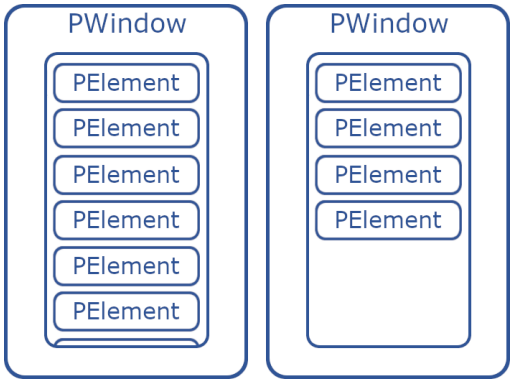
ESSENTIAL LABS

ProceduralUI Documentation

1. UI Structure



PManager (located in the ProceduralUIManager prefab) creates and manages all windows that has been created. There you can call a method that creates a new window. After that you can add some PElements to every window created. ("P" prefix stands for procedural).

By default, all PWindows are movable and resizable. If PElements overflow the window, the mask is used and a slider shows up.

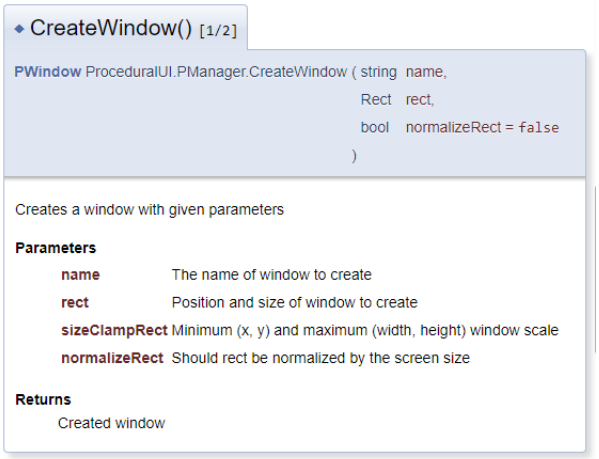
2. Setup

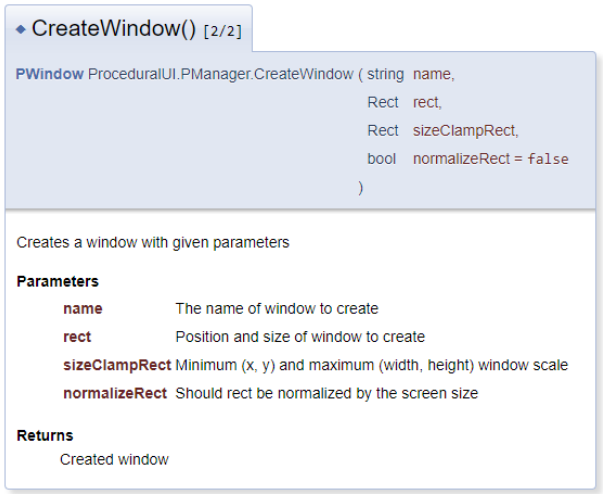
The first step is just to drag ProceduralUIManager prefab (*ProceduralUI/Resources/Prefabs*) onto the scene.

3. Creating your own window

It is really important to remember that window creation (as well as elements) should be done in the Awake function. Thanks to that we can listen to UI events without worrying about the order of scripts execution. To create a window, call a CreateWindow method in the PManager script (it's a singleton class). Example:

var window = PManager.Instance.CreateWindow("My_Window", windowRect, clampRect, PManager.Instance);





Name – used to identify a window by the PManager (that's why it should be unique). It should not contain any spaces (use "_" instead). It will be shown on the title bar (PWindow automatically replaces "_" with a space – don't worry about it).

Rect – Defines window's x and y position, as well as width and height.

SizeClampRect – X and y stands for the minimum window's width and height. Width and height arguments are seen as max width and height.

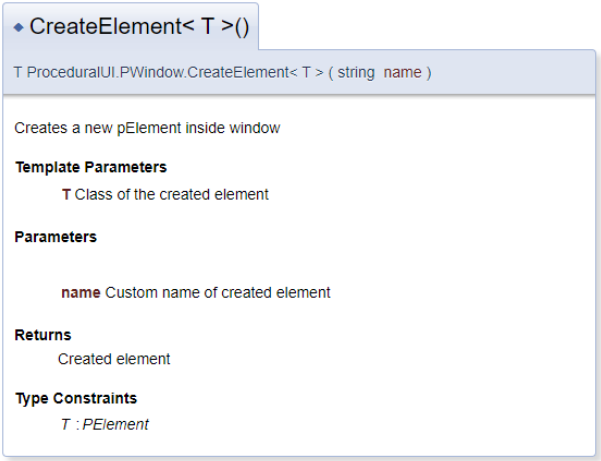
Normalized rect – If set the rect arguments will be treated as normalized (otherwise they will be seen as pixels). For example, *new Rect(0.5f, 0.5f, 0.25f, 0.25f)* passed as the **rect** argument would make a window whose bottom-left corner would be located at the center of the screen. It's width and height would be set to 1/4th of the screen. Keep in mind that the same rule goes with the **SizeClampRect** parameter.

It's worth mentioning that you can turn on/off window resizing/moving by calling one of those two methods:

void SetDraggingActive (bool active)
Disable ability to move the window
void SetResizingActive (bool active)
Disable ability to resize the window

4. Adding elements to a window

CreateWindow method returns the PWindow class and that's what we need. To create an element, call the CreateElement method.



For example, if you want to add an input field it should look like this:

PInputField textField = loginWindow.CreateElement<PInputField>("Text_Field");

Name – element's name follows the same rules as window's name. However, it is not visible in-game – just manager's id purposes. If you want to initialize an element with some real, visible variables, you need to call p*Element.Initialize(foo)*.

textField.Initialize("Login");

Each elements have its own initialization parameters (each one has different needs). Here you have a list of the default PElements:

- **PText –** just a normal text. Its initialization requires a string argument that represents a text visible in a window.

  pText.Initialize (string text)

- **PInputField –** initializes with a placeholder text.

  pInputField.Initialize(string placeholderText)
  Initialize input field with a placeholder text (text visible when no value)
  pInputField.SetText(string text)
  Overwrites input field's text value

- **PSlider –** requires more variables during the initialization.

  pSlider.Initialize(float min, float max, float value = 0f, bool wholeNumbers = false)

- **PToggle –** requires a string that represents a text next to the toggle. It also needs a default Bool value.

  pToggle.Initialize(string text, bool isOn = false)

- **PButton –** just needs a string that shows on the button.

  pButton.Initialize(string text)

- **PVector2 –** initialized with a default value

  pVector2.Initalize(Vector2 value)
  pVector2.SetValue(Vector2 value)

  pVector2.X [get, private set]
  pVector2.Y [get, private set]
  pVector2.Value [get, private set]

- **PVector3 –** same as **PVector2**

  pVector3.Initalize(Vector3 value)
  pVector3.SetValue(Vector3 value)

  pVector3.X [get, private set]
  pVector3.Y [get, private set]
  pVector3.Z [get, private set]
  pVector3.Value [get, private set]

- **PSpace –** lets you add some space between other elements. Requires a space height in pixels.

  pSpace.Initialize(float height)
  pSpace.SetHeight(float height)

- **PMultilineInputField –** just as the normal input field, it requires a placeholder text.

pMultilineInputField.Initialize(string placeholderText)
Initialize input field with a text that is shown when no value
pMultilineInputField.SetText(string text)
Overwrites current field text value

It is really important to know that every PElement stores the information about Unity's UI elements that the is created of. Because of that you can call them in order to, for example, listen to OnValueChanged event of PInputFIeld, or OnClick for PButton (more about that in demo scenes).

Also, you **need** to call the window's Refresh() method right after creating the last PElement to make sure that elements are positioned correctly.

5. Creating your own PElement

It's a good practice not to modify the ProceduralUI folder. It will allow you to update the package easily. Everything that we will do right now should be done in a separate directory.

For the showcase purposes we're going to create a slider element. We'll call it PCustomSlider. Let's create a script with that name. Every PElement needs to be a derivative of the PElement class.

```
using ProceduralUI;

public class PCustomSlider : PElement {

}
```

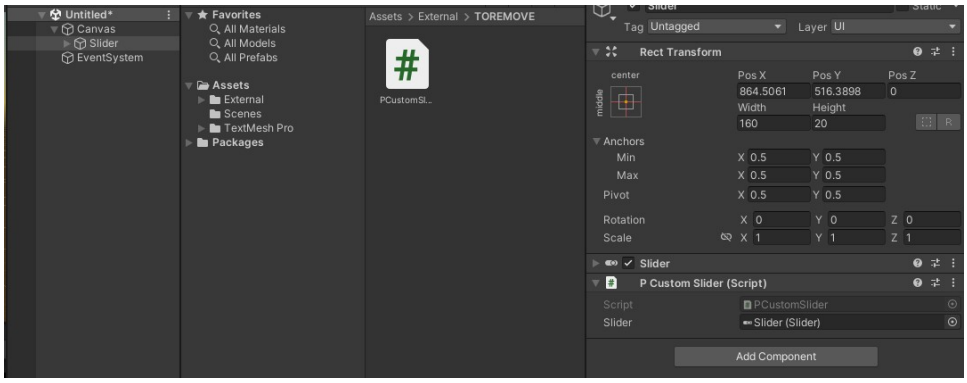
We can use unity's built in Slider object. Let's add *UnityEngine.UI* namespace and a slider reference.

```
using UnityEngine.UI;
using ProceduralUI;

public class PCustomSlider : PElement {
    public Slider slider;
}
```

The last step is to create an Initialization method (it's optional though).

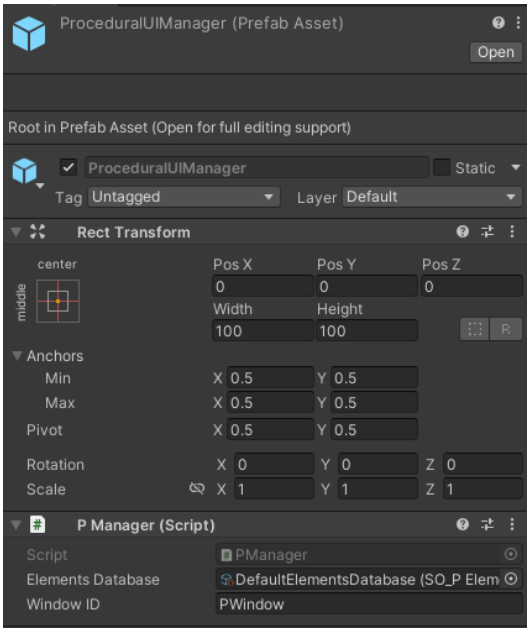
```
using UnityEngine.UI;
using ProceduralUI;

public class PCustomSlider : PElement {
    public Slider slider;

    public void Initialize(float min, float max, float value = 0f, bool wholeNumbers = false) {
        slider.minValue = min;
        slider.maxValue = max;
        slider.value = value;
        slider.wholeNumbers = wholeNumbers;
    }
}
```

That's all we need to do if it goes to scripting. The next step is to create a Slider prefab. Go to *GameObject/UI/Slider* and attach our script.



We can define element's height, however width is calculated by the window. Save the slider as a prefab (**its name needs to match the class name**). The last step is to go to the *ProceduralUI/Resources/Data/DefaultElementsDatabase* and drag the prefab there.

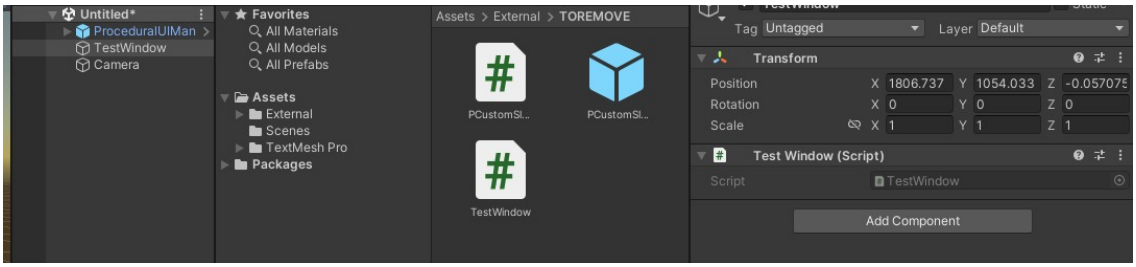
If you don't want to modify package assets, you can create your own database by rightclicking project window and then *Create/Essential Labs/ProceduralUI/ElementsDatabase.* Then you need to assign your scriptable object to the PManager singleton. WindowID informs the script which element from the database is the window prefab.



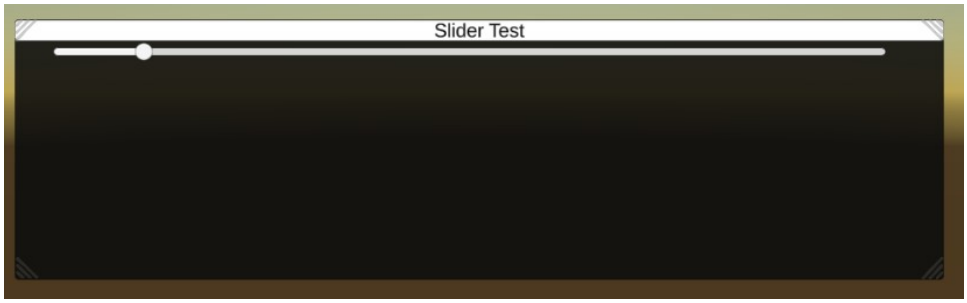
Now we can test if it works! Let's create a window script that contains our *PCustomSlider*.

```
using UnityEngine;
using ProceduralUI;

public class TestWindow : MonoBehaviour {
    private void Awake() {
        PManager manager = PManager.Instance;

        Rect position = new Rect(0.25f, 0.32f, 0.5f, 0.25f);
        Rect sizeClamp = new Rect(0.15f, 0.15f, 0.75f, 0.75f);
        PWindow createdWindow = manager.CreateWindow("Slider_Test", position, sizeClamp, true);

        PCustomSlider ourSlider = createdWindow.CreateElement<PCustomSlider>("custom_slider");
        ourSlider.Initialize(0, 10, 1);

        createdWindow.Refresh();
    }
}
```

The last step is to attach our script to the empty game object. It should look like this:



And here we have it:



If you want to call a function whenever the slider value has changed, you can do

OurSlider.slider.OnValugeChanged.AddListener(OurFunction);

6. Closure

Did you find a bug? Create an issue on the github page [here]

ProceduralUI is in active development feel free to share your ideas: jachow.mateusz@gmail.com