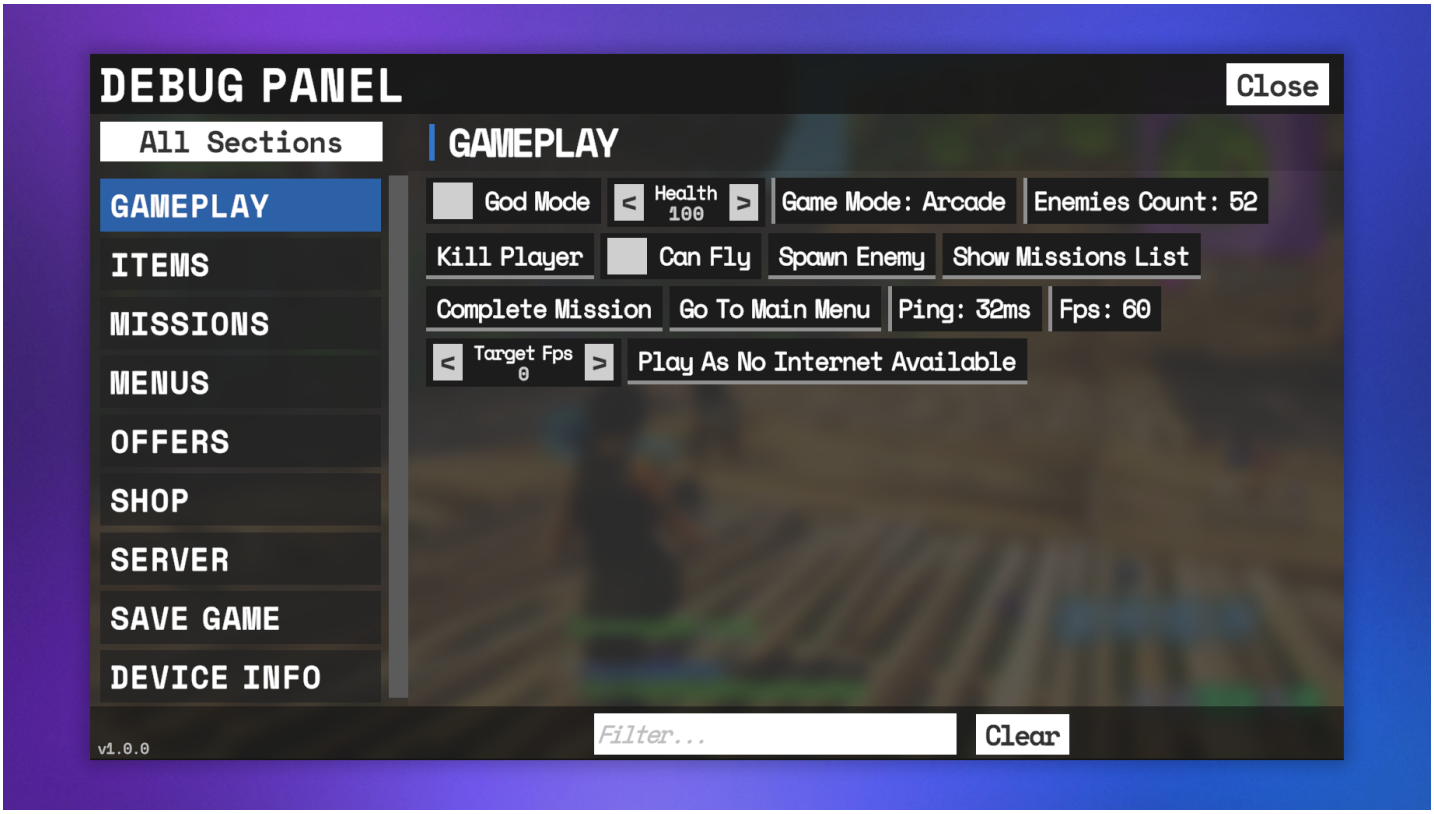
Unity Debug Panel is a lightweight and versatile ingame debug panel for Unity with C#.
It can be incredible useful to be able to modify gameplay parameters while on the target device.
This asset simplifies the process of creating a panel with debut options in your Unity projects, allowing you to focus on what matters: gameplay.

A debug panel, is a user interface that provides developers with tools and information to aid in debugging and profiling during the development of a software application or game.

This asset provides a suit of premade elements (buttons, int selector, float selector, enum selection, etc), while allowing for the creation of new ones, with ease.

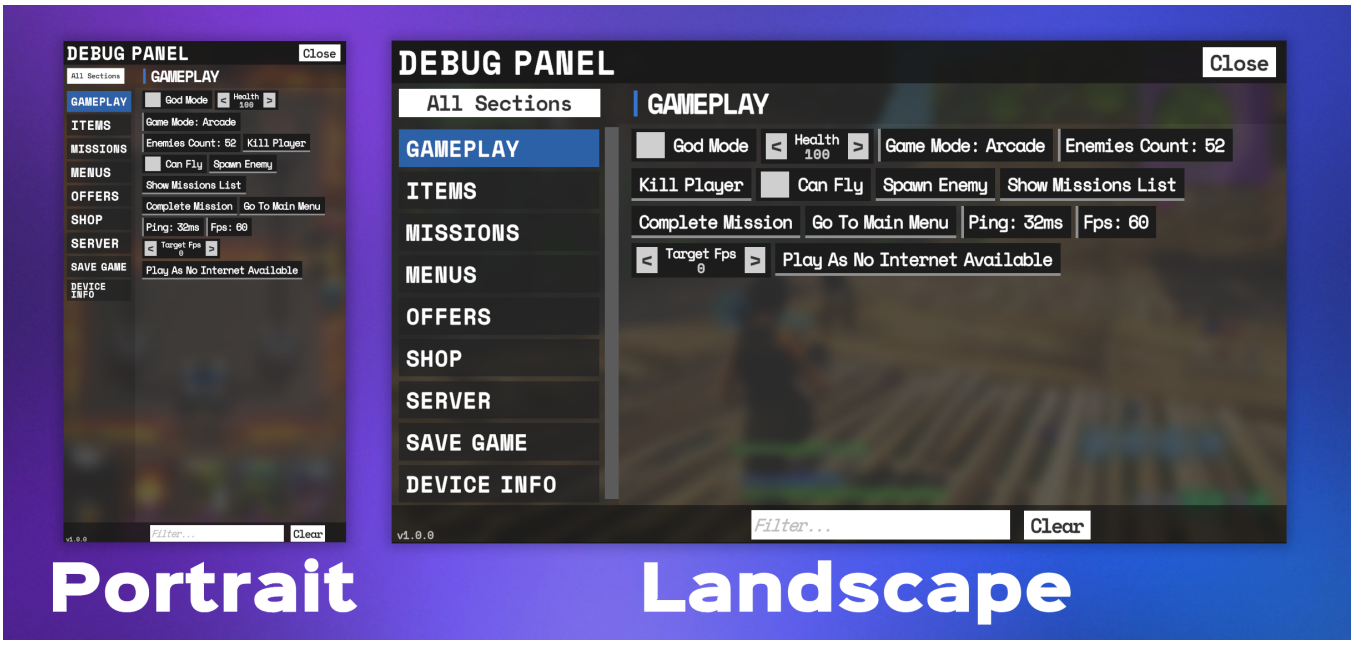


Features

- **Simple API**: Unity Debug Panel provides an intuitive and easy-to-use API with C#.

```
UDebugPanel.Show();
UDebugPanel.Hide();

IDebugActionsSection section = UDebugPanel.AddSection("Section name")
section.AddButton("Button name", () => Debug.Log("Button click"));
```

- **Adaptative**: The different widgets support and adapt to different screen aspect ratios, making it a good fit for both, desktop and mobile.



- **Smart**: You can automatically generate a debug options section using a class. Using reflection, changes that occur on the debug panel will affect the class instance.

- **Organization**: Organize your options using collapsable sections.
- **Fuzzy search**: Quicly find the options you were looking for with the search bar!
- **Lightweight**: While your game is running, the panel does not exist at all until you want to show it. When is hidden again, the panel is completely destroyed, so it does not affect to the preformance of your game.

Installation

Download and import from the Unity Asset Store. You can move the root folder anywhere you want in your project.
The package has the following dependencies:

- TextMeshPro (Mandatory).

> [!NOTE]
> Works with both New and Old input systems

After installing

To quickly check if everything has been setup properly, you can go to DebugPanel/Examples/Scenes/ and open any of the example scenes. When you run any of those scenes, a simple functionality example should play.

Getting started

Showing / Hiding

- For **showing** the panel, you just need to call the `Show` method. The internal logic will take care of everything else.

```
UDebugPanel.Show();
```

- For **hiding** the panel, just call the `Hide` method.

```
UDebugPanel.Hide();
```

- For convenience, you can also **toggle** the panel. Just call the `Toggle` method.

```
UDebugPanel.Toggle();
```

Automatic Toggle

We already provide some handy automatic input toggles for you.

```
UDebugPanel.SetupToggleInput();
```

By calling `UDebugPanel.SetupToggleInput()` , you will automatically setup toggle for:

- Keyboard F1 key.
- Middle mouse button.
- Triple finger tap on screen.

You can modify this behaviour using the method parameters.

Controller navigation

The panel has support for navigation with a controller or keyboard.
You can enable controller navigation by calling:

```
UDebugPanel.SetControllerSupport(true);
```

> [!NOTE]
> Must be called before showing the panel, to take effect.

Sections

Debug actions are divided within different sections. These sections allow you to better organize your actions.
You cannot create a debug action outside of a section.

- **Creating** a new section is very simple, you just need to call `AddSection` and provide a section name:

  ```
  IDebugActionsSection section = UDebugPanel.AddSection("Section name")
  ```

- **Removing** a section is equally as simple. Just call `RemoveSection` , and provide the section you want to remove.

  ```
  UDebugPanel.RemoveSection(section);
  ```

> [!NOTE]
> You don't need to show the panel before creating actions/sections.

> [!NOTE]
> You can see this functionality on the example DebugPanel.Sections.

Automatic debug actions

This asset has the ability of scanning for properties and methods in C# classes, to automatically create the adecuate widgets.

One of such classes may look like this:

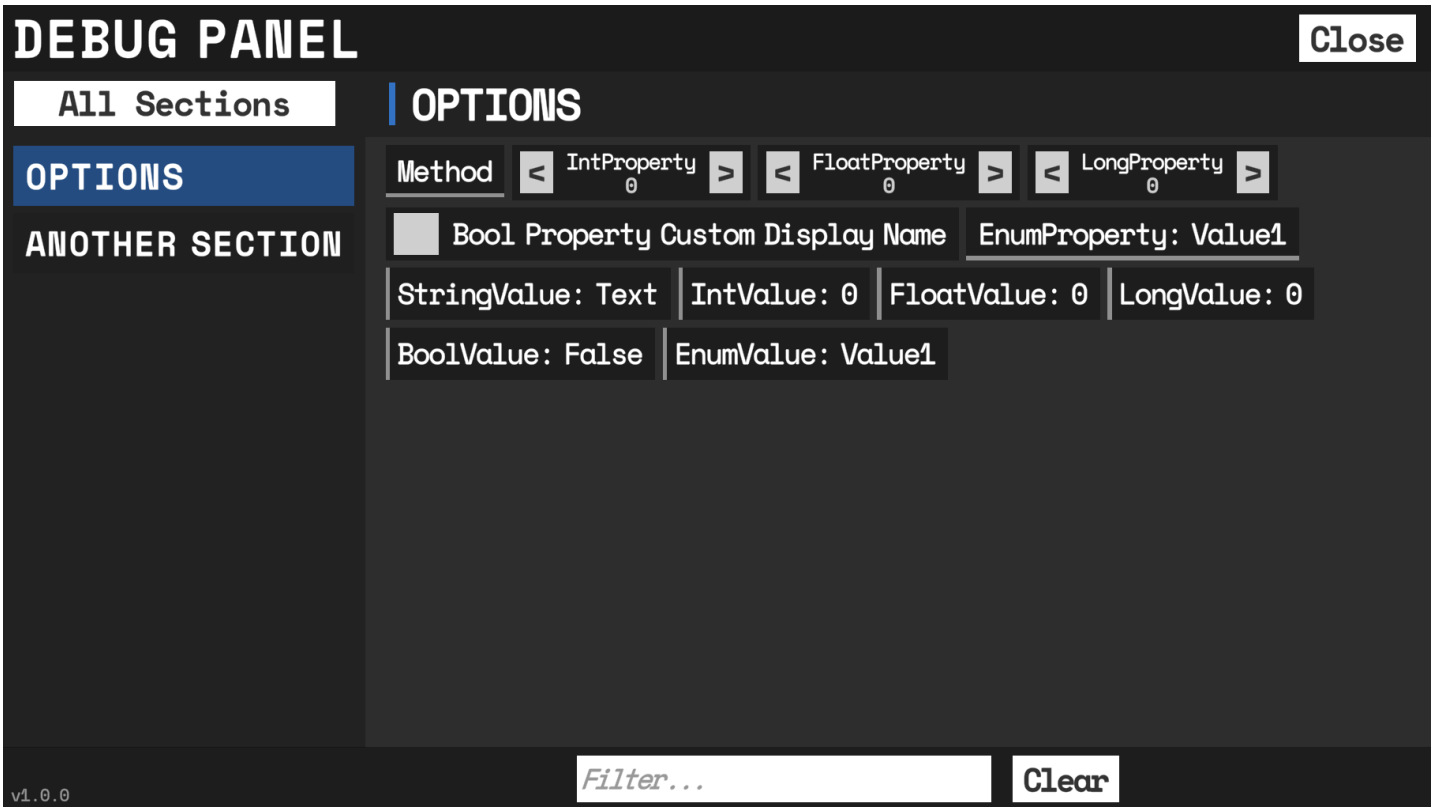
```
class Options
{
    [NumberStep(5)]
    public int IntProperty { get; set; }

    [NumberStep(2.5f)]
    public float FloatProperty { get; set; }

    [NumberRange(-10, 10)]
    public long LongProperty { get; set; }

    [DisplayName("Bool Property Custom Display Name")]
    public bool BoolProperty { get; set; }
    public EnumProperty EnumProperty { get; set; }

    public string StringValue { get; } = "Text";
    public int IntValue => IntProperty;
    public float FloatValue => FloatProperty;
    public long LongValue => LongProperty;
    public bool BoolValue => BoolProperty;
    public EnumProperty EnumValue => EnumProperty;

    public void Method()
    {
        Debug.Log("Calling Method");
    }

    [Category("Another Section")]
    public int IntProperty2 { get; set; }

    [Category("Another Section")]
```

```
    public int FloatProperty2 { get; set; }
}
```

Then we add the class like this:

```
UDebugPanel.AddOptionsObject(new ExampleOptionsObject());
```

And we will get debug options like this:



> [!NOTE]
> You can see this functionality on the example DebugPanel.Reflection.

Manual debug actions

This is the most important part of this asset, the debug actions (or widgets). Once you have a section, you add debug actions to it:

Info:

- Info: a static string that cannot be changed one submited.

```
    section.AddInfo("Some info that never changes");
```

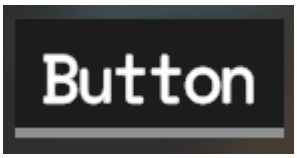
- Dynamic Info: a getter for a string that it's updated every frame.

```
section.AddInfoDynamic(() => "Some info that can change");
```

Buttons:

- Button: a simple button with a name.

```
section.AddButton("Button name", () => Debug.Log("Pressed"));
```



Toggle:

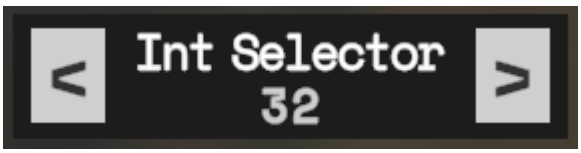
- Toggle: a simple bool toggle with a name. Requests a setter and a getter for the value.

```
bool someBool = false;
section.AddToggle("Toggle name", val => someBool = val, () => someBoo
```

Number selectors:

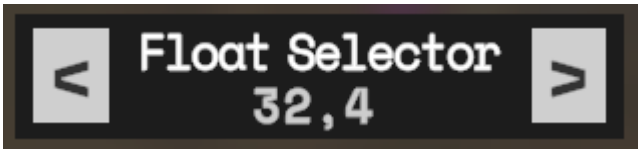
- Int: an int selector with a name. Requests a setter and a getter for the value.

```
int someInt = 0;
section.AddIntSelector("Int name", val => someInt = val, () => someIn
```



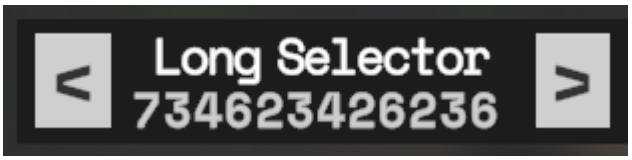
- Float: a float selector with a name. Requests a setter and a getter for the value.

```
float someFloat = 0f;
section.AddFloatSelector("Float name", val => someFloat = val, () =>
```



- Long: a long selector with a name. Requests a setter and a getter for the value.

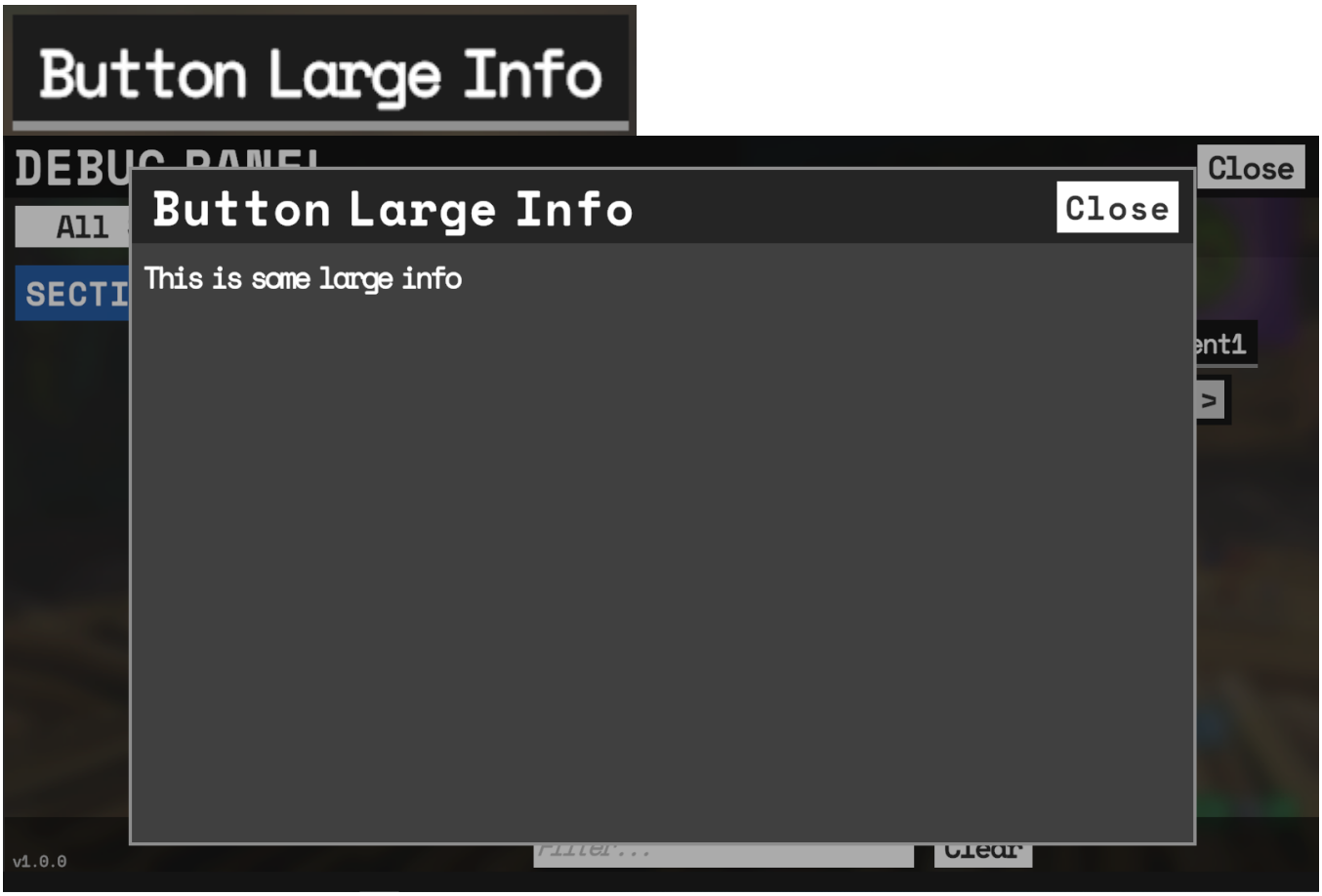
```
long someLong = 0;
section.AddLongSelector("Long name", val => someLong = val, () => sor
```



Advanced Buttons:

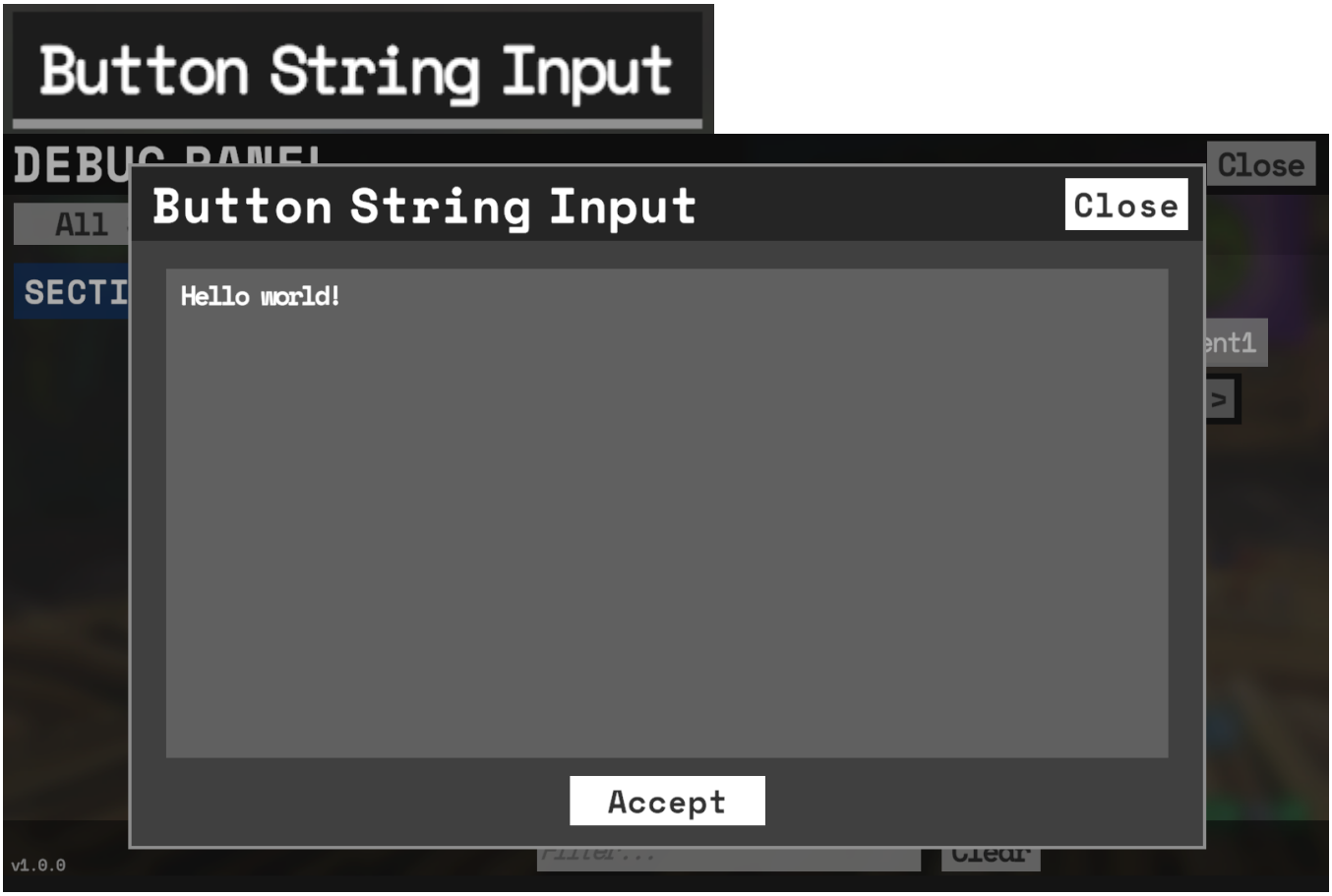
- Button large info: a button that opens a popup which can shown information as text.

```
section.AddButtonLargeInfo("Button name", () => "This is some large i
```



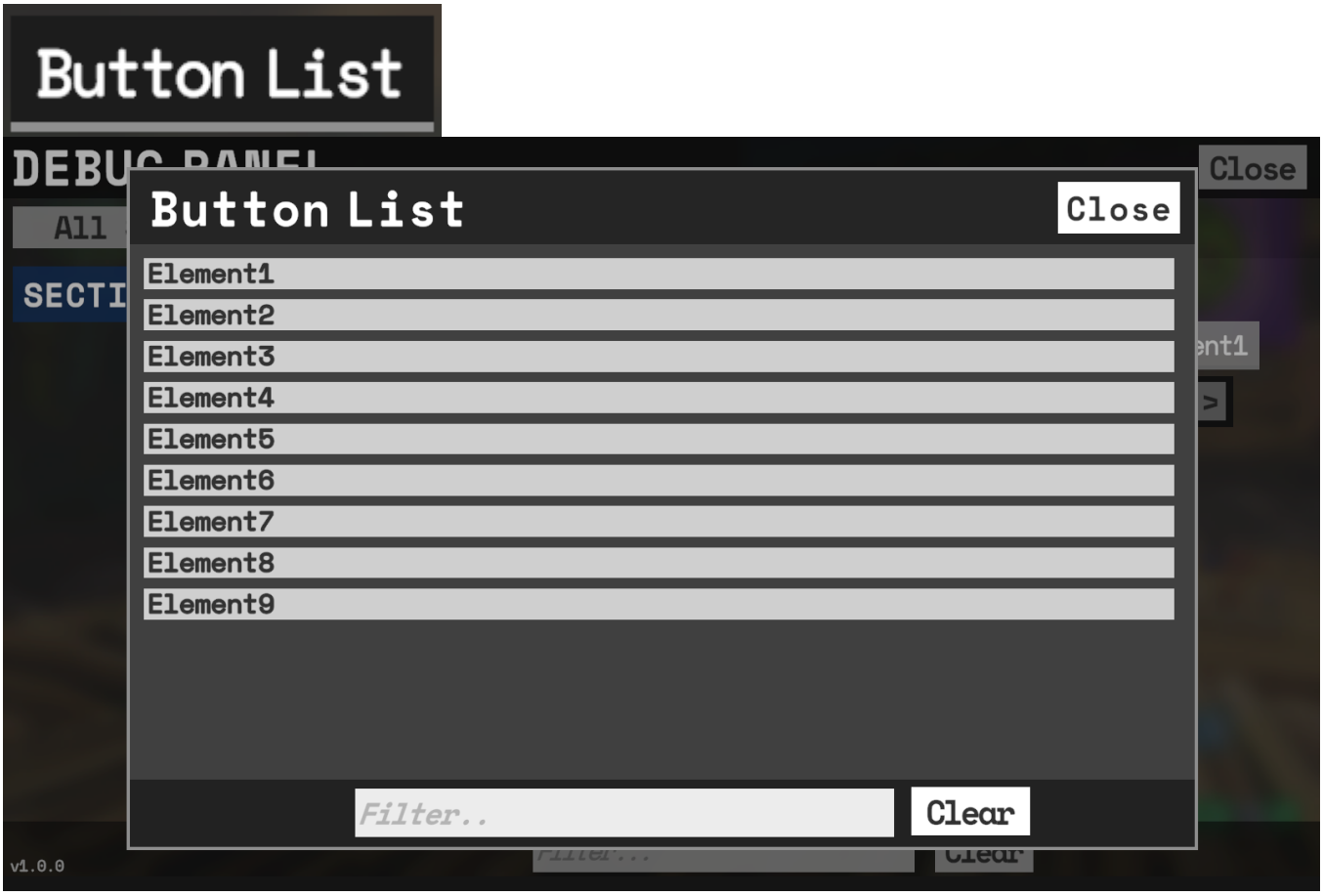
- Button string input: a button that opens a popup where you can set a string value.

```
string _stringInput = "Empty";
section.AddButtonStringInput("Button name", () => _stringInput, v =>
```



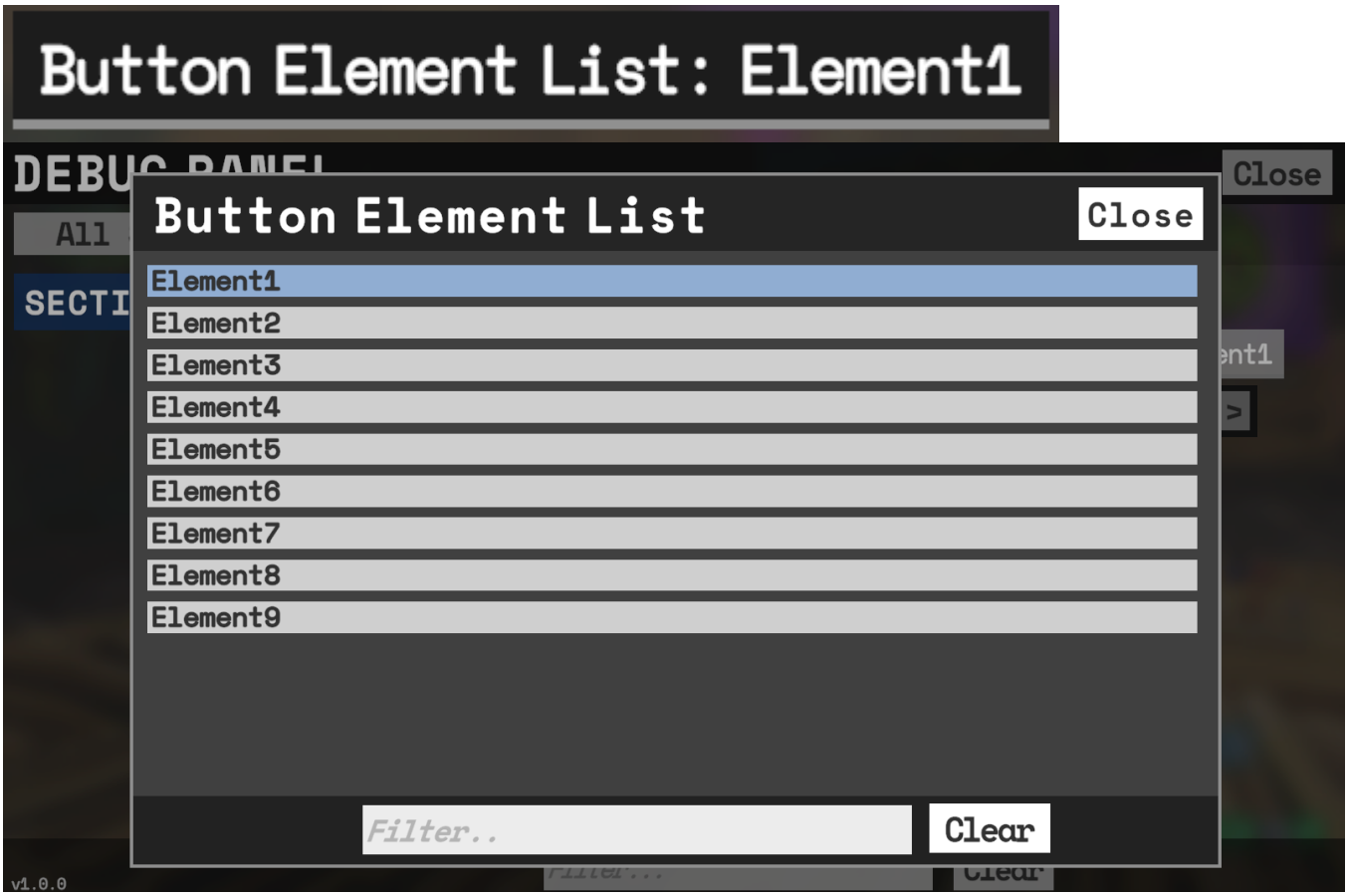
- Button list selector: a button that opens a popup where you can select an item from a list of items.

```
List<string> _elementsList = new() {"Element1", "Element2", "Element3
section.AddButtonListSelector("Button name", () => _elementsList, i =
```



- Button element list selector: a button that opens a popup where you can select an item from a list of items. The current selected value is stored internally and shown on the button text.

```
List<string> _elementsList = new() {"Element1", "Element2", "Element3
section.AddButtonElementListSelector("Button name", () => _elementsLi
```



- Button enum selector: a button that opens a popup where you can select the value of an enum. The current selected value is stored internally and shown on the button text.

```
TestEnum _elementEnumSelected = TestEnum.Element1;
section.AddButtonEnumSelector("Button name",  i => _elementEnumSelect
```

Nested Containers:

- Nested conatiner: a button that opens a popup which can have more debug actions.

```
section.AddButtonActionsContainer("Actions", s =>
{
    s.AddButton("Another Button 1", () => Debug.Log("Button pressed")
    s.AddButton("Another Button 2", () => Debug.Log("Button pressed")
    s.AddButton("Another Button 3", () => Debug.Log("Button pressed")
});
```

> [!NOTE]
> You can see debug actions functionality on the example DebugPanel.Widgets.

Creating more debug actions

Some times, your game may have specific needs that cannot be properly met by the default provided widgets. That's why you can create your own.
We are going to use the Info action as an example.

1. The first thing you need to do is create a new class and inherit from `DebugAction`. This interface will force you to implement `DebugActionWidget InitWidget(DebugActionWidget viewInstance)` method, which is responsable for setting the values from the action, to the widget Ui. We will not implement it for now.
   We should first set the information that the action will hold. In this case it's a string for the info.

```
public sealed class InfoDebugAction : DebugAction
{
    readonly string _info

    public InfoDebugAction(string info)
    {
        _info = info;
        ActionName = info; // ActionName is used for search box funct
```

```
    }

    public override void InitWidget(DebugActionWidget viewInstance)
    {
        throw new NotImplementedException();
    }
}
```

2. Next, we need a new `DebugActionWidget`, which will be the actual GameObject placed on the Ui. Since the widget is made of a label, we will add a reference to it. We will also add an Init method to set the label value.

```
public sealed class InfoDebugActionWidget : DebugActionWidget
{
    public TextMeshProUGUI Label;

    public void Init(string info)
    {
        Label!.text = info;
    }
}
```

3. Going back to the `InfoDebugAction` class, we need to implement `InitWidget`. The widget itself will be automatically instantiated internally. We need to init it here.

```
public sealed class IntDebugAction : IDebugAction
{
    readonly string _info

    public InfoDebugAction(string info)
    {
        _info = info;
        ActionName = info;
    }

    public override void InitWidget(DebugActionWidget viewInstance)
    {
        InfoDebugActionWidget widget = (InfoDebugActionWidget)viewIns
        widget.Init(ActionName);
    }
}
```

4. For being able to use this new action on a section, just add an extension method that does this:

```
public static IDebugAction AddInfo(this IDebugActionsSection section,
{
    InfoDebugAction debugAction = new InfoDebugAction(info);
    section.Add(debugAction);
    return debugAction;
}
```

5. Cool! Finally, befor using the action, you just need to let the Debug Panel know that it should link the new debug action with the new widget prefab.

```
UDebugPanel.RegisterWidgetPrefab<InfoDebugAction>(InfoDebugActionWi
```

6. That's it. Everything is set up for using your new debug action.

> [!NOTE]
> You can see an example of a custom widget at the example scene DebugPanel.CustomWidgets.

Creating more popups

> [!NOTE]
> You can see an example of how to create custom popups on DebugPanel.CustomPopups.