Voltage Framework Quickstart (WIP)

Table of Contents

Areas

Areas can hold any element inside of them, including other areas. Areas help you define your layout in an easy and fast way, you just need to know what you are doing. This section will help you understand how each area behaves.
Some important settings to keep in mind are:

- **Horizontal:** The orientation the elements will be drawn. Set it to true for horizontal or false for vertical.
- **Element Margin:** by default all elements will be put without any space between them. Use this setting to separate them.
- **Padding:** The padding of an area will determine how much space will be left in between the area's border and the elements inside.
- **Alignment:** This setting will only be used in the case that the area has some free space. This will not work with the elements internal alignment, as in labels won't align their text accordingly with this setting.

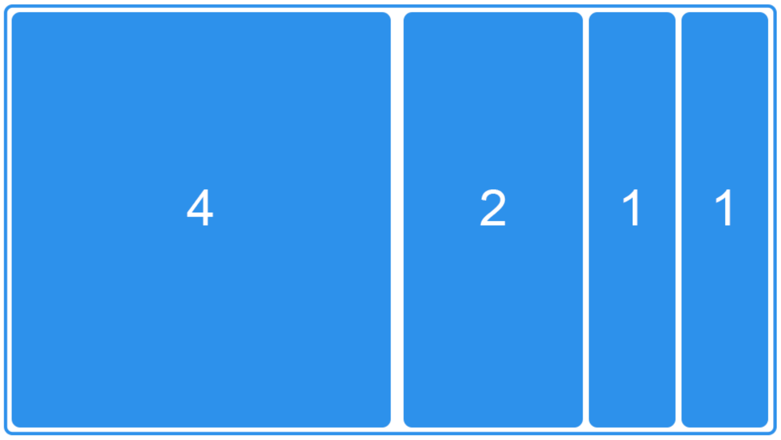
You can find this settings on all areas, and you can set them on their constructors by passing an AreaSettings struct as a parameter.

Weight Areas

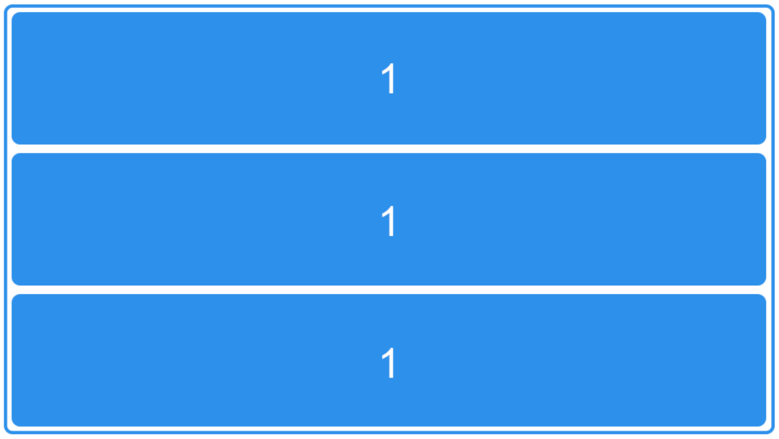
Weight areas work by distributing their available space between all their elements according to their weight. For example, an element with a weight of 2 will be twice as large as an element with a weight of 1.

To set an element's weight you can either define it on it's constructor passing an **ElementSettings** struct, or afterwards by accessing the elements Weight property (Check the API Reference for more details). Weights are not managed inside weight areas, they are only read to calculate the elements proper size.

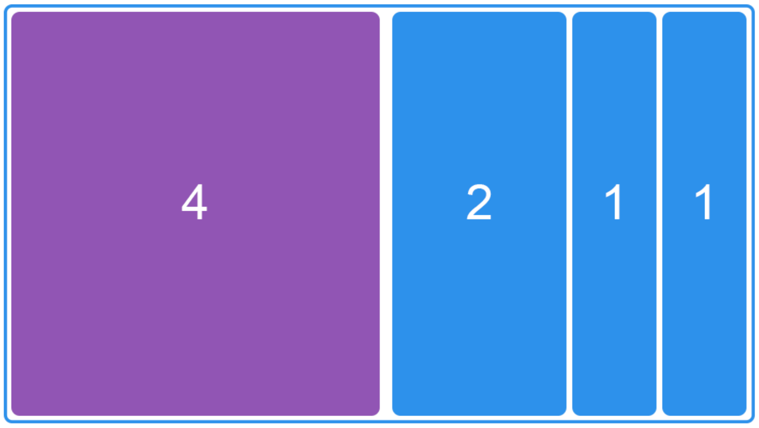
If the area is horizontal all elements will take the full height.



If the area is vertical they will take the full width.



If you add an area (in violet) it will behave the same way as any other element.



Using Wild WeightAreas

```
protected void VoltageGUI()
{
    Constructor.WeightAreaStart(/*Your Settings*/);
    {
        // Add wild elements to this area
    }
    Constructor.EndArea();
}
```

```
WeightArea myWeightArea;

protected void VoltageInit()
```

```
{
    // Initialize the area
    myWeihtArea = new WeightArea(/*Your Settings*/);
}
protected void VoltageGUI()
{
    // Use your area
    Constructor.WeightAreaStart(myWeightArea);
    {
        // Add wild elements to your area
    }
    Constructor.EndArea();
}
```

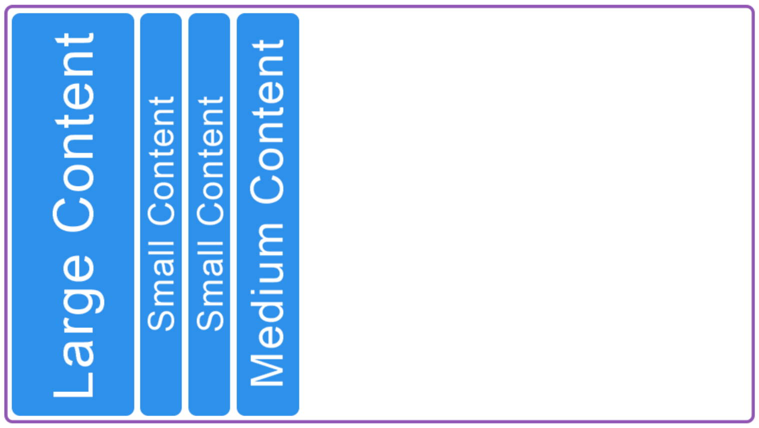
Using Stored WeightAreas

```
WeightArea myWeightArea;

protected void VoltageInit()
{
    // Initialize the area
    myWeihtArea = new WeightArea(/*Your Settings*/);

    // Start the constructor for the area
    Constructor.StartStoredConstructor(myWeightArea);
    {
        // Add stored elements to your area
    }
    Constructor.EndStoredConstructor();
}

protected void VoltageGUI()
{
    // Use your area
    Constructor.WeightAreaStart(myWeightArea);
    {
        //Add wild elements to your area
        //They will be displayed after the stored elements
    }
    Constructor.EndArea();
}
```

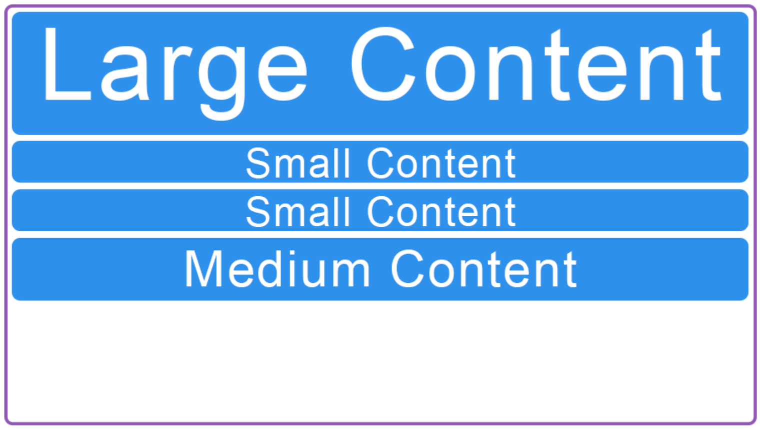
Stream Areas

Stream areas organize their elements by fitting them one next to the other with each element's minimum size. Areas will be fitted too, so there is no need to keep adding stream areas. Stream areas may end up with free space, and they can also overflow out of their assigned area (the overflow won't be visible and out of bounds). To show the overflowed content wrap the stream area inside a scroll area, which will add a set of scrollbars to navigate the area.

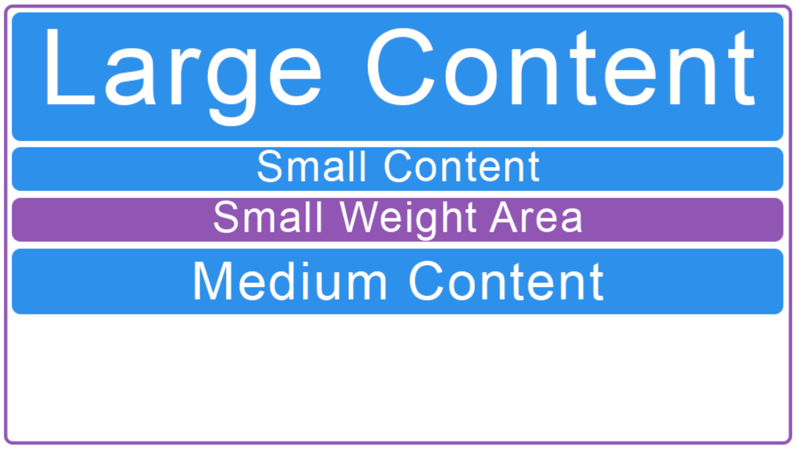
If the area is horizontal all elements will take the full height.(Text won't show sideways, this is only for demonstration)



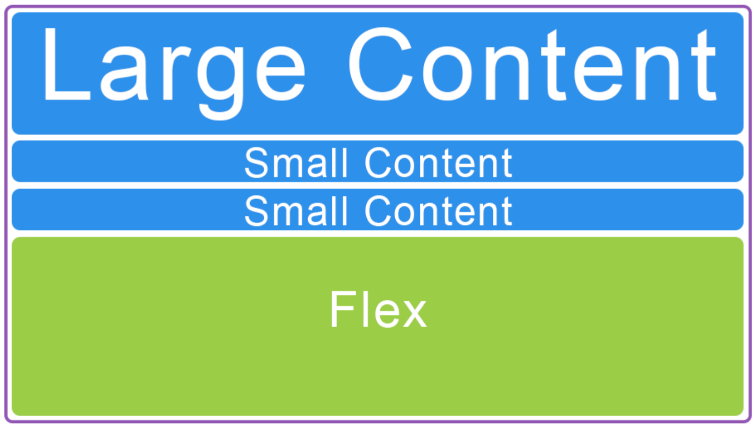
If the area is vertical all elements will take the full width.



Areas(in violet) will be fitted too.



If there are any elements on a stream area set as flex (green) then they will be sharing the rest of the space available according to their weights (same as in a weight area). To set an element's flex and weight you can either define it on it's constructor passing an **ElementSettings** struct, or afterwards by accessing their Flex or Weight property respectively (Check the API Reference for more details).



Using Wild StreamAreas

```
protected void VoltageGUI()
{
    Constructor.WeightAreaStart(/*Your Settings*/);
    {
        // Add wild elements to this area
    }
    Constructor.EndArea();
}
```

```
StreamArea myStreamArea;

protected void VoltageInit()
{
    // Initialize the area
    myStreamArea = new StreamArea(/*Your Settings*/);
}
protected void VoltageGUI()
{
    Constructor.StreamAreaStart(myStreamArea );
    {
        // Add wild elements to your area
    }
    Constructor.EndArea();
}
```

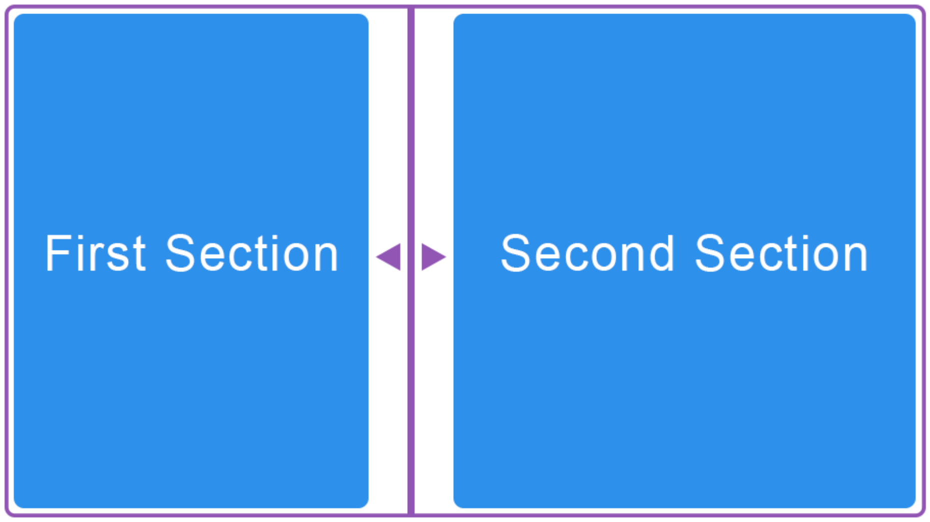
Using Stored StreamAreas

```
StreamArea myStreamArea;

protected void VoltageInit()
{
    // Initialize the area
    myStreamArea = new StreamArea(/*Your Settings*/);

    // Start the constructor for the area
    Constructor.StartStoredConstructor(myStreamArea);
    {
        //Add stored elements to your area
    }
    Constructor.EndStoredConstructor();
}

protected void VoltageGUI()
{
    // Use your are
    Constructor.StreamAreaStart(myStreamArea);
    {
        //Add wild elements to your area
        //They will be displayed after the stored elements
    }
    Constructor.EndArea();
}
```

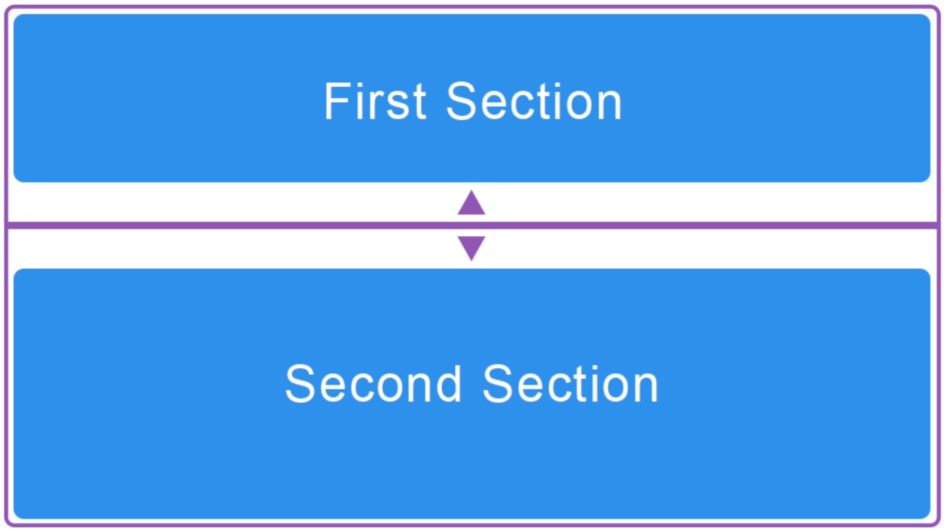
Split Areas

Split area lets you define an area divided in two, which division can be resized. You always need to have a SplitArea variable so the division value can persist (see examples below). Both sides of the split will start as horizontal weight areas, to override them just add another area inside of them before you start adding elements.

Horizontal Split



Vertical Split.



Using Wild SplitAreas

```
// Always keep a reference to all your SplitAreas
SplitArea mySplitArea;

protected void VoltageInit()
{
    // Initialize the area
    mySplitArea = new SplitArea(/*Your Settings*/);
}
protected void VoltageGUI()
{
    Constructor.SplitAreaStart(mySplitArea);
    {
        // Add wild elements to the first section
    }
    Constructor.SplitCurrentArea()
    {
        // Add wild elements to the second section
    }
    Constructor.EndArea();
}
```

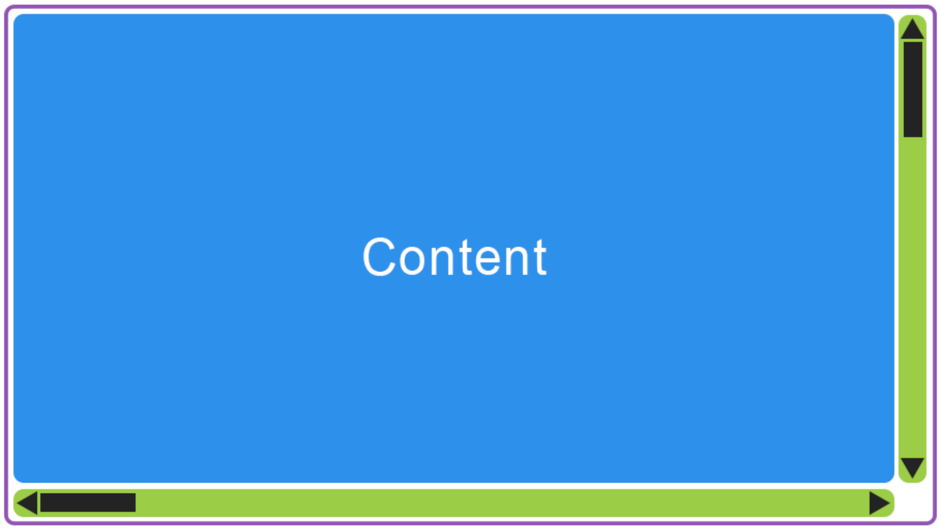
Using Stored SplitAreas

```
// Always keep a reference to all your SplitAreas
SplitArea mySplitArea;

protected void VoltageInit()
{
    // Initialize the area
    mySplitArea = new SplitArea(/*Your Settings*/);

    // Start the constructor for the area
    Constructor.StartStoredConstructor(mySplitArea);
    {
        // Add stored elements to the first section
    }
    Constructor.SplitCurrentArea()
    {
        // Add stored elements to the second section
    }
    Constructor.EndStoredConstructor();
}

protected void VoltageGUI()
{
    Constructor.SplitAreaStart(mySplitArea);
    {
        // Add wild elements to the first section here
        // They will be displayed after the stored elements of the first
section
    }
    Constructor.SplitCurrentArea()
    {
        // Add wild elements to the second section here
        // They will be displayed after the stored elements of the second
section
    }
    Constructor.EndArea();
}
```

Scroll Areas

Scroll areas are used to wrap other areas or elements and provide scrollbars in the case that the size of it's content is larger than the size the scroll area is assigned to. The most common type of area that will require a scroll area wrapping it, is the stream area. But it will work with any other kind of areas as well.

Scroll areas' horizontal setting won't be used.



Using Wild ScrollAreas

```
// Always keep a reference to all your ScrollAreas
ScrollArea myScrollArea;

protected void VoltageInit()
{
    // Initialize the area
    myScrollArea = new ScrollArea(/*Your Settings*/);
}
protected void VoltageGUI()
{
    Constructor.ScrollAreaStart(myScrollArea);
    {
        // Add wild elements to your area
    }
    Constructor.EndArea();
}
```

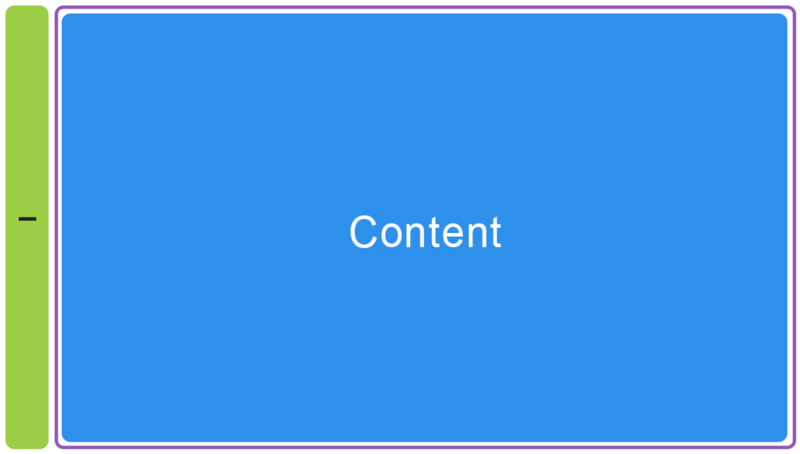
Using Stored ScrollAreas

```
// Always keep a reference to all your ScrollAreas
ScrollArea myScrollArea;

protected void VoltageInit()
{
    // Initialize the area
    myScrollArea = new ScrollArea(/*Your Settings*/);

    // Start the constructor for the area
    Constructor.StartStoredConstructor(myScrollArea);
    {
        // Add stored elements to your area
    }
    Constructor.EndStoredConstructor();
}

protected void VoltageGUI()
{
    // Use your area
    Constructor.ScrollAreaStart(myScrollArea);
    {
        // Add wild elements to your area
        // They will be displayed after the stored elements
    }
    Constructor.EndArea();
}
```

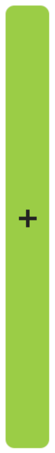
Foldout Areas

Foldout areas are a collapsible area. They allow you to separate your elements into discrete sections, and allow the user to show or hide them. You need to have your foldouts on a variable for them to be collapsible.

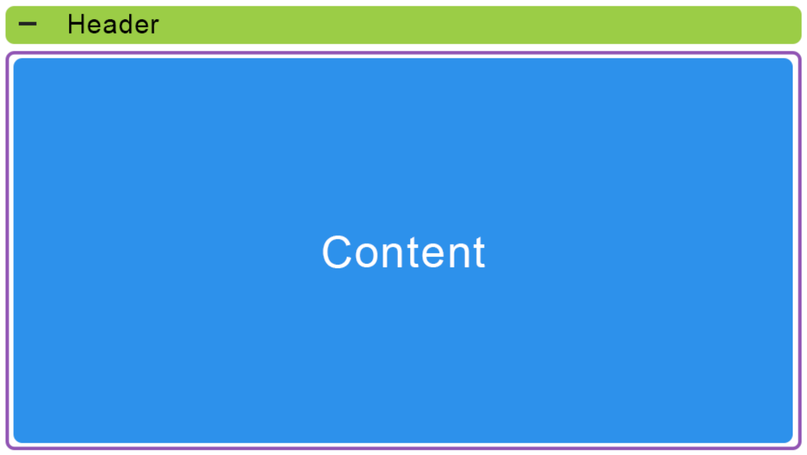
Horizontal Unfolded



Horizontal Folded



Vertical Unfolded



Vertical Folded

Using Wild FoldoutAreas

```
// Always keep a reference to all your FoldoutAreas
FoldoutArea myFoldoutArea;

protected void VoltageInit()
{
    // Initialize the area
    myFoldoutArea = new FoldoutArea(/*Your Settings*/);
}
protected void VoltageGUI()
{
    Constructor.FoldoutAreaStart(myFoldoutArea );
    {
        // Add wild elements to your area
    }
    Constructor.EndArea();
}
```

Using Stored FoldoutAreas

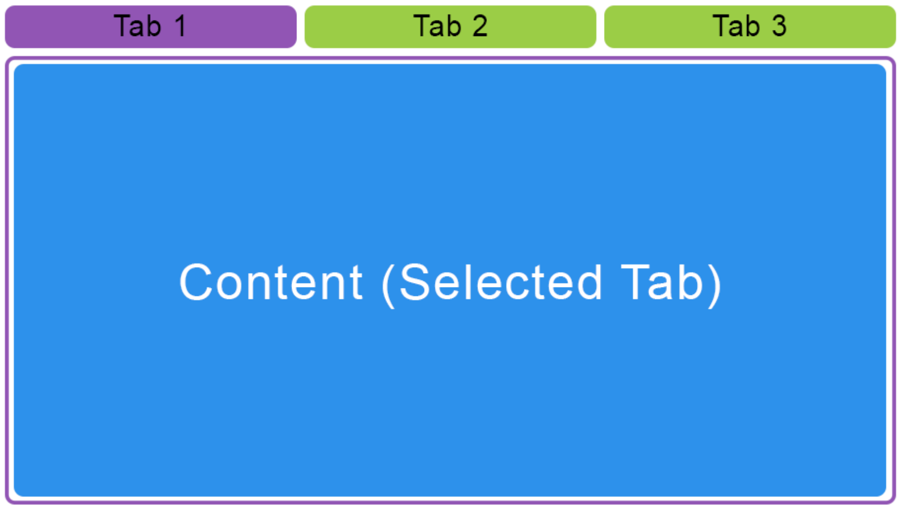
```
// Always keep a reference to all your FoldoutAreas
FoldoutArea myFoldoutArea;

protected void VoltageInit()
{
    // Initialize the area
    myFoldoutArea = new FoldoutArea(/*Your Settings*/);
    // Start the constructor for the area
    Constructor.StartStoredConstructor(myScrollArea);
    {
        // Add stored elements to your area
    }
    Constructor.EndStoredConstructor();
}

protected void VoltageGUI()
{
    // Use your area
    Constructor.ScrollAreaStart(myScrollArea);
    {
        // Add wild elements to your area
        // They will be displayed after the stored elements
    }
    Constructor.EndArea();
}
```

Tab Areas

Tab areas are a quick way to implement tabs with Voltage. They function the same way as any other tabs you are use to. As with any special area the default Voltage area for each tab is a weight area. If you wish to replace it, you can add the area you want inside and that should do it.

Each tab has their own area, so you should always fill them even if they aren’t the active tab. For better performance try to use stored elements whenever possible. The active tab, and tab switching is handled internally so you don’t have to worry about it.

Vertical Tab Area. Active tab in violet.



Using Wild TabAreas

```
// Always keep a reference to all your TabAreas
TabArea myTabArea;

protected void VoltageInit()
{
    // Initialize the area
    myTabArea = new TabArea(/*Your Settings*/);

    // Add 3 tabs
    myTabArea.AddTab(/*Your Settings for the first tab*/);
    myTabArea.AddTab(/*Your Settings for the second tab*/);
    myTabArea.AddTab(/*Your Settings for the third tab*/);
}
protected void VoltageGUI()
{
    // Use your area
    Constructor.TabAreaStart(myTabArea);
    {
        // Add wild elements to your first tab
    }
    Constructor.NextTab();
    {
        // Add wild elements to your second tab
    }
    Constructor.NextTab();
    {
        // Add wild elements to your third tab
    }
    Constructor.EndArea();
}
```

Using Stored TabAreas

```
//Always keep a reference to all your TabAreas
TabArea myTabArea;

protected void VoltageInit()
{
    //Initialize the area
    myTabArea = new TabArea(/*Your Settings*/);

    //Add 3 tabs
    myTabArea.AddTab(/*Your Settings for the first tab*/);
    myTabArea.AddTab(/*Your Settings for the second tab*/);
    myTabArea.AddTab(/*Your Settings for the third tab*/);

    //Start the constructor for the area
    Constructor.StartStoredConstructor(myTabArea);
    {
        //Add stored elements to your first tab
    }
    Constructor.NextTab();
    {
        //Add stored elements to your second tab
    }
    Constructor.NextTab();
    {
        //Add stored elements to your third tab
    }
    Constructor.EndStoredConstructor();
}

protected void VoltageGUI()
{
    // Use your area
    Constructor.TabAreaStart(myTabArea);
    {
        // Add wild elements to your first tab
        // They will be displayed after the stored elements
    }
    Constructor.NextTab();
    {
        // Add wild elements to your second tab
        // They will be displayed after the stored elements
    }
    Constructor.NextTab();
    {
        // Add wild elements to your third tab
        // They will be displayed after the stored elements
    }
    Constructor.EndArea();
}
```

```
// Always keep a reference to all your TabAreas
TabArea myTabArea;

protected void VoltageInit()
{
    // Initialize the area
    myTabArea = new TabArea(/*Your Settings*/);
    // Add 3 tabs
    myTabArea.AddTab(/*Your Settings for the first tab*/);
    myTabArea.AddTab(/*Your Settings for the second tab*/);
    myTabArea.AddTab(/*Your Settings for the third tab*/);

    // Start the constructor for the area
    Constructor.StartStoredConstructor(myTabArea);
    {
        // Add stored elements to your first tab
    }
    Constructor.NextTab();
    {
        // Add stored elements to your second tab
    }
    Constructor.NextTab();
    {
        // Add stored elements to your third tab
    }
    Constructor.EndStoredConstructor();
}

protected void VoltageGUI()
{
    // Use your area
    Constructor.TabAreaStart(myTabArea);
    {
        // Or do not add any wild elements
    }
    Constructor.EndArea();
}
```

Common Concepts

Voltage Steps

Voltage Init

Voltage Init is the first step on our UI construction. It will be called on the script instantiation. For a window that would be the first time you open it, or when the editor recompiles your scripts.

On this step you should initialize your variables, areas and elements. Stored constructors should also be used on the Init phase.

Voltage GUI

Voltage GUI is the second step, and is where the layout is constructed. In here you should add your wild areas and elements. You should also start your stored areas in here, we will see a more detail explanation later.

This phase will only put the layouts in a sort of queue to be later drawn.

Voltage Draw (Internal)

This step is internal, for now you shouldn’t worry about this. On this step the elements are drawn and the values of the classic OnGUI dependent elements are updated. All the calculations on where to draw your elements are performed here.

Voltage Event (Internal)

The event step is where the Voltage Event system kicks in and the pure voltage elements are updated according to the actions performed to the interface. This phase is also internal.

Voltage Serialization

This is the third, and final step that you should worry about. This is called after the interface has already been drawn and updated, so all your elements values will be up to date here.

Contrary to it’s name, nothing actually happens here unless you write it. This step is purely for you, in here you can serialize your data or perform any other operation with the updated values of your fields.

Stored vs Wild

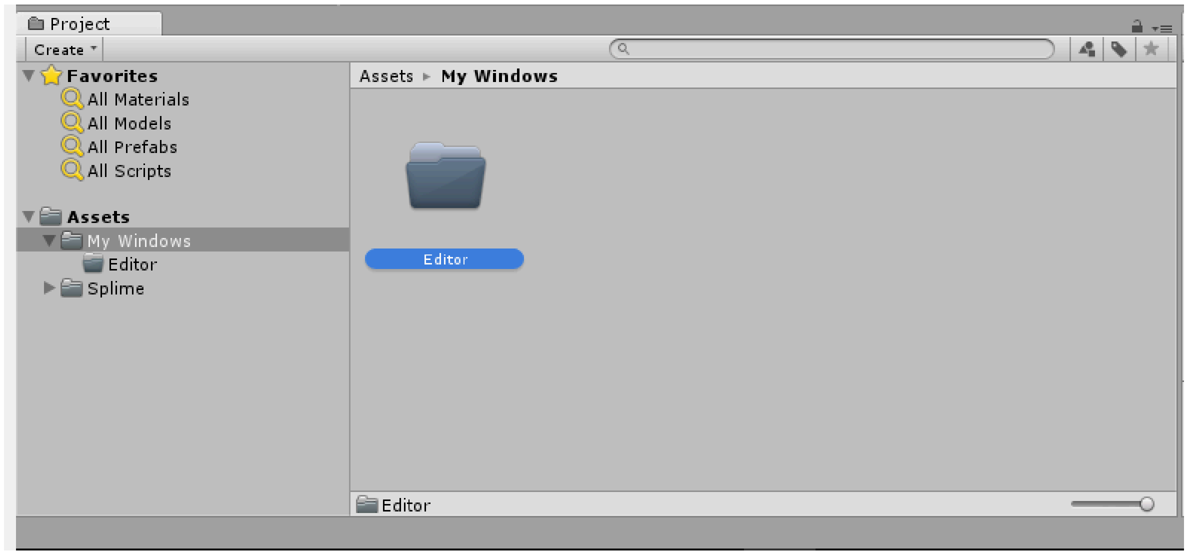
On simple words, stored elements and areas are build on the Init phase and shown on the GUI phase, while wild elements are build and shown on the GUI phase. This means that stored elements are only instantiated once, and wild elements are being instantiated and destroyed once per GUI update (A lot).

Usually there won’t be any problem with building your interface on the VoltageGUI (meaning building a wild interface), but if your editor is starting to tank consider moving a few parts of the interface to VoltageInit instead.

My First Voltage Window

Creating a Voltage Window

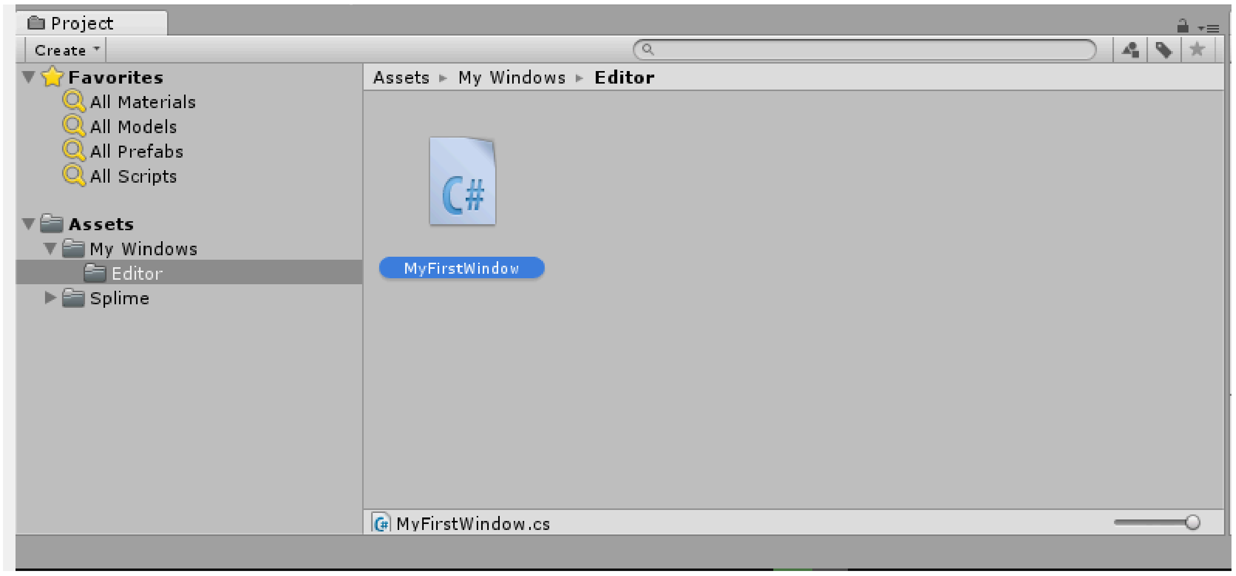
Now that we know how Voltage steps work, we can start with our window. Once you have imported Voltage into your project, the first thing we want to do is to create a folder named “Editor”. You can add this folder anywhere you want as long as it is under the project’s “Assets” folder.



The way this works, is that Unity will use all files under all folders named “Editor” only on the editor. This means two things will happen, the scripts under an “Editor” folder won’t be build with your release and they will be able to access the *UnityEditor* library in addition to the *UnityEngine* library.

The *UnityEditor* library is the one that holds all you need to create editor extensions and functions. In this case Voltage is using this library to improve upon the OnGUI methods used to build windows and editors.

Now that we have our *Editor* folder we can create our window script. We will call it *MyFirstWindow* and is going to be just a new c# script.



Once you open it, you should see something like this.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MyFirstWindow : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

To make MyFirstWindow into an actual window we need to inherit from VoltageWindow.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MyFirstWindow : VoltageWindow {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

If you have already tried this, you will get the error:

‘The type or namespace name `VoltageWindow' could not be found. Are you missing `Voltage' using directive?’

To solve this simply add a **using Voltage;** line at the top of the document to tell this class that it will use the *Voltage* library. We will also use this chance to add **using UnityEditor;** and to clean the methods of our window.

```
using UnityEngine;
using UnityEditor;
using Voltage;

public class MyFirstWindow : VoltageWindow
{

}
```

Now our window is an actual window, but we will get three new errors.

`MyFirstWindow' does not implement inherited abstract member `Voltage.VoltageWindow.VoltageInit()'

`MyFirstWindow' does not implement inherited abstract member `Voltage.VoltageWindow.VoltageGUI()'

`MyFirstWindow' does not implement inherited abstract member `Voltage.VoltageWindow.VoltageSerialization()'

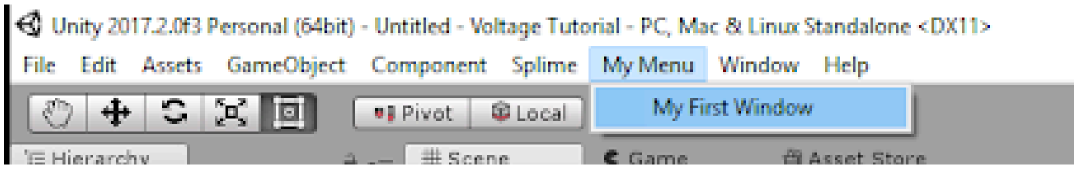
On any Voltage window or editor you need to implement these three abstract methods, even if you won't use them. Most IDEs will prompt you to add them automatically, you can also write them yourself. Either way you should end up with something like this.

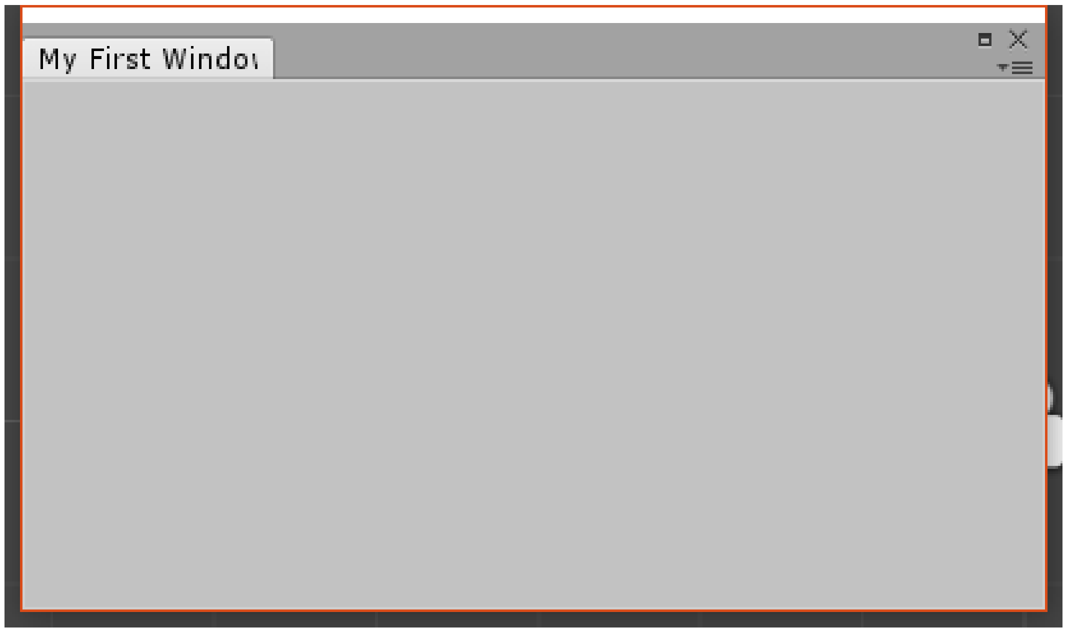
```
using UnityEngine;
using UnityEditor;
using Voltage;

public class MyFirstWindow : VoltageWindow
{
    protected override void VoltageGUI()
    {

    }

    protected override void VoltageInit()
    {
    }

    protected override void VoltageSerialization()
    {

    }
}
```

MyFirstWindow is now ready to be shown, but as it is you won't be able to open it from anywhere. To open this window we are going to add a new public static function to our window responsible for creating, and showing it on the editor, and a menu item to Unity's interface to call this function.

```
using UnityEngine;
using UnityEditor;
using Voltage;

public class MyFirstWindow : VoltageWindow
{
    // To add a menu item that will call a specific method you need to add
the next macro, and make the method public and static.
    [MenuItem("My Menu/My First Window")]
    public static void OpenWindow()
    {
       // VoltageWindow.GetWindow<T> will retrieve an existing window of
type T (in this case MyFirstWindow) or create a new one if none exist.
       // Pass the name you want your window to display as a parameter.
       VoltageWindow window = VoltageWindow.GetWindow<MyFirstWindow>("My
First Window");

       // Once a window is retrieved, show it.
       window.Show();
    }

    protected override void VoltageGUI()
    {

    }

    protected override void VoltageInit()
    {
    }

    protected override void VoltageSerialization()
    {

    }
}
```

Now you should see the menu, and you will be able to open the window. Congrats!


Unity 2017.2.0f3 Personal (64bit) - Untitled - Voltage Tutorial - PC, Mac & Linux Standalone <DX11>
File Edit Assets GameObject Component Splime My Menu Window Help
My First Window
Pivot Local
Hierarchy
Scene
Game
Asset Store



Building the Layout

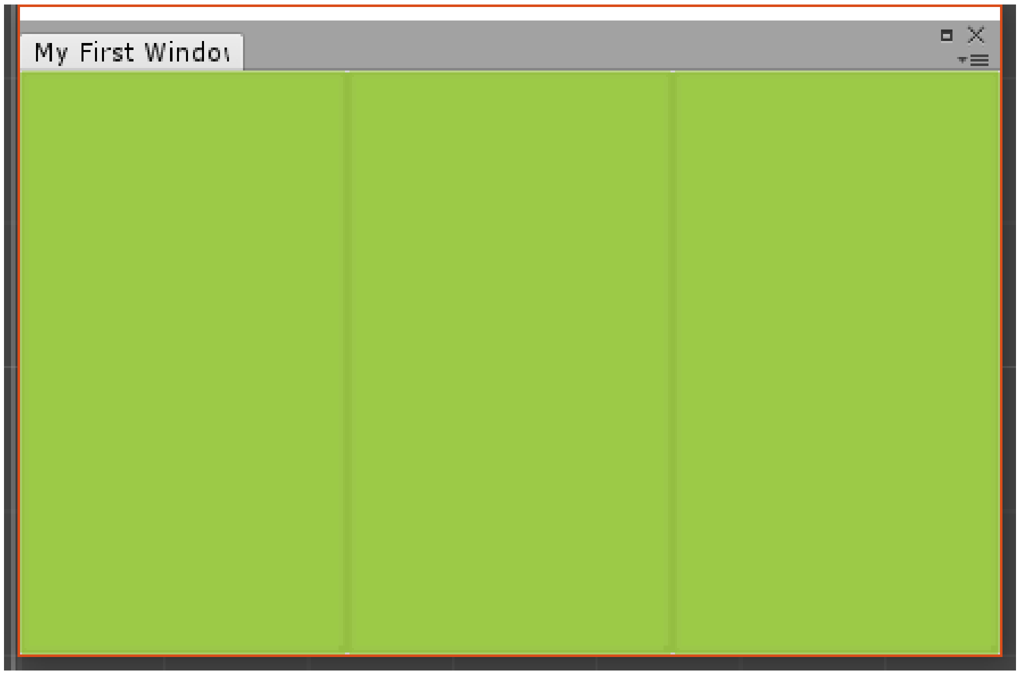
Now that we have created our window, we can start adding things to it (I will assume you have already read the **Areas** section of this guide). By default every Voltage Window starts with a weight area, to understand how this work we will add three new areas with different weights. Some areas can be added from the Constructor property without declaring them first, we will use another three weight areas with a *BoxGreen* style so we can see them.

To use styles in Voltage, simply call the static methods on the ***Styles*** class (remember to use Voltage namespace). For retrieving a style you can call:

```
//For styles on the VoltageStyles bundle
Styles.GetStyle("Style Name");

//For styles on your own bundles
Styles.GetStyle("Bundle Name", "Style Name");
```

On our window it should look like this.

```
using UnityEngine;
using UnityEditor;
using Voltage;

public class MyFirstWindow : VoltageWindow
{
    [MenuItem("My Menu/My First Window")]
    public static void OpenWindow()
    {
       VoltageWindow window = VoltageWindow.GetWindow<MyFirstWindow>("My
First Window");
       window.Show();
    }

    protected override void VoltageInit()
    {

    }

    protected override void VoltageGUI()
    {
       // Be sure to close this areas or else they will end up inside one
another
       Constructor.WeightAreaStart(Styles.GetStyle("BoxGreen"));
       Constructor.EndArea();

       Constructor.WeightAreaStart(Styles.GetStyle("BoxGreen"));
       Constructor.EndArea();

       Constructor.WeightAreaStart(Styles.GetStyle("BoxGreen"));
       Constructor.EndArea();
    }
    protected override void VoltageSerialization()
    {

    }
}
```

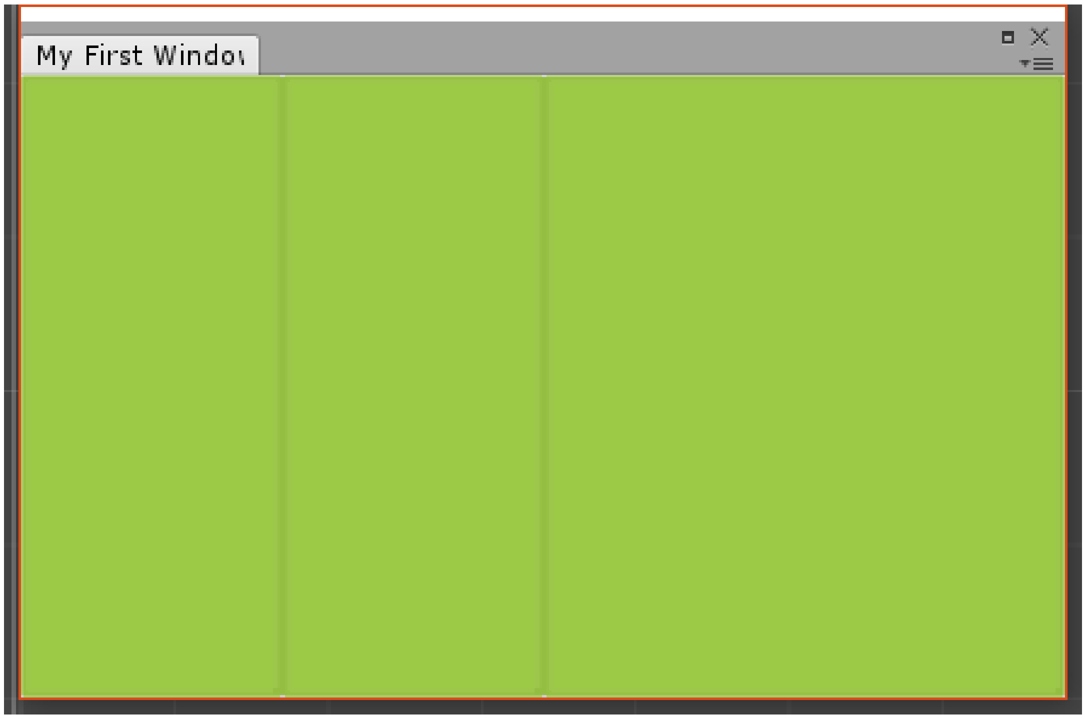


You can see the three areas one next to the other with no space in between. By default all elements and areas start with a weight of 1, so in this case the three areas are assigned the same amount of space.

Let’s change that and assign a weight of 2 to the last area. To do so, we will pass a new ElementSettings struct to the WeightAreaStart function.

```
using UnityEngine;
using UnityEditor;
using Voltage;

public class MyFirstWindow : VoltageWindow
{
    [MenuItem("My Menu/My First Window")]
    public static void OpenWindow()
    {
        VoltageWindow window = VoltageWindow.GetWindow<MyFirstWindow>("My
First Window");
```

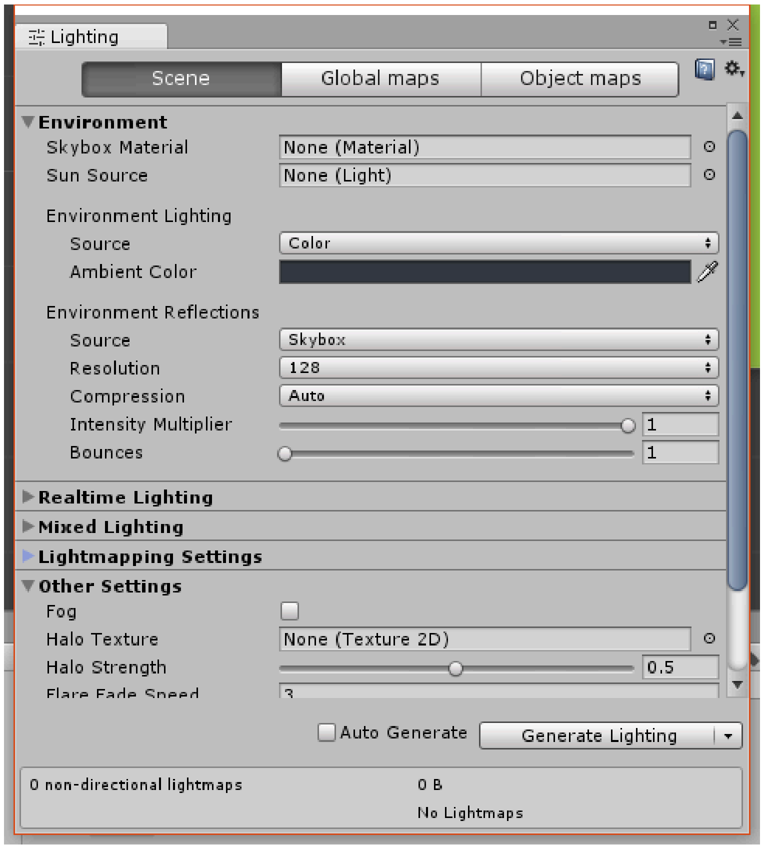
```
        window.Show();
    }

    protected override void VoltageInit()
    {

    }

    protected override void VoltageGUI()
    {
        Constructor.WeightAreaStart(Styles.GetStyle("BoxGreen"));
        Constructor.EndArea();

        Constructor.WeightAreaStart(Styles.GetStyle("BoxGreen"));
        Constructor.EndArea();

        Constructor.WeightAreaStart(new ElementSettings(2),
Styles.GetStyle("BoxGreen"));
        Constructor.EndArea();
    }

    protected override void VoltageSerialization()
    {

    }
}
```



You can see that the last area is now taking half the space on the window. This happens because the total added weight of the three areas is 4 (1+1+2), and since the last area has a weight of 2 (half the total weight) it takes half the space.

The Real Layout

We will scratch what we just did, and start making a window that we can use. We will replicate the layout of the Lighting window of Unity. We won't get into serialization or the functions behind it though, just the layout.



Skybox Material None (Material)
Sun Source None (Light)

Environment Lighting
Source Color
Ambient Color

Environment Reflections
Source Skybox
Resolution 128
Compression Auto
Intensity Multiplier 1
Bounces 1

Fog
Halo Texture None (Texture 2D)
Halo Strength 0.5
Flare Fade Speed 3

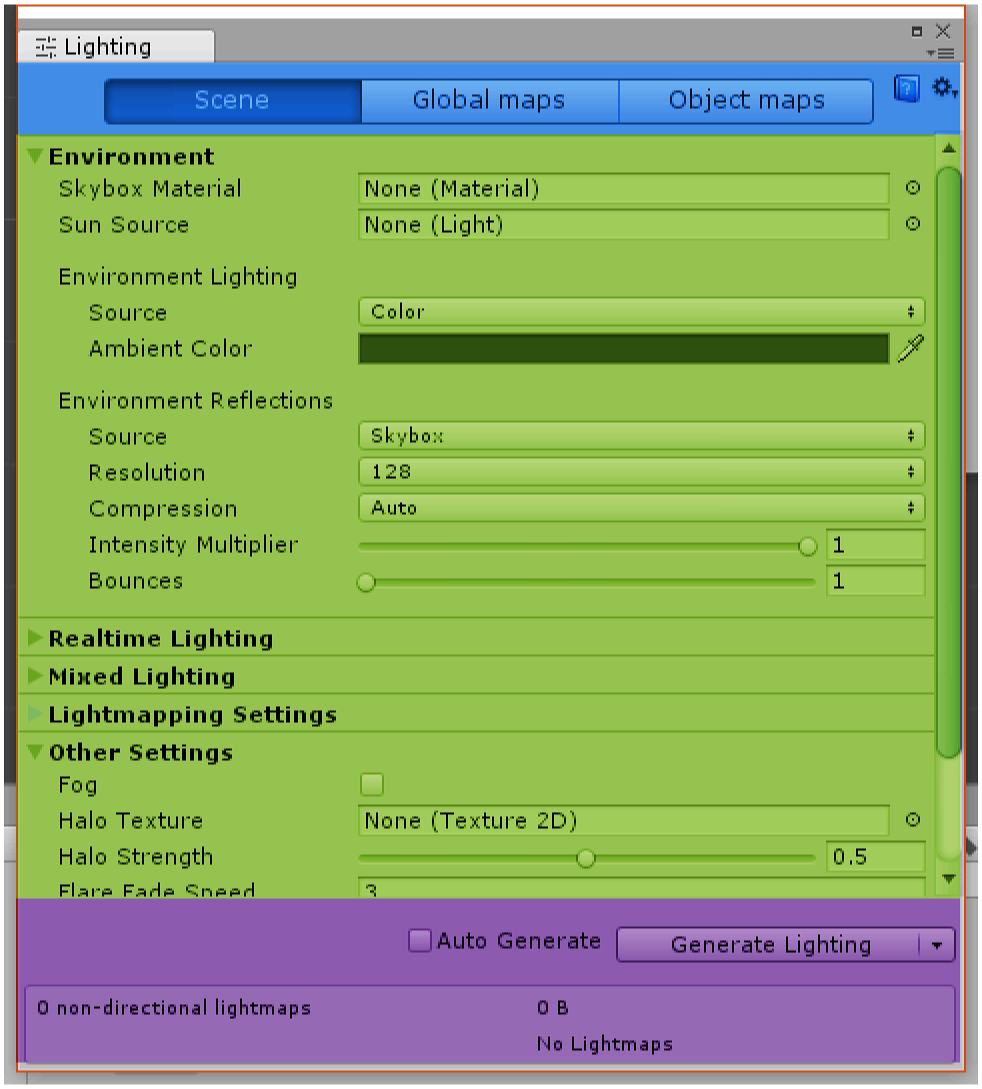
0 non-directional lightmaps 0 B
No Lightmaps

This is a complex layout with a lot of nested areas. So lets start easy. The base area we will need is a vertical Stream Area, which will provide us with the top to bottom structure that this window has. We will use curly brackets for better code structure and auto indentation. Feel free to use a vertical Weight Area instead to understand the difference between them.

Previously I said that by default every Voltage Window starts with a Weight Area, this can be changed, by adding a Stream Area inside of it, all our elements will behave according to this new Stream Area. And since the base Weight Area only has one element, this Stream area won't be sharing space with anyone else like we saw before and will take the whole window for itself.

```
using UnityEngine;
using UnityEditor;
using Voltage;

public class MyFirstWindow : VoltageWindow
{
    [MenuItem("My Menu/My First Window")]
    public static void OpenWindow()
    {
       VoltageWindow window = VoltageWindow.GetWindow<MyFirstWindow>("My
First Window");
        window.Show();
    }

    protected override void VoltageInit()
    {

    }

    protected override void VoltageGUI()
    {
        // --Start our Stream Area--
        Constructor.StreamAreaStart();
        {

        }
        Constructor.EndArea();
    }

    protected override void VoltageSerialization()
    {

    }
}
```

Now we can divide the window in three primary sections. The first one is the header where the tabs are, the second one is the body where the settings are sitting, and the third is the footer where the Generate Lighting button sits.



Lighting

Scene | Global maps | Object maps

Environment

Skybox Material: None (Material)

Sun Source: None (Light)

Environment Lighting

Source: Color

Ambient Color

Environment Reflections

Source: Skybox

Resolution: 128

Compression: Auto

Intensity Multiplier: 1

Bounces: 1

Realtime Lighting

Mixed Lighting

Lightmapping Settings

Other Settings

Fog

Halo Texture: None (Texture 2D)

Halo Strength: 0.5

Flare Fade Speed: 3

Auto Generate | Generate Lighting

0 non-directional lightmaps 0 B

No Lightmaps

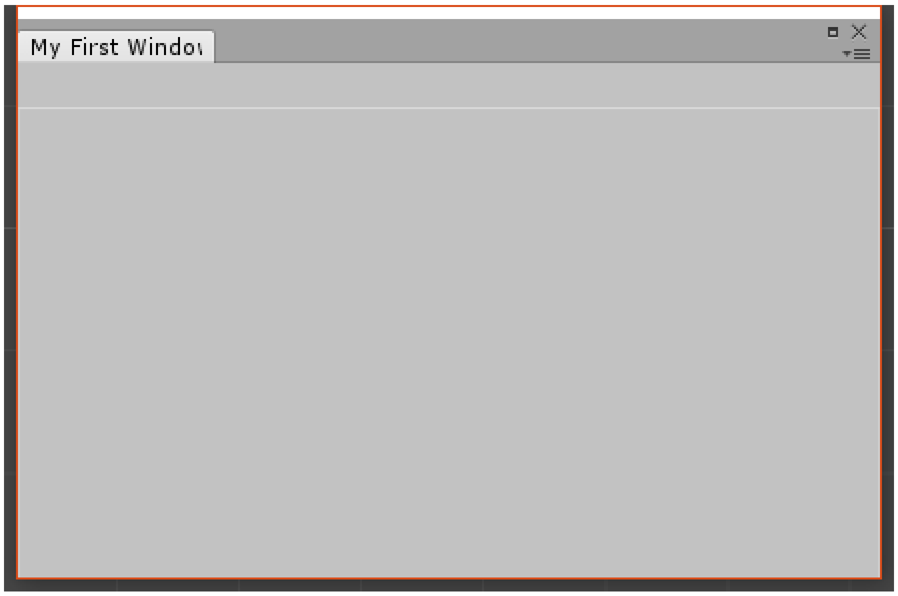
On Voltage we can combine the top two areas (Tabs and Settings) into just one, A Tab Area. Tab areas in Voltage come with their header and their body, which will automatically change according to the active tab.

We need to do three things to add our Tab area.

1. We need to declare the area as a private member of our window. Tab areas need to be declared so they can maintain the value of the active area. You can't add them directly from the Constructor the same way you would add a Stream Area or a Weight Area.
2. Initialize the area on VoltageInit. Since TabAreas need some extra initialization apart of their constructor, initializing them on VoltageInit will make a more comprehensive code, more later.
3. Use the Constructor to add this area to the layout on VoltageGUI.

```
using UnityEditor;
using Voltage;

public class MyFirstWindow : VoltageWindow
{
    // --Add the tab as a private member--
    TabArea lightTab;

    [MenuItem("My Menu/My First Window")]
    public static void OpenWindow()
    {
       VoltageWindow window = VoltageWindow.GetWindow<MyFirstWindow>("My
First Window");
       window.Show();
    }

    protected override void VoltageInit()
    {
       // --Initialize the area--
       lightTab = new TabArea();
    }

    protected override void VoltageGUI()
    {
        Constructor.StreamAreaStart();
        {
            // --Add your area to the layout--
            Constructor.TabAreaStart(lightTab);
            {

            }
            Constructor.EndArea();
        }
        Constructor.EndArea();
    }

    protected override void VoltageSerialization()
    {

    }
}
```

You will end up with a window looking like this, and a few errors.

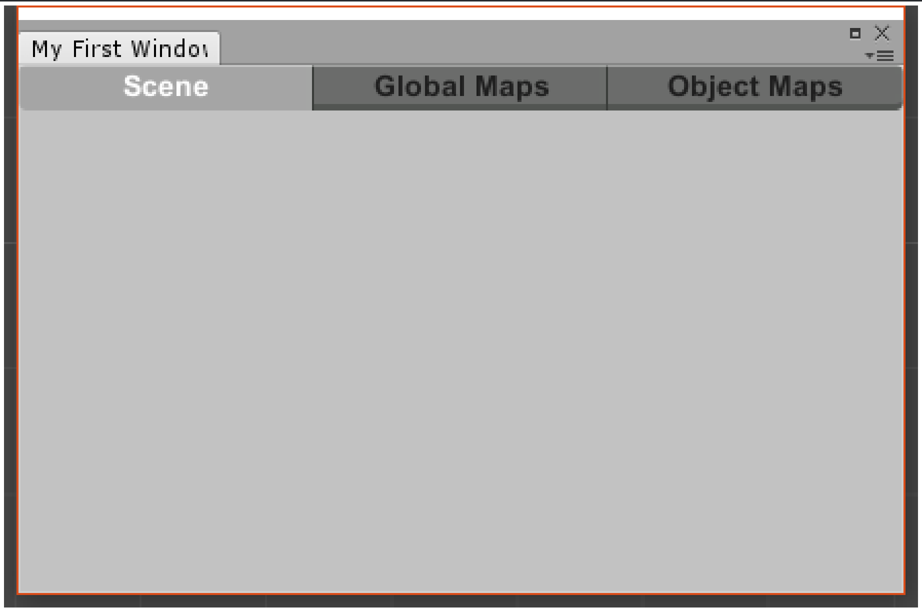


The thing with tab areas is that you have to add the tabs you want it to have. To add a tab to a TabArea use AddTab:

```
myTabArea.AddTab("Tab Name");
```

Now we add our three tabs to our Tab Area after it’s constructor on VoltageInit.

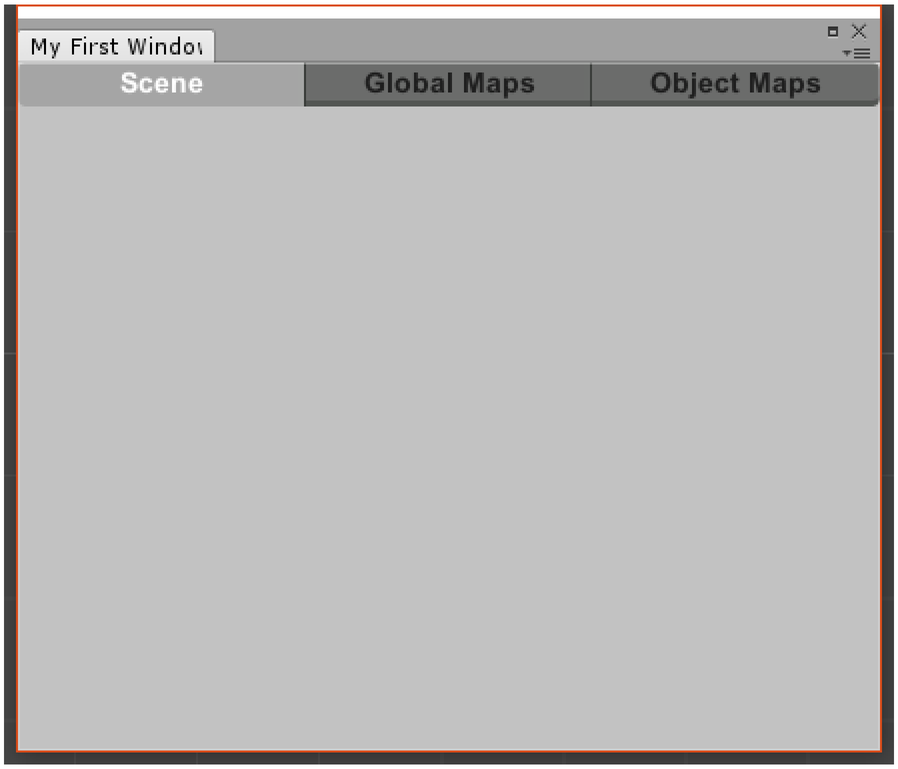
```
using UnityEngine;
using UnityEditor;
using Voltage;

public class MyFirstWindow : VoltageWindow
{

    TabArea lightTab;

    [MenuItem("My Menu/My First Window")]
    public static void OpenWindow()
    {
        VoltageWindow window = VoltageWindow.GetWindow<MyFirstWindow>("My 
First Window");
        window.Show();
    }

    protected override void VoltageInit()
    {
       lightTab = new TabArea();

       // --Add the three tabs--
       lightTab.AddTab("Scene");
       lightTab.AddTab("Global Maps");
       lightTab.AddTab("Object Maps");
    }

    protected override void VoltageGUI()
    {
       Constructor.StreamAreaStart();
       {
             Constructor.TabAreaStart(lightTab);
             {

             }
             Constructor.EndArea();
       }
       Constructor.EndArea();
    }

    protected override void VoltageSerialization()
    {

    }
}
```



To better see how our layout is being distributed we will start adding BoxGreen and BoxPurple styles to areas. To add a style to a declared area you have to do it from the constructor. But for TabAreas you will have to add a new area with a style to each tab. Lets add a Stream Area since we will need it later for our fields.

```
using UnityEngine;
using UnityEditor;
using Voltage;

public class MyFirstWindow : VoltageWindow
{

    TabArea lightTab;

    [MenuItem("My Menu/My First Window")]
    public static void OpenWindow()
    {
       VoltageWindow window = VoltageWindow.GetWindow<MyFirstWindow>("My
First Window");
       window.Show();
    }

    protected override void VoltageInit()
    {
       lightTab = new TabArea();

       lightTab.AddTab("Scene");
       lightTab.AddTab("Global Maps");
       lightTab.AddTab("Object Maps");
    }

    protected override void VoltageGUI()
    {
       Constructor.StreamAreaStart();
       {
             Constructor.TabAreaStart(lightTab);
             {
                   // --Add a new area

Constructor.StreamAreaStart(Styles.GetStyle("BoxGreen"));
                   {
                   }
```

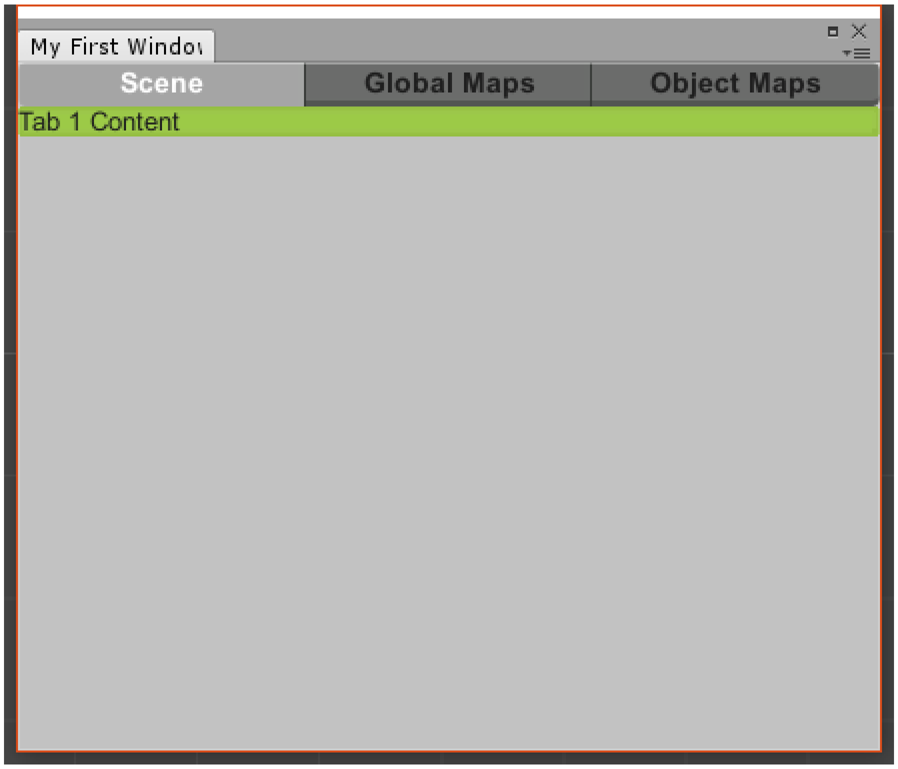
```
                    Constructor.EndArea();
                }
                Constructor.EndArea();
            }
            Constructor.EndArea();
        }

        protected override void VoltageSerialization()
        {

        }
    }
```



Now, you can’t see the green box because of how a stream area works.

Remember that a stream area will draw the elements and areas inside with the minimal size they can have (since it’s a vertical Stream, it will draw them with the minimal height). And since our tab doesn’t have anything inside, it’s minimal height is 0. Add a new label to the bottom of the hierarchy to see what’s going on (The last stream area inside our tab area).

```
using UnityEngine;
using UnityEditor;
using Voltage;
public class MyFirstWindow : VoltageWindow
{

    TabArea lightTab;

    [MenuItem("My Menu/My First Window")]
    public static void OpenWindow()
    {
       VoltageWindow window = VoltageWindow.GetWindow<MyFirstWindow>("My
First Window");
       window.Show();
    }

    protected override void VoltageInit()
    {
       lightTab = new TabArea();

       lightTab.AddTab("Scene");
       lightTab.AddTab("Global Maps");
       lightTab.AddTab("Object Maps");
    }

    protected override void VoltageGUI()
    {
       Constructor.StreamAreaStart();
       {
             Constructor.TabAreaStart(lightTab);
             {
Constructor.StreamAreaStart(Styles.GetStyle("BoxGreen"));
                   {
                         // -- Add a label --
                         Constructor.Label("Tab 1 Content");
                   }
                   Constructor.EndArea();
             }
             Constructor.EndArea();
       }
       Constructor.EndArea();
    }
    protected override void VoltageSerialization()
    {

    }
}
```



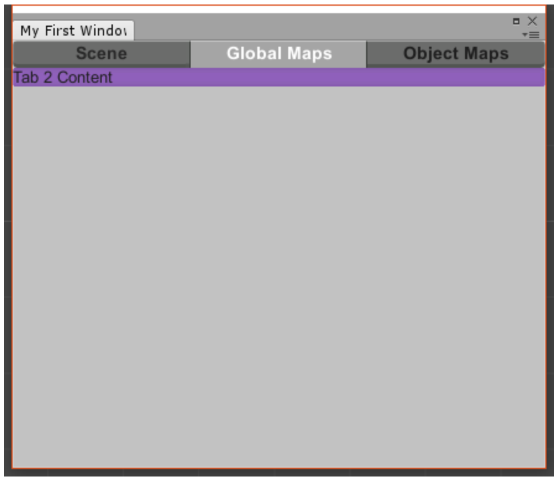
Tab 1 Content

If you change tabs you will see that the other two don’t have anything on them. To add elements to the other tabs you need to use:

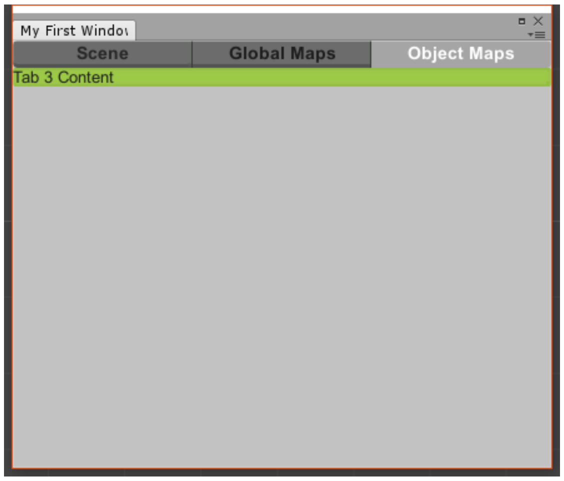
```
Constructor.NextTab();
```

Add new content to the other areas.

```
using UnityEngine;
using UnityEditor;
using Voltage;

public class MyFirstWindow : VoltageWindow
{

    TabArea lightTab;

    [MenuItem("My Menu/My First Window")]
    public static void OpenWindow()
    {
       VoltageWindow window = VoltageWindow.GetWindow<MyFirstWindow>("My
First Window");
       window.Show();
    }

    protected override void VoltageInit()
    {
       lightTab = new TabArea();

       lightTab.AddTab("Scene");
       lightTab.AddTab("Global Maps");
       lightTab.AddTab("Object Maps");
    }

    protected override void VoltageGUI()
    {
       Constructor.StreamAreaStart();
       {
             Constructor.TabAreaStart(lightTab);
             {

Constructor.StreamAreaStart(Styles.GetStyle("BoxGreen"));
                   {
                         Constructor.Label("Tab 1 Content");
                   }
                   Constructor.EndArea();
             }
             // --Use NextTab to change the Constructor focus to the next
tab--
             Constructor.NextTab();
             {

Constructor.StreamAreaStart(Styles.GetStyle("BoxPurple"));
                   {
                         Constructor.Label("Tab 2 Content");
                   }
                   Constructor.EndArea();
             }
             // --Use NextTab to change the Constructor focus to the next
tab--
             Constructor.NextTab();
             {

Constructor.StreamAreaStart(Styles.GetStyle("BoxGreen"));
                   {
                         Constructor.Label("Tab 3 Content");
                   }
                   Constructor.EndArea();
             }
             Constructor.EndArea();
       }
       Constructor.EndArea();
    }

    protected override void VoltageSerialization()
    {

    }
}
```



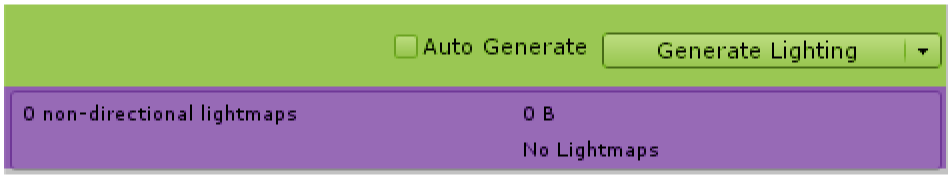
Tab 2 Content



Tab 3 Content

Now that we know how to populate our tabs, let’s add the footer. We will have to add three fields, a toggle, a label and a button. Labels can be added directly from the constructor, but other fields need to be declared as members of the window.

To achieve both parts of the footer we will need a vertical Stream Area to wrap it, and two horizontal Stream areas, one for the section with the button, and another one for the info section.



0 non-directional lightmaps

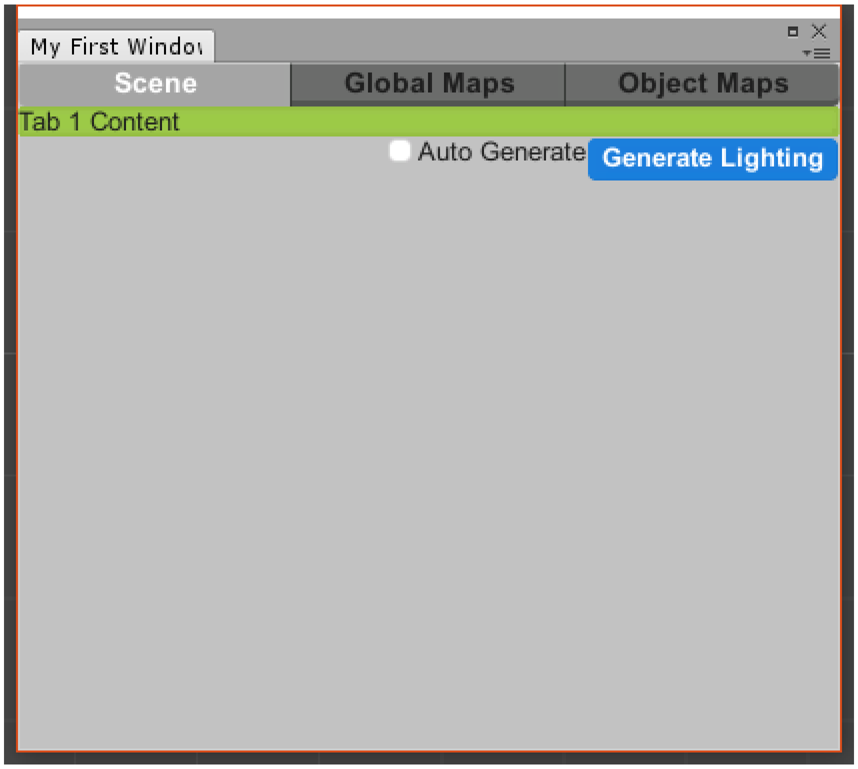
0 B

No Lightmaps

We will start with the top part of the footer.

```
using UnityEditor;
using Voltage;

public class MyFirstWindow : VoltageWindow
{

    TabArea lightTab;
    // -- Declare your fields --
    VoltageToggle autoGenerate;
    VoltageButton generate;

    [MenuItem("My Menu/My First Window")]
    public static void OpenWindow()
    {
       VoltageWindow window = VoltageWindow.GetWindow<MyFirstWindow>("My
First Window");
       window.Show();
    }

    protected override void VoltageInit()
    {
       lightTab = new TabArea();

       lightTab.AddTab("Scene");
       lightTab.AddTab("Global Maps");
       lightTab.AddTab("Object Maps");
       // -- Initialize your fields --
       autoGenerate = new VoltageToggle(false);
       // -- We will set the callback method for our button to null, we
won't use it.
       generate = new VoltageButton("Generate Lighting", null);
    }

    protected override void VoltageGUI()
    {
       Constructor.StreamAreaStart();
       {
             Constructor.TabAreaStart(lightTab);
             {

Constructor.StreamAreaStart(Styles.GetStyle("BoxGreen"));
                   {
                         Constructor.Label("Tab 1 Content");
                   }
                   Constructor.EndArea();
             }
             Constructor.NextTab();
             {

Constructor.StreamAreaStart(Styles.GetStyle("BoxPurple"));
                   {
                         Constructor.Label("Tab 2 Content");
                   }
                   Constructor.EndArea();
             }
             Constructor.NextTab();
             {

Constructor.StreamAreaStart(Styles.GetStyle("BoxGreen"));
                   {
                         Constructor.Label("Tab 3 Content");
                   }
                   Constructor.EndArea();
             }
             Constructor.EndArea();

             // -- The Footer should be added after the tab area is closed
             Constructor.StreamAreaStart();
             {
                   Constructor.StreamAreaStart(new
AreaSettings(true,VoltageElementAlignment.BottomRight));
                   {
                         // -- Add your fields and your label --
                         Constructor.Field(autoGenerate);
                         Constructor.Label("Auto Generate");
                         Constructor.Field(generate);
                   }
```

```
            }
            Constructor.EndArea();
        }
        Constructor.EndArea();
    }

    protected override void VoltageSerialization()
    {

    }
}
```

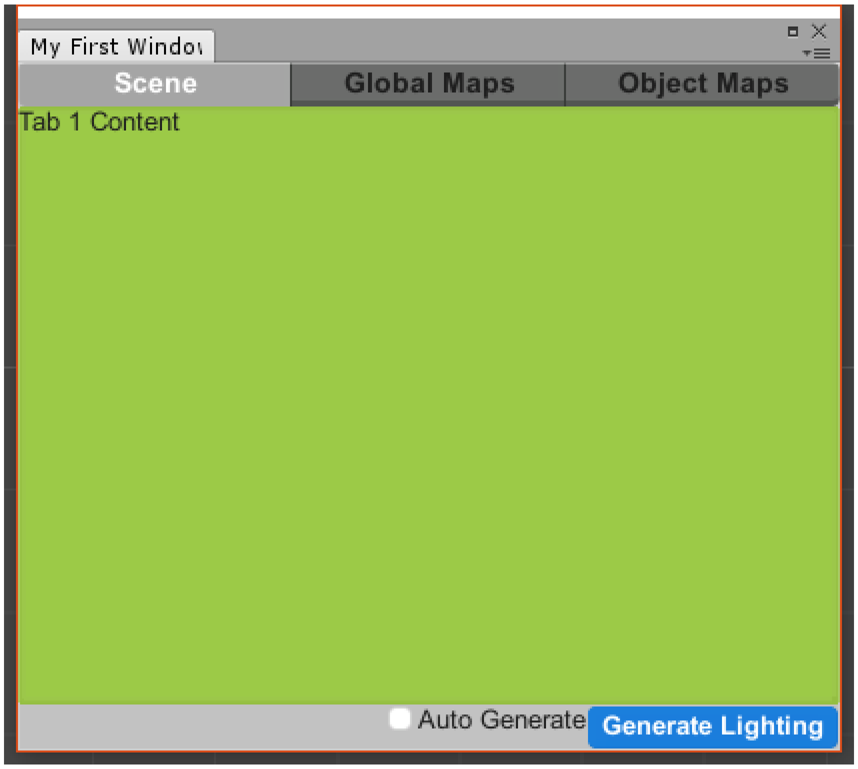


The footer looks fine, but it is not behaving like a footer. We need to push it to the bottom of the page. To do so we can use the flex property. On a stream area, a flex element will take all the free space that the stream area is not using.

The element that should be flex, is the tab area itself. This way the tab area will grow to fill the empty space on the window and the footer will remain the same height. Add a new ElementSettings struct as a parameter on the tab area constructor and set it’s first parameter to true (the constructor overload of ElementSettings with a boolean sets the flex value).

```
using UnityEngine;
using UnityEditor;
using Voltage;

public class MyFirstWindow : VoltageWindow
{

    TabArea lightTab;

    VoltageToggle autoGenerate;
    VoltageButton generate;

    [MenuItem("My Menu/My First Window")]
    public static void OpenWindow()
    {
       VoltageWindow window = VoltageWindow.GetWindow<MyFirstWindow>("My
First Window");
       window.Show();
    }

    protected override void VoltageInit()
    {
       // -- Add a new ElementSettings()
       lightTab = new TabArea(new ElementSettings(true));

       lightTab.AddTab("Scene");
       lightTab.AddTab("Global Maps");
       lightTab.AddTab("Object Maps");

       autoGenerate = new VoltageToggle(false, new ElementSettings(new
Vector2(16f, 16f)));

       generate = new VoltageButton("Generate Lighting", null);
    }

    protected override void VoltageGUI()
    {
       Constructor.StreamAreaStart();
       {
             Constructor.TabAreaStart(lightTab);
             {

Constructor.StreamAreaStart(Styles.GetStyle("BoxGreen"));
```

```
                    {
                          Constructor.Label("Tab 1 Content");
                    }
                    Constructor.EndArea();
              }
              Constructor.NextTab();
              {

Constructor.StreamAreaStart(Styles.GetStyle("BoxPurple"));
                    {
                          Constructor.Label("Tab 2 Content");
                    }
                    Constructor.EndArea();
              }
              Constructor.NextTab();
              {

Constructor.StreamAreaStart(Styles.GetStyle("BoxGreen"));
                    {
                          Constructor.Label("Tab 3 Content");
                    }
                    Constructor.EndArea();
              }
              Constructor.EndArea();

              Constructor.StreamAreaStart();
              {
                    Constructor.StreamAreaStart(new
AreaSettings(true,VoltageElementAlignment.BottomRight));
                    {
                          Constructor.Field(autoGenerate);
                          Constructor.Label("Auto Generate");
                          Constructor.Field(generate);
                    }
              }
              Constructor.EndArea();
        }
        Constructor.EndArea();
     }

     protected override void VoltageSerialization()
     {
    }
}
```



Now the tab area is pushing the footer to the bottom of the window. This will still be the case even if you change tabs. If you add elements to the footer and it changes in size, or you resize the window the tab area will readjust.

(More to come)

My First Voltage Editor

Until the guide is completed:

- Building editors with Voltage is very similar to building windows. Instead of inheriting from UnityEditor.Editor like you would to create a classic editor, inherit from Voltage.VoltageEditor.
- Please check the included ExampleEditor.cs script under “Assets/Splime/Voltage Framework/Editor/Demos” which is linked to “Assets/Splime/Voltage

Framework/CustomEditorExample.cs” MonoBehaviour. Add this component to an object to see this example working.
- All functions of the Constructor (Areas, fields, labels, etc) that work on a Voltage Window, will also work on a Voltage Editor.
- Instead of starting with a Weight Area like windows do, Editors start with a Stream Area.

My First Voltage Helper

My First Voltage List