Yunasawa の Library

Runtime Debug Command - Documentation

***Documentation* shows you how to use *RUNTIME DEBUG COMMAND*.**

Script DOCUMENTATION Debug Command Contact yunasawa200@gmail.com

★ About

- ***Runtime Debug Command*** provides you an input field used to handle and debug your logics, events or you can use it as a feature of you game.
- See **Version** for more updated features.
- This tool is not really perfect and complete so if you have any errors or bugs or difficulties when use this, please feedback and I will response as soon as possible.

★ Table On Contents

★ How to create Debug Command GUI

From Menu Item / Hierarchy (Currently not supported)

- First, you can select an object in hierarchy to be parent of RDC.
- Right-click in hierarchy, select `YのL` > `2D` > `Runtime Debug Command` or select `Tools` > `YのL` > `Create Object` > `2D` > `Runtime Debug Command` on window bar.
- In case you don't select a canvas to be the parent, a new Canvas and Event System will be created automatically.

From Prefab / Asset

- Find RDC in `Assets` > `Yunasawa の Library` > `Runtime Debug Command` > `Prefabs` > `Debug Command GUI`.
- Drag it into an object that you want it to be the parent of RDC.

★ How to create new Debug Command

- I recommend you to create a new `DebugCommand` right inside `Assets > Yunasawa の Library > Runtime Debug Command > Custom Commands > Commands`, otherwise you will have some small troubles.
- Here is a sample code for a `DebugCommand`, I call it DC_Debug. It is used to display a message inside log window with a general command of `/debug selection message`.

```
2 references
public class DC_Debug : DebugCommand // /debug selection message
{
    1 reference
    public DC_Debug() : base()
    {
        CommandNodes = new CommandNode[]
        {
            new("debug", new[] { "debug" }, true, false), // 0

            new("selection", new[] { "log", "warning", "caution", "notify", "error" }, true, false), // 1
            new("message", new string[0], false, true) // 2
        };
    }

    2 references
    public override void Execute(string[] value)
    {
        if (CheckWrongCommand(value)) return;

        string content = string.Join(" ", value.Skip(2));

        string color = "";
        switch (value[1])
        {
            case "log": color = "9EFFF9"; break;
            case "warning": color = "FFE045"; break;
            case "caution": color = "FF983D"; break;
            case "notify": color = "79FF53"; break;
            case "error": color = "FF3A3A"; break;
        }
        Log($"<#{color}>{value[1].ToTitleCase()}:</color> <#FFFFFF>{content}</color>");
        Debug.Log($"<color=#{color}><b>{value[1].ToTitleCase()} ►</b></color> <color=#FFFFFF>{content}</color>");
    }
}
```

- As you can see on the sample picture, now I will show you how to make one step by step:
  - First, create a new class (name it whatever you want, I recommend to put DC_ in the beginning), inherited from `DebugCommand`.
  - Make a constructor for it, now you have to concentrate on this step. **CommandNodes** is a **List** of **CommandNode**. Here is **CommandNode** class:

```
public class CommandNode
{
    public string Nodes = "";
    public string[] Suggestions = new string[0];
    public bool MustStartWith = false;
    public bool Customable = false;

    7 references
    public CommandNode(string nodeCommand, string[] suggestions, bool startWith = false, bool customable = false)
    {
        Nodes = nodeCommand;
        Suggestions = suggestions;
        MustStartWith = startWith;
        Customable = customable;
    }
}
```

As you can see, **CommandNode** has 3 properties, Nodes, Suggestions, MustStartWith and Customable.

- **Nodes** is the general name of node in a command. For example, in the command `/debug selection message`, **Nodes** are "*debug*", "*selection*", "*message*".
- **Suggestions** will show up when you typing the commands so you can `Tab` to finish it automatically, when you typing the "*selection*" node of `/debug selection message`, a list of `"log", "warning", "caution", "notify", "error"` will show up.
- **MustStartWith**, if you enable this only the suggestions which start with the word you're typing will appear. You are in "*selection*" node, then you type "n" then only "notify" appears, but if it's disabled, "warning", "caution" and "notify" will show up (Those 3 contain "n").
- **Customable** allows you to set that node is customable or not, it means you can type anything in that node and no need to follow the suggestions.

- Back to sample DebugCommand, you can see inside the constructor, I assign **CommandNodes** with a new list, inside I make new **CommandNode** objects with inputing params are **Nodes, Suggestions, MustStartWith, Customable**.
  - Node 0: `new("debug", new[] { "debug" }, true, false), // 0`, it have "*debug*" as **Nodes**, "*debug*" as **Suggestions** and **MustStartWith** is *true*.
  - Node 1:

    `new("selection", new[] { "log", "warning", "caution", "notify", "error" }, true, false), // 1`

    is similar.
  - Node 2: `new("message", new string[0], false, true) // 2`, because I don't need **Suggestions** for this so I don't put anything inside, **StartWith** is defaulted by *false* and this node is **Customable**, you can type anything in this node.
- After finish the constructor, call an override void named `Execute(string[] value)`, **value** is an array of words separated by *space*. For example, in command `/debug log Hello world`, **value** is *{ "debug", "log", "Hello", "world" }*. Inside this method, you can do your own code to handle the command just like above sample code.
- You can use `if (CheckWrongCommand(value)) return;` to automatically check for wrong command and block the executer.
- Use `Log(string log, LogType type = LogType.Executed)` to show your command's result on Command Log as a message.

- Here is another sample for `DebugCommand` called DC_Time, used to manage time in game.

```
public class DC_Time : DebugCommand // /time selection amount
{
    1 reference
    public DC_Time() : base()
    {
        CommandNodes = new CommandNode[]
        {
            new("time", new[] { "time" }, true, false), // 0

            new("selection", new[] { "timescale" }, true, false), // 1
            new("amount", new[] { "0", "1" }, false, true) // 2
        };
    }

    2 references
    public override void Execute(string[] value)
    {
        if (CheckWrongCommand(value)) return;

        if (int.TryParse(value[2], out int amount)) { }
        else
        {
            Log($"<#FF7070>Failed:</color> <#FFF087>{value[2]}</color> is in wrong format", LogType.Failed);
            return;
        }
        Time.timeScale = amount;
        Log($"Time set to <#FFE045>{value[2]}</color>");
    }

}
```

- After you created a new `DebugCommand`, go inside `DebugCommandObject.cs` and there're 2 things you have to notice:

○ Find this **CreateCommand()** method in **DebugCommandList** class.
○ Inside this **switch**, add the first node of commands as **case** and return the **command** like this.
	`public DebugCommand CreateCommand(string key)`
	`{`
	`  switch (key)`
	`  {`
	`  case "debug": return new DC_Debug();`
	`  case "time": return new DC_Time();`
	`  }`
	`  return new DC_Detector();`
	`}`

○ Then find this **CommandList** inside **DebugCommandObject** class.
○ Add the first node of the command inside the list of string like this.
	`public static string[] CommandList = new string[]`
	`{`
	`  "debug", "time"`
	`};`

- Finish those thing and you now can use your own `DebugCommand` right inside your project.

★ How to use Debug Command Setting

You can select **Debug Command GUI** object to make your custom setting for RDC.