Contents

DialogueQuest User Manual

DialogueQuest for non-coders

DialogueQuest features a standalone program called DialogueQuestTester that allows running dialogues without a Godot environment.

Writing Dialogue - DQD

Basics

DQD stands for `DialogueQuest Dialogue` and is the dialogue format of DialogueQuest.

The DQD format uses the `.dqd` file extension.

DQD is a simple text-based format, that goes something like this:

```
statement | param1 | param2 | ...
```

Every line starts with a statement which 'moves forward' in the line like a pipeline.

The most basic and most used statement is the say statment, which looks like this:

```
say | joe | Hello DialogueQuest
say | You don't even need a character
```

Comments

DQD Support comments.

A line that starts with `//` is considered a comment, and will not be parsed/executed.

Comments are useful for explaining things like branches, flags, or even leaving a comment for your team on their good work :)

Comments can also be used to temporarily disable parts of the dialogue without deleting them.

An example of comments:

```
// The line bellow is commented and will not run. This
   is a comment too by the way!
// say | This is a comment, you will not see this
   dialogue
```

```
say | This is not a comment, you will see it
```

Flag Solving

See flag

If you have set a flag, you can get it's value with the special syntax `${flag}`

For example:

```
flag | inc | 5 | monkeys
say | There are ${monkeys} little monkeys jumping on the
   bed.
```

BBCode and Text Effects

In order to have text effects and formatting such as **bold text**, *italic text*, and much more.

BBCode is a well-known format, and you can find out more about it on the Godot documentation, but here's a basic example:

```
say | italian_man | [i]I am speaking in italic! No not
   italian...
say | brave_man | I am brave and [b]bold[/b] in the face
   of danger.
say | small_man | [font_size=8]Please don't make fun of
   my font size, I'm quite insecure about it.
```

See Also

characters

say

choice

branch

DialogueQuest BBCodes

Characters

Characters are simply a collection of data, and have the following properties:

An ID - This is how they will be referred to in DQD. The ID is not shown in-game.

A Name - The name that will be displayed in the in-game dialogue.

A Color - The color Associated with the character, used mainly for displaying their name.

A Portrait - An image that will be displayed when the character is speaking.

Importing Characters into Dialogue-QuestTester

In the top menubar, toggle the `DialogueQuest Menu` button.

To import, press `Import`, then select all relevant `.dqc` character files.

To export, select your characters, and click `Export`. You will then be prompted for the directory to save the characters in.

The Say Statement

The say statement is the most common statement in DialogueQuest.

It's usage is:

```
say | [character_id] | [speech]
say | [character_id] | [speech] | [speech2]
say | [character_id] | [speech] | [speech2] |
say | [speech]
say | [speech] | [speech2]
say || [speech] | [speech2] |
```

NOTE: The `say` statement has a special use case, where the statement can be skipped:

```
[character_id] | ...
```

So writing `say |` is optional, as long as a valid character ID is provided. This setting may be disabled by a developer (however it is enabled by default).

The basic use case would be:

```
say | my_character | Hey, I am saying something
```

And:

```
say | There is dialogue without character. Perhaps it is
   a ghost...
```

The character_id field can also be provided empty for the same result:

```
say | | I am still a ghost...
```

If you want to pause in the middle, you can use multiple speech pipes as so:

```
say | DialogueQuest is absolutely | legen|dary!
```

If you end the say statement with an empty pipe, the dialogue will advance without user input:

```
say | dude1 | Hey man so I heard about this game called
   DeshanimQuest and |
say | dude2 | Yeah whatever dude
say | dude1 | Hey don't cut me off like that!
```

If using it without a character, you **must** provide an empty character:

```
say | This is not going to work... |
```

```
say | | This does work though! |
```

See Also

Writing Dialogue

BBCode and Text Effects

The flag statement

A `flag`, is simply a value that can exist, or not exist.

The act of creating a flag is called `raising` it, afterwards we can check if it exists, and what it is set to.

It's usage is:

```
flag | raise | [flag]
flag | set | [value] | [flag]
flag | inc | [flag]
flag | inc | [amount] | [flag]
```

```
flag | dec | [flag]
flag | dec | [amount] | [flag]
flag | delete | [flag]
```

A basic example would be:

```
flag | raise | is_using_dialogue_quest

// This will happen
branch | flag | is_using_dialogue_quest
    say | We are using DialogueQuest.
branch | end

// This will not happen
branch | no_flag | is_using_dialogue_quest
    say | We are NOT using DialogueQuest.
branch | end
```

You can also use `flag | inc` and `flag | dec` to use integer (whole number) flags:

```
flag | inc | money

// Will say `I have 1 money`
say | I have ${money} money

flag | inc | 6 | money

// Will say `I have 7 money now`
say | I have ${money} money now

flag | dec | money

// Will say `I have 6 money now`
say | I have ${money} money now
```

You can use `flag | set` to set a flag as an arbitrary value like so:

```
flag | set | Mage | player_class

// Will say `Oh sick! I am a Mage`
say | Oh sick! I am a ${player_class}

flag | set | 20 | number_of_enemies

// Will say `We have 20 enemies here, that's a lot!`
say | We have ${number_of_enemies} enemies here, that's
   a lot!
```

And finally, you can delete a flag as well:

```
flag | raise | road_is_safe

// Will say `The player proceeds forward` (in Italics)
branch | flag | road_is_safe
    say | [i]The player proceeds forward
branch | end
branch | no_flag | road_is_safe
    say | [i]The player stays back
branch | end

flag | delete | road_is_safe

// Will say `The player stays back`
branch | flag | road_is_safe
    say | [i]The player proceeds forward
branch | end
branch | no_flag | road_is_safe
    say | [i]The player stays back
branch | end
```

The choice statement

The choice statement will bring up a menu with items that the user has to choose from.

It is inherently dependant on the branch statement

It's usage is:

```
choice | [choice1] | [choice2]...
```

For example:

```
say | Which one do you like better? Apples or Oranges?
choice | Apples | Oranges | You can't compare

branch | choice | Apples
    say | Doctors hate you
branch | end
branch | choice | Oranges
    say | Juicy!
branch | end
branch | choice | You can't compare
    say | You're just so smart, aren't you?
branch | end
```

The Branch Statement

The branch statement allows dialogue to happen in different ways depending on a variety of factors.

When a branch statement is encountered, the dialogue can go in one way or another, like a fork in the road or *branch*es of a tree.

It is recommended to first understand flag, choice, and flag solving as they are essential for undertstanding branching.

It's usage is:

```
branch | choice | [choice1] | [choice2]...
branch | evaluate | [expression]
branch | end

branch | flags | [flag1] | [flag2]...
branch | flag | [flag]
branch | flag | [flag1] | [flag2]...
branch | no_flag | [flag]
branch | no_flag | [flag1] | [flag2]...
branch | flag > | [flag] | [value]
branch | flag < | [flag] | [value]
branch | flag = | [flag] | [value]
branch | flag != | [flag] | [value]
branch | flag >= | [flag] | [value]
branch | flag <= | [flag] | [value]
```

A simple example of a branch would be:

```
say | Let's see about this branching thing

flag | raise | loves_dialogue_quest

branch | flag | loves_dialogue_quest
    // We will see this
    say | I love DialogueQuest!
branch | end

branch | no_flag | loves_dialogue_quest
    // We will not see this
    say | I HATE DialogueQuest!
branch | end
```

A branch checks a **condition**, and if it finds that condition to be **true**, it runs the contents until it reaches the next `branch | end` statement.

Choice

When using choices, we must use the `branch | choice` statement, like so

```
choice | a | b

branch | choice | a
    say | We picked A
branch | end
branch | choice | b
    say | We picked B
branch | end
```

We do not have to provide a branch for every choice.

We can also check for multiple choices, like so:

```
choice | a | b | c | d

branch | choice | a | b
    say | We picked either A or B!
branch | end
```

Evalute

`evaluate` is the most complex branch statement, and will use GDScript to solve the branch.

It can be used like the following:

```
branch | evaluate | true
    say | This will always happen.
branch | end

branch | evaluate | false
    say | This will never happen.
branch | end

branch | evaluate | 5 == 10
    say | This won't happen because 5 is not 10 :)
branch | end

branch | evaluate | 10 > 5
    say | This will happen.
branch | end

branch | evaluate | 5 != 10
```

```
    say | This will happen.
branch | end

branch | evaluate | 5 >= 5
    say | This will happen.
branch | end

branch | evaluate | this == that
    say | This won't happen.
branch | end

branch | evaluate | that == that
    say | This will happen.
branch | end
```

evaluate can also be used with flag solving

```
branch | evaluate | "${main_character}" == "joe"
    say | joe | Yo, uh-huh
branch | end

branch | evaluate | ${number_of_corners} == 3
    say | This is my hat
branch | end

// You can also use the 'or', 'and', '&&', '||'
   statements to check multiple conditions.
branch | evaluate | ${number_of_corners} > 3 or
   ${number_of_corners} < 3
    say | This is not my hat
branch | end
```

Flags

The `branch | flag` statement is quite versatile, and can be used in a few ways:

This is the simplest flag check:

```
branch | flag | some_basic_flag
    say | This really is basic
branch | end
```

We can also check multiple flags at the same time:

```
branch | flag | red_flag | green_flag
    say | I'll choose either anyway
```

```
branch | end

flag | raise | green_flag
branch | flag | red_flag | green_flag
    // We will see this
    say | Great!
branch | end

flag | delete | green_flag
flag | raise | red_flag
// Only red_flag is raised at this point
branch | flag | red_flag | green_flag
    // We will see this
    say | Still great...?
branch | end

branch | flag | orange_flag
    // We will not see this
    say | What does an orange flag mean?
branch | end
```

In the above example, the branch will be entered if *any* of the flags are raised.

There is also the alternative `branch | flags` (note the `s` for plural), which will only be entered if *all* flags are raised

```
flag | raise | table
flag | raise | tea

branch | flags | table | tea
    // We will see this
    say | Tea is great.
branch | end

branch | flags | table | plate
    // We will not see this
    say | Cookies are great.
branch | end
```

Finally there is the `branch | no_flag` statement, which will only be entered if *none* of the flags are raised

```
flag | raise | pickles
flag | raise | lettuce

branch | no_flag | knife
```

```
    // We will see this
    say | How will I cut my pickles?
branch | end

branch | no_flag | pickles | tomatoes | lettuce
    // We will not see this because
    say | I love my burgers dry as the desert.
branch | end
```

Flag operators

The `branch | flag` also has versions for using comparison operators, such as `> (greater than)`, `< (lesser than)`, `= (equals)`, `>= (greater than or equals)`, and `<= (lesser than or equals)`.

```
flag | set | 10 | stairs

branch | flag >= | stairs | 11
    // We will not see this
    say | I'm gonna take the elevator.
branch | end

branch | flag != | stairs | 0
    // We will see this
    say | We have stairs
branch | end

branch | flag < | stairs | 2
    // We will not see this
    say | Even a baby can climb these
branch | end

branch | flag = | stairs | 10
    // We will see this
    say | The perfect amount of stairs
branch | end
```

See Also

flag

choice

flag solving

GDScript Control Flow

What are Expressions?

GDScript Expression class

The signal statement

The signal statement does not quite do anything for the user.

It's functionality is sending a "message" of sorts for the Godot developer to implement into concrete functionality.

It's usage is:

```
signal | [param1] | [param2]...
```

For example:

```
signal | play_song | Nightcall - Kavinsky
```

The developer can for example check for the signal value "play_song", and play the song accordingly.

Also see:

Developer manual entry for `signals`

The call statement

This is advanced functionality, and requires coding knowledge to use

The call statement allows you to run GDScript code directly from a DQD.

It's usage is:

```
call | [GDScript code]
```

Using call, you can run any GDScript code.

By default, this will run code as an `Expression` object.

DialogueQuest has a setting that runs the code in a GDScript instance, which is more powerful, however it is experimental.

The exit statement

The exit statement will end the dialogue early.

It's usage is:

```
exit |
```

DialogueQuest specific BBCode

If you haven't already, check out the BBCode and Text Effects section.

DialogueQuest implements a few custom BBCodes:

The speed bbcode

The `speed` bbcode sets the dialogue speed (letters per second) within the bounds of the BBCode.

For example:

```
say | This is a regular say statment
say | [speed=1]This is a suuuper slooow say statment
say | [speed=500]This is a really fast say statment
say | [speed=10]Now I'm slow[/speed][speed=100] And now
   I'm fast
```

The pause statement

The `pause` statement makes the dialogue pause for a specified time (in seconds) before automatically continuing.

For example:

```
say | I have hi[pause=0.5]-hiccups and a bit of a
   s[pause=0.1]-s[pause=0.1]-s[pause=0.1]-tutter
```

See Also

DQD

BBCode and Text Effects