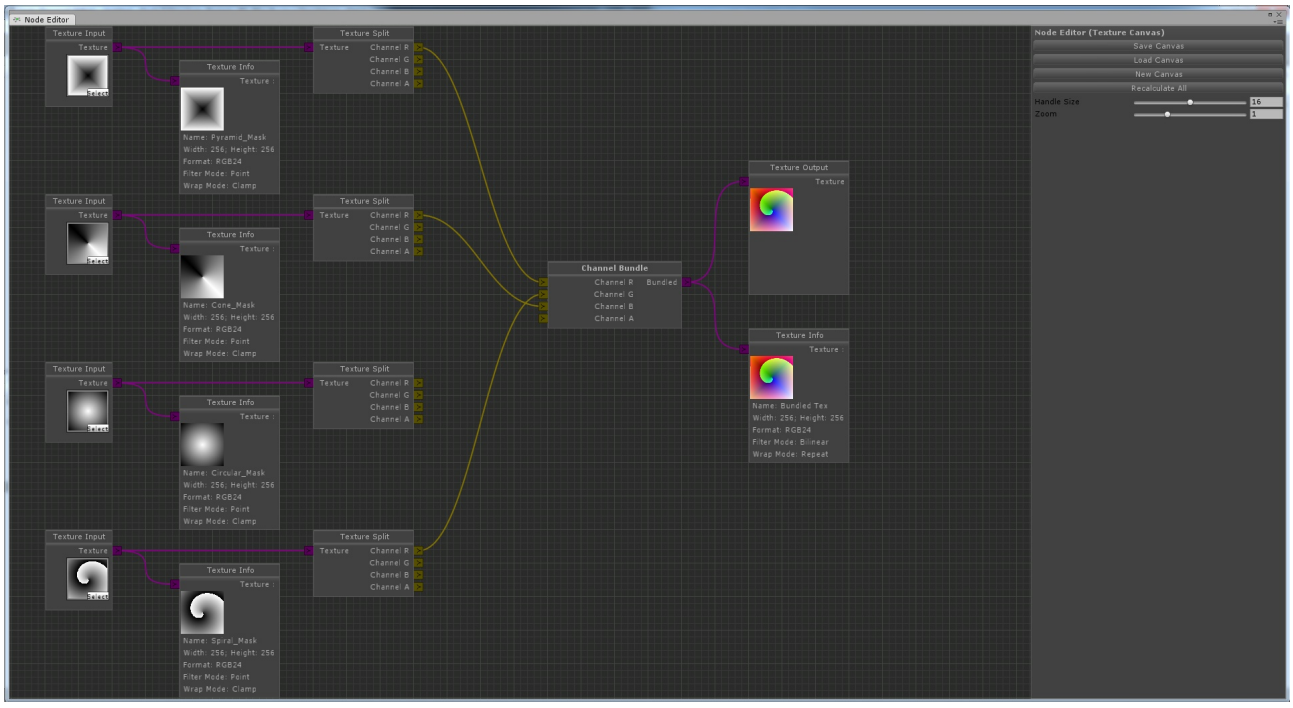
NODE EDITOR

Free and versatile Node Editor Framework for Unity 3D



(Texture Composer, an example avaiable on the forums)

Documentation by Seneral

Version 1.0 (23.01.16)

Repository - Forums

Contents

Preface

This Documentation intends to give you an overview of the Node Editor. It was initially posted by me as a personal project on the Unity forums. After receiving a great amount of positive feedback I continued improving and supporting it, now featuring alot of major things a Node Editor needs to have! The GitHub repository was set up by Baste during the early developement, and now it's the main platform to share and contribute to Node Editor.

Features

The Node Editor is special in that it's open source, but still is packed full of features, some of which are unique in their implementation and cleverness:

- Full interface featuring all expected controls like zooming/panning
- Flexible and interchangeable GUI
- Canvas and Editor State system featuring alot of advanced controls
- Super easy to extend with custom nodes and types without modifying the provided code
- Flexible calculation system, soon supporting for state-like behaviour
- Extreme Node customization to create totally unique node appearances achieved by a custom windowing system
- Growing Runtime support, including every bit from the EditorWindow

The framework also shines with clean and easy to modify code. It's clearly seperated in multiple parts, some of which can even be taken and used somewhere else, such as the unique, generic scaling approach!

Accessing the Examples

You can start off by checking out the Editor Window at 'Window/Node Editor' and loading one of the example canvases, such as the Calculation Canvas. Do that by either loading it with the button at the top right or by locating it in the project folder and double-clicking it.
With right-click you can add additional nodes, using drag'n'drop you can connect node Outputs and Inputs with each other.

Getting Started

Here you'll find some help on how to get you started creating custom Nodes and ConnectionTypes in the simplest form, without touching the Framework code. These two methods already can help you make little extensions requiring the Node Editor Framework to be installed seperately.

Custom Nodes

The implementation of additional, custom nodes is fairly easy. You have to create a script anywhere in the project extending the Node class of the NodeEditorFramework namespace. It will provide the Framework all needed information about the node itself, the optional 'Node' Attribute contains information about the presentation in the editor ('accept transitions' being an exception). The Framework will search all script assemblies for additional nodes, so extra setup is not required.

In the following are the necessary Node members outlined, based upon the ExampleNode found at 'Plugins/Node_Editor/Nodes'. First to mention is that even though the Framework is programmed in C#, you can add nodes in JavaScript with the limitation that they have to be compiled in phase 2,3 or 4, as described here. Therefore the following members are described language independently.

- NodeEditorFramework dependency
- Class extending 'Node'
- Optional: **Attribute 'Node'**; Params: *[Bool]* Is Node hidden in the context menu?; *[String]* Directory in the context menu; *[Bool]* Accepting transitions (experimental state)?
- Unique **Node ID**, specified in 'ID' *[constant string]* and exposed through the get-property 'GetID' *[override]*
- Method '**Create**' *[override / Params: [Vector2] Position / Returns: [Node] Created Node]*
  - Create a new Instance of your node type using 'CreateInstance'. Assign it's property 'rect' a value using the position parameter and give it a name.
  - Add connections using the CreateInput/CreateOutput, passing a name and a type (e.g. 'Float'). Alternatively use NodeInput.Create/NodeOutput.Create passing your Node.
  - Perform any other additional setup and return your created node.
- Method '**NodeGUI**' *[override]*
  - Draw you Node's GUI in it, you may use GUILayout funtions savely.
  - Using the Inputs/Outputs array you can access your created Inputs/Outputs in the order in which you created them. Calling DisplayLayout on them will draw a name-label for them and set the knob next to that label. Using the overloads you can specify GUIContent and GUIStyle of the label.
- Method '**Calculate**' *[override / Returns: [Bool] Was calculation attempt a success?]*
  - Use the methods 'allInputsReady', 'hasUnassignedInputs' and 'descendantsCalculated' to check right at the start whether the node is ready, based on the needs and purposes of it.
  - Access the inputs as described above, calling 'GetValue' with the connection type on them will return the value stored in the connection. Similarily, you set the value of the outputs by calling SetValue on them.
  - Return true when you're done calculating and false when you are not ready yet and need another attempt. But be aware, you cannot yield calculation that way, after a maximum of a thousand repeated tries the calculation will be aborted!

Custom ConnectionTypes

Implementing custom ConnectionTypes is similar to Node implementation, as it uses the same fetching system: Declare a class inheriting from the 'ITypeDeclaration' interface and specify it's properties.

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

// TODO: Node Editor: Built-In Connection Types
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 Type { get { return typeof(float); } }
}
```

Top Block: ITypeDeclaration; Bottom Block: Built-in Float type (from ConnectionTypes.cs)

- The **string 'name'** is used to adress the type later on.
- The **Color 'col'** is the color associated with the type, and the following knob textures aswell as the node connection curves are tinted with it.
- The **strings 'InputKnob_TexPath'** and **'OutputKnob_TexPath'** contain the paths to the input and output knob textures relative to **Node_Editor/Resources**. Default are 'Textures/In_Knob.png' and 'Textures/Out_Knob.png'
- The type this declaration representates, e.g. 'typeof(float)', is stored in the **Type 'Type'**.

Customizing

Even though you can already built small extensions with methods described above pretty well, to natively integrate Node Editor into your own editor extension you may want to cutomize it, from building your own editor interface to modify and extend the framework itself.

Editor/Runtime Interface

The provided Editor Window basically serves as the default Node Canvas explorer for all dependant extensions and single canvases, but also as a starting point to develop a custom Editor Window from. That means, you can savely delete it if you don't want it in your extension. In the following I'll outline all things you need to consider to build a basic Node Editor Interface in both Runtime and the Editor.

Saving and Loading Canvas and Editor State

The Editor needs to store the currently opened Canvas ('NodeCanvas') and it's Editor State ('NodeEditorState'). For an explanation of these, please look up the Framework Overview.

You can save both using 'NodeEditor.SaveNodeCanvas' and load them with 'NodeEditor.LoadNodeCanvas' and 'NodeEditor.LoadEditorStates'. Take reference from the default NodeEditorWindow to see how exactly these functions are integrated. The function 'AutoOpenCanvas' also shows how to automatically open a canvas by double-clicking it's asset in the Editor.

Bringing the Canvas onscreen

First, you need to make sure that the NodeEditor is initiated using 'NodeEditor.checkInit', and that there is always a canvas loaded, else create a new one.
Before drawing you'll want to define the rect in which the canvas resides. No boundaries are set anymore in where the canvas is set, in how many subgroups, etc. Only the case of modifying the GUI.matrix scale before is not yet supported. You currently must assign the rect to the 'canvasRect' property of the EditorState you're using.
In order to best account for errors, the following drawing statement is embedded in a try-catch-block catching only UnityExceptions, unloading the old and creating a new canvas when an error occurs. Draw the canvas by passing both the NodeCanvas and the EditorState into 'NodeEditor.DrawCanvas', which will behave like an ordinary GUI control in most cases.

Custom GUISkin

The GUISkin of the Node Editor can currently only be changed by modifying the NodeEditorGUI.cs source file or simply replacing the textures.

Framework Overview

This section aims to bring you a decent overview on how the framework is structured, so you can get to modify it quickly. This does not necessarily include implementation details – code sections that need extra detailing are commented. Also, this section is not only for those planning to get into the code, but for everyone to get an overview what he's working with:)

Canvas and EditorState

Those two components essentially make up something you can load up into the EditorWindow. Basically, the canvas is the important part with all the nodes and any additional information directly related to the Canvas. In contrary, the EditorState holds all information on the state, or in other words, on how the Canvas is presented. This includes zoom and pan values, selected Nodes, the rect the Canvas is drawn in, etc. Not all of these values are actually saved with the asset. That structure allows for multiple 'views' on the same Canvas and edit it simultaneously.

The DrawCanvas function

This function acts very similar to any other GUI control, with a few exceptions, and is responsible for drawing the Canvas. On the first glance it's just a wrapper for DrawSubCanvas, with the exception that it holds the OverlayGUI and NodeGUISkin code. DraSubCanvas is used in the future for SubCanvases, as the name proposes.
In the first major block, the background texture is splattered across the screen where needed, accounting for pan and zoom, relative to the rect center.
In the function 'InputEvents' all inputs are catched. It's well commented, so I won't explain it any further here. It accounts for rects that should be ignored for input.
Then, the scale area gets initiated with a call to my custom solution 'GUIScaleUtility.BeginScale'. Any GUI code after is getting scaled appropriately.
In the following, everything that needs to be scaled gets drawn, including temporal connections, node transitions, connections, bodies and knobs.
Thereafter, the scale area gets closed again with another call to 'GUIScaleUtility.EndScale'.
The 'LateEvents' function checks all inputs with secondary priority after the nodes just like 'inputEvent' does, in this case it only makes sure the node can only be dragged when not clicking on a control (GUI.hotControl is empty).

Events

(WIP)
The Framework supports a multitude of Events which might be important during the editing process. Those Events can either be received by subsribing to the appropriate delegate in the 'NodeEditorCallbacks' class or by extending from 'NodeEditorCallbackReceiver' (which is a MonoBehaviour) and overriding the appropriate method. Both classes can be found in 'NodeEditorCallbackReceiver.cs'.
Current Events include:

- OnEditorStartup (): Gets called each time NodeEditor gets inited (can also happen when switching scene or playmode)
- OnLoadCanvas (NodeCanvas): Gets called when this canvas gets loaded
- OnLoadEditorState (NodeEditorState): Gets called when this editorState gets loaded

- OnSaveCanvas (NodeCanvas): Gets called when this canvas gets saved
- OnSaveEditorState (NodeEditorState): Gets called when this editorState gets saved
- OnAddNode (Node): Gets called when a new Node was added or duplicated
- OnDeleteNode (Node): Gets called when a Node gets deleted
- OnMoveNode (Node): Gets called when a Node moves
- OnAddConnection (NodeInput): Gets called when a connection was added to this input. If there was one previously, OnRemoveConnection gets called before, too.
- OnRemoveConnection (Nodeinput): Gets called when the connection was removed from this input

Conclusion

(WIP)
I'm happy that the Node Editor has received so much positive comments and helpful critism since I posted it back in May 2015.
Since then a few people took the time and motivation to contribute to the project which I appreciate very much! You can check all contributions out on the repo page. But also those who use, test and report any bugs are very important for this project.
If you wish to contribute, you may take a look at the roadmap as a rough guideline what is planned and how you can help. Of course, own ideas are just as fine.
Make sure to post or tell me if you are making an extension using Node Editor, may it be big or small, and notify me about any problems you may encounter:) This is vital to the project! Just make sure to account for the MIT License included with Node Editor.

Also, you can always contact me with a PM on the Forums or per Email at lev.gaeher@gmail.com:)
I hope this Documentation has helped you understanding the NodeEditor, else feel free to suggest imrovements and ask me!