Instructions & User Guide

Hello, Creative Adventurer!

The **Dark** engine is a tool that you can use to create Android, iOS, and web app versions of your own interactive stories.

By **stories**, we mean that you will use images and/or text to depict characters or events that change as the story progresses.

By **interactive**, we mean that the story's progression will be influenced by the way that the person reading your story uses the app (i.e. where they move their finger or mouse pointer).

Check out the video above to see what these stories will look like. Better yet, click on the **Alice**, **Cow**, or **Mini** button in the navigation bar to the left to experience one of these stories for yourself.

Without further ado, here's how to use **Dark**:

1. Read through this entire document.
2. Download **Dark**, by clicking on the pink button to the left. Then, open up that folder. You will find five things inside:
   - A **stories** folder.
   - An **engine** folder.
   - A **Create** program.
   - A **New Story** program.
   - A **User Guide** file, which is identical to the document you are currently reading.
3. Double-click on the **New Story** icon to run the program. It will prompt you to type in the name that you would like to give your new story. It will then create a folder for your new story within the **stories** folder.
4. Look inside the folder for your new story and open the **code.txt** file in a plain text editor of your choice, such as **TextEdit**.
5. Be sure to add all of the image and audio files necessary for your story to your story's **img** and **music** folders.
6. When your story is ready, double-click on the **Create** icon, select the name of your story, and then select the type of app (i.e. Android, iOS, or web) that you would like to make. You will then find these newly-created apps in your story's **output** folder.

The rest of this document explains **Step #4** in greater detail.

Feel free to experiment and play with the three demo stories (**Alice**, **Cow**, and **Mini**), located both inside the **stories** folder you downloaded and also in the navigation bar to the left.

System Requirements

All you need is a **Mac** computer with **Python** installed. You can download Python here (https://www.python.org/downloads/).

Introduction

Each **code.txt** file needs exactly one `GRID` line, at least one `SCENE` line, and at least one `LOGIC` line. Don't worry about what `GRID`, `SCENE`, and `LOGIC` mean yet -- we'll explain them in a little while.

For now, all you need to know is that these three types of code lines are similar in that they are each composed of a label (e.g. `GRID`, `SCENE`, or `LOGIC`), followed by a series of components. Each component must contain a keyword and a value, separated by an equals sign. Multiple components of a `GRID`, `SCENE`, or `LOGIC` line should be separated by colons.

Here's the general format of a line of code:

```
LABEL : KEYWORD = VALUE : KEYWORD = VALUE
```

Now, here's an example of a line of code that uses a real label, keywords, and values:

```
GRID : COLUMNS = 4 : ROWS = 3
```

In the example above, the label is `GRID`, the 1st component sets the value of `COLUMNS` to `4`, and the 2nd component sets the value of `ROWS` to `3`. `COLUMNS` and `ROWS` are keywords specific to `GRID` lines. We'll explain more about `GRID` lines in the next section.

The number of spaces or tabs between words, colons, and equal signs never matters. It also doesn't matter what order you list the components, so long as the label of the line always comes first. So, an equally valid line of code is listed below:

```
GRID: ROWS= 3 : COLUMNS= 4
```

Finally, you also have the option of only using the first letter of keywords. So, you can rewrite the above example like this:

```
GRID : R = 3 : C = 4
```

You are encouraged to write your code in whichever format is easiest for you.

Coding GRID Lines

In order to place the text and images that make up your story in particular positions on the screen of your app, you will have to refer to locations on a grid. The `GRID` line code sets the total number of `ROWS` and `COLUMNS` that in the grid the the screen. For example, if your app contains 3 rows and 4 columns, you could place a character in the top-left corner by refering to the 1st row and 1st column. Alternatively, you could instead place the character in the bottom-right corner by refering to the 3rd row and 4th column.

Here's an example of a `GRID` line of code that establishes very few rows and columns:

```
GRID: C= 2 : R= 1
```

In contrast, here's a `GRID` line that uses a rather large grid:

```
GRID: C= 10 : R= 8
```

Keep in mind that the circular spotlight that follows the user's finger or mouse pointer when reading your story will be resized to match the size of a single grid box, so the more rows and columns you use, the smaller the individual grid boxes and the user's spotlight will be. You are encouraged to play through the **Mini** and **Cow** demos on the left side of this page, to experiment with the effects of changing the number of rows and columns. The **Cow** demo uses 3 rows and 4 columns while the **Mini** demo uses only 1 row and 2 columns. Consequently, the spotlight in the **Mini** demo is much larger.

In order to help you visualize the organization of your app's grid and how to refer to locations inside it, below are two 'maps' of differently sized grids.

Below is a map of how a grid with 1 row and 2 columns would be organized:

```
(1, 1) (2, 1)
```

In contrast, here is a map of a grid with 3 rows and 4 columns:

```
(1, 1) (2, 1) (3, 1) (4, 1)
(1, 2) (2, 2) (3, 2) (4, 2)
(1, 3) (2, 3) (3, 3) (4, 3)
```

Dark uses the convention of always listing the number of columns before the number of rows in parentheses. For example, `(1, 2)` indicates the 1st column and 2nd row while `(2, 1)` refers to the 2nd column and 1st row.

Coding SCENE Lines

Now, it is time for you to code the first scene of your story.

Each scene needs to start with a `SCENE` line, which must have at least two components: a `NAME` and a `BACKGROUND`. You also have the option of adding `MUSIC`.

The `NAME` of your scene can be anything you want, but the `BACKGROUND` must be identical to that of an image file that is saved in your **img** folder.

Below is an example of a `SCENE` line without `MUSIC`:

```
SCENE: NAME = My First Scene : BACKGROUND = myBackground.jpg
```

Now, here is an example of the same `SCENE` line with `MUSIC`:

```
SCENE: NAME = My First Scene : BACKGROUND = myBackground.jpg : MUSIC = myMusic.mp3
```

Coding LOGIC Lines

Finally, you should add some interactivity to your scene. To do this, you must add a few `LOGIC` lines. Each `LOGIC` line typically uses three components: a `NAME`, a `CAUSE`, and an `EFFECT`; however, there are some special cases, which we will discuss shortly, in which you might want to use alternative components.

`LOGIC` lines can either be `ACTIVE` or `INACTIVE`. `INACTIVE` logics do nothing whereas `ACTIVE` ones usually display images on the screen that describe a specific event in your story. These images are chosen by using the `CAUSE` component. You can also use `LOGIC` lines to activate the next event or `LOGIC` in your story, by using the `EFFECT` component.

For example, below is an initially `ACTIVE` `LOGIC` line, called `Question`, that displays `howAreYou.png` on the screen. Once the user moves their finger or mouse pointer over the image, they will activate another `LOGIC` line, called `Answer`. Don't worry about the details of this line of code yet, since we will soon explain each component of it in greater detail.

```
LOGIC: NAME = Question->ACTIVE : CAUSE = howAreYou.png->(1, 1) : EFFECT = Answer->ACTIVE
```

Now, here is another `LOGIC`, called `Answer`, which does nothing yet because it is initially `INACTIVE`; however, once another `LOGIC` activates it, such as the one called `Question` above, then it will display `fantastic.png` on the screen. Once the user views this image, the `LOGIC` called `Next Question` will be activated.

```
LOGIC: NAME = Answer->INACTIVE : CAUSE = fantastic.png->(2, 1) : EFFECT = Next Question->ACTIVE
```

In this way, `LOGIC` lines are used to control the chain of events that occur in your story.

NAME

Just like with `SCENE` lines, the `NAME` of your `LOGIC` line can be anything you want; however, no two lines can have the same name. To set the initial activation state, you should type an arrow (`->`) next to the name that points to either `ACTIVE` or `INACTIVE`.

Below is an `ACTIVE` `LOGIC`:

```
LOGIC: NAME = First Logic->ACTIVE
```

In contrast, here's an `INACTIVE` `LOGIC`:

```
LOGIC: NAME = Second Logic->INACTIVE
```

You should always have at least one `LOGIC` that is initially `ACTIVE`. Otherwise, nothing will ever appear in your story other than the background image.

CAUSE

The `CAUSE` of your `LOGIC` is an image that will appear on the screen only when the `LOGIC` is `ACTIVE`. To set the location of the image on the screen, you must type an arrow (`->`) pointing the column and row number where you would like the image to appear. The coordinates should be enclosed in parentheses and separated by a comma, just like the notation used in the maps shown earlier. **Remember:** the number of columns always comes before the number of rows.

The following `LOGIC` shows `howAreYou.png` in the 1st row and 2nd column of the screen:

```
LOGIC: NAME = Question->ACTIVE : CAUSE = howAreYou.png->(2, 1)
```

In contrast, the example below shows the same image in the 2nd row and 1st column:

```
LOGIC: NAME = Question->ACTIVE : CAUSE = howAreYou.png->(1, 2)
```

You may absolutely make more than one image appear at once, but you must always have at least one spot on the screen empty at all times. So, if you are using a tiny grid with only 2 boxes (as is the case in the **Mini** demo), you can only use one image in the `CAUSE` componenent of the `LOGIC` line, since there are only 2 grid boxes total.

Below is a `LOGIC` line that displays two images in two different places:

```
LOGIC: NAME = Busy Logic->ACTIVE : CAUSE = image1.png->(1, 2), image2.png->(2, 2)
```

EFFECT

Each `LOGIC` line will most likely also have an `EFFECT` triggered by the user's spotlight landing on all of the images listed in the `CAUSE` component. This `EFFECT` would be to either activate or deactivate another `LOGIC`. To code this you should type out the name of the other `LOGIC` that you hope to affect, followed by an arrow to either `ACTIVE` or `INACTIVE`.

The `LOGIC` line below, called `First Event`, activates `Second Event` only if `myImage.png` is viewed:

```
LOGIC: NAME = First Event->ACTIVE : CAUSE = myImage.png->(1,1) : EFFECT = Second Event->ACTIVE
```

Once `Second Event` is activated by viewing `myImage.png`, `First Event` will automatically get deactivated. If you want `First Event` to stay `ACTIVE` even after it activates `Second Event`, you have to code it like this, instead:

```
LOGIC: NAME = First Event->A : CAUSE = myImage.png->(1,1) : EFFECT = Second Event->A, First Event->A
```

As exemplified above, any `LOGIC` can activate or deactivate as many other `LOGIC` lines as you want. Simply add them to the list. As you can see, single-letter shortcuts for `ACTIVE` were used in that example too. This is just a matter of personal preference.

Finally, here's an example of a `LOGIC` that deactivates another `LOGIC`:

```
LOGIC: NAME = First Event->ACTIVE : CAUSE = myImage.png->(1,1) : EFFECT = Second Event->INACTIVE
```

ORDERED

If you have more than one image listed in the `CAUSE` component of your `LOGIC` line, you can specify the order in which they must be viewed in order to trigger the associated `EFFECT`. To do this, simply add `ORDERED` as a component in your `LOGIC` line. This will make it so that the images must be viewed in the order in which they are listed.

For example, a normal, unordered `LOGIC` line looks like this:

```
LOGIC: NAME = First Event->A : CAUSE = image1.png->(1,1), image2.png->(2,1) : EFFECT = Second Event->A
```

In the line above, as long as both images are viewed (in any order), viewing them will activate `Second Event`.

In contrast, the line below requires that `image1.png` be viewed before `image2.png`, in order for `Second Event` to activate. Otherwise, nothing will happen, and `First Event` will remain `ACTIVE` while `Second Event` remains `INACTIVE`.

```
LOGIC: N = First Event->A : C = image1.png->(1,1), image2.png->(2,1) : E = Second Event->A : ORDERED
```

DEPENDENT

If you are trying to create a more complicated story or game that would require multiple `LOGIC` lines to be `ACTIVE` simultaneously before their `EFFECT` occurs, then you can use the `DEPENDENT` component. Simply list the names of the other `LOGIC` lines that you want the current `LOGIC` to be `DEPENDENT` on.

For example, in the following `LOGIC` line, `First Event` is `ACTIVE`, so `myImage.png` appears on the screen; however, viewing `myImage.png` only activates `Forth Event` if `Second Event` and `Third Event` (defined in separate `LOGIC` lines) are simultaneously `ACTIVE`:

```
LOGIC: N = First Event->A : C = myImage.png->(1,1) : E = Forth Event->A : DEPENDENT = Second Event, Third Event
```

MUSIC

If you would like viewing the `LOGIC` images to change the music in the scene, then you should use the `MUSIC` component. All you have to do is type out the name of the audio file that you want to start playing, like this:

```
LOGIC: NAME = Change Music->A : CAUSE = myImage.png->(1,1) : MUSIC = newMusic.mp3
```

In the example above, viewing `myImage.png` will change the music to newMusic.mp3; however, it is also possible to both change the music and activate another `LOGIC`, like this:

```
LOGIC: NAME = First Event->A : CAUSE = myImage.png->(1,1) : EFFECT = Second Event->A : MUSIC = newMusic.mp3
```

SCENE & TRANSITION

Finally, at the end of your scene, you should write a `LOGIC` line that will change the current scene to the next one, even if the current scene is the last or the only scene of your story. This `LOGIC` line must contain a `SCENE` component and a `TRANSITION` component.

The `SCENE` should simply be a name that matches one that you will use in a `SCENE` line somewhere else in your **code.txt** file, like this:

```
LOGIC: N = End of Scene->A : C = myImage.png->(1,1) : SCENE = Second Scene : TRANSITION = transition.jpg->DOWN
```

The example above starts `Second Scene` after `myImage.png` is viewed. It assumes that somewhere in your **code.txt** file you have also written a line of code that looks like this:

```
SCENE: NAME = Second Scene : BACKGROUND = myBackground.jpg
```

If you want the current scene, called `First Scene`, to repeat itself rather than changing to a different scene, simply set the value of `SCENE` to the name of the current scene, like this:

```
LOGIC: N = End of Scene->A : C = myImage.png->(1,1) : S = First Scene : T = transition.jpg->DOWN
```

The `TRANSITION` should equal the name of an image in your **img** folder, which you want to appear in between showing the background of this scene and the background of the next scene. This `TRANSITION` should also have an arrow pointing in the direction that the scene transition will occur in. Your choices are `UP`, `DOWN`, `LEFT`, and `RIGHT`.

For example, the `LOGIC` coded above transitions downwards because it uses an arrow that points to `DOWN`. This means that `transition.jpg` enters from the top of the screen and travels `DOWN` the screen until it disappears off the bottom of the screen.

Alternatively, the example below will show `transition.jpg` travelling from the left side of the screen to the right side of the screen. In other words, the arrow always points to when you want the final destination of the image to be -- in this case, the far `RIGHT` of the screen.

```
LOGIC: N = End of Scene->A : C = myImage.png->(1,1) : S = First Scene : T = transition.jpg->RIGHT
```

Putting It All Together

Now, let's take a look at how everything ties together by coding an entire (tiny) story, line by line. Below you'll find the code that was used to generate the **Mini** demo, which you should quickly play through right now, by clicking on the pink button to the left.

```
(1) GRID:  COLUMNS = 2 : ROWS = 1

(2) SCENE: NAME = Mini Demo : BACKGROUND = miniBackground.jpg : MUSIC = miniMusic.mp3

(3) LOGIC: N = First Event->A  : C = Hello.png->(1,1)     : E = Second Event->A
(4) LOGIC: N = Second Event->I : C = Hey.png->(2,1)       : E = Third Event->A
(5) LOGIC: N = Third Event->I  : C = HowAreYou.png->(1,1) : E = Forth Event->A
(6) LOGIC: N = Forth Event->I  : C = Fantastic.png->(2,1) : E = Fifth Event->A
(7) LOGIC: N = Fifth Event->I  : C = MeToo.png->(1,1)     : E = Last Event->A
(8) LOGIC: N = Last Event->I   : C = Yay.png->(2,1)       : S = Mini Demo : T= miniTransition.jpg->DOWN
```

1. This `GRID` line establishes the size of the grid that the story will use: `2` columns and `1` row. This means that the screen will essentially be split in half horizontally, and the user's spotlight will be so large that it will span half of the screen.
2. This `SCENE` line first names the scene `Mini Demo`, loads the background image called `miniBackground.jpg`, and then starts playing the music file called `miniMusic.mp3`.
3. This `LOGIC` line defines `First Event`, which starts out as `ACTIVE`. This means that at the beginning of the scene, its image `Hello.png` will already be visible. The `(1,1)` indicates that the image will show up on the left half of the screen (i.e. the 1st column and 1st row). Finally, we also know that viewing this image will have the effect of activating `Second Event`.
4. Lines 4 through 7 are quite similar to the `LOGIC` in line #3, except that they start out as `INACTIVE`, which means that their images will not show up until something activates them. Specifically, line #4 shows that once `Second Event` is activated, it will present the image `Hey.png` in location `(2,1)` - that is, on the right half of the screen. Once clicked, this image image will activate `Third Event`.
5. The next `LOGIC` line behaves the same way, except that it displays `HowAreYou.png` on the left half of the screen.

8. Lastly, line #8 will show `Yay.png` on the right half of the screen. When the image is viewed, the scene will repeat because `Mini Demo` is chosen as the next scene. But, before the scene resets, the image `miniTransition.jpg` will slide `DOWN` the screen.

Phew, You Made It!

Thank you for reading through this rather lengthy set of instructions.

You are now ready to begin creating your own adventures inside of your own **code.txt** file.

I hope that this document has been helpful. If you are confused about anything, or if you have any questions at all, please do not hesitate to email **Erica Silverstein** at **e.o.silverstein@gmail.com**.