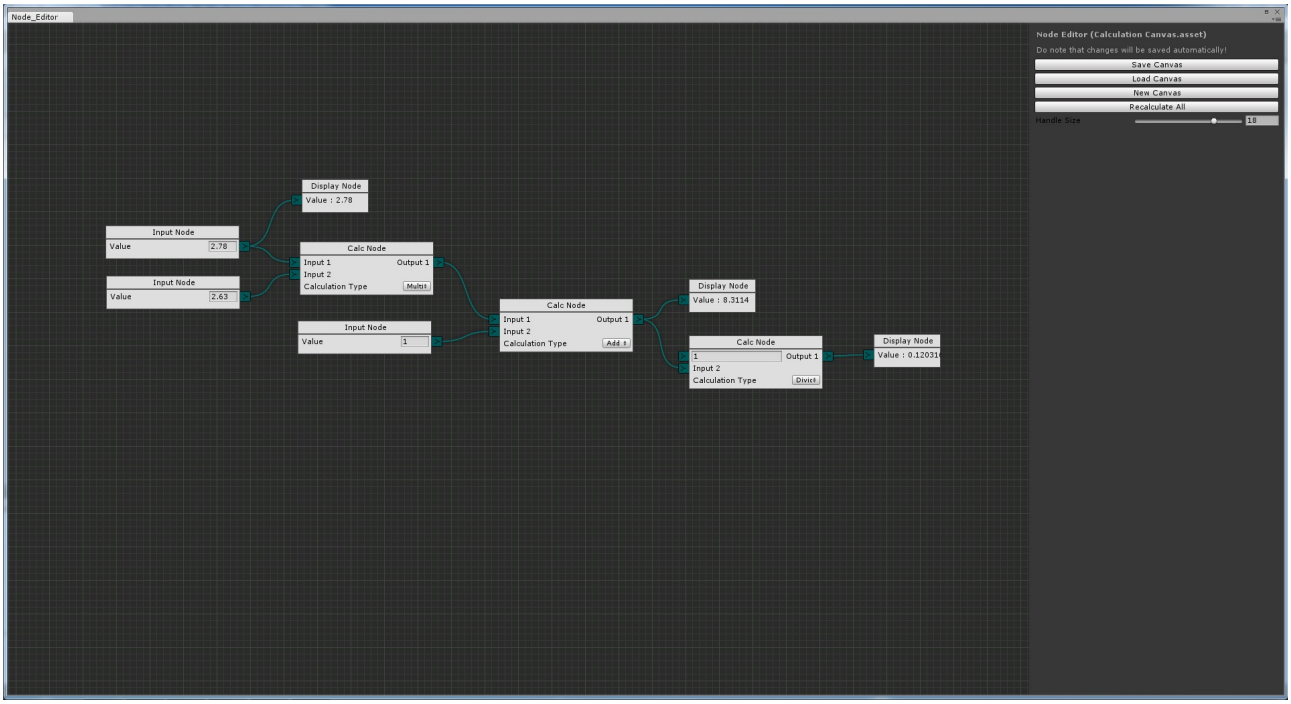
Last updated: 13.06.2015

Node Editor
Documentation

by Seneral



Content

Tell me about...

Connection Type
Node Regristration

General

- Place the Node Editor inside „Assets/Plugins/Node Editor/Editor/“
- You can access it through „Window/Node Editor“
- Use context clicks!
- Example included at „/Saves/Calculation Canvas“

Feature Overview

- Customisable Node Editor GUI
- Reliable calculation system adaptable to most use cases
- Convenient drag'n'drop connection system
- Connections colored by type
- Save/Load system using *ScriptableObject*
- Custom windowing system for most flexibility
- Fully documented, clean code
- Very easy implementation of custom nodes

Code Breakdown

Let's break down the code a bit for better understanding, although the comments should get you started quickly as well.
The code is structured into 5 regions: GUI, GUI Functions, Events, Calculation and Save/Load.

GUI calls to draw every node and it's connection, as well as the side window.

GUI Functions has a lot of helper functions, but also contains *ContextCallback,* the function where you have to register your custom nodes.

Events catches the events and handles the connection system, panning and other shortcuts (for now only 'N': *Navigation* helps you find your way back). It also draws the background (on Repaint).

Calculation contains the functions used by the calculation system, and also allows for manually calculating. Primarily these functions are called in appropriate spots, for example when connecting, breaking connections, changing a value and so forth.

Save/Load has the functions to Save/Load the current Node Canvas in it.

For further details, I recommend you to read the comments. They are the most valueable source of information.

Some notes about features and their implementation:

The **Custom Windowing System** allows for more flexibilty regarding the style and any other modifications. Though, if you want to disable it, I prepared some TODOs you can check to switch over to Unity's default windows.

The **GUI Textures and Styles** are intended to be replaced. I provided some default texture work but depending on your extension, you probably want to replace them.

Implementation of custom Nodes

The Node Editor Framework is designed for you to have the least work possible to create your Node Editor based Editor Extension.
Thus, creating your own nodes is fairly easy. This section lines out which steps you have to perform in order to implement a simple node.

First, create a new class inheriting from *Node* similar to the one below. Give it three functions *Create*, *NodeGUI* and *Calculate* with these signatures and contents:

```
using UnityEngine;
using UnityEditor;
using System.Collections;

[System.Serializable]
public class ExampleNode : Node
{
    public static ExampleNode Create (Rect NodeRect)
    {
        ExampleNode node = CreateInstance<ExampleNode> ();
        return node;
    }

    public override void NodeGUI ()
    {

    }

    public override bool Calculate ()
    {
        return true;
    }
}

```

Node Regristration
Before continuing, let's first register the Node in the *NodeEditor.ContextCallback* function inside the *GUIFunctions* region. Add the code similar to the following inside the switch-statement using a string identifier as a case:

```
case "exampleNode":
    ExampleNode.Create (new Rect (mousePos.x, mousePos.y, 100, 50));
    break;
```

It's called from a context menu, so we have to add it there as well. This context menu can be found in the *Events* region. Look for 'Right click on empty canvas -> Editor Context Click'. Note the last argument has to be your string identifier:

```
menu.AddItem(new GUIContent("Add Example Node"), false, ContextCallback, "exampleNode");
```

Your Node is now registered and can be created like any other with a context click.

We start to fill **Create**:
This static member is the function that inits your node:
Set the name and the rect of it and perform any other setups here as well.
Also, you have to call *node.Init* to initiate the base at the end.

```
public static ExampleNode Create (Rect NodeRect)
{
    ExampleNode node = CreateInstance<ExampleNode> ();

    node.name = "Example Node";
    node.rect = NodeRect;

    node.Init ();
    return node;
}
```

Of course, we need some Inputs and Outputs for our Node. Our *ExampleNode* will have one Input, one Output. Let's add them before initing our base:

```
NodeInput.Create (node, "Value", TypeOf.Float);
NodeOutput.Create (node, "Output val", TypeOf.Float);
```

In both *NodeInput* and *NodeOutput*, you have to pass in the parent node, the name of it (which you may override later in the GUI) and the type of the connection.

Connection Type
The *TypeOf* enum consists of all types that you use in your Extension. To add a type, you have to edit it's source at the top of *Node_Editor.cs:*

```
public enum TypeOf { Float }
```

Then, register it in the *Node_Editor.typeData* Dictionary (→ *Node_Editor.checkInit*), choosing a color and textures:

```
typeData = new Dictionary<TypeOf, TypeData> ()
{
    { TypeOf.Float, new TypeData (Color.cyan, InputKnob, OutputKnob) }
};
```

Those textures will be tinted with the color for you.

The next function, **NodeGUI,** is responsible for drawing your GUI:

```
public override void NodeGUI ()
{
    GUILayout.Label ("This is a custom Node!");
}
```

We want to display our Inputs and Outputs, of course. So we add something like this:

```
GUILayout.Label ("Input");
if (Event.current.type == EventType.Repaint)
    Inputs [0].SetRect (GUILayoutUtility.GetLastRect ());
```

And with the Output respectively. Basically, your have to pass the rect of your Input/Output GUI control to *SetRect*, in order to update the rect of the knob drawn at the left/right edge of the node respectively.

The **Calculate** function performs the calculation related things. It's called whenever a change in values or dependecies was done.
Often functions like *base.allInputsReady*, *base.hasNullInputs* or *base.hasNullInputValues* are used to check whether the node is ready to calculate it's stuff.
Note that, if your nodes is not ready, *Calculate* has to return false, and it will be resolved at a later stage.

For now, *ExampleNode* just calculates the product of Input and 5:

```
public override bool Calculate ()
{
    if (!allInputsReady ())
        return false;
    Outputs [0].value = (float)Inputs [0].connection.value * 5;
    return true;
}
```

Note the way input and output values are accessed.

Conclusion

I really hope Node Editor helps you as I intended it to do. The same goes for this documentation, and as this is my first one I'd really like to hear feedback about it;)

Of course, if you made an Editor Extension using this, we'd really like to hear about it;)

Node Editor is published under the **MIT License** found next to this documentation**:**
In short, you may do anything to Node Editor, but you have to include the license file.

You're also free to contribute your changes and updates to the **GitHub repository** hosted by Baste (Thanks!). I just ask you to not use auto-formatter which will result in everything proposely changed;)

Any major updates on Node Editor will still be published in this **post** though.

Thanks for your interest in Node Editor!
Best regards,

Seneral