) must be a valid TwineScript expression, of the `<<set>>` macro variety, which will be evaluated late (i.e. when the link is clicked on).

The `Image` component value may be any valid URL to an image resource (local or remote) or the title of an [embedded image passage \(pre-Twine 2 only\)](#). The `Link` component value may be the title of a passage or any valid URL to a resource (local or remote).

Also, in addition to the standard pipe separator (`|`) used to separate the `Image` and `Title` components (as seen below), SugarCube also supports the arrow separators (`->` & `<-`). Particular to the arrow separators, the arrows' direction determines the order of the components, with the arrow always pointing at the `Image` component (i.e. the right arrow works like the pipe separator, `Title->Image`, while the left arrow is reversed, `Image<-Title`).

For the following examples assume: `$src` is `home.png`, `$go` is `Home`, and `$show` is `Go home`

Type	Syntax	Example	Result	
Image	<code>[img[Image]]</code>	<code>[img[home.png]]</code> <code>[img[\$src]]</code>	Image:	<code>home.png</code>
Image w/ Link	<code>[img[Image][Link]]</code>	<code>[img[home.png][Home]]</code> <code>[img[\$src][\$go]]</code>	Image:	<code>home.png</code>
			Link:	<code>Home</code>
Image w/ Link & Setter	<code>[img[Image][Link][Setter]]</code>	<code>[img[home.png][Home][\$done to true]]</code> <code>[img[\$src][\$go][\$done to true]]</code>	Image:	<code>home.png</code>
			Link:	<code>Home</code>
			Setter:	<code>\$done to true</code>
Image w/ Title	<code>[img[Title Image]]</code>	<code>[img[Go home home.png]]</code> <code>[img[\$show \$src]]</code>	Title:	<code>Go home</code>
			Image:	<code>home.png</code>
Image w/ Title & Link	<code>[img[Title Image][Link]]</code>	<code>[img[Go home home.png][Home]]</code> <code>[img[\$show \$src][\$go]]</code>	Title:	<code>Go home</code>
			Image:	<code>home.png</code>
			Link:	<code>Home</code>
Image w/ Title, Link, & Setter	<code>[img[Title Image][Link][Setter]]</code>	<code>[img[Go home home.png][Home][\$done to true]]</code> <code>[img[\$show \$src][\$go][\$done to true]]</code>	Title:	<code>Go home</code>
			Image:	<code>home.png</code>
			Link:	<code>Home</code>
			Setter:	<code>\$done to true</code>

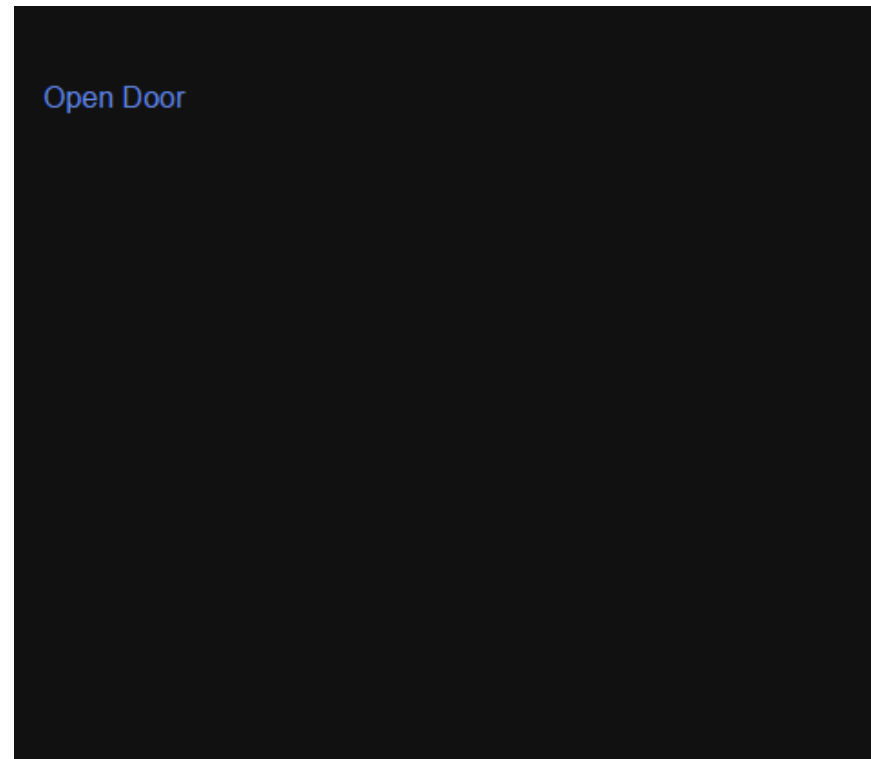
Links



Single Link

+ Tag

[[Open Door]]





Named Links



+ Tag

```
[[Open the spooky door->Spooky door]]  
[[Eat a jellybean|Eat]]  
[[Run<-Run away as fast as you can]]|
```

Named Links

```
[[Open the spooky  
door->Spooky door]]  
[[Eat a jellybean|Eat]]  
[[Run<-Run away as  
fast as you ...
```

Spooky door

Double-click this
passage to edit it.

Eat

Double-click this
passage to edit it.

Run

Double-click this
passage to edit it.

Open the spooky door

Eat a jellybean

Run away as fast as you can

Linking to Media



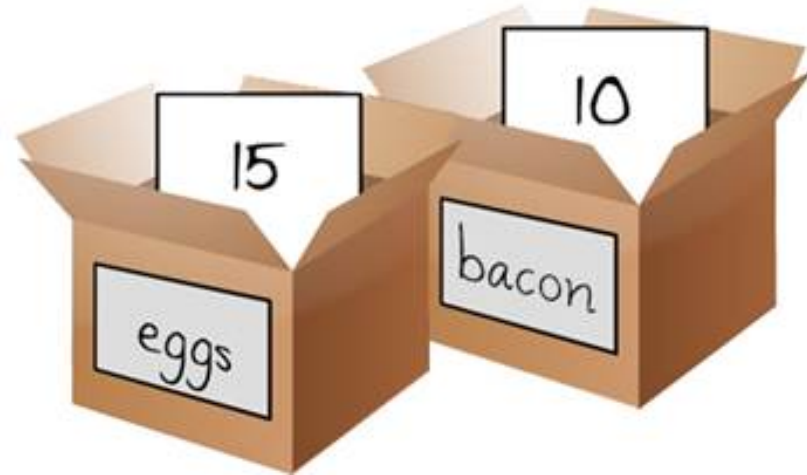
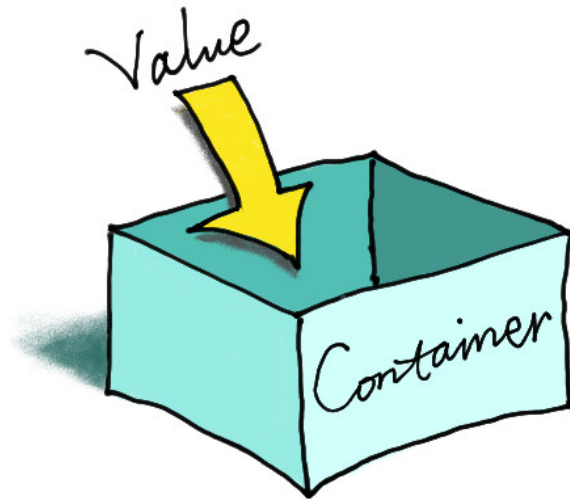
Strategies

- Hotlinking
 - Linking to a file someone else has hosted online
 - If that person takes the file down, you lose access to it
- Hosting Yourself
 - Create a GitHub repository called MyTwines
 - Upload images/sounds/video to folders there
 - Link to them using a URL:
<http://mikewesthad.github.io/MyTwine/images/cat.png>
- Encoding & Embedding
 - More info later...



Variables

Variables Are Containers



```
Variables ×  
+Tag  
  
<<set Sbacon = 10>>  
<<set Seggs = 15>>
```


Twine Variable Types

- Strings
 - A series of characters inside of double quotation marks
- Numbers
 - Whole numbers or decimals
- Booleans
 - Either true or false

```
<<set $name = "Charizard">>  
<<set $weight = 200>>  
<<set $canFly = true>>
```

Twine Story Variables

- Must start with a dollar sign (\$)
- Second character: A – Z, a – z, \$ or _
- Any other characters: A – Z, a – z, 0 – 9, \$ or _
- Pick meaningful, descriptive names!
- We will use [lowerCamelCase](#)

```
$cash  
$age  
$preferredNickname  
$hasKeyCard2  
$hasMetRobot
```

Macros



Macros

- A macro is a piece of programming functionality
 - Under the hood, macros run a piece of JavaScript
- We've already used a macro:

```
<<set ScanFly = true>>
```

- In SugarCube, there are three general types of macros
 - Documentation: motoslave.net/sugarcube/2/docs/macros.html



Set Macro

(Single Expression Macro)

```
<<set expression>>
```

Sets story \$variables and temporary _variables based on the given expression.

Arguments:

- **expression:** A valid expression. See [Variables](#) and [Expressions](#) for more information.

Set Macro

(Single Expression Macro)

Start of Macro

TwineScript Expression



The diagram illustrates the structure of a Set Macro in TwineScript. The macro is represented as a string: `<<set $canFly = true>>`. This string is enclosed in a light gray rectangular box. Above the box, a purple bracket labeled "Start of Macro" spans the first two characters, "<<". Another purple bracket labeled "TwineScript Expression" spans the entire content of the box. Below the box, a purple bracket labeled "Macro Name" spans the characters "set \$canFly", and a final purple bracket labeled "End of Macro" spans the last two characters, ">>".

Macro Name

End of Macro



Textbox Macro

(Multiple Expression Macro)

Oh, hello there... *Er, I know this...* Okay. I knew this once. What's your name?

Continue



Textbox Macro

(Multiple Expression Macro)

```
<<textbox variable_name default_value [passage_name] [autofocus]>>
```

Creates a text input box, used to modify the value of the \$variable with the given name, optionally forwarding the user to another passage.

SEE: [Interactive macro warning](#).

Arguments:

- **variable_name:** The name of the \$variable to modify, which *must* be quoted (e.g. "\$foo"). Object and array property references are also supported (e.g. "\$foo.bar", "\$foo['bar']", & "\$foo[0]").
- **default_value:** The default value of the text box.
- **passage_name:** (optional) The name of the passage to go to if the return/enter key is pressed.
- **autofocus:** (optional) Keyword, used to signify that the text box should automatically receive focus. Only use the keyword *once* per page; attempting to focus more than one element is undefined behavior.

Usage:

```
→ Creates a text box which modifies $pie
What's your favorite pie? <<textbox "$pie" "Blueberry">>

→ Creates an automatically focused text box which modifies $pie
What's your favorite pie? <<textbox "$pie" "Blueberry" autofocus>>

→ Creates a text box which modifies $pie and forwards to the "Cakes" passage
What's your favorite pie? <<textbox "$pie" "Blueberry" "Cakes">>

→ Creates an automatically focused text box which modifies $pie and forwards to the "Cakes" passage
What's your favorite pie? <<textbox "$pie" "Blueberry" "Cakes" autofocus>>
```

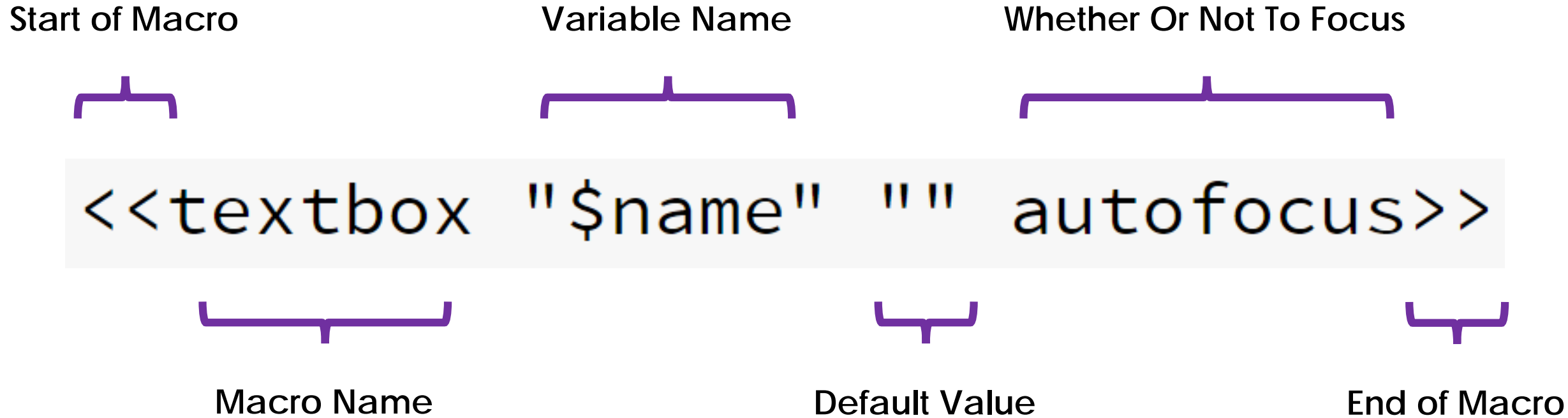
Textbox Macro

(Multiple Expression Macro)

Start of Macro

Variable Name

Whether Or Not To Focus



```
<<textbox "$name" "" autofocus>>
```

The diagram illustrates the components of the Textbox Macro. Above the code, three purple brackets are positioned: a small one under the opening double angle brackets, a medium one under the variable name, and a long one under the focus parameter. Below the code, three purple brackets are positioned: a medium one under the macro name, a small one under the default value, and a small one under the closing double angle brackets. Labels are placed above and below these brackets to identify each part of the macro.

Macro Name

Default Value

End of Macro

Note: When a macro takes multiple parameters, you need to separate each parameter with whitespace!



Conditional Macros

Conditionals

- Oftentimes, you will need to do something only under certain conditions
 - E.g. if the player has the key, let them open the chest
 - E.g. if the player's health drops to zero, restart the game
- In order to express this in code, we need two things:
 - A way to compare variables (conditional operators)
 - A new set of macros (if, elseif, else)



JavaScript conditional operators: (not an exhaustive list)

<code>==</code>	Evaluates to true if both sides are <u>equal</u> (e.g. <code>\$bullets == 6</code>).
<code>!=</code>	Evaluates to true if both sides are <u>not equal</u> (e.g. <code>\$pie != "cherry"</code>).
<code>===</code>	Evaluates to true if both sides are <u>strictly equal</u> (e.g. <code>\$bullets === 6</code>).
<code>!==</code>	Evaluates to true if both sides are <u>strictly not equal</u> (e.g. <code>\$pie !== "cherry"</code>).
<code>></code>	Evaluates to true if the left side is greater than the right side (e.g. <code>\$cash > 5</code>).
<code>>=</code>	Evaluates to true if the left side is greater than or equal to the right side (e.g. <code>\$foundStars >= \$neededStars</code>).
<code><</code>	Evaluates to true if the left side is less than the right side (e.g. <code>\$shoeCount < (\$peopleCount * 2)</code>).
<code><=</code>	Evaluates to true if the left side is less than or equal to the right side (e.g. <code>\$level <= 30</code>).
<code>!</code>	Flips a true evaluation to false, and vice versa (e.g. <code>!\$hungry</code>).
<code>&&</code>	Evaluates to true if all subexpressions evaluate to true (e.g. <code>\$age >= 20 && \$age <= 30</code>).
<code> </code>	Evaluates to true if any subexpressions evaluate to true (e.g. <code>\$friend === "Sue" \$friend === "Dan"</code>).



Conditional Macros

(Opening/Closing Macro)

```
<<if conditional_expression>>, <<elseif conditional_expression>>, & <<else>>
```

Executes its contents if the given conditional expression evaluates to `true`. If the condition evaluates to `false` and an `<<elseif>>` or `<<else>>` exists, then other contents can be executed.

NOTE: SugarCube does not trim whitespace from the contents of `<<if>>` macros, so that authors don't have to resort to various kludges to get whitespace where they want it. However, this means that extra care must be taken when writing them to ensure that unwanted whitespace is not created within the final output.

Arguments:

- **conditional_expression:** A valid conditional expression, evaluating to either `true` or `false`. See [Expressions](#) for more information.



Conditional Macros

(Opening/Closing Macro)

Conditional Expression

Start of
Macro

{

```
<<if $score >= 50>>
```

```
    Hey, you passed.
```

End of
Macro

{

```
<</if>>
```

Note: assume we already have
a variable called \$score



Conditional Macros

(Opening/Closing Macro)

```
<<if $score >= 50>>
```

```
    Hey, you passed.
```

```
<<else>>
```

```
    Well, better luck next time.
```

```
<</if>>
```



Conditional Macros

(Opening/Closing Macro)

```
<<if $score >= 90>>
```

```
    Nice, an A.
```

```
<<elseif $score >= 50>>
```

```
    Hey, you passed.
```

```
<<else>>
```

```
    Well, better luck next time.
```

```
<</if>>
```

```
<!-- Checking a Boolean -->  
<<if $isRaining>>  
    Bring an umbrella.  
<</if>>
```

```
<!-- Comparing a String -->  
<<if $name == "Lassie">>  
    What, Timmy's fallen in the well?  
<</if>>
```

Cycling Link Macro

Custom Macros

- If you know JavaScript, you can write your own macro
- The <<cyclinglink>> macro allows you to create a link that cycles through a series of options when you click on it
- Installing the macro
 - Download the macro [here](#)
 - Open up cyclinglink-macro.min.js in VS Code
 - Copy and paste it into you “Story JavaScript” in Twine

Variable to Store
Current Link



```
<<cyclingle link "$pieType" "apple" "blueberry" "pumpkin">>
```



Values to Cycle Through