MAKING YOUR TWINE GAME MORE GAME-LIKE WITH PROGRAMMING

This handout explains how to use variables and do basic programming in Twine. All these instructions are based on the SugarCube story format. Before beginning, make sure that your Twine game is set up for the SugarCube format. To do so, click on the name of your story in its main "story map" view. Select "Change Story Format" and check the box next to "Sugarcube."

What is a variable?

A **variable** is container whose contents can be changed. (It gets its name from the fact that its contents are "variable.") Think of it as an envelope. You might put a piece of paper into the envelope that says "Adam." You might put a piece of paper into it that says 9. You might put an entire novel into it.

Variables have **names** and **values**. The **value** of a variable is the "content" described above — the word "Adam," or the number 9, or the entire novel. The **name** is just the shorthand that Twine will use to access whatever is in it. You have to decide on the variable's name, but you can call it whatever you want. The only rule is that Twine variables always need to start with a dollar sign (`$`). To bring a variable to life, use SugarCube's `<<set>>` command, which does two things: creates a variable with a certain name, and gives it its initial "value."

```
<<set $myvariable to "Adam">>
<<set $myothervariable to 99>>
<<set $yetanother to true>>
```

These are the three main types of variables might want to make: text (aka "strings"), numbers (aka "numeric" variables), and true/false (aka "booleans"). Notice that you need to put quotation marks around the contents of text variables. Don't use quotes for numbers or for true/false variables.

Application #1: Using a key in your game

Let's say that you're making a game about escaping from a castle. In order to be able to exit the castle, your player needs to find the key to the door. This key is hidden in some obscure passage of your game; you've made it deliberately hard to find.

You'll want a variable called something like `$hasKey` which will take one of two values: `false` when the player doesn't have it, and `true` when she finds it.

By default, the player doesn't have the key. So in the first passage of your game, you'll want to create your variable, and set its to `false`, like this:

```
<<set $hasKey to false>>
```

In the passage when your reader enters the hidden room and discovers the key, you'll want to have set the value of `$hasKey` to `true`. You can do that by putting the following line of code into that passage:

```
<<set $hasKey to true>>
```

(Note that `<<set>>` can *create* variables — but if the variable already exists, it just *modifies* the value of the variable.)

Okay, so now imagine we're in the passage of your game where the player has finally reached the main door of the castle and is trying to escape. If they have the key, they can escape. If they don't, they can't. To implement this, you could use SugarCube's `<<if>>` function, like this:

```
<<if $hasKey is true>>You insert your key into the door and it opens. You
[[walk through the door|outside]].
<<elseif $hasKey is false>>You try to open the door, but it's locked, and it
won't budge. You'll need to [[keep looking for the key|start]].
<</if>>
```

Twine begins by evaluating the first line of the `if` statement; if that's not true, it looks at the first `elseif` line, then looks at the rest of the `elseif` lines (if there are any more), and then stops working when it reaches `<</if>>` which means that the `if` statement is over. In addition to `if` and `elseif`, you can also write `else`, which just means "If none of the `if` or `elseif` conditions are met, then do *this*.) Also, note that if you're working with numbers, you can use conditional operators like `gt` ("greater than") and `lt` ("less than") instead of just `is`.

What this all means is the following. First, Twine will check whether the value of `$hasKey` is `true`. If it is, it will display the text "You insert your key into the door and it opens" and give the user the option to click on a link to a passage where they're outside the castle. If `$hasKey` isn't true, Twine *won't* display that text to the user, and the user *won't* be able to click on that link. Now, Twine will evaluate the next possibility. Here it will evaluate whether `$hasKey` is `false`. If it's `false` (which is it by default, since we set that as its initial value in the opening passage), then Twine will display the text "You try to open the door, but it's locked, and it won't budge," and the only option will be to click on a link that takes them back to the start of the game, to re-start their search for that obscure passage where `$hasKey` is set to true.

Application #2: Checking a player's name

Let's say you want to make a game in which you ask for a player's name, and then give some customized feedback if you recognize it.

On your first passage, use SugarCube's built-in code for displaying a text box and sticking whatever the player writes into a variable:

```
Enter your name:
<<textbox "$name" "">>
When you're ready, click [[here]].
```

This code specifies that whatever the player enters into the text box will be stored in a variable called `$name` (the `""` just means that there is no default text in the box — if you wrote `"Enter your name here"` in that space, the text box would initially show up with "Enter your name here" written in it.)

Now, on the next page, you could make it so that your game displays a special message if someone has entered their name as "Adam."

```
<<if $name is "Adam">>Hey, your name's Adam! So is mine!
<<else>>Hi, $name.
<</if>>
```

Twine will only display "Hey, your name's Adam! So is mine!" if the variable `$name` is equal to `"Adam"`. Otherwise — `<<else>>` — it just will say "Hi," and repeat the person's name back to her (yes, this is a cool thing about variables. If you write the variable name into a regular span of text, Twine will replace it with the value of the variable when it actually shows it to the player.)

Application #3: Recording a player's "happiness"

Let's say you want your game to record a running tally of how "happy" your player is. You could do this with a numeric variable. For instance, at the beginning of your game, you could include:

```
<<set $happiness to 0>>
```

Then, whenever something happens to make your player happy, you could include this line:

```
<<set $happiness to $happiness + 1>>
```

If the player's happiness level is zero going in to this passage, this line of code will set it to 1 (0 + 1 = 1). If the player's happiness level is 3 going in, this line of code will set it to 4 (3 + 1 = 4). The reason you don't want this line to be `<<set $happiness to 1>>` is that this would erase any "running tally" you've got going,

and just set the value of happiness to 1, regardless of how happy your player was before stumbling into this particular passage.

When something happens to make your player unhappy, you could include this line of code:

```
<<set $happiness to $happiness - 1>>
```

Later on, let's say that your player gets a phone call from a friend. If their happiness level is above a certain threshold, they decide to go out for ice cream. If their happiness level is below this threshold, they don't pick up the phone and stay inside. You might code this as follows:

```
<<if $happiness gte 5>>You pick up the phone and she asks you out for ice
cream and you [[totally go]].
<<else>>You don't feel like picking up. I guess you'll never know what she
wanted to ask you about. You [[stay at home]].
<</if>
```

Here, if your `$happiness` variable has a value of 5 or higher (`gte` is SugarCube-speak for "greater than or equal to"), your player will get the chance to navigate to the ice cream passage. Otherwise — `<<else>>` — their only option is to proceed to the "stay at home" passage.

These are just a few examples to get you started. If you get excited by this, you will want to visit this page for a full list of SugarCube 1.x macros: **http://www.motoslave.net/sugarcube/1/docs/macros.html**
If you *really* get into all this stuff, I would *highly* recommend that you then install SugarCube 2.0, which is a bit more complete and more fun. Instructions for downloading it, and full documentation, are at:
http://www.motoslave.net/sugarcube/2/