Contents

DialogueQuest Developer Manual

DialogueQuest for non-coders

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

Installation and Setup

The recommended way to install DialogueQuest is vida Godot's builtin Asset Library.

However if you want to get the latest features, you should install via the repository as such:

On Linux / Mac:

```
cd my_godot_project
git clone https://github.com/hohfchns/DialogueQuest
mkdir -p addons/
mv ./DialogueQuest/addons/DialogueQuest ./addons
rm -rf DialogueQuest
```

Or in one line: `git clone https://github.com/hohfchns/DialogueQuest && mkdir -p addons/ && mv ./DialogueQuest/addons/DialogueQuest ./addons && rm -rf DialogueQuest`

On Windows:

In your Godot project:

- Clone the repository
- Make directory called `addons`
- Move the folder `DialogueQuest\addons\DialogueQuest` inside `addons`
- Delete the cloned repository

Open your Godot project, go to `Project -> Project Settings -> Plugins` and enable `DialogueQuest`

The Data Directory

Go to `Project -> Project Settings -> General` and search for `Dialogue Quest`, then set `Data Directory` to a folder where you will

store DialogueQuest files (characters, dialogues, etc.)

This folder is by default set to `res://dialogue_quest/`

This folder is where you will be storing your characters and dialogues

---

Examples

To see a basic example of DialogueQuest, see the `examples` folder of the repository.

A more advanced example would be the DialogueQuestTester application.

Writing Dialogue

Writing dialogue is done in the `DQD` format and is explained in detail in the `User Manual`.

This manual is for usage and extension of DialogueQuest within a Godot project.

For implementing the dialogue, see Creating Dialogue

Creating Characters

Before we starting creating dialogues, we need to know how to create characters.

Creating characters is quite simple, simply create a new `DQCharacter` Resource in your Data Directory, and DialogueQuest will automatically be able to find and use it.

When creating a characters, you must provide a `character_id`. This is how the character will be referred to in DQD.

Creating Dialogue

To create a dialogue, simply create a new `.dqd` file.

If you put the file in your data directory, you will not have to specify a full path for it. See Starting the Dialogue

Playing Dialogue

Scene setup

To play dialogue, you should set up your scene as follows:

```
...
CanvasLayer
    DQDialoguePlayer
    DQDialogueBox
    DQChoiceMenu
...
```

Click on the `DQDialoguePlayer` and provide it with the `DQDialogueBox`, as well as `DQChoiceMenu`.

Also create a `DQDialoguePlayerSettings` for it. It is recommended to save this resource as a file in your project.

You can also do this setup through code, however make sure the `DQDialoguePlayer` node set up before it is added to the scene.

Starting the dialogue

In order to start the dialogue, use the `DQDialoguePlayer.play()` method.

```
dialogue_player.play("my_dialogue_name")
# These are also valid ways to provide the dialogue
# dialogue_player.play("my_dialogue_name.dqd")
#
   dialogue_player.play("res://dialogue_quest/my_dialogue_name.dqd")
```

Take note - If your `.dqd` file is not in your data directory, you will have to provide the full filepath.

Stopping the dialogue

If you want to stop the dialogue early, you can call the `DQDialoguePlayer.stop()` method which will end the dialogue early.

Settings

Settings are saved in the `.dialogue_quest_settings.conf` file.

Currently there is only one setting, the data directory, which should be configured in the Project Settings rather than directly in the config file.

Extending DialogueQuest

Theming

DialogueQuest uses mostly Godot's native Theme system for designing how the interface looks.

You can create a new Theme and import the settings from the default DialogueQuest theme, or create one completely from scratch as the default theme is quite small.

The main way of customizing dialogue components in DialogueQuest is simply creating an inherited scene, and changing it however you like.

Some nodes such as `DQDialogueBox` have `settings` objects, for example `DQDialogueBoxSettings`, which provides some common customizations.

See Also

Theme

Using the theme editor

Custom Statements

DialogueQuest allows you to add custom statements to `DQD` and extend the featureset of DqdParser.

To do so, do the following:

```
## my_node_or_autoload.gd

class SectionMySection extends DQDqdParser.DqdSection:
    var statement: String

    func solve_flags() -> void:
        pass

func _ready() -> void:
    DQDqdParser.statements.append(
        DQDqdParser.Statement.new("my_statement",
            _my_statement_func)
    )

## Returns SectionPipeline on success
```

```
## Returns DQDqdParser.DqdError on failure
static func _my_statement_func(pipeline:
    PackedStringArray):
    if pipeline.size() <= 2:
        var error := DQDqdParser.DqdError.new("Error!
            Cannot parse statement my_statement, please
            provide at least 2 arguments.")
        return error

    var sec := SectionMySection.new()
    sec.statement = pipeline[1] + pipeline[2]
    return sec
```

First we create a new section class which extends `DQDqdParser.DqdSection`, this can be either a localy defined class like the example, or a new script with `class_name` definition.

We can give it the `solve_flags()` method which will define how the `${flag}` syntax works in DQD.

Now we need to create our parser function, in this case `_my_statement_func`. Note that the `static` is optional, however the rest of the signature is critical.

The function must take in an argument of type `PackedStringArray`, and must return either an object of class inheriting `DqdSection` indicating it is successful, or `DqdError` indicating it has failed.

The `pipeline` argument is an array of every pipe-seperated argument in the line the statement was found in.

Note that: - It contains the statement itself (always, at index 0) - It contains whitespace, you can use the helper functions `DQScriptingHelper.remove_whitespace`, `DQScriptingHelper.trim_whitespace`, `DQScriptingHelper.trim_whitespace_prefix`, `DQScriptingHelper.trim_whitespace_suffi`

Lastly we must add a new `DQDqdParser.Statement` object to `DQDqdParser.statements`.

The `DQDqdParser.Statement` constructor takes 2 arguments: - The statement itself, the word that will be referred to in DQD. - The callback function that will be used to parse the the statement.

Right now, your statement is parsed, however it cannot actually do anything until you implement it's logic. See Custom Logic

Custom Logic

The logic of DialogueQuest is handled in the `DQDialoguePlayer` class.

In order to add custom logic, you must create a new class extending `DQDialoguePlayer`.

Once you do, you can handle your custom statement like so:

```
## my_dialogue_player.gd
extends DQDialoguePlayer

func _ready() -> void:
    self.section_handlers.append(
        SectionHandler.new(SectionMySection,
            _handle_my_section),
    )

    super._ready()


func _handle_my_section(section: SectionMySection) ->
   void:
    # Here you have access to all parts of the
       DQDialoguePlayer
    print("This section doesn't do anything yet... It's
       statement is %s" section.statement)
```

To add a handler, we must add a `SectionHandler` object to the `section_handlers` array.

The constructor of `SectionHandler` takes two parameters, a class (object of type `GDScript`), and a `Callable`.

When the parser returns the class you provided, the function you provided will be called with the parser's returned object.

Now we need to create our handler function, in this case `_handle_my_section`.

The function must take in an argument of your section class, in this case `SectionMySection`. It does not return anything.

The Flags system

Flags are global variables that can be accessed from both code and dialogue.

They are accessible via the global `DQFlags` instance `DialogueQuest.Flags`

An example:

```
DialogueQuest.Flags.raise("flag1")

DialogueQuest.Flags.set_flag("flag2", 2)

DialogueQuest.Flags.set_flag("flag3", "a third flag")

# Outputs 2
print(DialogueQuest.Flags.get("flag2"))

# Outputs 'A third flag'
print(DialogueQuest.Flags.get("flag3"))
```

Also see

The user manual entry on the `flag` statement

DialogueQuest signals

There are a few important signals in DialogueQuest:

The error signal

DialogueQuest uses `assert` statements for it's critical errors, which will pause the game when running in the editor, however will not do so in a release build.

For the purpose of handling errors in release builds as well as GUI, DialogueQuest emits the `DialogueQuest.error(message: String)` signal when an error occures.

DQSignals

Other main signals are available via the `DQSignals` instance `DialogueQuest.Signals`

The signals are:

- dialogue_started(dialogue_path: String)
- dialogue_ended(dialogue_path: String)
- dialogue_signal(params: Array)
  - Emitted via the `signal` statment in dialogue.
- choice_made(choice: String)
  - Emitted when a player makes a choice during dialogue.

See also

The `signal` statement in the user manual.

The `choice` statement in the user manual.