NARRATIVE CREATION GUIDE

Narrative technical structure

Every block of text (block of text = a series of lines that are always said by the same person and in the same order, one right after another) is represented by a Dialogue Message object. Each Dialogue Message links to one or more Dialogue Messages that may come after it.

DIALOGUE MESSAGES

Dialogue Message object creation

Dialogue Messages are a type of scriptable object, which means that they exist as files (DialogueMessageName.asset) within the Dialogue folder in the Assets folder of the Unity project. Here is how they are created:

1. Right click in the project hierarchy in Unity or click on the Assets menu tab at the top left of the Unity editor
2. Go to Create -> Dialogue Message
3. Put the newly created Dialogue Message in the correct folder according to the folder naming convention. For conversation pieces:

   Assets → Dialogue → Floor Number → Character(s) → Name of Conversation

   Where “characters” is the names of the characters involved in the conversation (apart from Constance, since she’s implied). For flavor text, there are two categories: simple flavor text and complex flavor text. Simple flavor text is flavor text that never changes and has no interactive components, so every time the object is inspected, the same response is generated. Complex flavor text describes every object that may produce more than one response when inspected or has interactive components. Here is the folder naming convention for simple flavor text:

   Assets → Flavor Text → Floor Number

   For complex flavor text, here is the folder naming convention:

   Assets → Flavor Text → Floor Number → Object

Where “object” refers to what is inspected to generate the flavor text.

4. Name the Dialogue Message according to the correct Dialogue Message naming convention. For conversations:

   Number - Speaker - Summary

   Where number is the message’s sequence in the conversation. If multiple messages are possible, append them with a letter (2a, 2b, 2c, etc.). Try to order them in corresponding ways when possible (e.g. 2a results in 3a). It’s important to note that during conversations, if any text needs to appear that isn’t dialogue, Speaker should be set to “Flavor.” Summary should summarize what is said in the Dialogue Message in as few words as possible (no more than three). For simple flavor text:

   Object

   Where the object is what was inspected to produce the flavor text. For complex flavor text:

   Number - Entity - Summary

   Where number is the message’s sequence in the conversation. If multiple messages are possible, append them with a letter (2a, 2b, 2c, etc.). Try to order them in corresponding ways when possible (e.g. 2a results in 3a). Entity refers to the object if the dialogue is flavor text or a character if it’s a character-induced action or dialogue message. Summary should summarize what is said in the Dialogue Message in as few words as possible (no more than three).

Example: #1a Constance and Leah are having a conversation on the 8th floor about what kind of olive is best. Leah starts the conversation by rambling about how ‘in’ olives are right now and then asking Constance if green olives or black olives are better. The player can choose either, but Leah will be disgusted if the player says green olives are better, and will vomit on the spot after berating Constance about that choice. Create Dialogue Message objects with the appropriate names and folder.

First we need to make a folder for these messages to go in.
Floor number = 8
Character(s) involved = Leah (Constance is a given, since she is the player character)

Name of Conversation = Olive talk
Assets -> Dialogue -> 8 -> Leah -> Olive talk

Now we need to make the message objects themselves. There are six Dialogue Messages here, which will be explained in detail.

1. Leah rambles about how 'in' olives are right now and then asks Constance if green olives or black olives are better.
   *Note that even though Leah says a lot here, it can be consolidated as one dialogue message, since Leah always says the same things in the same order with no insertions or deletions. They can be expressed as multiple Dialogue Messages, but it's less messy to keep them as one Dialogue Message.
   Number = 1
   Speaker = Leah
   Summary = Loves Olives (this can be anything, just try to communicate what is said)
   1 - Leah - Loves Olives
2. Constance says she prefers green olives.
   Number = 2a
   Speaker = Constance
   Summary = Green Better
   2a - Constance - Green Better
3. Constance says she prefers black olives.
   Number = 2b
   Speaker = Constance
   Summary = Black Better
   2b - Constance - Black Better
4. Leah reacts with disgust to Constance preferring green olives.
   Number = 3a
   Speaker = Leah
   Summary = Ew Green
   3a - Leah - Ew Green
5. Leah reacts well to Constance preferring black olives.
   Number = 3b
   Speaker = Leah
   Summary = Yay Black
   3b - Leah - Yay Black
6. Leah vomits on the spot after berating Constance about choosing green olives.
   Number = 4a *note that while there is no other possible 4th message in this conversation, it's good practice to say 4a instead of 4 here, since 4 implies this is

always the fourth message of every possible conversation, whereas 4a implies this is the 4th message only in the path of messages with number #a.
Speaker = Flavor *since this is a narration and not Leah saying, “Leah vomits at your choice!” This counts as flavor text.
Summary = Leah Vomits
4a - Flavor - Leah Vomits

Example: #1b On the 13th floor, there is a green tapestry that can be inspected (there’s also a red tapestry on this floor). The same result is always produced when inspected. Create Dialogue Message objects for the green tapestry with the appropriate names and folder.

Since the inspection of the tapestry never changes, this counts as simple flavor text.
Floor number = 13
Assets → Flavor Text → 13

Now we name the Dialogue Message. Note that it’s referred to as “Green Tapestry” as opposed to “Tapestry 1” or something, since this communicates which tapestry the Dialogue Message is referring to most clearly.
Object = Green Tapestry
Green Tapestry

Example: #1c On the 1st floor, there is a hyper realistic statue of an old friend of Constance’s who she had a falling out with. Players have the option to either tell it to go away or give it a hug, and each option produces an additional piece of flavor text. If you tell it to go away, a message says, “You feel a bit better. The statues ignore you. You stop feeling better.” If you give it a hug, a message says, “You hold the statue and weep softly. It’s oddly therapeutic.” Create the appropriate Dialogue Message objects with the appropriate names and folder.

The statue is interactable, so this counts as complex flavor text.
Floor number = 1
Object = Statue
Assets → Flavor Text → 1 → Statue

There are five dialogue messages to create here.

1. Initial description of the statue as a hyper realistic statue of an old friend of Constance’s who she had a falling out with.
   Number = 1

Entity = Statue
Summary = Resembles Ex-Friend
1 - Statue - Resembles Ex-Friend

2. Constance tells the statue to go away.
   Number = 2a
   Entity = Constance
   Summary = Insult
   2a - Constance - Insult
3. Constance hugs the statue.
   Number = 2b
   Entity = Statue
   Summary = Hug
   2b - Constance - Hug
4. If you tell it to go away, a message says, “You feel a bit better. The statues ignore you. You stop feeling better.”
   Number = 3a
   Entity = Statue *note that this is the statue, not Constance, because this is driven by the statue’s lack of a response. This distinction gets a bit muddy, but the way I like to think of it is to imagine this as a game of D&D, and if the message is said by a player, then entity = character, but if it’s more like something the GM would say, it’s entity = character. Alternatively, you can look at 2a as Constance’s turn to say something, so she’s the entity, and this is the statue’s turn to do something, so the statue is the entity.
   Summary = Insulted
   3a - Statue - Insulted
5. If you give it a hug, a message says, “You hold the statue and weep softly. It’s oddly therapeutic.”
   Number = 2b
   Entity = Statue
   Summary = Hugged
   3b - Statue - Hugged

Dialogue Message structure

A Dialogue Message is composed of six parts:

- DialogueMessage.text is an array of strings that refers to every line of dialogue that is displayed in the Dialogue Message. Every string in the array is displayed separately, appearing one after another in the text box on screen.
    - Important note: If a DialogueMessage is the direct result of a dialogue choice, text[0] will be the text that appears on the button that must be

pressed to access this dialogue choice. No effects should be associated with button text.

- DialogueMessage.speaker is a string which refers to who is talking, and will be reflected on screen. If nobody is talking (flavor text), leave this blank or give a name that contains “Flavor” with a capital F. Also, if a message is a result of a narrative choice that does not involve dialogue (i.e. choose to fight or run away), the speaker should be “Constance Flavor.”
- DialogueMessage.next is an array of Dialogue Messages listing every line of dialogue that may come next.
    - If the length of the array is zero, the conversation ends after this Dialogue Message.
    - If the length is one, this shows a linear conversation, and the Next[0] will be loaded once the current DialogueMessage is finished.
    - If the length is greater than one, this shows some nonlinearity. The way this works changes based on whether the next speaker is Constance (a dialogue choice) or not (nonlinear based on what’s happened so far)
        - If the next speaker is Constance, all viable dialogue messages will appear as buttons to choose the next thing the player may say. Make sure at least one is always possible!
        - If the next speaker is not Constance, it will run through every Dialogue Message in the list, and the next one will be the first one in the list that is possible
- DialogueMessage.reqs is an array of strings listing any requirements that must be met for any Dialogue Message in DialogueMessage.next to happen. The order of the items in the arrays corresponds to each other (e.g. the second item in DialogueMessage.reqs details what must be true for the second item in DialogueMessage.next to happen). The naming convention of these is detailed later in this document.
- DialogueMessage.effects is an array of strings listing any effects that may happen when different lines in DialogueMessage.text are first displayed. The order of the strings in the arrays correspond to each other (e.g. when the second string in DialogueMessage.text is displayed, the second item of DialogueMessage.effects happens). The naming convention of these is detailed later in this document.
- DialogueMessage.effectReqs is an array of strings listing what requirements must be met for the effects in DialogueMessage.effects to take place. The order of the strings in the arrays correspond to each other (e.g. the second string in DialogueMessage.effectReqs must be met in order for the second item in

DialogueMessage.effects to take place). The naming convention of these is detailed later in this document.

REQUIREMENTS

Requirement checks

Requirements are fed into another script called Information.cs. Information deals with every piece of knowledge characters have that are relevant to the world and subject to change as well as the state of the game world. It's essentially a giant dictionary that takes in a string (hasFinishedFloor6, graceBecameSelfAware, bankBalance, etc.) and returns an int representing its current value. An object with an Information script is automatically generated if none already exists, so there's no need to worry about whether or not one exists in any given scene.

Any given requirement, whether in DialogueObject.reqs or DialogueMessage.effectReqs, checks a given requirement against a value, and returns either true (requirement is met) or false (requirement is not met).

Requirement structure

Requirements in their simplest form are structured as one string with no spaces as:

[name of requirement][conditional][value]

Where [name of requirement] is the case-sensitive string identifying the information that is being checked, as detailed in the Information object (e.g. ancientKeyCollected). The conditional shows how the values are being compared. Here's a table showing how they're written:

Conditional	Meaning
req>n	Is the value of req greater than n?
req>=n	Is the value of req greater than or equal to n?
req=n	Is the value of req equal to n?
req!=n	Is the value of req not equal to n?
req<=n	Is the value of req less than or equal to n?

req<n	Is the value of req less than n?

Conditionals can also be checked against each other in more complex ways as shown:

Conditional	Meaning
!conditional	Is the conditional not met?
conditional1&conditional2	Are both conditional1 and conditional2 met?
conditional1|conditional2	Are either conditional1 or conditional2 met? (Note that red mark is a vertical bar)
(conditional1&conditional2)|conditional3	Are either both conditional 1 and conditional2 or conditional3 met?
conditional1&(conditional2|conditional3)	Are both conditional1 and either conditional2 or conditional3 met?

Note that NO combination of requirements should have spaces.

EFFECTS

Effect types

Effects come in three types, referred to as effectTypes, referencing values and functions in Information.cs.

- Set: Set the value of one piece of information to a specific value
- Adjust: Increase or decrease the value of one piece of information by a set amount
- Trigger: Run a function in Information.cs (feel free to add functions to that script when necessary). Note that these functions cannot take in any arguments. If you want to add a function to Information.cs that would make sense to have arguments, add the arguments as keys in the dictionary Information.info, and then call those values in the function.

Effect structures

Effect	Type	Meaning
itemsCollected=6	Set	Set itemsCollected to 6
itemsCollected+1	Adjust	Add 1 to itemsCollected
itemsCollected-1	Adjust	Subtract 1 from itemsCollected
CollectItem	Trigger	Run Information.CollectItem()
Effect1,Effect2 or Effect1&Effect2	Multiple	Run both Effect1 and Effect2

Example: #2 Create the full Dialogue Messages for each message in Example 1a in the conversation between Leah and Constance about olives.

1. 1 - Leah - Loves Olives
   1 - Leah - Loves Olives.text =
   [“Connie! I just found a jar of olives. Can you believe it?”,
   “I love olives SO much. They’re so round and hollow and TRENDY.”,
   “What kind of olives do you like most?”]
   1 - Leah - Loves Olives.speaker = “Leah”
   1 - Leah - Loves Olives.next = [2a - Constance - Green Better, 2b - Constance - Black Better]
   1 - Leah - Loves Olives.reqs = []
   1 - Leah - Loves Olives.effects = []
   1 - Leah - Loves Olives.effectReqs = []
2. 2a - Constance - Green Better
   2a - Constance - Green Better.text =
   [“Say green is better”, *remember, since this is a choice, text[0] will be text on a button
   “I like green olives.”]
   2a - Constance - Green Better.speaker = “Constance”
   2a - Constance - Green Better.next = [3a - Leah - Ew Green]
   2a - Constance - Green Better.reqs = []
   2a - Constance - Green Better.effects = [“”, “leahLikesConstance-1”] *note that this effect is the second item in the array, since it happens when Constance says green is better, not when the “Say green is better” button appears.
   2a - Constance - Green Better.effectReqs = []

3. 2b - Constance - Black Better
   2b - Constance - Black Better.text =
   [“Say black is better”,
   “I like black olives.”]
   2b - Constance - Black Better.speaker = “Constance”
   2b - Constance - Black Better.next = [3b - Leah - Yay Black]
   2b - Constance - Black Better.reqs = []
   2b - Constance - Black Better.effects = [“”, “leahLikesConstance+1”]
   2b - Constance - Black Better.effectReqs = []
4. 3a - Leah - Ew Green
   3a - Leah - Ew Green.text = [“Holy mother of hoes that’s disgusting!”]
   3a - Leah - Ew Green.speaker = “Leah”
   3a - Leah - Ew Green.next = [4a - Flavor - Leah Vomits]
   3a - Leah - Ew Green.reqs = []
   3a - Leah - Ew Green.effects = []
   3a - Leah - Ew Green.effectReqs = []
5. 3b - Leah - Yay Black
   3b - Leah - Yay Black.text =
   [“Oh my god same!”,
   “People who prefer green olives deserve to die.”]
   3b - Leah - Yay Black.speaker = “Leah”
   3b - Leah - Yay Black.next = []
   3b - Leah - Yay Black.reqs = []
   3b - Leah - Yay Black.effects = []
   3b - Leah - Yay Black.effectReqs = []
6. 4a - Flavor - Leah Vomits
   4a - Flavor - Leah Vomits.text = [“Leah vomits at your choice!”]
   4a - Flavor - Leah Vomits.speaker = “”
   4a - Flavor - Leah Vomits.next = []
   4a - Flavor - Leah Vomits.reqs = []
   4a - Flavor - Leah Vomits.effects = [LeahVomits] *this would only be necessary if this is reflected in the physical world (i.e. Leah does an animation, a collectible puddle of vomit appears, etc.)
   4a - Flavor - Leah Vomits.effectReqs = []

DYNAMIC TEXT

If you want to have text appear slightly differently based on certain actions the player has taken, especially if there are potentially infinite ways a piece of text may be presented (e.g. a character referencing exactly how many times a player performed a specific action, referencing a player-generated name, etc.), it's a good idea to use dynamic text. Dynamic text is essentially a fill-in-the-blank style of text that takes in a key and returns the value stored in the Information class.

Dynamic text can be used anywhere in a dialogue interface (main text, button text, or speaker text). It is represented by surrounding the key to be retrieved with vertical bars.

Example: #3 Fiona adopted a pet werewolf that she has been using in battle. Create a dialogue message where she states how many enemies her pet werewolf has defeated and mentions the werewolf by the name the player chose.

We need to store both Fiona's pet werewolf's name and the number of enemies the werewolf defeated in the Information class, since those are decisions/outcomes that must be remembered. Thus, we need to make a key for each by which we can retrieve these data. Let's call the key for the werewolf's name FionaPetWerewolfName, and the key for the number of enemies that werewolf defeated FionaPetWerewolfDefeats.

For a dialogue message called 1 - Fiona - Pet Defeats:

1 - Fiona - Pet Defeats.text = ["|FionaPetWerewolfName| has defeated |FionaPetWerewolfDefeats| enemies! He's so talented!"]
1 - Fiona - Pet Defeats.speaker = "Fiona"

For example, if the werewolf's name was Corn Cob, and Corn Cob had defeated 17 enemies, this text would appear in game as:

> Corn Cob has defeated 17 enemies! He's so talented!

DOCUMENTATION

Effect and Requirement Documentation

All set/adjust effects and requirements absolutely must be documented in this spreadsheet. This is vital to making sure we're consistent throughout the production of the game, we're not redundant with the information that exists, and no narrative paths are impossible to reach.

Here is how the documentation works (see examples in the spreadsheet):

- Name: the name of the piece of information, exactly as shown in the code (case sensitive!)
- Description: A representation of the question that is essentially asked whenever a requirement concerning that piece of information is called
- Target: The character, object, etc. that this piece of information concerns. May be multiple
- Floors: A list of all floors where this piece of information either changes or is checked
- Start Val: The value of this piece of information at the start of a new game
- Value Retrieved: A list of all the times this piece of information is checked, and for what values. It should follow this format:

  - File path → Dialogue Message (condition checked)

  Where “File path” shows the location of the Dialogue Message where the value is checked. The naming convention for condition checked is the same as for requirements without the requirement itself being explicitly mentioned (see examples in spreadsheet)
- Value Changed: A list of all the times this piece of information is changed, and to what values. It should follow this format:

  - File path → Dialogue Message (adjustment)

  The convention for the adjustment should follow the same convention as the effect structures, but without the piece of information itself being explicitly mentioned (see examples in spreadsheet).

FAQs

What if I want the first text to be nonlinear in some way?

In this situation, create a Dialogue Message called “0 - Empty” and start the conversation with that. Make sure 0 - Empty.text is an empty array, and add the possible first lines of dialogue/flavor text into 0 - Empty.next as you normally would.

What if I want to use randomness in some way?

To randomize effects, you'll need to generate a random number that determines the outcome. Run the effect:

"maxRandomValue=n&Randomize"

This generates a random integer between 0 (inclusive) and n (exclusive), and stores that value in information as "randomSeed".

Example: #4 There is a big pool of slime in the middle of the hallway on the 5th floor, which Constance may choose to cross. The only way to cross it is by jumping over it, but there's only a 65% chance that she'll succeed. Write out the Dialogue Messages for this interaction.

This counts as complex flavor text, since Constance doesn't talk to anyone, and the interaction is interactable.
Floor number = 5
Object = Slimepool
Assets → Flavor Text → 5 → Slimepool

Name the dialogue messages

1. First description of the pool and its purpose
   Number = 1
   Entity = Slimepool
   Summary = Blocking Way
   1 - Slimepool - Blocking Way
2. Deciding not to jump over the slimepool
   Number = 2a
   Entity = Constance
   Summary = No Jump
   2a - Constance - No Jump
3. Deciding to jump over the slimepool
   Number = 2b
   Entity = Constance
   Summary = Jump
   2b - Constance - Jump
4. Successfully jumping over the slimepool
   Number = 3a
   Entity = Constance

Summary = Good Jump
3a - Constance - Good jump

5. Unsuccessfully jump over the slimepool
   Number = 3b
   Entity = Constance
   Summary = Bad Jump
   3b - Constance - Bad Jump

Create the dialogue messages

1. 1 - Slimepool - Blocking Way
   1 - Slimepool - Blocking Way.text =
   [“A large pool of slime blocks your path.”,
   “It doesn’t look dangerous, but it does look very gross.”,
   “What are you going to do about it?”]
   1 - Slimepool - Blocking Way.speaker = “”
   1 - Slimepool - Blocking Way.next = [2a - Constance - No Jump, 2b - Constance - Jump]
   1 - Slimepool - Blocking Way.reqs = []
   1 - Slimepool - Blocking Way.effects = []
   1 - Slimepool - Blocking Way.effectReqs = []
2. 2a - Constance - No Jump
   2a - Constance - No Jump.text =
   [“Absolutely nothing”,
   “You just stand around and stare at the slime.”]
   2a - Constance - No Jump.speaker = “Constance Flavor”
   2a - Constance - No Jump.next = []
   2a - Constance - No Jump.reqs = []
   2a - Constance - No Jump.effects = []
   2a - Constance - No Jump.effectReqs = []
3. 2b - Constance - Jump
   2b - Constance - Jump.text =
   [“Jump over it”,
   “You square up, and do a running jump over the slime…”]
   2b - Constance - Jump.speaker = “Constance Flavor”
   2b - Constance - Jump.next = [3a - Constance - Good Jump, 3b - Constance - Bad Jump]
   2b - Constance - Jump.reqs = [randomSeed<65, randomSeed>=65]
   2b - Constance - Jump.effects = [“”,“maxRandomValue=100&Randomize”]
   2b - Constance - Jump.effectReqs = []

4. 3a - Constance - Good Jump
   3a - Constance - Good Jump.text = [“...and clear it successfully!”]
   3a - Constance - Good Jump.speaker = “”
   3a - Constance - Good Jump.next = []
   3a - Constance - Good Jump.reqs = []
   3a - Constance - Good Jump.effects = []
   3a - Constance - Good Jump.effectReqs = []
5. 3a - Constance - Bad Jump
   3a - Constance - Bad Jump.text = [“you plummet right into the slime.”]
   3a - Constance - Bad Jump.speaker = “”
   3a - Constance - Bad Jump.next = []
   3a - Constance - Bad Jump.reqs = []
   3a - Constance - Bad Jump.effects = [constanceIsSlimy=1]
   3a - Constance - Bad Jump.effectReqs = []

Is there a good way for me to structure dialogue before putting it in the game that’s easy to read?
Yeah I’d highly recommend mapping everything out in Twine before putting it in the game. You can download Twine here.

Color Coding in this Document

I’m self aware enough to know this might come up.

Color	Instructional Text	Examples
Black	Main text	Given information
Blue	Headings	Main work
Red	Vocab words and important side notes	Tertiary work, example number, and important side notes
Green	BIG headings	Secondary work
Purple	Formulas	Final answers