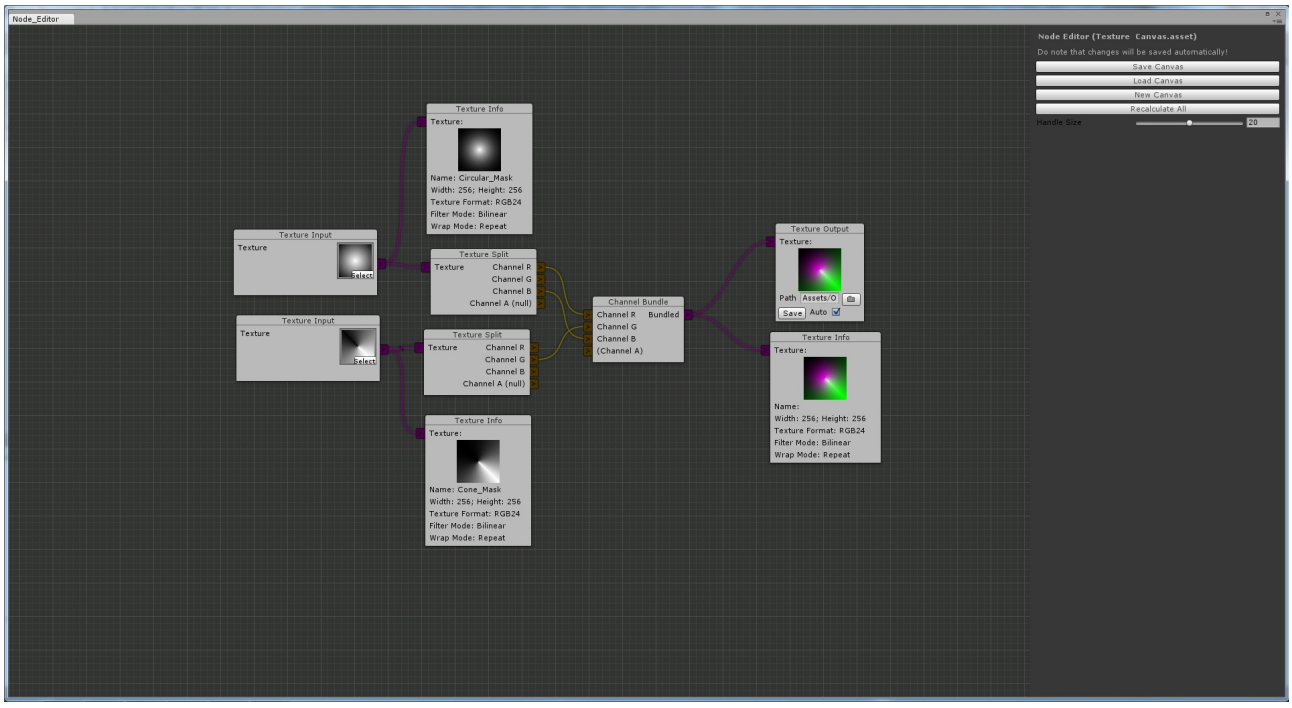
Last updated: 03.08.2015

Node Editor
Documentation

by Seneral



Free Node Editor Framework for Unity3D

Table of Contents

Preface

Node Editor started off as a project I made out of fun, but as it evolved, I decided to put it up on the forums... Now, Node Editor receives so much positive feedback and gains in Contributors/Contributions, that I decided to dedicate alot of my freetime on it and hope that Node Editor will once be the free Node Editor for every occasion, built by the incredible community of Unity itself!
To keep Node Editor evolve that way, there's still alot ahead, but it goes very well;)

First Test:
To get a feel of how it performs, you can test it out using three example nodes I created.
First, make sure Node Editor is placed in the folder „Assets/Plugins/Node Editor/“.
If there are no other errors in your project, you can access Node Editor through „Window/Node Editor“. Open the example „Calculation Canvas.asset“.
Controls: Pan, scroll, drag and use context clicks!
Current Shortcuts include:

- 'N' to Navigate: This helps you find your way back whenever you're lost.
- 'Control' to Snap: Snaps positions of nodes.

After you've got an idea, don't hesitate to start modifiying and improving it! We're happy if you share your improvements with us over GitHub.
We'd also like to see your Editor Extensions you made;)

But let's get going. I hope I can give you a point to start learning how Node Editor works. It's simple to modify, and easily adaptable to most use cases already.

Feature Overview

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

On the first glance, the features resemble a calculation-oriented node editor system. In fact, that was it's original purpose. We are currently working on making it flexible and support the known „Statemachine“ style, too.

Code Breakdown

This section aims to get you a slight overview how Node Editor is structured, although the details can best be explored in the code itself, as nearly every function is well-documented.

First off, we take a look at how the canvases are handled.
You'll find two scripts, *NodeCanvas* and *NodeEditorState*, which both represent a canvas together. *NodeCanvas* holds the actual canvas data, whereas *NodeEditorState* contains everything about the state (view, zoom, …). This enables for multiple views of the same Canvas.
Both classes inherit from *ScriptableObject*, as everything that needs to be saved does (like Nodes). That's what the editorside Save/Load system depends on.

Nodes are simply classes inherited from *Node*, automatically caught by the framework and presented as an options in the context menus:) They have to feature a unique ID, a initialization function, a gui function and a calculation function (which will be optional later on). See Implementation of custom Nodes

Currently, there are connection types which represent the types of inputs/outputs. They feature a color, type, texture, and later a condition on which they will pass only. See Connection Types

Feature Implementation
You'll find some TODO's in order to give you quick access to some of the features and important locations in the code of Node Editor. They can be accessed through the 'Tasks' Pad in MonoDevelop. In the following are some notes about features:

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.

Zooming is a very fragile and hard to implement feature, but I'm proud Node Editor features it. Due to Unity's clipping and grouping system, there has yet to be found a methode to zoom inside a group.

In the future, you'll have **preprocessor settings** to toggle specific settings of Node Editor by preprocessor defines. You have to add them in the project settings in the Compiler tab or add them at the very top of each file. Those include:

- NODE_EDITOR_LINE_CONNECTION : Define to replace cuves with lines.
- NODE_EDITOR_STATEMACHINE : Switches various Statemachine-related things, including
  - Transitions instead of input/output connections
  - Calculation system replaced with path-walking based on transition conditions
- NODE_EDITOR_NO_TOOLBAR : Removes the Toolbar GUI Element at the top
- NODE_EDITOR_NO_PANEL : Removes the Side Panel GUI Element
- NODE_EDITOR_PARAMETER_PANEL and !NODE_EDITOR_NO_PANEL : Shows connections on node only, instead calls NodeGUI on the side panel when node is active

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 Extensions. Due to the recent addition of searching the assembly for nodes and types, there is no need for a regristration anymore. That means, you can enable your users to extent your Extension by simply searching the full assembly.

The Node declaration consists of a class such as the following, using *NodeEditorFramework*, extending *Node* and implementing the const ID and static property GetID aswell as the override functions *Create*, *NodeGUI* and *Calculate* with these signatures and contents:

```
using UnityEngine;
using NodeEditorFramework;

public class ExampleNode : Node
{
    public const string ID = "exampleNode";
    public override string GetID { get { return ID; } }

    public override Node Create (Vector2 pos)
    {
        ExampleNode node = CreateInstance<ExampleNode> ();
        return node;
    }

    public override void NodeGUI ()
    {
    }

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

The static function **Create** inits your node taking the node position as a parameter. It starts by creating an instance of your Node using *ScriptableObject.CreateInstance* (short: *CreateInstance*). Set the name and the rect of it, followed by your Inputs and Outputs. Perform any other setup steps here as well.

```
public override Node Create (Vector2 pos)
{
    ExampleNode node = CreateInstance<ExampleNode> ();

    node.rect = new Rect (pos.x, pos.y, 150, 60);
    node.name = "Example Node";

    CreateInput ("Value", "Float");
    CreateOutput ("Output val", "Float");

    return node;
}
```

In both *CreateInput* and *CreateOutput*, you have to pass in the name of the connection (which you may override later in the GUI) and the type of the connection (Currently identified by the string defined in the *TypeDeclaration*).

The next function, **NodeGUI,** is responsible for drawing the node's GUI. It's the place to expose any parameters and the Inputs/Outputs. Here is a simple example:

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

    GUILayout.BeginHorizontal ();
    GUILayout.BeginVertical ();

    Inputs [0].DisplayLayout ();

    GUILayout.EndVertical ();
    GUILayout.BeginVertical ();

    Outputs [0].DisplayLayout ();

    GUILayout.EndVertical ();
    GUILayout.EndHorizontal ();
}
```

There are several ways to show the Inputs/Outputs:
DisplayLayout () shows the name,
DisplayLayout (GUIContent) displays a custom Label.
Node also features a function which places the knobs right next to the last *GUIElement*, *InputKnob* and *OutputKnob* respectively.

The **Calculate** function performs the calculation related things. it's called whenever a change in values or dependecies was performed. Later on, this will be optional.
Functions like *allInputsReady*, *hasNullInputs* or *hasNullInputValues* are used to quickly 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].GetValue<FloatValue> ().value = Inputs[0].connection.GetValue<FloatValue>().value * 5;
    return true;
}
```

Note the way to access the Input/Output values.

NoteAttribute

To provide more flexibility, there is an optional NodeAttribute where you can pass in whether to hide the node from the user (as if it does not exist) or to change the context menu path. Though that is it for now, the attribute has much potential and will provide more options in the future.

Connection Types

Connection types define the color, type, texture and any other type-related data used by the extension. They are declared by extending the interface *ITypeDeclaration*:

```
public interface ITypeDeclaration
{
    string name { get; }
    Color col { get; }
    string InputKnob_TexPath { get; }
    string OutputKnob_TexPath { get; }
    Type InputType { get; }
    Type OutputType { get; }
}
```

Here is the example of the float decalaration. As a type, choose a class (reference type) containing the value:

```
public class FloatType : ITypeDeclaration
{
    public string name { get { return "Float"; } }
    public Color col { get { return Color.cyan; } }
    public string InputKnob_TexPath { get { return "Textures/In_Knob.png"; } }
    public string OutputKnob_TexPath { get { return "Textures/Out_Knob.png"; } }
    public Type InputType { get { return null; } }
    public Type OutputType { get { return typeof(FloatValue); } }
}

[Serializable]
public class FloatValue
{
    [NonSerialized]
    public float value;
}
```

Later on, this will also feature a passing condition system rather than an Input/Output type. As long as this is in the same assembly, you now can access it through it's name „Float“. To make sure it's in the same assembly, enclose it in the namespace *NodeEditorFramework*.

Events

Node Editor features an Event system in it's early stages. They are accessible to a MonoBehaviour by inheriting from *NodeEditorCallbackReceiver* and to any other class by subscribing to the *NodeEditorCallbacks* delegates. For now, these most important Events are implemented:

OnEditorStartUp ()
OnLoadCanvas (NodeCanvas)
OnLoadEditorState (NodeEditorState)
OnSaveCanvas (NodeCanvas)
OnSaveEditorState (NodeEditorState)
OnAddNode (Node)
OnDeleteNode (Node)
OnMoveNode (Node)
OnAddConnection (NodeInput)
OnRemoveConnection (NodeInput)

Conclusion

I was surprised by how positive my initial cobtribution was accepted, but now, Node Editor is not anymore just my contribution:) Here you can see who has contributed to the project so far!
Of couse, you're welcome to contribute your changes and updates to the **GitHub repository** hosted by Baste.
There's also a Trello Board where we invite you to join discussions about Node Editor. If you want to become a member of it, PM me or any other member of the trello board for an invitation:)
Also, if you made an Editor Extension using this, we'd really like to lean about it;)

Major stable updates on Node Editor will still be published in this original **post**.

Node Editor is published under the **MIT License** found next to this documentation.

Thanks for your interest in Node Editor!
Best regards,

Seneral