.gitignore

/[Ll]ibrary/  
/[Tt]emp/  
/[Oo]bj/  
/[Bb]uild/  
/[Bb]uilds/  
/[Ll]ogs/  
/[Uu]ser[Ss]ettings/  
/[Mm]emoryCaptures/  
/[Rr]ecordings/  
/[Aa]ssets/Plugins/Editor/JetBrains\*  
.vs/  
.gradle/  
ExportedObj/  
.consulo/  
sysinfo.txt  
crashlytics-build.properties  
/[Aa]ssets/[Aa]ddressable[Aa]ssets[Dd]ata/\*/\*.bin\*  
/[Aa]ssets/[Ss]treamingAssets/aa.meta  
/[Aa]ssets/[Ss]treamingAssets/aa/\*

AmplifyShaderEditor.asmdef.meta

fileFormatVersion: 2  
guid: f540dafdfbc0586439d98823585550d4  
AssemblyDefinitionImporter:  
 externalObjects: {}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

ChangeLog.txt.meta

fileFormatVersion: 2  
guid: 580cccd3e608b7f4cac35ea46d62d429  
timeCreated: 1481127071  
licenseType: Store  
TextScriptImporter:  
 userData:   
 assetBundleName:   
 assetBundleVariant:

CreatingTerrainsWithASE.txt.meta

fileFormatVersion: 2  
guid: f11d5aaf59fc38544b8419242801ff97  
timeCreated: 1513615640  
licenseType: Store  
TextScriptImporter:  
 userData:   
 assetBundleName:   
 assetBundleVariant:

Credits.txt.meta

fileFormatVersion: 2  
guid: 451790f45b4e5434586d16e924540ee7  
timeCreated: 1481127071  
licenseType: Store  
TextScriptImporter:  
 userData:   
 assetBundleName:   
 assetBundleVariant:

DownloadSamplesInstructions.txt.meta

fileFormatVersion: 2  
guid: 03fe3bdcb7262a84ca060ff336c7d8d1  
timeCreated: 1481127071  
licenseType: Store  
TextScriptImporter:  
 userData:   
 assetBundleName:   
 assetBundleVariant:

README Samples.txt.meta

fileFormatVersion: 2  
guid: 5c5208e121a880c4fbb9de7aaeaa8aa2  
timeCreated: 1541776170  
licenseType: Store  
TextScriptImporter:  
 userData:   
 assetBundleName:   
 assetBundleVariant:

Examples.meta

fileFormatVersion: 2  
guid: 425c3aed47dd05444960ca41af18e591  
folderAsset: yes  
timeCreated: 1481126943  
licenseType: Store  
DefaultImporter:  
 userData:   
 assetBundleName:   
 assetBundleVariant:

ActionData.cs

using System;  
using UnityEngine;  
namespace AmplifyShaderEditor  
{  
 public class ActionData  
 {  
 public virtual void ExecuteForward() { }  
 public virtual void ExecuteReverse() { }  
 }  
 public class CreateNodeActionData : ActionData  
 {  
 private int m\_nodeId;  
 private System.Type m\_nodeType;  
 private Vector2 m\_nodePos;  
 public CreateNodeActionData( ParentNode node )  
 {  
 m\_nodeId = node.UniqueId;  
 m\_nodePos = node.Vec2Position;  
 m\_nodeType = node.GetType();  
 }  
 public CreateNodeActionData( int nodeId, System.Type nodeType, Vector2 nodePos )  
 {  
 m\_nodeId = nodeId;  
 m\_nodePos = nodePos;  
 m\_nodeType = nodeType;  
 }  
 public override void ExecuteForward()  
 {  
 UIUtils.CreateNode( m\_nodeType, false, m\_nodePos, m\_nodeId );  
 }  
 public override void ExecuteReverse()  
 {  
 UIUtils.DestroyNode( m\_nodeId );  
 }  
 public override string ToString()  
 {  
 return "Create Node - Type: " + m\_nodeType + " Node: " + m\_nodeId + " Position: " + m\_nodePos;  
 }  
 }  
 public class DestroyNodeActionData : ActionData  
 {  
 private int m\_nodeId;  
 private System.Type m\_nodeType;  
 private Vector2 m\_nodePos;  
 public DestroyNodeActionData( ParentNode node )  
 {  
 m\_nodeId = node.UniqueId;  
 m\_nodePos = node.Vec2Position;  
 m\_nodeType = node.GetType();  
 }  
 public DestroyNodeActionData( int nodeId, System.Type nodeType, Vector2 nodePos )  
 {  
 m\_nodeId = nodeId;  
 m\_nodePos = nodePos;  
 m\_nodeType = nodeType;  
 }  
 public override void ExecuteForward()  
 {  
 UIUtils.DestroyNode( m\_nodeId );  
 }  
 public override void ExecuteReverse()  
 {  
 UIUtils.CreateNode( m\_nodeType, false, m\_nodePos, m\_nodeId );  
 }  
 public override string ToString()  
 {  
 return "Destroy Node - Type: " + m\_nodeType + " Node: " + m\_nodeId + " Position: " + m\_nodePos;  
 }  
 }  
 public class MoveNodeActionData : ActionData  
 {  
 private int m\_nodeId;  
 private Vector2 m\_nodeInitalPos;  
 private Vector2 m\_nodeFinalPos;  
 public MoveNodeActionData( int nodeId, Vector2 nodeInitialPos, Vector2 nodeFinalPos )  
 {  
 m\_nodeId = nodeId;  
 m\_nodeInitalPos = nodeInitialPos;  
 m\_nodeFinalPos = nodeFinalPos;  
 }  
 public override void ExecuteForward()  
 {  
 ParentNode node = UIUtils.GetNode( m\_nodeId );  
 if ( node != null )  
 node.Vec2Position = m\_nodeFinalPos;  
 }  
 public override void ExecuteReverse()  
 {  
 ParentNode node = UIUtils.GetNode( m\_nodeId );  
 if ( node != null )  
 node.Vec2Position = m\_nodeInitalPos;  
 }  
 public override string ToString()  
 {  
 return "Move Node - Node: " + m\_nodeId + " Initial Position: " + m\_nodeInitalPos + " Final Position: " + m\_nodeFinalPos;  
 }  
 }  
 public class CreateConnectionActionData : ActionData  
 {  
 private int m\_inputNodeId;  
 private int m\_inputPortId;  
 private int m\_outputNodeId;  
 private int m\_outputPortId;  
 public CreateConnectionActionData( int inputNodeId, int inputPortId, int outputNodeId, int outputPortId )  
 {  
 m\_inputNodeId = inputNodeId;  
 m\_inputPortId = inputPortId;  
 m\_outputNodeId = outputNodeId;  
 m\_outputPortId = outputPortId;  
 }  
 public override void ExecuteForward()  
 {  
 UIUtils.ConnectInputToOutput( m\_inputNodeId, m\_inputPortId, m\_outputNodeId, m\_outputPortId );  
 }  
 public override void ExecuteReverse()  
 {  
 UIUtils.DeleteConnection( true, m\_inputNodeId, m\_inputPortId, false, true );  
 }  
 public override string ToString()  
 {  
 return "Create Connection Node - Input Node: " + m\_inputNodeId + " Input Port: " + m\_inputPortId + " Output Node: " + m\_outputNodeId + " Output Port: " + m\_outputPortId;  
 }  
 }  
 public class DestroyConnectionActionData : ActionData  
 {  
 private int m\_inputNodeId;  
 private int m\_inputPortId;  
 private int m\_outputNodeId;  
 private int m\_outputPortId;  
 public DestroyConnectionActionData( int inputNodeId, int inputPortId, int outputNodeId, int outputPortId )  
 {  
 m\_inputNodeId = inputNodeId;  
 m\_inputPortId = inputPortId;  
 m\_outputNodeId = outputNodeId;  
 m\_outputPortId = outputPortId;  
 }  
 public override void ExecuteForward()  
 {  
 UIUtils.DeleteConnection( true, m\_inputNodeId, m\_inputPortId, false, true );  
 }  
 public override void ExecuteReverse()  
 {  
 UIUtils.ConnectInputToOutput( m\_inputNodeId, m\_inputPortId, m\_outputNodeId, m\_outputPortId );  
 }  
 public override string ToString()  
 {  
 return "Destroy Connection Node - Input Node: " + m\_inputNodeId + " Input Port: " + m\_inputPortId + " Output Node: " + m\_outputNodeId + " Output Port: " + m\_outputPortId;  
 }  
 }  
 public class MoveInputConnectionActionData : ActionData  
 {  
 private int m\_oldInputNodeId;  
 private int m\_oldInputNodePortId;  
 private int m\_newInputNodeId;  
 private int m\_newInputNodePortId;  
 private int m\_outputNodeId;  
 private int m\_outputPortId;  
 public MoveInputConnectionActionData( int oldInputNodeId, int oldInputPortId, int newInputNodeId, int newInputPortId, int outputNodeId, int outputPortId )  
 {  
 m\_oldInputNodeId = oldInputNodeId;  
 m\_oldInputNodePortId = oldInputPortId;  
 m\_newInputNodeId = newInputNodeId;  
 m\_newInputNodePortId = newInputPortId;  
 m\_outputNodeId = outputNodeId;  
 m\_outputPortId = outputPortId;  
 }  
 public override void ExecuteForward()  
 {  
 UIUtils.DeleteConnection( true, m\_oldInputNodeId, m\_oldInputNodePortId, false, true );  
 UIUtils.ConnectInputToOutput( m\_newInputNodeId, m\_newInputNodePortId, m\_outputNodeId, m\_outputPortId );  
 }  
 public override void ExecuteReverse()  
 {  
 base.ExecuteReverse();  
 UIUtils.DeleteConnection( true, m\_newInputNodeId, m\_newInputNodePortId, false, true );  
 UIUtils.ConnectInputToOutput( m\_oldInputNodeId, m\_oldInputNodePortId, m\_outputNodeId, m\_outputPortId );  
 }  
 public override string ToString()  
 {  
 return "Move Input Connection Node - Old Input Node: " + m\_oldInputNodeId + " Old Input Port: " + m\_oldInputNodePortId + " New Input Node: " + m\_newInputNodeId + " New Input Port: " + m\_newInputNodePortId + " Output Node: " + m\_outputNodeId + " Output Port: " + m\_outputPortId;  
 }  
 }  
 public class MoveOutputConnectionActionData : ActionData  
 {  
 private int m\_inputNodeId;  
 private int m\_inputPortId;  
 private int m\_newOutputNodeId;  
 private int m\_newOutputPortId;  
 private int m\_oldOutputNodeId;  
 private int m\_oldOutputPortId;  
 public MoveOutputConnectionActionData( int inputNodeId, int inputPortId, int newOutputNodeId, int newOutputPortId, int oldOutputNodeId, int oldOutputPortId )  
 {  
 m\_inputNodeId = inputNodeId;  
 m\_inputPortId = inputPortId;  
 m\_newOutputNodeId = newOutputNodeId;  
 m\_newOutputPortId = newOutputPortId;  
 m\_oldOutputNodeId = oldOutputNodeId;  
 m\_oldOutputPortId = oldOutputPortId;  
 }  
 public override void ExecuteForward()  
 {  
 UIUtils.DeleteConnection( false, m\_oldOutputNodeId, m\_oldOutputNodeId, false, true );  
 UIUtils.ConnectInputToOutput( m\_inputNodeId, m\_inputPortId, m\_newOutputNodeId, m\_newOutputPortId );  
 }  
 public override void ExecuteReverse()  
 {  
 base.ExecuteReverse();  
 UIUtils.DeleteConnection( false, m\_newOutputNodeId, m\_newOutputPortId, false, true );  
 UIUtils.ConnectInputToOutput( m\_inputNodeId, m\_inputPortId, m\_oldOutputNodeId, m\_oldOutputPortId );  
 }  
 public override string ToString()  
 {  
 return "Move Input Connection Node - Input Node: " + m\_inputNodeId + " Input Port: " + m\_inputPortId + " Old Output Node: " + m\_oldOutputNodeId + " Old Output Port: " + m\_oldOutputPortId + " New Output Node: " + m\_newOutputNodeId + " New Output Port: " + m\_newOutputPortId;  
 }  
 }  
 public class CreateNewGraphActionData : ActionData  
 {  
 private string m\_name;  
 public CreateNewGraphActionData( string name )  
 {  
 m\_name = name;  
 }  
 public override void ExecuteForward()  
 {  
 UIUtils.CreateNewGraph( m\_name );  
 }  
 }  
 public class ChangeNodePropertiesActionData : ActionData  
 {  
 private string m\_originalProperties;  
 private string m\_newProperties;  
 private int m\_nodeId;  
 public ChangeNodePropertiesActionData( ParentNode node, string originalProperties )  
 {  
 m\_nodeId = node.UniqueId;  
 m\_originalProperties = originalProperties;  
 m\_newProperties = string.Empty;  
 string trash = string.Empty;  
 node.WriteToString( ref m\_newProperties, ref trash );  
 }  
 public ChangeNodePropertiesActionData( int nodeId, string originalProperties )  
 {  
 m\_nodeId = nodeId;  
 m\_originalProperties = originalProperties;  
 m\_newProperties = string.Empty;  
 string trash = string.Empty;  
 UIUtils.GetNode( nodeId ).WriteToString( ref m\_newProperties, ref trash );  
 }  
 public override void ExecuteForward()  
 {  
 string[] properties = m\_newProperties.Split( IOUtils.FIELD\_SEPARATOR );  
 UIUtils.GetNode( m\_nodeId ).ReadFromString( ref properties );  
 }  
 public override void ExecuteReverse()  
 {  
 string[] properties = m\_originalProperties.Split( IOUtils.FIELD\_SEPARATOR );  
 UIUtils.GetNode( m\_nodeId ).ReadFromString( ref properties );  
 }  
 public override string ToString()  
 {  
 return "Change Node Propertie - Node: " + m\_nodeId + "\nOriginal Properties:\n" + m\_originalProperties + "\nNew Properties:\n" + m\_newProperties;  
 }  
 }  
}

ActionData.cs.meta

fileFormatVersion: 2  
guid: 29204f353101f46439a93f1c503d3197  
timeCreated: 1481126954  
licenseType: Store  
MonoImporter:  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

ActionLog.cs

using System.Collections.Generic;  
namespace AmplifyShaderEditor  
{  
 public class ActionLog  
 {  
 private int m\_maxCount;  
 private int m\_index;  
 private List<ActionData> m\_sequence;  
 public ActionLog(int maxCount)  
 {  
 m\_maxCount = maxCount;  
 m\_index = 0;  
 m\_sequence = new List<ActionData>();  
 }  
 public void AddToLog(ActionData actionData)  
 {  
 if (m\_sequence.Count > m\_maxCount)  
 {  
 m\_sequence.RemoveAt(0);  
 }  
 m\_sequence.Add(actionData);  
 m\_index = m\_sequence.Count - 1;  
 }  
 public void UndoLastAction()  
 {  
 if ( m\_index > -1 && m\_index < m\_sequence.Count )  
 m\_sequence[m\_index--].ExecuteReverse();  
 }  
 public void RedoLastAction()  
 {  
 if (m\_index < (m\_sequence.Count - 1))  
 m\_sequence[++m\_index].ExecuteForward();  
 }  
 public void ClearLog()  
 {  
 m\_sequence.Clear();  
 m\_index = 0;  
 }  
 public void Destroy()  
 {  
 m\_sequence.Clear();  
 m\_sequence = null;  
 }  
 }  
}

ActionLog.cs.meta

fileFormatVersion: 2  
guid: bc089a69595d8994cb89946a919517c2  
timeCreated: 1481126958  
licenseType: Store  
MonoImporter:  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

ActionSequence.cs

using System.Collections.Generic;  
namespace AmplifyShaderEditor  
{  
 public class ActionSequence  
 {  
 private string m\_name;  
 private List<ActionData> m\_sequence;  
 public ActionSequence( string name )  
 {  
 m\_name = name;  
 m\_sequence = new List<ActionData>();  
 }  
 public void AddToSequence( ActionData actionData )  
 {  
 m\_sequence.Add( actionData );  
 }  
 public void Execute()  
 {  
 for ( int i = 0; i < m\_sequence.Count; i++ )  
 {  
 m\_sequence[ i ].ExecuteForward();  
 }  
 }  
 public void Destroy()  
 {  
 m\_sequence.Clear();  
 m\_sequence = null;  
 }  
 public string Name { get { return m\_name; } }  
 }  
}

ActionSequence.cs.meta

fileFormatVersion: 2  
guid: 43bd963fa46ee9c4680dacff1d8dc0b9  
timeCreated: 1481126955  
licenseType: Store  
MonoImporter:  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

Actions.meta

fileFormatVersion: 2  
guid: a2ba8588f45692f4ea2fa5afa9faf434  
folderAsset: yes  
timeCreated: 1481126944  
licenseType: Store  
DefaultImporter:  
 userData:   
 assetBundleName:   
 assetBundleVariant:

Constants.cs

using UnityEngine;  
using System.Collections.Generic;  
namespace AmplifyShaderEditor  
{  
 public struct Constants  
 {  
 public readonly static string[] FaceMacros =  
 {  
 "#if defined(SHADER\_API\_GLCORE) || defined(SHADER\_API\_GLES) || defined(SHADER\_API\_GLES3) || defined(SHADER\_API\_D3D9)",  
 "#define FRONT\_FACE\_SEMANTIC VFACE",  
 "#define FRONT\_FACE\_TYPE float",  
 "#else",  
 "#define FRONT\_FACE\_SEMANTIC SV\_IsFrontFace",  
 "#define FRONT\_FACE\_TYPE bool",  
 "#endif"  
 };  
 {  
 "#if defined(SHADER\_API\_D3D11) || defined(SHADER\_API\_XBOXONE) || defined(UNITY\_COMPILER\_HLSLCC)//ASE Args Macros",  
 "#define ASE\_TEXTURE2D\_ARGS(textureName) Texture2D textureName, SamplerState sampler##textureName",  
 "#define ASE\_TEXTURE3D\_ARGS(textureName) Texture3D textureName, SamplerState sampler##textureName",  
 "#define ASE\_TEXTURECUBE\_ARGS(textureName) TextureCube textureName, SamplerState sampler##textureName",  
 "#define ASE\_TEXTURE2D\_PARAMS(textureName) textureName, sampler##textureName",  
 "#define ASE\_TEXTURE3D\_PARAMS(textureName) textureName, sampler##textureName",  
 "#define ASE\_TEXTURECUBE\_PARAMS(textureName) textureName, sampler##textureName",  
 "#define ASE\_TEXTURE2D\_ARRAY\_PARAMS(textureName) textureName, sampler##textureName",  
 "#else//ASE Args Macros",  
 "#define ASE\_TEXTURE2D\_ARGS(textureName) sampler2D textureName",  
 "#define ASE\_TEXTURE3D\_ARGS(textureName) sampler3D textureName",  
 "#define ASE\_TEXTURECUBE\_ARGS(textureName) samplerCUBE textureName",  
 "#define ASE\_TEXTURE2D\_PARAMS(textureName) textureName",  
 "#define ASE\_TEXTURE3D\_PARAMS(textureName) textureName",  
 "#define ASE\_TEXTURECUBE\_PARAMS(textureName) textureName",  
 "#define ASE\_TEXTURE2D\_ARRAY\_PARAMS(textureName) textureName",  
 "#endif//ASE Args Macros\n"  
 };  
 public readonly static string[] CustomASEDeclararionMacros =  
 {  
 "#define ASE\_TEXTURE2D(textureName) {0}2D(textureName)",  
 "#define ASE\_TEXTURE2D\_ARRAY(textureName) {0}2D\_ARRAY(textureName)",  
 "#define ASE\_TEXTURE3D(textureName) {0}3D(textureName)",  
 "#define ASE\_TEXTURECUBE(textureName) {0}CUBE(textureName)\n"  
 };  
 public readonly static string[] CustomASEStandarSamplingMacrosHelper =  
 {  
 "#if defined(SHADER\_API\_D3D11) || defined(SHADER\_API\_XBOXONE) || defined(UNITY\_COMPILER\_HLSLCC)//ASE Sampling Macros",  
 "#else//ASE Sampling Macros",  
 "#endif//ASE Sampling Macros\n"  
 };\*/  
 {  
 "#define ASE\_SAMPLE\_TEXTURE2D(textureName,{0}coords) {1}2D{2}(textureName,{0}coords)",  
 "#define ASE\_SAMPLE\_TEXTURE2D\_LOD(textureName, {0}coord2, lod) {1}2D{2}\_LOD(textureName, {0}coord2, lod)",  
 "#define ASE\_SAMPLE\_TEXTURE2D\_BIAS(textureName,{0}coord2, bias) {1}2D{2}\_BIAS(textureName,{0}coord2, bias)",  
 "#define ASE\_SAMPLE\_TEXTURE2D\_GRAD(textureName,{0}coord2, dpdx, dpdy) {1}2D{2}\_GRAD(textureName,{0}coord2, dpdx, dpdy)",  
 "#define ASE\_SAMPLE\_TEXTURE3D(textureName,{0}coord3) {1}3D{2}(textureName,{0}coord3)",  
 "#define ASE\_SAMPLE\_TEXTURE3D\_LOD(textureName,{0}coord3, lod) {1}3D{2}\_LOD(textureName,{0}coord3, lod)",  
 "#define ASE\_SAMPLE\_TEXTURE3D\_BIAS(textureName,{0}coord3, bias) {1}3D{2}\_BIAS(textureName,{0}coord3, bias)",  
 "#define ASE\_SAMPLE\_TEXTURE3D\_GRAD(textureName,{0}coord3, dpdx, dpdy) {1}3D{2}\_GRAD(textureName,{0}coord3, dpdx, dpdy)",  
 "#define ASE\_SAMPLE\_TEXTURECUBE(textureName,{0}coord3) {1}CUBE{2}(textureName,{0}coord3)",  
 "#define ASE\_SAMPLE\_TEXTURECUBE\_LOD(textureName,{0}coord3, lod) {1}CUBE{2}\_LOD(textureName,{0}coord3, lod)",  
 "#define ASE\_SAMPLE\_TEXTURECUBE\_BIAS(textureName,{0}coord3, bias) {1}CUBE{2}\_BIAS(textureName,{0}coord3, bias)\n"  
 };\*/  
 {  
 "#define ASE\_TEXTURE2D\_ARGS(textureName) TEXTURE2D(textureName), SAMPLER(textureName)",  
 "#define ASE\_TEXTURE3D\_ARGS(textureName) TEXTURE3D(textureName), SAMPLER(textureName)",  
 "#define ASE\_TEXTURECUBE\_ARGS(textureName) TEXTURECUBE(textureName), SAMPLER(textureName)",  
 "#define ASE\_TEXTURE2D\_PARAMS(textureName) textureName, sampler##textureName",  
 "#define ASE\_TEXTURE3D\_PARAMS(textureName) textureName, sampler##textureName",  
 "#define ASE\_TEXTURECUBE\_PARAMS(textureName) textureName, sampler##textureName",  
 "#define ASE\_TEXTURE2D\_ARRAY\_PARAMS(textureName) textureName, sampler##textureName\n"  
 };\*/  
 public readonly static RenderTextureFormat PreviewFormat = RenderTextureFormat.ARGBFloat;  
 public readonly static int PreviewSize = 128;  
 public readonly static List<string> UnityNativeInspectors = new List<string>  
 {  
 "Rendering.HighDefinition.LightingShaderGraphGUI",  
 "Rendering.HighDefinition.HDUnlitGUI",  
 "UnityEditor.Rendering.HighDefinition.HDLitGUI",  
 "UnityEditor.ShaderGraph.PBRMasterGUI",  
 "UnityEditor.Rendering.HighDefinition.DecalGUI",  
 "UnityEditor.Rendering.HighDefinition.FabricGUI",  
 "UnityEditor.Experimental.Rendering.HDPipeline.HDLitGUI",  
 "Rendering.HighDefinition.DecalGUI",  
 "Rendering.HighDefinition.LitShaderGraphGUI",  
 "Rendering.HighDefinition.DecalShaderGraphGUI",  
 "UnityEditor.ShaderGraphUnlitGUI",  
 "UnityEditor.ShaderGraphLitGUI",  
 "UnityEditor.Rendering.Universal.DecalShaderGraphGUI"  
 };  
 public readonly static Dictionary<string, string> CustomInspectorHD7To10 = new Dictionary<string, string>  
 {  
 { "UnityEditor.Rendering.HighDefinition.DecalGUI","Rendering.HighDefinition.DecalGUI"},  
 { "UnityEditor.Rendering.HighDefinition.FabricGUI","Rendering.HighDefinition.LightingShaderGraphGUI"},  
 { "UnityEditor.Rendering.HighDefinition.HDLitGUI","Rendering.HighDefinition.LitShaderGraphGUI"},  
 { "UnityEditor.Experimental.Rendering.HDPipeline.HDLitGUI","Rendering.HighDefinition.LitShaderGraphGUI"},  
 };  
 public readonly static Dictionary<string , string> CustomInspectorURP10To12 = new Dictionary<string , string>  
 {  
 { "UnityEditor.ShaderGraph.PBRMasterGUI","UnityEditor.ShaderGraphLitGUI"},  
 };  
 public readonly static Dictionary<string , string> CustomInspectorHDLegacyTo11 = new Dictionary<string , string>  
 {  
 { "UnityEditor.Rendering.HighDefinition.DecalGUI","Rendering.HighDefinition.DecalShaderGraphGUI"},  
 { "Rendering.HighDefinition.DecalGUI","Rendering.HighDefinition.DecalShaderGraphGUI"},  
 { "UnityEditor.Rendering.HighDefinition.FabricGUI","Rendering.HighDefinition.LightingShaderGraphGUI"},  
 { "UnityEditor.Rendering.HighDefinition.HDLitGUI","Rendering.HighDefinition.LitShaderGraphGUI"},  
 { "UnityEditor.Experimental.Rendering.HDPipeline.HDLitGUI","Rendering.HighDefinition.LitShaderGraphGUI"},  
 };  
 public readonly static string CustomASEStandardSamplerParams = "#define ASE\_TEXTURE\_PARAMS(textureName) textureName\n";  
 public readonly static string[] CustomASESRPTextureArrayMacros =   
 {  
 "#define ASE\_TEXTURE2D\_ARRAY\_ARGS(textureName) TEXTURE2D\_ARRAY\_ARGS(textureName,sampler##textureName)\n" ,  
 "#define ASE\_TEXTURE2D\_ARRAY\_PARAM(textureName) TEXTURE2D\_ARRAY\_PARAM(textureName,sampler##textureName)\n" ,  
 "#define ASE\_SAMPLE\_TEXTURE2D\_ARRAY(textureName, coord3) textureName.Sample(sampler##textureName, coord3)",  
 "#define ASE\_SAMPLE\_TEXTURE2D\_ARRAY\_LOD(textureName, coord3, lod) textureName.SampleLevel(sampler##textureName, coord3, lod)"  
 };  
 public readonly static string CustomASESRPSamplerParams = "#define ASE\_TEXTURE\_PARAMS(textureName) textureName, sampler##textureName\n";  
 public readonly static string[] CustomSRPSamplingMacros =  
 {  
 "#if defined(SHADER\_API\_D3D11) || defined(SHADER\_API\_XBOXONE) || defined(UNITY\_COMPILER\_HLSLCC) || defined(SHADER\_API\_PSSL) || (defined(SHADER\_TARGET\_SURFACE\_ANALYSIS) && !defined(SHADER\_TARGET\_SURFACE\_ANALYSIS\_MOJOSHADER))//3D SRP MACROS",  
 "#define SAMPLE\_TEXTURE3D\_GRAD(textureName, samplerName, coord3, dpdx, dpdy) textureName.SampleGrad(samplerName, coord3, dpdx, dpdy)",  
 "#define SAMPLE\_TEXTURE3D\_BIAS(textureName, samplerName, coord3, bias) textureName.SampleBias(samplerName, coord3, bias)",  
 "#else//3D SRP MACROS",  
 "#define SAMPLE\_TEXTURE3D\_GRAD(textureName, samplerName, coord3, dpdx, dpdy) SAMPLE\_TEXTURE3D(textureName, samplerName, coord3)",  
 "#define SAMPLE\_TEXTURE3D\_BIAS(textureName, samplerName, coord3, bias) SAMPLE\_TEXTURE3D(textureName, samplerName, coord3)",  
 "#endif//3D SRP MACROS\n"  
 };  
 public readonly static Dictionary<TextureType, string> TexDeclarationSRPMacros = new Dictionary<TextureType, string>  
 {  
 { TextureType.Texture2D,"TEXTURE2D({0}); SAMPLER(sampler{0});"},  
 { TextureType.Texture3D,"TEXTURE3D({0}); SAMPLER(sampler{0});"},  
 { TextureType.Cube,"TEXTURECUBE({0}); SAMPLER(sampler{0});"},  
 { TextureType.Texture2DArray,"TEXTURE2D\_ARRAY({0}); SAMPLER(sampler{0});"},  
 };  
 public readonly static Dictionary<TextureType, string> SamplerDeclarationSRPMacros = new Dictionary<TextureType, string>  
 {  
 { TextureType.Texture2D,"SAMPLER(sampler{0});"},  
 { TextureType.Texture3D,"SAMPLER(sampler{0});"},  
 { TextureType.Cube,"SAMPLER(sampler{0});"},  
 { TextureType.Texture2DArray,"SAMPLER(sampler{0});"},  
 };  
 public readonly static Dictionary<TextureType, string> TexDeclarationNoSamplerSRPMacros = new Dictionary<TextureType, string>  
 {  
 { TextureType.Texture2D,"TEXTURE2D({0})"},  
 { TextureType.Texture3D,"TEXTURE3D({0})"},  
 { TextureType.Cube,"TEXTURECUBE({0})"},  
 { TextureType.Texture2DArray,"TEXTURE2D\_ARRAY({0})"},  
 };  
 public readonly static Dictionary<TextureType, string> TexSampleSRPMacros = new Dictionary<TextureType, string>  
 {  
 { TextureType.Texture2D,"SAMPLE\_TEXTURE2D{0}( {1}, {2}, {3} )"},  
 { TextureType.Texture3D,"SAMPLE\_TEXTURE3D{0}( {1}, {2}, {3} )"},  
 { TextureType.Cube,"SAMPLE\_TEXTURECUBE{0}( {1}, {2}, {3} )"},  
 { TextureType.Texture2DArray,"SAMPLE\_TEXTURE2D\_ARRAY{0}( {1}, {2}, {3} )"},  
 };  
 public readonly static Dictionary<TextureType, string> TexParams = new Dictionary<TextureType, string>  
 {  
 { TextureType.Texture2D,"ASE\_TEXTURE2D\_PARAMS({0})"},  
 { TextureType.Texture3D,"ASE\_TEXTURE3D\_PARAMS({0})"},  
 { TextureType.Cube,"ASE\_TEXTURECUBE\_PARAMS({0})"},  
 { TextureType.Texture2DArray,"ASE\_TEXTURE2D\_ARRAY\_PARAMS({0})"},  
 };  
 public readonly static Dictionary<WirePortDataType, TextureType> WireToTexture = new Dictionary<WirePortDataType, TextureType>  
 {  
 { WirePortDataType.SAMPLER1D,TextureType.Texture1D},  
 { WirePortDataType.SAMPLER2D,TextureType.Texture2D},  
 { WirePortDataType.SAMPLER3D,TextureType.Texture3D},  
 { WirePortDataType.SAMPLERCUBE,TextureType.Cube},  
 { WirePortDataType.SAMPLER2DARRAY,TextureType.Texture2DArray},  
 };  
 public readonly static Dictionary<TextureType, WirePortDataType> TextureToWire = new Dictionary<TextureType, WirePortDataType>  
 {  
 { TextureType.Texture1D,WirePortDataType.SAMPLER1D},  
 { TextureType.Texture2D,WirePortDataType.SAMPLER2D},  
 { TextureType.Texture3D, WirePortDataType.SAMPLER3D},  
 { TextureType.Cube,WirePortDataType.SAMPLERCUBE},  
 { TextureType.Texture2DArray,WirePortDataType.SAMPLER2DARRAY},  
 { TextureType.ProceduralTexture,WirePortDataType.SAMPLER2D},  
 };  
 public readonly static string SamplingMacrosDirective = "#define ASE\_USING\_SAMPLING\_MACROS 1";  
 public readonly static string[] CustomASEStandarSamplingMacrosHelper =  
 {  
 "#if defined(SHADER\_API\_D3D11) || defined(SHADER\_API\_XBOXONE) || defined(UNITY\_COMPILER\_HLSLCC) || defined(SHADER\_API\_PSSL) || (defined(SHADER\_TARGET\_SURFACE\_ANALYSIS) && !defined(SHADER\_TARGET\_SURFACE\_ANALYSIS\_MOJOSHADER))//ASE Sampler Macros",  
 "#if defined(SHADER\_API\_D3D11) || defined(SHADER\_API\_XBOXONE) || defined(UNITY\_COMPILER\_HLSLCC) || defined(SHADER\_API\_PSSL)//ASE Sampler Macros",  
 "#else//ASE Sampling Macros",  
 "#endif//ASE Sampling Macros\n"  
 };  
 public readonly static string[] CustomASEArraySamplingMacrosRecent =  
 {  
 "#define UNITY\_SAMPLE\_TEX2DARRAY(tex,coord) tex.Sample(sampler##tex,coord)",  
 "#define UNITY\_SAMPLE\_TEX2DARRAY\_LOD(tex,coord,lod) tex.SampleLevel(sampler##tex,coord, lod)",  
 "#define UNITY\_SAMPLE\_TEX2DARRAY\_BIAS(tex,coord,bias) tex.SampleBias(sampler##tex,coord,bias)",  
 "#define UNITY\_SAMPLE\_TEX2DARRAY\_GRAD(tex,coord,ddx,ddy) tex.SampleGrad(sampler##tex,coord,ddx,ddy)",  
 };  
 public readonly static string[] CustomASEArraySamplingMacrosOlder =  
 {  
 "#define UNITY\_SAMPLE\_TEX2DARRAY(tex,coord) tex2DArray(tex,coord)",  
 "#define UNITY\_SAMPLE\_TEX2DARRAY\_LOD(tex,coord,lod) tex2DArraylod(tex, float4(coord,lod))",  
 "#define UNITY\_SAMPLE\_TEX2DARRAY\_BIAS(tex,coord,bias) tex2DArray(tex,coord)",  
 "#define UNITY\_SAMPLE\_TEX2DARRAY\_GRAD(tex,coord,ddx,ddy) tex2DArray(tex,coord)",  
 };  
 public readonly static string[] CustomASEStandarSamplingMacrosRecent =  
 {  
 "#define SAMPLE\_TEXTURE2D(tex,samplerTex,coord) tex.Sample(samplerTex,coord)",  
 "#define SAMPLE\_TEXTURE2D\_LOD(tex,samplerTex,coord,lod) tex.SampleLevel(samplerTex,coord, lod)",  
 "#define SAMPLE\_TEXTURE2D\_BIAS(tex,samplerTex,coord,bias) tex.SampleBias(samplerTex,coord,bias)",  
 "#define SAMPLE\_TEXTURE2D\_GRAD(tex,samplerTex,coord,ddx,ddy) tex.SampleGrad(samplerTex,coord,ddx,ddy)",  
 "#define SAMPLE\_TEXTURE3D(tex,samplerTex,coord) tex.Sample(samplerTex,coord)",  
 "#define SAMPLE\_TEXTURE3D\_LOD(tex,samplerTex,coord,lod) tex.SampleLevel(samplerTex,coord, lod)",  
 "#define SAMPLE\_TEXTURE3D\_BIAS(tex,samplerTex,coord,bias) tex.SampleBias(samplerTex,coord,bias)",  
 "#define SAMPLE\_TEXTURE3D\_GRAD(tex,samplerTex,coord,ddx,ddy) tex.SampleGrad(samplerTex,coord,ddx,ddy)",  
 "#define SAMPLE\_TEXTURECUBE(tex,samplerTex,coord) tex.Sample(samplerTex,coord)",  
 "#define SAMPLE\_TEXTURECUBE\_LOD(tex,samplerTex,coord,lod) tex.SampleLevel(samplerTex,coord, lod)",  
 "#define SAMPLE\_TEXTURECUBE\_BIAS(tex,samplerTex,coord,bias) tex.SampleBias(samplerTex,coord,bias)",  
 "#define SAMPLE\_TEXTURECUBE\_GRAD(tex,samplerTex,coord,ddx,ddy) tex.SampleGrad(samplerTex,coord,ddx,ddy)",  
 "#define SAMPLE\_TEXTURE2D\_ARRAY(tex,samplerTex,coord) tex.Sample(samplerTex,coord)",  
 "#define SAMPLE\_TEXTURE2D\_ARRAY\_LOD(tex,samplerTex,coord,lod) tex.SampleLevel(samplerTex,coord, lod)",  
 "#define SAMPLE\_TEXTURE2D\_ARRAY\_BIAS(tex,samplerTex,coord,bias) tex.SampleBias(samplerTex,coord,bias)",  
 "#define SAMPLE\_TEXTURE2D\_ARRAY\_GRAD(tex,samplerTex,coord,ddx,ddy) tex.SampleGrad(samplerTex,coord,ddx,ddy)",  
 };  
 public readonly static string[] CustomASEStandarSamplingMacrosOlder =  
 {  
 "#define SAMPLE\_TEXTURE2D(tex,samplerTex,coord) tex2D(tex,coord)",  
 "#define SAMPLE\_TEXTURE2D\_LOD(tex,samplerTex,coord,lod) tex2Dlod(tex,float4(coord,0,lod))",  
 "#define SAMPLE\_TEXTURE2D\_BIAS(tex,samplerTex,coord,bias) tex2Dbias(tex,float4(coord,0,bias))",  
 "#define SAMPLE\_TEXTURE2D\_GRAD(tex,samplerTex,coord,ddx,ddy) tex2Dgrad(tex,coord,ddx,ddy)",  
 "#define SAMPLE\_TEXTURE3D(tex,samplerTex,coord) tex3D(tex,coord)",  
 "#define SAMPLE\_TEXTURE3D\_LOD(tex,samplerTex,coord,lod) tex3Dlod(tex,float4(coord,lod))",  
 "#define SAMPLE\_TEXTURE3D\_BIAS(tex,samplerTex,coord,bias) tex3D(tex,coord)",  
 "#define SAMPLE\_TEXTURE3D\_GRAD(tex,samplerTex,coord,ddx,ddy) tex3D(tex,coord)",  
 "#define SAMPLE\_TEXTURECUBE(tex,samplertex,coord) texCUBE(tex,coord)",  
 "#define SAMPLE\_TEXTURECUBE\_LOD(tex,samplertex,coord,lod) texCUBElod (tex,half4(coord,lod))",  
 "#define SAMPLE\_TEXTURECUBE\_BIAS(tex,samplertex,coord,bias) texCUBE(tex,coord)",  
 "#define SAMPLE\_TEXTURECUBE\_GRAD(tex,samplertex,coord,ddx,ddy) texCUBE(tex,coord)",  
 "#define SAMPLE\_TEXTURE2D\_ARRAY(tex,samplertex,coord) tex2DArray(tex,coord)",  
 "#define SAMPLE\_TEXTURE2D\_ARRAY\_LOD(tex,samplertex,coord,lod) tex2DArraylod(tex, float4(coord,lod))",  
 "#define SAMPLE\_TEXTURE2D\_ARRAY\_BIAS(tex,samplerTex,coord,bias) tex2DArray(tex,coord)",  
 "#define SAMPLE\_TEXTURE2D\_ARRAY\_GRAD(tex,samplerTex,coord,ddx,ddy) tex2DArray(tex,coord)",  
 };  
 public readonly static string[] CustomArraySamplingMacros =  
 {  
 "#if defined(UNITY\_COMPILER\_HLSL2GLSL) || defined(SHADER\_TARGET\_SURFACE\_ANALYSIS)//ASE Array Sampler Macros",  
 "#define ASE\_SAMPLE\_TEX2DARRAY\_GRAD(tex,coord,dx,dy) UNITY\_SAMPLE\_TEX2DARRAY (tex,coord)",  
 "#else//ASE Array Sampler Macros",  
 "#define ASE\_SAMPLE\_TEX2DARRAY\_GRAD(tex,coord,dx,dy) tex.SampleGrad (sampler##tex,coord,dx,dy)",  
 "#endif//ASE Array Sampler Macros\n"  
 };  
 public readonly static Dictionary<TextureType, string> TexDeclarationStandardMacros = new Dictionary<TextureType, string>  
 {  
 { TextureType.Texture2D,"UNITY\_DECLARE\_TEX2D({0});"},  
 { TextureType.Texture3D,"UNITY\_DECLARE\_TEX3D({0});"},  
 { TextureType.Cube,"UNITY\_DECLARE\_TEXCUBE({0});"},  
 { TextureType.Texture2DArray,"UNITY\_DECLARE\_TEX2DARRAY({0});"}  
 };  
 public readonly static Dictionary<TextureType, string> TexDeclarationNoSamplerStandardMacros = new Dictionary<TextureType, string>  
 {  
 { TextureType.Texture2D,"UNITY\_DECLARE\_TEX2D\_NOSAMPLER({0})"},  
 { TextureType.Texture3D,"UNITY\_DECLARE\_TEX3D\_NOSAMPLER({0})"},  
 { TextureType.Cube,"UNITY\_DECLARE\_TEXCUBE\_NOSAMPLER({0})"},  
 { TextureType.Texture2DArray,"UNITY\_DECLARE\_TEX2DARRAY\_NOSAMPLER({0})"}  
 };  
 public readonly static Dictionary<TextureType, string> TexSampleStandardMacros = new Dictionary<TextureType, string>  
 {  
 { TextureType.Texture2D,"UNITY\_SAMPLE\_TEX2D{0}( {1}, {3} )"},  
 { TextureType.Texture3D,"UNITY\_SAMPLE\_TEX3D{0}( {1}, {3} )"},  
 { TextureType.Cube,"UNITY\_SAMPLE\_TEXCUBE{0}( {1}, {3} )"},  
 { TextureType.Texture2DArray,"UNITY\_SAMPLE\_TEX2DARRAY{0}( {1}, {3} )"}  
 };  
 public readonly static Dictionary<TextureType, string> TexSampleSamplerStandardMacros = new Dictionary<TextureType, string>  
 {  
 { TextureType.Texture2D,"SAMPLE\_TEXTURE2D{0}( {1}, {2}, {3} )"},  
 { TextureType.Texture3D,"SAMPLE\_TEXTURE3D{0}( {1}, {2}, {3} )"},  
 { TextureType.Cube,"SAMPLE\_TEXTURECUBE{0}( {1}, {2}, {3} )"},  
 { TextureType.Texture2DArray,"SAMPLE\_TEXTURE2D\_ARRAY{0}( {1}, {2}, {3} )"}  
 };  
 public readonly static Dictionary<TextureType, string> TexSampleStandard = new Dictionary<TextureType, string>  
 {  
 { TextureType.Texture2D,"tex2D{0}( {1}, {2} )"},  
 { TextureType.Texture3D,"tex3D{0}( {1}, {2} )"},  
 { TextureType.Cube,"texCUBE{0}( {1}, {2} )"},  
 { TextureType.Texture2DArray,"tex2DArray{0}( {1}, {2} )"}  
 };  
 public readonly static char LineFeedSeparator = '$';  
 public readonly static char SemiColonSeparator = '@';  
 public readonly static string AppDataFullName = "appdata\_full";  
 public readonly static string CustomAppDataFullName = "appdata\_full\_custom";  
 public readonly static string CustomAppDataFullBody =  
 "\n\t\tstruct appdata\_full\_custom\n" +  
 "\t\t{\n" +  
 "\t\t\tfloat4 vertex : POSITION;\n" +  
 "\t\t\tfloat4 tangent : TANGENT;\n" +  
 "\t\t\tfloat3 normal : NORMAL;\n" +  
 "\t\t\tfloat4 texcoord : TEXCOORD0;\n" +  
 "\t\t\tfloat4 texcoord1 : TEXCOORD1;\n" +  
 "\t\t\tfloat4 texcoord2 : TEXCOORD2;\n" +  
 "\t\t\tfloat4 texcoord3 : TEXCOORD3;\n" +  
 "\t\t\tfloat4 color : COLOR;\n" +  
 "\t\t\tUNITY\_VERTEX\_INPUT\_INSTANCE\_ID\n";  
 public readonly static string IncludeFormat = "#include \"{0}\"";  
 public readonly static string PragmaFormat = "#pragma {0}";  
 public readonly static string DefineFormat = "#define {0}";  
 public readonly static string RenderTypeHelperStr = "RenderType";  
 public readonly static string RenderQueueHelperStr = "Queue";  
 public readonly static string DisableBatchingHelperStr = "DisableBatching";  
 public readonly static string DefaultShaderName = "New Amplify Shader";  
 public readonly static string UndoReplaceMasterNodeId = "Replacing Master Node";  
 public readonly static string UnityLightingLib = "Lighting.cginc";  
 public readonly static string UnityAutoLightLib = "AutoLight.cginc";  
 public readonly static string UnityBRDFLib = "UnityStandardBRDF.cginc";  
 public readonly static string LocalValueDecWithoutIdent = "{0} {1} = {2};";  
 public readonly static string CustomTypeLocalValueDecWithoutIdent = "{0} {1} =({0}){2};";  
 public readonly static string LocalValueDefWithoutIdent = "{0} {1} {2};";  
 public readonly static string TilingOffsetFormat = "{0} \* {1} + {2}";  
 public static string InvalidPostProcessDatapath = "\_\_DELETED\_GUID\_Trash";  
 public static float PlusMinusButtonLayoutWidth = 15;  
 public static float NodeButtonSizeX = 16;  
 public static float NodeButtonSizeY = 16;  
 public static float NodeButtonDeltaX = 5;  
 public static float NodeButtonDeltaY = 11;  
 public readonly static string SafeNormalizeInfoStr = "With Safe Normalize division by 0 is prevented over the normalize operation at the expense of additional instructions on shader.";  
 public readonly static string ReservedPropertyNameStr = "Property name '{0}' is reserved and cannot be used";  
 public readonly static string NumericPropertyNameStr = "Property name '{0}' is numeric thus cannot be used";  
 public readonly static string DeprecatedMessageStr = "Node '{0}' is deprecated. Use node '{1}' instead.";  
 public readonly static string DeprecatedNoAlternativeMessageStr = "Node '{0}' is deprecated and should be removed.";  
 public readonly static string UndoChangePropertyTypeNodesId = "Changing Property Types";  
 public readonly static string UndoChangeTypeNodesId = "Changing Nodes Types";  
 public readonly static string UndoMoveNodesId = "Moving Nodes";  
 public readonly static string UndoRegisterFullGrapId = "Register Graph";  
 public readonly static string UndoAddNodeToCommentaryId = "Add node to Commentary";  
 public readonly static string UndoRemoveNodeFromCommentaryId = "Remove node from Commentary";  
 public readonly static string UndoCreateDynamicPortId = "Create Dynamic Port";  
 public readonly static string UndoDeleteDynamicPortId = "Destroy Dynamic Port";  
 public readonly static string UndoRegisterNodeId = "Register Object";  
 public readonly static string UndoUnregisterNodeId = "Unregister Object";  
 public readonly static string UndoCreateNodeId = "Create Object";  
 public readonly static string UndoPasteNodeId = "Paste Object";  
 public readonly static string UndoDeleteNodeId = "Destroy Object";  
 public readonly static string UndoDeleteConnectionId = "Destroy Connection";  
 public readonly static string UndoCreateConnectionId = "Create Connection";  
 public readonly static float MenuDragSpeed = -0.5f;  
 public readonly static string DefaultCustomInspector = "ASEMaterialInspector";  
 public readonly static string ReferenceTypeStr = "Mode";  
 public readonly static string AvailableReferenceStr = "Reference";  
 public readonly static string InstancePostfixStr = " (Reference) ";  
 public readonly static string ASEMenuName = "Amplify Shader";  
 public readonly static string LodCrossFadeOption2017 = "dithercrossfade";  
 public readonly static string UnityShaderVariables = "UnityShaderVariables.cginc";  
 public readonly static string UnityCgLibFuncs = "UnityCG.cginc";  
 public readonly static string UnityStandardUtilsLibFuncs = "UnityStandardUtils.cginc";  
 public readonly static string UnityPBSLightingLib = "UnityPBSLighting.cginc";  
 public readonly static string UnityDeferredLightLib = "UnityDeferredLibrary.cginc";  
 public readonly static string ATSharedLibGUID = "ba242738c4be3324aa88d126f7cc19f9";  
 public readonly static string CameraDepthTextureValue = "UNITY\_DECLARE\_DEPTH\_TEXTURE( \_CameraDepthTexture );";  
 public readonly static string CameraDepthTextureValue = "uniform sampler2D \_CameraDepthTexture;";  
 public readonly static string CameraDepthTextureLWEnabler = "REQUIRE\_DEPTH\_TEXTURE 1";  
 public readonly static string CameraDepthTextureTexelSize = "uniform float4 \_CameraDepthTexture\_TexelSize;";  
 public readonly static string InstanceIdMacro = "UNITY\_VERTEX\_INPUT\_INSTANCE\_ID";  
 public readonly static string InstanceIdVariable = "UNITY\_GET\_INSTANCE\_ID({0})";  
 public readonly static string HelpURL = "http://wiki.amplify.pt/index.php?title=Unity\_Products:Amplify\_Shader\_Editor";  
 public readonly static string NodeCommonUrl = "http://wiki.amplify.pt/index.php?title=Unity\_Products:Amplify\_Shader\_Editor/";  
 public readonly static string CommunityNodeCommonUrl = "http://wiki.amplify.pt/index.php?title=Unity\_Products:Amplify\_Shader\_Editor/";  
 public readonly static Color InfiniteLoopColor = Color.red;  
 public readonly static Color DefaultCategoryColor = new Color( 0.26f, 0.35f, 0.44f, 1.0f );  
 public readonly static Color NodeBodyColor = new Color( 1f, 1f, 1f, 1.0f );  
 public readonly static Color ModeTextColor = new Color( 1f, 1f, 1f, 0.25f );  
 public readonly static Color ModeIconColor = new Color( 1f, 1f, 1f, 0.75f );  
 public readonly static Color PortTextColor = new Color( 1f, 1f, 1f, 0.5f );  
 public readonly static Color PortLockedTextColor = new Color( 1f, 1f, 1f, 0.35f );  
 public readonly static Color BoxSelectionColor = new Color( 0.5f, 0.75f, 1f, 0.33f );  
 public readonly static Color SpecialRegisterLocalVarSelectionColor = new Color( 0.27f, 0.52f, 1.0f, 1f );  
 public readonly static Color SpecialGetLocalVarSelectionColor = new Color( 0.2f, 0.8f, 0.4f, 1f );  
 public readonly static Color NodeSelectedColor = new Color( 0.85f, 0.56f, 0f, 1f );  
 public readonly static Color NodeDefaultColor = new Color( 1f, 1f, 1f, 1f );  
 public readonly static Color NodeConnectedColor = new Color( 1.0f, 1f, 0.0f, 1f );  
 public readonly static Color NodeErrorColor = new Color( 1f, 0.5f, 0.5f, 1f );  
 public readonly static string NoSpecifiedCategoryStr = "<None>";  
 public readonly static int MINIMIZE\_WINDOW\_LOCK\_SIZE = 630;  
 public readonly static int FoldoutMouseId = 0; // Left Mouse Button  
 public readonly static float SNAP\_SQR\_DIST = 200f;  
 public readonly static int INVALID\_NODE\_ID = -1;  
 public readonly static float WIRE\_WIDTH = 7f;  
 public readonly static float WIRE\_CONTROL\_POINT\_DIST = 0.7f;  
 public readonly static float WIRE\_CONTROL\_POINT\_DIST\_INV = 1.7f;  
 public readonly static float IconsLeftRightMargin = 5f;  
 public readonly static float PropertyPickerWidth = 16f;  
 public readonly static float PropertyPickerHeight = 16f;  
 public readonly static float PreviewExpanderWidth = 16f;  
 public readonly static float PreviewExpanderHeight = 16f;  
 public readonly static float TextFieldFontSize = 11f;  
 public readonly static float DefaultFontSize = 15f;  
 public readonly static float DefaultTitleFontSize = 13f;  
 public readonly static float PropertiesTitleFontSize = 11f;  
 public readonly static float MessageFontSize = 40f;  
 public readonly static float SelectedObjectFontSize = 30f;  
 public readonly static float PORT\_X\_ADJUST = 10;  
 public readonly static float PORT\_INITIAL\_X = 10;  
 public readonly static float PORT\_INITIAL\_Y = 40;  
 public readonly static float INPUT\_PORT\_DELTA\_Y = 5;  
 public readonly static float PORT\_TO\_LABEL\_SPACE\_X = 5;  
 public readonly static float NODE\_HEADER\_HEIGHT = 32;  
 public readonly static float NODE\_HEADER\_EXTRA\_HEIGHT = 5;  
 public readonly static float NODE\_HEADER\_LEFTRIGHT\_MARGIN = 10;  
 public readonly static float MULTIPLE\_SELECION\_BOX\_ALPHA = 0.5f;  
 public readonly static float RMB\_CLICK\_DELTA\_TIME = 0.1f;  
 public readonly static float RMB\_SCREEN\_DIST = 10f;  
 public readonly static float CAMERA\_MAX\_ZOOM = 2f;  
 public readonly static float CAMERA\_MIN\_ZOOM = 1f;  
 public readonly static float CAMERA\_ZOOM\_SPEED = 0.1f;  
 public readonly static float ALT\_CAMERA\_ZOOM\_SPEED = -0.05f;  
 public readonly static object INVALID\_VALUE = null;  
 public readonly static float HORIZONTAL\_TANGENT\_SIZE = 100f;  
 public readonly static float OUTSIDE\_WIRE\_MARGIN = 5f;  
 public readonly static string SubTitleNameFormatStr = "Name( {0} )";  
 public readonly static string SubTitleSpaceFormatStr = "Space( {0} )";  
 public readonly static string SubTitleTypeFormatStr = "Type( {0} )";  
 public readonly static string SubTitleValueFormatStr = "Value( {0} )";  
 public readonly static string SubTitleConstFormatStr = "Const( {0} )";  
 public readonly static string SubTitleVarNameFormatStr = "Var( {0} )";  
 public readonly static string SubTitleRefNameFormatStr = "Ref( {0} )";  
 public readonly static string CodeWrapper = "( {0} )";  
 public readonly static string InlineCodeWrapper = "{{\n{0}\n}}";  
 public readonly static string NodesDumpFormat = "{0}:,{1},{2}\n";  
 public readonly static string TagFormat = " \"{0}\" = \"{1}\"";  
 public readonly static string LocalVarIdentation = "\t\t\t";  
 public readonly static string SimpleLocalValueDec = LocalVarIdentation + "{0} {1};\n";  
 public readonly static string LocalValueDec = LocalVarIdentation + LocalValueDecWithoutIdent + '\n';  
 public readonly static string LocalValueDef = LocalVarIdentation + "{0} = {1};\n";  
 public readonly static string CastHelper = "({0}).{1}";  
 public readonly static string PropertyLocalVarDec = "{0} {1} = {0}({2});";  
 public readonly static string[] UniformDec = { "uniform {0} {1};", "{0} {1};" };  
 public readonly static string PropertyValueLabel = "Value( {0} )";  
 public readonly static string ConstantsValueLabel = "Const( {0} )";  
 public readonly static string PropertyFloatFormatLabel = "0.###";  
 public readonly static string PropertyBigFloatFormatLabel = "0.###e+0";  
 public readonly static string PropertyIntFormatLabel = "0";  
 public readonly static string PropertyBigIntFormatLabel = "0e+0";  
 public readonly static string PropertyVectorFormatLabel = "0.##";  
 public readonly static string PropertyBigVectorFormatLabel = "0.##e+0";  
 public readonly static string PropertyMatrixFormatLabel = "0.#";  
 public readonly static string PropertyBigMatrixFormatLabel = "0.#e+0";  
 public readonly static string NoPropertiesLabel = "No assigned properties";  
 public readonly static string ValueLabel = "Value";  
 public readonly static string DefaultValueLabel = "Default Value";  
 public readonly static string MaterialValueLabel = "Material Value";  
 public readonly static GUIContent DefaultValueLabelContent = new GUIContent( "Default Value" );  
 public readonly static GUIContent MaterialValueLabelContent = new GUIContent( "Material Value" );  
 public readonly static string InputVarStr = "i";//"input";  
 public readonly static string OutputVarStr = "o";//"output";  
 public readonly static string CustomLightOutputVarStr = "s";  
 public readonly static string CustomLightStructStr = "Custom";  
 public readonly static string VertexShaderOutputStr = "o";  
 public readonly static string VertexShaderInputStr = "v";//"vertexData";  
 public readonly static string VertexDataFunc = "vertexDataFunc";  
 public readonly static string VirtualCoordNameStr = "vcoord";  
 public readonly static string VertexVecNameStr = "vertexVec";  
 public readonly static string VertexVecDecStr = "float3 " + VertexVecNameStr;  
 public readonly static string VertexVecVertStr = VertexShaderOutputStr + "." + VertexVecNameStr;  
 public readonly static string NormalVecNameStr = "normalVec";  
 public readonly static string NormalVecDecStr = "float3 " + NormalVecNameStr;  
 public readonly static string NormalVecFragStr = InputVarStr + "." + NormalVecNameStr;  
 public readonly static string NormalVecVertStr = VertexShaderOutputStr + "." + NormalVecNameStr;  
 public readonly static string IncidentVecNameStr = "incidentVec";  
 public readonly static string IncidentVecDecStr = "float3 " + IncidentVecNameStr;  
 public readonly static string IncidentVecDefStr = VertexShaderOutputStr + "." + IncidentVecNameStr + " = normalize( " + VertexVecNameStr + " - \_WorldSpaceCameraPos.xyz)";  
 public readonly static string IncidentVecFragStr = InputVarStr + "." + IncidentVecNameStr;  
 public readonly static string IncidentVecVertStr = VertexShaderOutputStr + "." + IncidentVecNameStr;  
 public readonly static string WorldNormalLocalDecStr = "WorldNormalVector( " + Constants.InputVarStr + " , {0}( 0,0,1 ))";  
 public readonly static string VFaceVariable = "ASEVFace";  
 public readonly static string VFaceInput = "half ASEVFace : VFACE";  
 public readonly static string ColorVariable = "vertexColor";  
 public readonly static string ColorInput = "float4 vertexColor : COLOR";  
 public readonly static string NoStringValue = "None";  
 public readonly static string EmptyPortValue = " ";  
 public readonly static string[] OverallInvalidChars = { "\r", "\n", "\\", " ", ".", ">", ",", "<", "\'", "\"", ";", ":", "[", "{", "]", "}","|", "=", "+", "`", "~", "/", "?", "!", "@", "#", "$", "%", "^", "&", "\*", "(", ")", "-" };  
 public readonly static string[] ShaderInvalidChars = { "\r", "\n", "\\", "\'", "\"", };  
 public readonly static string[] EnumInvalidChars = { "\r", "\n", "\\", ".", ">", ",", "<", "\'", "\"", ";", ":", "[", "{", "]", "}", "=", "+", "`", "~", "/", "?", "!", "@", "#", "$", "%", "^", "&", "\*", "(", ")", "-" };  
 public readonly static string[] AttrInvalidChars = { "\r", "\n", "\\", ">", "<", "\'", "\"", ";", ":", "[", "{", "]", "}", "=", "+", "`", "~", "/", "?", "!", "@", "#", "$", "%", "^", "&", "\*" };  
 public readonly static string[] HeaderInvalidChars = { "\r", "\n", "\\", ">", ",", "<", "\'", "\"", ";", ":", "[", "{", "]", "}", "=", "+", "`", "~", "/", "?", "!", "@", "#", "$", "%", "^", "&", "\*", "(", ")", "-" };  
 public readonly static string[] WikiInvalidChars = { "#", "<", ">", "[", "]", "|", "{", "}", "%", "+", "?", "\\", "/", ",", ";", "." };  
 public readonly static string[,] UrlReplacementStringValues =   
 {  
 { " = ", "Equals" },  
 { " == ", "Equals" },  
 { " != ", "NotEqual" },  
 { " \u2260 ", "NotEqual" },  
 { " > ", "Greater" },  
 { " \u2265 " , "GreaterOrEqual" },  
 { " >= ", "GreaterOrEqual" },  
 { " < ", "Less" },  
 { " \u2264 ", "LessOrEqual" },  
 { " <= ", "LessOrEqual" },  
 { " ", "\_" },  
 { "[", string.Empty },  
 { "]", string.Empty }  
 };  
 public readonly static int UrlReplacementStringValuesLen = UrlReplacementStringValues.Length / 2;  
 public readonly static string[,] ReplacementStringValues =  
 {  
 { " = ", "Equals" },  
 { " == ", "Equals" },  
 { " != ", "NotEqual" },  
 { " \u2260 ", "NotEqual" },  
 { " > ", "Greater" },  
 { " \u2265 ", "GreaterOrEqual" },  
 { " >= ", "GreaterOrEqual" },  
 { " < ", "Less" },  
 { " \u2264 ", "LessOrEqual" },  
 { " <= ", "LessOrEqual" }  
 };  
 public readonly static int ReplacementStringValuesLen = ReplacementStringValues.Length / 2;  
 public readonly static string InternalData = "INTERNAL\_DATA";  
 public readonly static string NoMaterialStr = "None";  
 public readonly static string OptionalParametersSep = " ";  
 public readonly static string NodeUndoId = "NODE\_UNDO\_ID";  
 public readonly static string NodeCreateUndoId = "NODE\_CREATE\_UNDO\_ID";  
 public readonly static string NodeDestroyUndoId = "NODE\_DESTROY\_UNDO\_ID";  
 public readonly static string CNIP = "#IP";  
 public readonly static float FLOAT\_DRAW\_HEIGHT\_FIELD\_SIZE = 16f;  
 public readonly static float FLOAT\_DRAW\_WIDTH\_FIELD\_SIZE = 45f;  
 public readonly static float FLOAT\_WIDTH\_SPACING = 3f;  
 public readonly static Color LockedPortColor = new Color( 0.3f, 0.3f, 0.3f, 0.5f );  
 public readonly static int[] AvailableUVChannels = { 0, 1, 2, 3, 4, 5, 6, 7 };  
 public readonly static string[] AvailableUVChannelsStr = { "0", "1", "2", "3", "4", "5", "6", "7"};  
 public readonly static string AvailableUVChannelLabel = "UV Channel";  
 public readonly static int[] AvailableUVSets = { 0, 1, 2, 3, 4, 5, 6, 7 };  
 public readonly static string[] AvailableUVSetsStr = { "1", "2", "3", "4","5", "6", "7", "8" };  
 public readonly static string AvailableUVSetsLabel = "UV Set";  
 public readonly static int[] AvailableUVChannels = { 0, 1, 2, 3 };  
 public readonly static string[] AvailableUVChannelsStr = { "0", "1", "2", "3" };  
 public readonly static string AvailableUVChannelLabel = "UV Channel";  
 public readonly static int[] AvailableUVSets = { 0, 1, 2, 3 };  
 public readonly static string[] AvailableUVSetsStr = { "1", "2", "3", "4" };  
 public readonly static string AvailableUVSetsLabel = "UV Set";  
 public readonly static int[] AvailableUVSizes = { 2, 3, 4 };  
 public readonly static string[] AvailableUVSizesStr = { "Float 2", "Float 3", "Float 4" };  
 public readonly static string AvailableUVSizesLabel = "Coord Size";  
 public readonly static string LineSeparator = "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_";  
 public readonly static Vector2 CopyPasteDeltaPos = new Vector2( 40, 40 );  
 public readonly static string[] VectorSuffixes = { ".x", ".y", ".z", ".w" };  
 public readonly static string[] ColorSuffixes = { ".r", ".g", ".b", ".a" };  
 public const string InternalDataLabelStr = "Internal Data";  
 public const string AttributesLaberStr = "Attributes";  
 public const string ParameterLabelStr = "Parameters";  
 public static readonly string[] ReferenceArrayLabels = { "Object", "Reference" };  
 public static readonly string[] ChannelNamesVector = { "X", "Y", "Z", "W" };  
 public static readonly string[] ChannelNamesColor = { "R", "G", "B", "A" };  
 public static readonly string SamplerFormat = "sampler{0}";  
 public static readonly string SamplerDeclFormat = "SamplerState {0}";  
 public static readonly string SamplerDeclSRPFormat = "SAMPLER({0})";  
 }  
}

Constants.cs.meta

fileFormatVersion: 2  
guid: d833dd0968f913f449477da6bcd56b48  
timeCreated: 1481126959  
licenseType: Store  
MonoImporter:  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

ASEBeginDecorator.cs

using UnityEngine;  
using UnityEditor;  
using System;  
using AmplifyShaderEditor;  
public class ASEBeginDecorator : MaterialPropertyDrawer  
{  
 const int Separator = 2;  
 public override void OnGUI( Rect position, MaterialProperty prop, String label, MaterialEditor editor )  
 {  
 Rect button = position;  
 button.height = EditorGUIUtility.singleLineHeight;  
 if( GUI.Button( button, "Open in Shader Editor" ) )  
 {  
 Material mat = editor.target as Material;  
 ASEPackageManagerHelper.SetupLateMaterial( mat );  
 AmplifyShaderEditorWindow.LoadMaterialToASE( mat );  
 }  
 }  
 public override float GetPropertyHeight( MaterialProperty prop, string label, MaterialEditor editor )  
 {  
 return EditorGUIUtility.singleLineHeight + Separator;  
 }  
}

ASEBeginDecorator.cs.meta

fileFormatVersion: 2  
guid: 508788a7fa76e1d42ad5fdfb1c941ed2  
MonoImporter:  
 externalObjects: {}  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

ASEEndDecorator.cs

using UnityEngine;  
using UnityEditor;  
using System;  
using AmplifyShaderEditor;  
public class ASEEndDecorator : MaterialPropertyDrawer  
{  
 bool m\_applyNext = false;  
 public override void OnGUI( Rect position, MaterialProperty prop, String label, MaterialEditor editor )  
 {  
 if( prop.applyPropertyCallback == null )  
 prop.applyPropertyCallback = Testc;  
 if( GUI.changed || m\_applyNext )  
 {  
 m\_applyNext = false;  
 Material mat = editor.target as Material;  
 UIUtils.CopyValuesFromMaterial( mat );  
 }  
 }  
 bool Testc( MaterialProperty prop, int changeMask, object previousValue )  
 {  
 m\_applyNext = true;  
 return false;  
 }  
 public override float GetPropertyHeight( MaterialProperty prop, string label, MaterialEditor editor )  
 {  
 return 0;  
 }  
}

ASEEndDecorator.cs.meta

fileFormatVersion: 2  
guid: fdf2e52babbbbf040b3b9f6df50243f3  
MonoImporter:  
 externalObjects: {}  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

EditableIf.cs

using UnityEngine;  
using UnityEditor;  
using System;  
public enum ComparisonOperators  
{  
 EqualTo, NotEqualTo, GreaterThan, LessThan, EqualsOrGreaterThan, EqualsOrLessThan, ContainsFlags,  
 DoesNotContainsFlags  
}  
public class EditableIf : MaterialPropertyDrawer  
{  
 ComparisonOperators op;  
 string FieldName = "";  
 object ExpectedValue;  
 bool InputError;  
 public EditableIf()  
 {  
 InputError = true;  
 }  
 public EditableIf( object fieldname, object comparison, object expectedvalue )  
 {  
 if( expectedvalue.ToString().ToLower() == "true" )  
 {  
 expectedvalue = (System.Single)1;  
 }  
 else if( expectedvalue.ToString().ToLower() == "false" )  
 {  
 expectedvalue = (System.Single)0;  
 }  
 Init( fieldname, comparison, expectedvalue );  
 }  
 public EditableIf( object fieldname, object comparison, object expectedvaluex, object expectedvaluey )  
 {  
 float? x = expectedvaluex as float?;  
 float? y = expectedvaluey as float?;  
 float? z = float.NegativeInfinity;  
 float? w = float.NegativeInfinity;  
 x = GetVectorValue( x );  
 y = GetVectorValue( y );  
 Init( fieldname, comparison, new Vector4( x.Value, y.Value, z.Value, w.Value ) );  
 }  
 public EditableIf( object fieldname, object comparison, object expectedvaluex, object expectedvaluey, object expectedvaluez )  
 {  
 float? x = expectedvaluex as float?;  
 float? y = expectedvaluey as float?;  
 float? z = expectedvaluez as float?;  
 float? w = float.NegativeInfinity;  
 x = GetVectorValue( x );  
 y = GetVectorValue( y );  
 z = GetVectorValue( z );  
 Init( fieldname, comparison, new Vector4( x.Value, y.Value, z.Value, w.Value ) );  
 }  
 public EditableIf( object fieldname, object comparison, object expectedvaluex, object expectedvaluey, object expectedvaluez, object expectedvaluew )  
 {  
 var x = expectedvaluex as float?;  
 var y = expectedvaluey as float?;  
 var z = expectedvaluez as float?;  
 var w = expectedvaluew as float?;  
 x = GetVectorValue( x );  
 y = GetVectorValue( y );  
 z = GetVectorValue( z );  
 w = GetVectorValue( w );  
 Init( fieldname, comparison, new Vector4( x.Value, y.Value, z.Value, w.Value ) );  
 }  
 private void Init( object fieldname, object comparison, object expectedvalue )  
 {  
 FieldName = fieldname.ToString();  
 var names = Enum.GetNames( typeof( ComparisonOperators ) );  
 var name = comparison.ToString().ToLower().Replace( " ", "" );  
 for( int i = 0; i < names.Length; i++ )  
 {  
 if( names[ i ].ToLower() == name )  
 {  
 op = (ComparisonOperators)i;  
 break;  
 }  
 }  
 ExpectedValue = expectedvalue;  
 }  
 private static float? GetVectorValue( float? x )  
 {  
 if( x.HasValue == false )  
 {  
 x = float.NegativeInfinity;  
 }  
 return x;  
 }  
 public override void OnGUI( Rect position, MaterialProperty prop, String label, MaterialEditor editor )  
 {  
 if( InputError )  
 {  
 EditorGUI.LabelField( position, "EditableIf Attribute Error: Input parameters are invalid!" );  
 return;  
 }  
 var LHSprop = MaterialEditor.GetMaterialProperty( prop.targets, FieldName );  
 if( string.IsNullOrEmpty( LHSprop.name ) )  
 {  
 LHSprop = MaterialEditor.GetMaterialProperty( prop.targets, "\_" + FieldName.Replace( " ", "" ) );  
 if( string.IsNullOrEmpty( LHSprop.name ) )  
 {  
 EditorGUI.LabelField( position, "EditableIf Attribute Error: " + FieldName + " Does not exist!" );  
 return;  
 }  
 }  
 object LHSVal = null;  
 bool test = false;  
 switch( LHSprop.type )  
 {  
 case MaterialProperty.PropType.Color:  
 case MaterialProperty.PropType.Vector:  
 LHSVal = LHSprop.type == MaterialProperty.PropType.Color ? (Vector4)LHSprop.colorValue : LHSprop.vectorValue;  
 var v4 = ExpectedValue as Vector4?;  
 v4 = v4.HasValue ? v4 : new Vector4( (System.Single)ExpectedValue, float.NegativeInfinity, float.NegativeInfinity, float.NegativeInfinity );  
 if( LHSprop.type == MaterialProperty.PropType.Color )  
 {  
 test = VectorCheck( (Vector4)LHSVal, op, v4 / 255 );  
 }  
 else  
 test = VectorCheck( (Vector4)LHSVal, op, v4 );  
 break;  
 case MaterialProperty.PropType.Range:  
 case MaterialProperty.PropType.Float:  
 LHSVal = LHSprop.floatValue;  
 test = ( Check( LHSVal, op, ExpectedValue ) );  
 break;  
 case MaterialProperty.PropType.Texture:  
 LHSVal = LHSprop.textureValue;  
 test = ( CheckObject( LHSVal, op, ExpectedValue ) );  
 break;  
 }  
 GUI.enabled = test;  
 editor.DefaultShaderProperty( position, prop, label );  
 GUI.enabled = true;  
 }  
 private bool VectorCheck( Vector4 LHS, ComparisonOperators op, object expectedValue )  
 {  
 var RHS = (Vector4)expectedValue;  
 if( RHS.x != float.NegativeInfinity )  
 {  
 if( !Check( LHS.x, op, RHS.x ) )  
 return false;  
 }  
 if( RHS.y != float.NegativeInfinity )  
 {  
 if( !Check( LHS.y, op, RHS.y ) )  
 return false;  
 }  
 if( RHS.z != float.NegativeInfinity )  
 {  
 if( !Check( LHS.z, op, RHS.z ) )  
 return false;  
 }  
 if( RHS.w != float.NegativeInfinity )  
 {  
 if( !Check( LHS.w, op, RHS.w ) )  
 return false;  
 }  
 return true;  
 }  
 protected bool Check( object LHS, ComparisonOperators op, object RHS )  
 {  
 if( !( LHS is IComparable ) || !( RHS is IComparable ) )  
 throw new Exception( "Check using non basic type" );  
 switch( op )  
 {  
 case ComparisonOperators.EqualTo:  
 return ( (IComparable)LHS ).CompareTo( RHS ) == 0;  
 case ComparisonOperators.NotEqualTo:  
 return ( (IComparable)LHS ).CompareTo( RHS ) != 0;  
 case ComparisonOperators.EqualsOrGreaterThan:  
 return ( (IComparable)LHS ).CompareTo( RHS ) >= 0;  
 case ComparisonOperators.EqualsOrLessThan:  
 return ( (IComparable)LHS ).CompareTo( RHS ) <= 0;  
 case ComparisonOperators.GreaterThan:  
 return ( (IComparable)LHS ).CompareTo( RHS ) > 0;  
 case ComparisonOperators.LessThan:  
 return ( (IComparable)LHS ).CompareTo( RHS ) < 0;  
 case ComparisonOperators.ContainsFlags:  
 return ( (int)LHS & (int)RHS ) != 0; // Dont trust LHS values, it has been casted to a char and then to an int again, first bit will be the sign  
 case ComparisonOperators.DoesNotContainsFlags:  
 return ( ( (int)LHS & (int)RHS ) == (int)LHS ); // Dont trust LHS values, it has been casted to a char and then to an int again, first bit will be the sign  
 default:  
 break;  
 }  
 return false;  
 }  
 private bool CheckObject( object LHS, ComparisonOperators comparasonOperator, object RHS )  
 {  
 switch( comparasonOperator )  
 {  
 case ComparisonOperators.EqualTo:  
 return ( LHS == null );  
 case ComparisonOperators.NotEqualTo:  
 return ( LHS != null );  
 }  
 return true;  
 }  
}

EditableIf.cs.meta

fileFormatVersion: 2  
guid: 7a5504a2b7d04a846978416748dc6e0a  
timeCreated: 1520330108  
licenseType: Store  
MonoImporter:  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

NoKeywordToggle.cs

using UnityEngine;  
using UnityEditor;  
using System;  
public class NoKeywordToggle : MaterialPropertyDrawer  
{  
 public override void OnGUI(Rect position, MaterialProperty prop, String label, MaterialEditor editor) {  
 bool value = (prop.floatValue != 0.0f);  
 EditorGUI.BeginChangeCheck();  
 {  
 EditorGUI.showMixedValue = prop.hasMixedValue;  
 value = EditorGUI.Toggle( position, label, value );  
 EditorGUI.showMixedValue = false;  
 }  
 if (EditorGUI.EndChangeCheck())  
 {  
 prop.floatValue = value ? 1.0f : 0.0f;  
 }  
 }  
}

NoKeywordToggle.cs.meta

fileFormatVersion: 2  
guid: e1a000d43a26286499b39a7571e5c61b  
timeCreated: 1605540234  
licenseType: Store  
MonoImporter:  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

RemapSliders.cs

using UnityEngine;  
using UnityEditor;  
using System;  
public class RemapSliders : MaterialPropertyDrawer  
{  
 public override void OnGUI( Rect position, MaterialProperty prop, String label, MaterialEditor editor )  
 {  
 EditorGUI.BeginChangeCheck();  
 Vector4 value = prop.vectorValue;  
 EditorGUI.showMixedValue = prop.hasMixedValue;  
 var cacheLabel = EditorGUIUtility.labelWidth;  
 var cacheField = EditorGUIUtility.fieldWidth;  
 if( cacheField <= 64 )  
 {  
 float total = position.width;  
 EditorGUIUtility.labelWidth = Mathf.Ceil( 0.45f \* total ) - 30;  
 EditorGUIUtility.fieldWidth = Mathf.Ceil( 0.55f \* total ) + 30;  
 }  
 EditorGUI.MinMaxSlider(position, label, ref value.x, ref value.y, 0, 1 );  
 EditorGUIUtility.labelWidth = cacheLabel;  
 EditorGUIUtility.fieldWidth = cacheField;  
 EditorGUI.showMixedValue = false;  
 if( EditorGUI.EndChangeCheck() )  
 {  
 prop.vectorValue = value;  
 }  
 }  
}

RemapSliders.cs.meta

fileFormatVersion: 2  
guid: 314af1bcecbba6c4d92cbb5843c221ba  
MonoImporter:  
 externalObjects: {}  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

RemapSlidersFull.cs

using UnityEngine;  
using UnityEditor;  
using System;  
public class RemapSlidersFull : MaterialPropertyDrawer  
{  
 public override void OnGUI( Rect position, MaterialProperty prop, String label, MaterialEditor editor )  
 {  
 EditorGUI.BeginChangeCheck();  
 Vector4 value = prop.vectorValue;  
 EditorGUI.showMixedValue = prop.hasMixedValue;  
 var cacheLabel = EditorGUIUtility.labelWidth;  
 var cacheField = EditorGUIUtility.fieldWidth;  
 if( cacheField <= 64 )  
 {  
 float total = position.width;  
 EditorGUIUtility.labelWidth = Mathf.Ceil( 0.45f \* total ) - 30;  
 EditorGUIUtility.fieldWidth = Mathf.Ceil( 0.55f \* total ) + 30;  
 }  
 EditorGUI.MinMaxSlider( position, label, ref value.x, ref value.y, value.z, value.w );  
 EditorGUIUtility.labelWidth = cacheLabel;  
 EditorGUIUtility.fieldWidth = cacheField;  
 EditorGUI.showMixedValue = false;  
 if( EditorGUI.EndChangeCheck() )  
 {  
 prop.vectorValue = value;  
 }  
 }  
}

RemapSlidersFull.cs.meta

fileFormatVersion: 2  
guid: 9a724dcf5c5ddef40bcef06f0b2c8ec0  
MonoImporter:  
 externalObjects: {}  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

SingleLineTexture.cs

using UnityEngine;  
using UnityEditor;  
using System;  
public class SingleLineTexture : MaterialPropertyDrawer  
{  
 public override void OnGUI( Rect position, MaterialProperty prop, String label, MaterialEditor editor )  
 {  
 EditorGUI.BeginChangeCheck();  
 EditorGUI.showMixedValue = prop.hasMixedValue;  
 Texture value = editor.TexturePropertyMiniThumbnail( position, prop, label, string.Empty );  
 EditorGUI.showMixedValue = false;  
 if( EditorGUI.EndChangeCheck() )  
 {  
 prop.textureValue = value;  
 }  
 }  
}

SingleLineTexture.cs.meta

fileFormatVersion: 2  
guid: 85da32683d237ac4f8665251e2ac38dc  
MonoImporter:  
 externalObjects: {}  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

CustomDrawers.meta

fileFormatVersion: 2  
guid: 2206c4bd7f3d18643a6a3452b0c070d1  
folderAsset: yes  
timeCreated: 1522769470  
licenseType: Store  
DefaultImporter:  
 userData:   
 assetBundleName:   
 assetBundleVariant:

DoCreateFunction.cs

using UnityEditor;  
using UnityEditor.ProjectWindowCallback;  
namespace AmplifyShaderEditor  
{  
 public class DoCreateFunction : EndNameEditAction  
 {  
 public override void Action( int instanceId, string pathName, string resourceFile )  
 {  
 UnityEngine.Object obj = EditorUtility.InstanceIDToObject( instanceId );  
 AssetDatabase.CreateAsset( obj, AssetDatabase.GenerateUniqueAssetPath( pathName ) );  
 AmplifyShaderEditorWindow.LoadShaderFunctionToASE( (AmplifyShaderFunction)obj, false );  
 }  
 }  
}

DoCreateFunction.cs.meta

fileFormatVersion: 2  
guid: 3f2c950b0ed192943b7484f6b551965f  
timeCreated: 1493906087  
licenseType: Store  
MonoImporter:  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

DoCreateShader.cs

using UnityEngine;  
using UnityEditor;  
using UnityEditor.ProjectWindowCallback;  
using System.IO;  
namespace AmplifyShaderEditor  
{  
 public class DoCreateStandardShader : EndNameEditAction  
 {  
 public override void Action( int instanceId, string pathName, string resourceFile )  
 {  
 string uniquePath = AssetDatabase.GenerateUniqueAssetPath( pathName );  
 string shaderName = Path.GetFileName( uniquePath );  
 if( IOUtils.AllOpenedWindows.Count > 0 )  
 {  
 EditorWindow openedWindow = AmplifyShaderEditorWindow.GetWindow<AmplifyShaderEditorWindow>();  
 AmplifyShaderEditorWindow currentWindow = AmplifyShaderEditorWindow.CreateTab();  
 WindowHelper.AddTab( openedWindow, currentWindow );  
 UIUtils.CurrentWindow = currentWindow;   
 }  
 else  
 {  
 AmplifyShaderEditorWindow currentWindow = AmplifyShaderEditorWindow.OpenWindow( shaderName, UIUtils.ShaderIcon );  
 UIUtils.CurrentWindow = currentWindow;  
 }  
 Shader shader = UIUtils.CreateNewEmpty( uniquePath, shaderName );  
 ProjectWindowUtil.ShowCreatedAsset( shader );  
 }  
 }  
 public class DoCreateTemplateShader : EndNameEditAction  
 {  
 public override void Action( int instanceId, string pathName, string resourceFile )  
 {  
 string uniquePath = AssetDatabase.GenerateUniqueAssetPath( pathName );  
 string shaderName = Path.GetFileName( uniquePath );  
 if( !string.IsNullOrEmpty( UIUtils.NewTemplateGUID ) )  
 {  
 Shader shader = AmplifyShaderEditorWindow.CreateNewTemplateShader( UIUtils.NewTemplateGUID, uniquePath, shaderName );  
 ProjectWindowUtil.ShowCreatedAsset( shader );  
 }  
 }  
 }  
}

DoCreateShader.cs.meta

fileFormatVersion: 2  
guid: 2cfa7290f61ad684f99f8d81328ad52c  
timeCreated: 1573664425  
licenseType: Store  
MonoImporter:  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

EditorOptions.cs

using UnityEditor;  
namespace AmplifyShaderEditor  
{  
 [System.Serializable]  
 public class OptionsWindow  
 {  
 private AmplifyShaderEditorWindow m\_parentWindow = null;  
 private bool m\_coloredPorts = true;  
 private bool m\_multiLinePorts = true;  
 private const string MultiLineId = "MultiLinePortsDefault";  
 private const string ColorPortId = "ColoredPortsDefault";  
 public OptionsWindow( AmplifyShaderEditorWindow parentWindow )  
 {  
 m\_parentWindow = parentWindow;  
 }  
 public void Init()  
 {  
 Load();  
 }  
 public void Destroy()  
 {  
 Save();  
 }  
 public void Save()  
 {  
 EditorPrefs.SetBool( ColorPortId, ColoredPorts );  
 EditorPrefs.SetBool( MultiLineId, m\_multiLinePorts );  
 }  
 public void Load()  
 {  
 ColoredPorts = EditorPrefs.GetBool( ColorPortId, true );  
 m\_multiLinePorts = EditorPrefs.GetBool( MultiLineId, true );  
 }  
 public bool ColoredPorts  
 {  
 get { return m\_coloredPorts; }  
 set  
 {  
 if ( m\_coloredPorts != value )  
 EditorPrefs.SetBool( ColorPortId, value );  
 m\_coloredPorts = value;  
 }  
 }  
 public bool MultiLinePorts  
 {  
 get { return m\_multiLinePorts; }  
 set  
 {  
 if ( m\_multiLinePorts != value )  
 EditorPrefs.SetBool( MultiLineId, value );  
 m\_multiLinePorts = value;  
 }  
 }  
 public AmplifyShaderEditorWindow ParentWindow { get { return m\_parentWindow; } set { m\_parentWindow = value; } }  
 }  
}

EditorOptions.cs.meta

fileFormatVersion: 2  
guid: 44cb06bc7bfe6e84aa8b5e8b702eb2dd  
timeCreated: 1481126955  
licenseType: Store  
MonoImporter:  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

NodeGrid.cs

using System.Collections.Generic;  
using UnityEngine;  
using UnityEditor;  
namespace AmplifyShaderEditor  
{  
 public class NodeGrid  
 {  
 private bool m\_debugGrid = false;  
 private const float GRID\_SIZE\_X = 100;  
 private const float GRID\_SIZE\_Y = 100;  
 private const float GRID\_AREA\_X = 1000;  
 private const float GRID\_AREA\_Y = 1000;  
 private Dictionary<int, Dictionary<int, List<ParentNode>>> m\_grid;  
 private int m\_xMin = int.MaxValue;  
 private int m\_yMin = int.MaxValue;  
 private int m\_xMax = int.MinValue;  
 private int m\_yMax = int.MinValue;  
 public NodeGrid()  
 {  
 m\_grid = new Dictionary<int, Dictionary<int, List<ParentNode>>>();  
 }  
 public void AddNodeToGrid( ParentNode node )  
 {  
 Rect pos = node.Position;  
 if ( Mathf.Abs( pos.width ) < 0.001f || Mathf.Abs( pos.height ) < 0.001f )  
 {  
 return;  
 }  
 float initialXf = pos.x / GRID\_SIZE\_X;  
 float initialYf = pos.y / GRID\_SIZE\_Y;  
 int endX = Mathf.CeilToInt( initialXf + pos.width / GRID\_SIZE\_X );  
 int endY = Mathf.CeilToInt( initialYf + pos.height / GRID\_SIZE\_Y );  
 int initialX = Mathf.FloorToInt( initialXf );  
 int initialY = Mathf.FloorToInt( initialYf );  
 if ( initialX < m\_xMin )  
 {  
 m\_xMin = initialX;  
 }  
 if ( initialY < m\_yMin )  
 {  
 m\_yMin = initialY;  
 }  
 if ( endX > m\_xMax )  
 {  
 m\_xMax = endX;  
 }  
 if ( endY > m\_yMax )  
 {  
 m\_yMax = endY;  
 }  
 for ( int x = initialX; x < endX; x += 1 )  
 {  
 for ( int y = initialY; y < endY; y += 1 )  
 {  
 if ( !m\_grid.ContainsKey( x ) )  
 {  
 m\_grid.Add( x, new Dictionary<int, List<ParentNode>>() );  
 }  
 if ( !m\_grid[ x ].ContainsKey( y ) )  
 {  
 m\_grid[ x ].Add( y, new List<ParentNode>() );  
 }  
 m\_grid[ x ][ y ].Add( node );  
 }  
 }  
 node.IsOnGrid = true;  
 }  
 public void RemoveNodeFromGrid( ParentNode node, bool useCachedPos )  
 {  
 Rect pos = useCachedPos ? node.CachedPos : node.Position;  
 if ( Mathf.Abs( pos.width ) < 0.001f || Mathf.Abs( pos.height ) < 0.001f )  
 {  
 return;  
 }  
 float initialXf = pos.x / GRID\_SIZE\_X;  
 float initialYf = pos.y / GRID\_SIZE\_Y;  
 int endX = Mathf.CeilToInt( initialXf + pos.width / GRID\_SIZE\_X );  
 int endY = Mathf.CeilToInt( initialYf + pos.height / GRID\_SIZE\_Y );  
 int initialX = Mathf.FloorToInt( initialXf );  
 int initialY = Mathf.FloorToInt( initialYf );  
 bool testLimits = false;  
 int xMinCount = 0;  
 int xMaxCount = 0;  
 int yMinCount = 0;  
 int yMaxCount = 0;  
 for ( int x = initialX; x < endX; x += 1 )  
 {  
 for ( int y = initialY; y < endY; y += 1 )  
 {  
 if ( m\_grid.ContainsKey( x ) )  
 {  
 if ( m\_grid[ x ].ContainsKey( y ) )  
 {  
 m\_grid[ x ][ y ].Remove( node );  
 node.IsOnGrid = false;  
 if ( initialX == m\_xMin && x == initialX )  
 {  
 testLimits = true;  
 if ( m\_grid[ x ][ y ].Count != 0 )  
 {  
 xMinCount += 1;  
 }  
 }  
 if ( endX == m\_xMax && x == endX )  
 {  
 testLimits = true;  
 if ( m\_grid[ x ][ y ].Count != 0 )  
 {  
 xMaxCount += 1;  
 }  
 }  
 if ( initialY == m\_yMin && y == initialY )  
 {  
 testLimits = true;  
 if ( m\_grid[ x ][ y ].Count != 0 )  
 {  
 yMinCount += 1;  
 }  
 }  
 if ( endY == m\_yMax && y == endY )  
 {  
 testLimits = true;  
 if ( m\_grid[ x ][ y ].Count != 0 )  
 {  
 yMaxCount += 1;  
 }  
 }  
 }  
 }  
 }  
 }  
 if ( testLimits )  
 {  
 if ( xMinCount == 0 || xMaxCount == 0 || yMinCount == 0 || yMaxCount == 0 )  
 {  
 m\_xMin = int.MaxValue;  
 m\_yMin = int.MaxValue;  
 m\_xMax = int.MinValue;  
 m\_yMax = int.MinValue;  
 foreach ( KeyValuePair<int, Dictionary<int, List<ParentNode>>> entryX in m\_grid )  
 {  
 foreach ( KeyValuePair<int, List<ParentNode>> entryY in entryX.Value )  
 {  
 if ( entryY.Value.Count > 0 )  
 {  
 if ( entryX.Key < m\_xMin )  
 {  
 m\_xMin = entryX.Key;  
 }  
 if ( entryY.Key < m\_yMin )  
 {  
 m\_yMin = entryY.Key;  
 }  
 if ( entryX.Key > m\_xMax )  
 {  
 m\_xMax = entryX.Key;  
 }  
 if ( entryY.Key > m\_yMax )  
 {  
 m\_yMax = entryY.Key;  
 }  
 }  
 }  
 }  
 m\_xMax += 1;  
 m\_yMax += 1;  
 }  
 }  
 }  
 public void DebugLimits()  
 {  
 Debug.Log( "[ " + m\_xMin + " , " + m\_yMin + " ] " + "[ " + m\_xMax + " , " + m\_yMax + " ] " );  
 }  
 public List<ParentNode> GetNodesOn( Vector2 pos )  
 {  
 int x = Mathf.FloorToInt( pos.x / GRID\_SIZE\_X );  
 int y = Mathf.FloorToInt( pos.y / GRID\_SIZE\_Y );  
 if ( m\_grid.ContainsKey( x ) )  
 {  
 if ( m\_grid[ x ].ContainsKey( y ) )  
 {  
 return m\_grid[ x ][ y ];  
 }  
 }  
 return null;  
 }  
 public List<ParentNode> GetNodesOn( int x, int y )  
 {  
 if ( m\_grid.ContainsKey( x ) )  
 {  
 if ( m\_grid[ x ].ContainsKey( y ) )  
 {  
 return m\_grid[ x ][ y ];  
 }  
 }  
 return null;  
 }  
 public void DrawGrid( DrawInfo drawInfo )  
 {  
 if ( m\_debugGrid )  
 {  
 Handles.CircleHandleCap( 0, drawInfo.InvertedZoom \* ( new Vector3( drawInfo.CameraOffset.x, drawInfo.CameraOffset.y, 0f ) ), Quaternion.identity, 5,EventType.Layout );  
 for ( int x = -( int ) GRID\_AREA\_X; x < GRID\_AREA\_X; x += ( int ) GRID\_SIZE\_X )  
 {  
 Handles.DrawLine( drawInfo.InvertedZoom \* ( new Vector3( x + drawInfo.CameraOffset.x, drawInfo.CameraOffset.y - GRID\_AREA\_Y, 0 ) ), drawInfo.InvertedZoom \* ( new Vector3( drawInfo.CameraOffset.x + x, drawInfo.CameraOffset.y + GRID\_AREA\_Y, 0 ) ) );  
 }  
 for ( int y = -( int ) GRID\_AREA\_Y; y < GRID\_AREA\_X; y += ( int ) GRID\_SIZE\_Y )  
 {  
 Handles.DrawLine( drawInfo.InvertedZoom \* ( new Vector3( drawInfo.CameraOffset.x - GRID\_AREA\_X, drawInfo.CameraOffset.y + y, 0 ) ), drawInfo.InvertedZoom \* ( new Vector3( drawInfo.CameraOffset.x + GRID\_AREA\_X, drawInfo.CameraOffset.y + y, 0 ) ) );  
 }  
 }  
 }  
 public void Destroy()  
 {  
 foreach ( KeyValuePair<int, Dictionary<int, List<ParentNode>>> entryX in m\_grid )  
 {  
 foreach ( KeyValuePair<int, List<ParentNode>> entryY in entryX.Value )  
 {  
 entryY.Value.Clear();  
 }  
 entryX.Value.Clear();  
 }  
 m\_grid.Clear();  
 }  
 public float MaxNodeDist  
 {  
 get { return Mathf.Max( ( m\_xMax - m\_xMin )\*GRID\_SIZE\_X, ( m\_yMax - m\_yMin )\*GRID\_SIZE\_Y ); }  
 }  
 }  
}

NodeGrid.cs.meta

fileFormatVersion: 2  
guid: 6344917ce0eed6b43840632b98a2ed57  
timeCreated: 1481126956  
licenseType: Store  
MonoImporter:  
 serializedVersion: 2  
 defaultReferences: []  
 executionOrder: 0  
 icon: {instanceID: 0}  
 userData:   
 assetBundleName:   
 assetBundleVariant:

ParentGraph.cs

using UnityEngine;  
using UnityEditor;  
using System;  
using System.Collections.Generic;  
namespace AmplifyShaderEditor  
{  
 [Serializable]  
 public class ParentGraph : ScriptableObject, ISerializationCallbackReceiver  
 {  
 private const int MasterNodeLODIncrement = 100;  
 private const int MaxLodAmount = 9;  
 public enum NodeLOD  
 {  
 LOD0,  
 LOD1,  
 LOD2,  
 LOD3,  
 LOD4,  
 LOD5  
 }  
 [SerializeField]  
 private bool m\_samplingThroughMacros = false;  
 private NodeLOD m\_lodLevel = NodeLOD.LOD0;  
 private GUIStyle nodeStyleOff;  
 private GUIStyle nodeStyleOn;  
 private GUIStyle nodeTitle;  
 private GUIStyle commentaryBackground;  
 public delegate void LODMasterNodesAdded( int lod );  
 public event LODMasterNodesAdded OnLODMasterNodesAddedEvent;  
 public delegate void EmptyGraphDetected( ParentGraph graph );  
 public event EmptyGraphDetected OnEmptyGraphDetectedEvt;  
 public delegate void NodeEvent( ParentNode node );  
 public event NodeEvent OnNodeEvent = null;  
 public event NodeEvent OnNodeRemovedEvent;  
 public delegate void DuplicateEvent();  
 public event DuplicateEvent OnDuplicateEvent;  
 public event MasterNode.OnMaterialUpdated OnMaterialUpdatedEvent;  
 public event MasterNode.OnMaterialUpdated OnShaderUpdatedEvent;  
 private bool m\_afterDeserializeFlag = true;  
 private bool m\_lateOptionsRefresh = false;  
 private bool m\_foundDuplicates = false;  
 private AmplifyShaderEditorWindow m\_parentWindow = null;  
 [SerializeField]  
 private int m\_validNodeId;  
 [SerializeField]  
 private List<ParentNode> m\_nodes = new List<ParentNode>();  
 [SerializeField]  
 private UsageListSamplerNodes m\_samplerNodes = new UsageListSamplerNodes();  
 [SerializeField]  
 private UsageListFloatIntNodes m\_floatNodes = new UsageListFloatIntNodes();  
 [SerializeField]  
 private UsageListTexturePropertyNodes m\_texturePropertyNodes = new UsageListTexturePropertyNodes();  
 [SerializeField]  
 private UsageListTextureArrayNodes m\_textureArrayNodes = new UsageListTextureArrayNodes();  
 [SerializeField]  
 private UsageListPropertyNodes m\_propertyNodes = new UsageListPropertyNodes();  
 [SerializeField]  
 private UsageListPropertyNodes m\_rawPropertyNodes = new UsageListPropertyNodes();  
 [SerializeField]  
 private UsageListScreenColorNodes m\_screenColorNodes = new UsageListScreenColorNodes();  
 [SerializeField]  
 private UsageListRegisterLocalVarNodes m\_localVarNodes = new UsageListRegisterLocalVarNodes();  
 [SerializeField]  
 private UsageListGlobalArrayNodes m\_globalArrayNodes = new UsageListGlobalArrayNodes();  
 [SerializeField]  
 private UsageListFunctionInputNodes m\_functionInputNodes = new UsageListFunctionInputNodes();  
 [SerializeField]  
 private UsageListFunctionNodes m\_functionNodes = new UsageListFunctionNodes();  
 [SerializeField]  
 private UsageListFunctionOutputNodes m\_functionOutputNodes = new UsageListFunctionOutputNodes();  
 [SerializeField]  
 private UsageListFunctionSwitchNodes m\_functionSwitchNodes = new UsageListFunctionSwitchNodes();  
 [SerializeField]  
 private UsageListFunctionSwitchCopyNodes m\_functionSwitchCopyNodes = new UsageListFunctionSwitchCopyNodes();  
 [SerializeField]  
 private UsageListTemplateMultiPassMasterNodes m\_multiPassMasterNodes = new UsageListTemplateMultiPassMasterNodes();  
 [SerializeField]  
 private List<UsageListTemplateMultiPassMasterNodes> m\_lodMultiPassMasterNodes;  
 [SerializeField]  
 private UsageListCustomExpressionsOnFunctionMode m\_customExpressionsOnFunctionMode = new UsageListCustomExpressionsOnFunctionMode();  
 [SerializeField]  
 private UsageListStaticSwitchNodes m\_staticSwitchNodes = new UsageListStaticSwitchNodes();  
 [SerializeField]  
 private int m\_masterNodeId = Constants.INVALID\_NODE\_ID;  
 [SerializeField]  
 private bool m\_isDirty;  
 [SerializeField]  
 private bool m\_saveIsDirty = false;  
 [SerializeField]  
 private int m\_nodeClicked;  
 [SerializeField]  
 private int m\_loadedShaderVersion;  
 [SerializeField]  
 private int m\_instancePropertyCount = 0;  
 [SerializeField]  
 private int m\_virtualTextureCount = 0;  
 [SerializeField]  
 private int m\_graphId = 0;  
 [SerializeField]  
 private PrecisionType m\_currentPrecision = PrecisionType.Float;  
 [SerializeField]  
 private NodeAvailability m\_currentCanvasMode = NodeAvailability.SurfaceShader;  
 [SerializeField]  
 private TemplateSRPType m\_currentSRPType = TemplateSRPType.BuiltIn;  
 private List<ParentNode> m\_nodePreviewList = new List<ParentNode>();  
 private Dictionary<int, ParentNode> m\_nodesDict = new Dictionary<int, ParentNode>();  
 [NonSerialized]  
 private List<ParentNode> m\_selectedNodes = new List<ParentNode>();  
 [NonSerialized]  
 private List<ParentNode> m\_markedForDeletion = new List<ParentNode>();  
 [SerializeField]  
 private List<WireReference> m\_highlightedWires = new List<WireReference>();  
 private System.Type m\_masterNodeDefaultType;  
 [SerializeField]  
 private List<PropertyNode> m\_internalTemplateNodesList = new List<PropertyNode>();  
 private Dictionary<int, PropertyNode> m\_internalTemplateNodesDict = new Dictionary<int, PropertyNode>();  
 private NodeGrid m\_nodeGrid;  
 private bool m\_markedToDeSelect = false;  
 private int m\_markToSelect = -1;  
 private bool m\_markToReOrder = false;  
 private bool m\_hasUnConnectedNodes = false;  
 private bool m\_checkSelectedWireHighlights = false;  
 [SerializeField]  
 private List<WireBezierReference> m\_bezierReferences;  
 private const int MaxBezierReferences = 50;  
 private int m\_wireBezierCount = 0;  
 protected int m\_normalDependentCount = 0;  
 private bool m\_forceCategoryRefresh = false;  
 [SerializeField]  
 private bool m\_forceRepositionCheck = false;  
 private bool m\_isLoading = false;  
 private bool m\_isDuplicating = false;  
 private bool m\_changedLightingModel = false;  
 public void ResetEvents()  
 {  
 OnNodeEvent = null;  
 OnMaterialUpdatedEvent = null;  
 OnShaderUpdatedEvent = null;  
 OnEmptyGraphDetectedEvt = null;  
 OnNodeRemovedEvent = null;  
 }  
 public void Init()  
 {  
 Undo.undoRedoPerformed += OnUndoRedoCallback;  
 m\_normalDependentCount = 0;  
 m\_nodes = new List<ParentNode>();  
 m\_samplerNodes = new UsageListSamplerNodes();  
 m\_samplerNodes.ContainerGraph = this;  
 m\_samplerNodes.ReorderOnChange = true;  
 m\_floatNodes = new UsageListFloatIntNodes();  
 m\_floatNodes.ContainerGraph = this;  
 m\_texturePropertyNodes = new UsageListTexturePropertyNodes();  
 m\_texturePropertyNodes.ContainerGraph = this;  
 m\_textureArrayNodes = new UsageListTextureArrayNodes();  
 m\_textureArrayNodes.ContainerGraph = this;  
 m\_textureArrayNodes.ReorderOnChange = true;  
 m\_propertyNodes = new UsageListPropertyNodes();  
 m\_propertyNodes.ContainerGraph = this;  
 m\_rawPropertyNodes = new UsageListPropertyNodes();  
 m\_rawPropertyNodes.ContainerGraph = this;  
 m\_customExpressionsOnFunctionMode = new UsageListCustomExpressionsOnFunctionMode();  
 m\_customExpressionsOnFunctionMode.ContainerGraph = this;  
 m\_staticSwitchNodes = new UsageListStaticSwitchNodes();  
 m\_staticSwitchNodes.ContainerGraph = this;  
 m\_staticSwitchNodes.ReorderOnChange = true;  
 m\_screenColorNodes = new UsageListScreenColorNodes();  
 m\_screenColorNodes.ContainerGraph = this;  
 m\_screenColorNodes.ReorderOnChange = true;  
 m\_localVarNodes = new UsageListRegisterLocalVarNodes();  
 m\_localVarNodes.ContainerGraph = this;  
 m\_localVarNodes.ReorderOnChange = true;  
 m\_globalArrayNodes = new UsageListGlobalArrayNodes();  
 m\_globalArrayNodes.ContainerGraph = this;  
 m\_functionInputNodes = new UsageListFunctionInputNodes();  
 m\_functionInputNodes.ContainerGraph = this;  
 m\_functionNodes = new UsageListFunctionNodes();  
 m\_functionNodes.ContainerGraph = this;  
 m\_functionOutputNodes = new UsageListFunctionOutputNodes();  
 m\_functionOutputNodes.ContainerGraph = this;  
 m\_functionSwitchNodes = new UsageListFunctionSwitchNodes();  
 m\_functionSwitchNodes.ContainerGraph = this;  
 m\_functionSwitchCopyNodes = new UsageListFunctionSwitchCopyNodes();  
 m\_functionSwitchCopyNodes.ContainerGraph = this;  
 m\_multiPassMasterNodes = new UsageListTemplateMultiPassMasterNodes();  
 m\_multiPassMasterNodes.ContainerGraph = this;  
 m\_lodMultiPassMasterNodes = new List<UsageListTemplateMultiPassMasterNodes>( MaxLodAmount );  
 for( int i = 0; i < MaxLodAmount; i++ )  
 {  
 m\_lodMultiPassMasterNodes.Add( new UsageListTemplateMultiPassMasterNodes() );  
 }  
 m\_selectedNodes = new List<ParentNode>();  
 m\_markedForDeletion = new List<ParentNode>();  
 m\_highlightedWires = new List<WireReference>();  
 m\_validNodeId = 0;  
 IsDirty = false;  
 SaveIsDirty = false;  
 m\_masterNodeDefaultType = typeof( StandardSurfaceOutputNode );  
 m\_bezierReferences = new List<WireBezierReference>( MaxBezierReferences );  
 for( int i = 0; i < MaxBezierReferences; i++ )  
 {  
 m\_bezierReferences.Add( new WireBezierReference() );  
 }  
 }  
 public void ActivatePreviews( bool value )  
 {  
 int count = m\_nodes.Count;  
 if( value )  
 {  
 for( int i = 0 ; i < count ; i++ )  
 {  
 m\_nodes[ i ].PreviewIsDirty = true;  
 }  
 }  
 else  
 {  
 }  
 }  
 private void OnUndoRedoCallback()  
 {  
 DeSelectAll();  
 }  
 private void OnEnable()  
 {  
 hideFlags = HideFlags.HideAndDontSave;  
 m\_nodeGrid = new NodeGrid();  
 m\_internalTemplateNodesDict = new Dictionary<int, PropertyNode>();  
 m\_nodesDict = new Dictionary<int, ParentNode>();  
 nodeStyleOff = UIUtils.GetCustomStyle( CustomStyle.NodeWindowOff );  
 nodeStyleOn = UIUtils.GetCustomStyle( CustomStyle.NodeWindowOn );  
 nodeTitle = UIUtils.GetCustomStyle( CustomStyle.NodeHeader );  
 commentaryBackground = UIUtils.GetCustomStyle( CustomStyle.CommentaryBackground );  
 }  
 public void UpdateRegisters()  
 {  
 m\_samplerNodes.UpdateNodeArr();  
 m\_propertyNodes.UpdateNodeArr();  
 m\_rawPropertyNodes.UpdateNodeArr();  
 m\_customExpressionsOnFunctionMode.UpdateNodeArr();  
 m\_staticSwitchNodes.UpdateNodeArr();  
 m\_functionInputNodes.UpdateNodeArr();  
 m\_functionNodes.UpdateNodeArr();  
 m\_functionOutputNodes.UpdateNodeArr();  
 m\_functionSwitchNodes.UpdateNodeArr();  
 m\_functionSwitchCopyNodes.UpdateNodeArr();  
 m\_multiPassMasterNodes.UpdateNodeArr();  
 for( int i = 0; i < m\_lodMultiPassMasterNodes.Count; i++ )  
 {  
 m\_lodMultiPassMasterNodes[ i ].UpdateNodeArr();  
 }  
 m\_texturePropertyNodes.UpdateNodeArr();  
 m\_textureArrayNodes.UpdateNodeArr();  
 m\_screenColorNodes.UpdateNodeArr();  
 m\_localVarNodes.UpdateNodeArr();  
 m\_globalArrayNodes.UpdateNodeArr();  
 }  
 public int GetValidId()  
 {  
 return m\_validNodeId++;  
 }  
 void UpdateIdFromNode( ParentNode node )  
 {  
 if( node.UniqueId >= m\_validNodeId )  
 {  
 m\_validNodeId = node.UniqueId + 1;  
 }  
 }  
 public void ResetNodeConnStatus()  
 {  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 if( m\_nodes[ i ].ConnStatus == NodeConnectionStatus.Connected )  
 {  
 m\_nodes[ i ].ConnStatus = NodeConnectionStatus.Not\_Connected;  
 }  
 }  
 }  
 public void CleanUnusedNodes()  
 {  
 List<ParentNode> unusedNodes = new List<ParentNode>();  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 if( m\_nodes[ i ].ConnStatus == NodeConnectionStatus.Not\_Connected )  
 {  
 unusedNodes.Add( m\_nodes[ i ] );  
 }  
 }  
 for( int i = 0; i < unusedNodes.Count; i++ )  
 {  
 DestroyNode( unusedNodes[ i ] );  
 }  
 unusedNodes.Clear();  
 unusedNodes = null;  
 IsDirty = true;  
 }  
 public void ClearGraph()  
 {  
 List<ParentNode> list = new List<ParentNode>();  
 int count = m\_nodes.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 if( m\_nodes[ i ].UniqueId != m\_masterNodeId )  
 {  
 list.Add( m\_nodes[ i ] );  
 }  
 }  
 while( list.Count > 0 )  
 {  
 DestroyNode( list[ 0 ] );  
 list.RemoveAt( 0 );  
 }  
 }  
 public void CleanNodes()  
 {  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 if( m\_nodes[ i ] != null )  
 {  
 Undo.ClearUndo( m\_nodes[ i ] );  
 m\_nodes[ i ].Destroy();  
 GameObject.DestroyImmediate( m\_nodes[ i ] );  
 }  
 }  
 ClearInternalTemplateNodes();  
 m\_masterNodeId = Constants.INVALID\_NODE\_ID;  
 m\_validNodeId = 0;  
 m\_instancePropertyCount = 0;  
 m\_virtualTextureCount = 0;  
 m\_nodesDict.Clear();  
 m\_nodes.Clear();  
 m\_samplerNodes.Clear();  
 m\_propertyNodes.Clear();  
 m\_rawPropertyNodes.Clear();  
 m\_customExpressionsOnFunctionMode.Clear();  
 m\_staticSwitchNodes.Clear();  
 m\_functionInputNodes.Clear();  
 m\_functionNodes.Clear();  
 m\_functionOutputNodes.Clear();  
 m\_functionSwitchNodes.Clear();  
 m\_functionSwitchCopyNodes.Clear();  
 m\_multiPassMasterNodes.Clear();  
 for( int i = 0; i < m\_lodMultiPassMasterNodes.Count; i++ )  
 {  
 m\_lodMultiPassMasterNodes[ i ].Clear();  
 }  
 m\_texturePropertyNodes.Clear();  
 m\_textureArrayNodes.Clear();  
 m\_screenColorNodes.Clear();  
 m\_localVarNodes.Clear();  
 m\_globalArrayNodes.Clear();  
 m\_selectedNodes.Clear();  
 m\_markedForDeletion.Clear();  
 }  
 public void ResetHighlightedWires()  
 {  
 for( int i = 0; i < m\_highlightedWires.Count; i++ )  
 {  
 m\_highlightedWires[ i ].WireStatus = WireStatus.Default;  
 }  
 m\_highlightedWires.Clear();  
 }  
 public void HighlightWiresStartingNode( ParentNode node )  
 {  
 for( int outputIdx = 0; outputIdx < node.OutputPorts.Count; outputIdx++ )  
 {  
 for( int extIdx = 0; extIdx < node.OutputPorts[ outputIdx ].ExternalReferences.Count; extIdx++ )  
 {  
 WireReference wireRef = node.OutputPorts[ outputIdx ].ExternalReferences[ extIdx ];  
 ParentNode nextNode = GetNode( wireRef.NodeId );  
 if( nextNode && nextNode.ConnStatus == NodeConnectionStatus.Connected )  
 {  
 InputPort port = nextNode.GetInputPortByUniqueId( wireRef.PortId );  
 if( port.ExternalReferences.Count == 0 || port.ExternalReferences[ 0 ].WireStatus == WireStatus.Highlighted )  
 {  
 return;  
 }  
 port.ExternalReferences[ 0 ].WireStatus = WireStatus.Highlighted;  
 m\_highlightedWires.Add( port.ExternalReferences[ 0 ] );  
 HighlightWiresStartingNode( nextNode );  
 }  
 }  
 }  
 RegisterLocalVarNode regNode = node as RegisterLocalVarNode;  
 if( (object)regNode != null )  
 {  
 int count = regNode.NodeReferences.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 HighlightWiresStartingNode( regNode.NodeReferences[ i ] );  
 }  
 }  
 }  
 void PropagateHighlightDeselection( ParentNode node, int portId = -1 )  
 {  
 if( portId > -1 )  
 {  
 InputPort port = node.GetInputPortByUniqueId( portId );  
 port.ExternalReferences[ 0 ].WireStatus = WireStatus.Default;  
 }  
 if( node.Selected )  
 return;  
 for( int i = 0; i < node.InputPorts.Count; i++ )  
 {  
 if( node.InputPorts[ i ].ExternalReferences.Count > 0 && node.InputPorts[ i ].ExternalReferences[ 0 ].WireStatus == WireStatus.Highlighted )  
 {  
 return;  
 }  
 }  
 for( int outputIdx = 0; outputIdx < node.OutputPorts.Count; outputIdx++ )  
 {  
 for( int extIdx = 0; extIdx < node.OutputPorts[ outputIdx ].ExternalReferences.Count; extIdx++ )  
 {  
 WireReference wireRef = node.OutputPorts[ outputIdx ].ExternalReferences[ extIdx ];  
 ParentNode nextNode = GetNode( wireRef.NodeId );  
 PropagateHighlightDeselection( nextNode, wireRef.PortId );  
 }  
 }  
 RegisterLocalVarNode regNode = node as RegisterLocalVarNode;  
 if( (object)regNode != null )  
 {  
 int count = regNode.NodeReferences.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 PropagateHighlightDeselection( regNode.NodeReferences[ i ], -1 );  
 }  
 }  
 }  
 public void ResetNodesData()  
 {  
 int count = m\_nodes.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 m\_nodes[ i ].ResetNodeData();  
 }  
 }  
 public void FullCleanUndoStack()  
 {  
 Undo.ClearUndo( this );  
 int count = m\_nodes.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 if( m\_nodes[ i ] != null )  
 {  
 Undo.ClearUndo( m\_nodes[ i ] );  
 }  
 }  
 }  
 public void FullRegisterOnUndoStack()  
 {  
 Undo.RegisterCompleteObjectUndo( this, Constants.UndoRegisterFullGrapId );  
 int count = m\_nodes.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 if( m\_nodes[ i ] != null )  
 {  
 Undo.RegisterCompleteObjectUndo( m\_nodes[ i ], Constants.UndoRegisterFullGrapId );  
 }  
 }  
 }  
 public void CheckPropertiesAutoRegister( ref MasterNodeDataCollector dataCollector )  
 {  
 List<PropertyNode> propertyNodesList = m\_rawPropertyNodes.NodesList;  
 int propertyCount = propertyNodesList.Count;  
 for( int i = 0; i < propertyCount; i++ )  
 {  
 propertyNodesList[ i ].CheckIfAutoRegister( ref dataCollector );  
 }  
 propertyNodesList = null;  
 List<GlobalArrayNode> globalArrayNodeList = m\_globalArrayNodes.NodesList;  
 int globalArrayCount = globalArrayNodeList.Count;  
 for( int i = 0; i < globalArrayCount; i++ )  
 {  
 globalArrayNodeList[ i ].CheckIfAutoRegister( ref dataCollector );  
 }  
 globalArrayNodeList = null;  
 }  
 public void SoftDestroy()  
 {  
 OnNodeRemovedEvent = null;  
 m\_masterNodeId = Constants.INVALID\_NODE\_ID;  
 m\_validNodeId = 0;  
 m\_nodeGrid.Destroy();  
 ClearInternalTemplateNodes();  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 if( m\_nodes[ i ] != null )  
 {  
 m\_nodes[ i ].Destroy();  
 GameObject.DestroyImmediate( m\_nodes[ i ] );  
 }  
 }  
 m\_instancePropertyCount = 0;  
 m\_nodes.Clear();  
 m\_nodesDict.Clear();  
 m\_samplerNodes.Clear();  
 m\_propertyNodes.Clear();  
 m\_rawPropertyNodes.Clear();  
 m\_customExpressionsOnFunctionMode.Clear();  
 m\_staticSwitchNodes.Clear();  
 m\_functionInputNodes.Clear();  
 m\_functionNodes.Clear();  
 m\_functionOutputNodes.Clear();  
 m\_functionSwitchNodes.Clear();  
 m\_functionSwitchCopyNodes.Clear();  
 m\_texturePropertyNodes.Clear();  
 m\_textureArrayNodes.Clear();  
 m\_screenColorNodes.Clear();  
 m\_localVarNodes.Clear();  
 m\_globalArrayNodes.Clear();  
 m\_selectedNodes.Clear();  
 m\_markedForDeletion.Clear();  
 m\_nodePreviewList.Clear();  
 IsDirty = true;  
 OnNodeEvent = null;  
 OnDuplicateEvent = null;  
 OnMaterialUpdatedEvent = null;  
 OnShaderUpdatedEvent = null;  
 OnEmptyGraphDetectedEvt = null;  
 nodeStyleOff = null;  
 nodeStyleOn = null;  
 nodeTitle = null;  
 commentaryBackground = null;  
 OnLODMasterNodesAddedEvent = null;  
 }  
 public void Destroy()  
 {  
 Undo.undoRedoPerformed -= OnUndoRedoCallback;  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 if( m\_nodes[ i ] != null )  
 {  
 Undo.ClearUndo( m\_nodes[ i ] );  
 m\_nodes[ i ].Destroy();  
 GameObject.DestroyImmediate( m\_nodes[ i ] );  
 }  
 }  
 ClearInternalTemplateNodes();  
 m\_internalTemplateNodesDict = null;  
 m\_internalTemplateNodesList = null;  
 OnNodeRemovedEvent = null;  
 m\_masterNodeId = Constants.INVALID\_NODE\_ID;  
 m\_validNodeId = 0;  
 m\_instancePropertyCount = 0;  
 m\_nodeGrid.Destroy();  
 m\_nodeGrid = null;  
 m\_nodes.Clear();  
 m\_nodes = null;  
 m\_samplerNodes.Destroy();  
 m\_samplerNodes = null;  
 m\_propertyNodes.Destroy();  
 m\_propertyNodes = null;  
 m\_rawPropertyNodes.Destroy();  
 m\_rawPropertyNodes = null;  
 m\_customExpressionsOnFunctionMode.Destroy();  
 m\_customExpressionsOnFunctionMode = null;  
 m\_staticSwitchNodes.Destroy();  
 m\_staticSwitchNodes = null;  
 m\_functionInputNodes.Destroy();  
 m\_functionInputNodes = null;  
 m\_functionNodes.Destroy();  
 m\_functionNodes = null;  
 m\_functionOutputNodes.Destroy();  
 m\_functionOutputNodes = null;  
 m\_functionSwitchNodes.Destroy();  
 m\_functionSwitchNodes = null;  
 m\_functionSwitchCopyNodes.Destroy();  
 m\_functionSwitchCopyNodes = null;  
 m\_multiPassMasterNodes.Destroy();  
 m\_multiPassMasterNodes = null;  
 for( int i = 0; i < m\_lodMultiPassMasterNodes.Count; i++ )  
 {  
 m\_lodMultiPassMasterNodes[ i ].Destroy();  
 m\_lodMultiPassMasterNodes[ i ] = null;  
 }  
 m\_lodMultiPassMasterNodes.Clear();  
 m\_lodMultiPassMasterNodes = null;  
 m\_texturePropertyNodes.Destroy();  
 m\_texturePropertyNodes = null;  
 m\_textureArrayNodes.Destroy();  
 m\_textureArrayNodes = null;  
 m\_screenColorNodes.Destroy();  
 m\_screenColorNodes = null;  
 m\_localVarNodes.Destroy();  
 m\_localVarNodes = null;  
 m\_globalArrayNodes.Destroy();  
 m\_globalArrayNodes = null;  
 m\_selectedNodes.Clear();  
 m\_selectedNodes = null;  
 m\_markedForDeletion.Clear();  
 m\_markedForDeletion = null;  
 m\_nodesDict.Clear();  
 m\_nodesDict = null;  
 m\_nodePreviewList.Clear();  
 m\_nodePreviewList = null;  
 IsDirty = true;  
 OnNodeEvent = null;  
 OnDuplicateEvent = null;  
 OnMaterialUpdatedEvent = null;  
 OnShaderUpdatedEvent = null;  
 OnEmptyGraphDetectedEvt = null;  
 nodeStyleOff = null;  
 nodeStyleOn = null;  
 nodeTitle = null;  
 commentaryBackground = null;  
 OnLODMasterNodesAddedEvent = null;  
 }  
 void OnNodeChangeSizeEvent( ParentNode node )  
 {  
 m\_nodeGrid.RemoveNodeFromGrid( node, true );  
 m\_nodeGrid.AddNodeToGrid( node );  
 }  
 public void OnNodeFinishMoving( ParentNode node, bool testOnlySelected, InteractionMode interactionMode )  
 {  
 if( OnNodeEvent != null )  
 {  
 OnNodeEvent( node );  
 SaveIsDirty = true;  
 }  
 m\_nodeGrid.RemoveNodeFromGrid( node, true );  
 m\_nodeGrid.AddNodeToGrid( node );  
 {  
 for( int i = m\_nodes.Count - 1; i > -1; i-- )  
 {  
 if( node.UniqueId != m\_nodes[ i ].UniqueId )  
 {  
 switch( interactionMode )  
 {  
 case InteractionMode.Target:  
 {  
 node.OnNodeInteraction( m\_nodes[ i ] );  
 }  
 break;  
 case InteractionMode.Other:  
 {  
 m\_nodes[ i ].OnNodeInteraction( node );  
 }  
 break;  
 case InteractionMode.Both:  
 {  
 node.OnNodeInteraction( m\_nodes[ i ] );  
 m\_nodes[ i ].OnNodeInteraction( node );  
 }  
 break;  
 }  
 }  
 }  
 }  
 }  
 public void OnNodeReOrderEvent( ParentNode node, int index )  
 {  
 if( node.Depth < index )  
 {  
 Debug.LogWarning( "Reorder canceled: This is a specific method for when reordering needs to be done and a its original index is higher than the new one" );  
 }  
 else  
 {  
 m\_nodes.Remove( node );  
 m\_nodes.Insert( index, node );  
 m\_markToReOrder = true;  
 }  
 }  
 public void AddNode( ParentNode node, bool updateId = false, bool addLast = true, bool registerUndo = true, bool fetchMaterialValues = true )  
 {  
 if( registerUndo )  
 {  
 UIUtils.MarkUndoAction();  
 Undo.RegisterCompleteObjectUndo( ParentWindow, Constants.UndoCreateNodeId );  
 Undo.RegisterCompleteObjectUndo( this, Constants.UndoCreateNodeId );  
 Undo.RegisterCreatedObjectUndo( node, Constants.UndoCreateNodeId );  
 }  
 if( OnNodeEvent != null )  
 {  
 OnNodeEvent( node );  
 }  
 if( updateId )  
 {  
 node.UniqueId = GetValidId();  
 }  
 else  
 {  
 UpdateIdFromNode( node );  
 }  
 if( addLast )  
 {  
 m\_nodes.Add( node );  
 node.Depth = m\_nodes.Count;  
 }  
 else  
 {  
 m\_nodes.Insert( 0, node );  
 node.Depth = 0;  
 }  
 if( m\_nodesDict.ContainsKey( node.UniqueId ) )  
 {  
 m\_foundDuplicates = true;  
 }  
 else  
 {  
 m\_nodesDict.Add( node.UniqueId, node );  
 node.SetMaterialMode( CurrentMaterial, fetchMaterialValues );  
 }  
 m\_nodeGrid.AddNodeToGrid( node );  
 node.OnNodeChangeSizeEvent += OnNodeChangeSizeEvent;  
 node.OnNodeReOrderEvent += OnNodeReOrderEvent;  
 IsDirty = true;  
 }  
 public void CheckForDuplicates()  
 {  
 if( m\_foundDuplicates )  
 {  
 Debug.LogWarning( "Found duplicates:" );  
 m\_foundDuplicates = false;  
 m\_nodesDict.Clear();  
 int count = m\_nodes.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 if( m\_nodesDict.ContainsKey( m\_nodes[ i ].UniqueId ) )  
 {  
 m\_nodes[ i ].UniqueId = GetValidId();  
 m\_nodesDict.Add( m\_nodes[ i ].UniqueId, m\_nodes[ i ] );  
 Debug.LogWarning( "Assigning new ID to " + m\_nodes[ i ].TypeName );  
 }  
 else  
 {  
 m\_nodesDict.Add( m\_nodes[ i ].UniqueId, m\_nodes[ i ] );  
 }  
 }  
 }  
 }  
 public ParentNode GetClickedNode()  
 {  
 if( m\_nodeClicked < 0 )  
 return null;  
 return GetNode( m\_nodeClicked );  
 }  
 public PropertyNode GetInternalTemplateNode( int nodeId )  
 {  
 if( m\_internalTemplateNodesDict.Count != m\_internalTemplateNodesList.Count )  
 {  
 m\_internalTemplateNodesDict.Clear();  
 int count = m\_internalTemplateNodesList.Count;  
 for( int i = 0; i < m\_internalTemplateNodesList.Count; i++ )  
 {  
 if( m\_internalTemplateNodesList[ i ] != null )  
 m\_internalTemplateNodesDict.Add( m\_internalTemplateNodesList[ i ].UniqueId, m\_internalTemplateNodesList[ i ] );  
 }  
 }  
 if( m\_internalTemplateNodesDict.ContainsKey( nodeId ) )  
 return m\_internalTemplateNodesDict[ nodeId ];  
 return null;  
 }  
 public PropertyNode GetInternalTemplateNode( string propertyName )  
 {  
 return m\_internalTemplateNodesList.Find( ( x ) => x.PropertyName.Equals( propertyName ) );  
 }  
 public void AddInternalTemplateNode( TemplateShaderPropertyData data )  
 {  
 PropertyNode propertyNode = null;  
 switch( data.PropertyDataType )  
 {  
 case WirePortDataType.FLOAT:  
 propertyNode = CreateInstance<RangedFloatNode>(); break;  
 case WirePortDataType.FLOAT4:  
 propertyNode = CreateInstance<Vector4Node>();  
 break;  
 case WirePortDataType.COLOR:  
 propertyNode = CreateInstance<ColorNode>();  
 break;  
 case WirePortDataType.INT:  
 propertyNode = CreateInstance<IntNode>(); break;  
 case WirePortDataType.SAMPLER1D:  
 case WirePortDataType.SAMPLER2D:  
 case WirePortDataType.SAMPLER3D:  
 case WirePortDataType.SAMPLERCUBE:  
 case WirePortDataType.SAMPLER2DARRAY:  
 propertyNode = CreateInstance<SamplerNode>();  
 break;  
 default: return;  
 }  
 propertyNode.PropertyNameFromTemplate( data );  
 int uniqueId = -( m\_internalTemplateNodesList.Count + 2 );  
 propertyNode.SetBaseUniqueId( uniqueId );  
 if( data.PropertyDataType == WirePortDataType.FLOAT ||  
 data.PropertyDataType == WirePortDataType.INT )  
 m\_floatNodes.AddNode( propertyNode );  
 m\_internalTemplateNodesList.Add( propertyNode );  
 m\_internalTemplateNodesDict.Add( uniqueId, propertyNode );  
 }  
 public void ClearInternalTemplateNodes()  
 {  
 if( m\_internalTemplateNodesList != null )  
 {  
 int count = m\_internalTemplateNodesList.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 m\_internalTemplateNodesList[ i ].Destroy();  
 GameObject.DestroyImmediate( m\_internalTemplateNodesList[ i ] );  
 }  
 m\_internalTemplateNodesList.Clear();  
 m\_internalTemplateNodesDict.Clear();  
 }  
 }  
 public ParentNode GetNode( int nodeId )  
 {  
 if( m\_nodesDict.Count != m\_nodes.Count )  
 {  
 m\_nodesDict.Clear();  
 int count = m\_nodes.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 if( m\_nodes[ i ] != null && !m\_nodesDict.ContainsKey( m\_nodes[ i ].UniqueId ) )  
 m\_nodesDict.Add( m\_nodes[ i ].UniqueId, m\_nodes[ i ] );  
 }  
 }  
 if( m\_nodesDict.ContainsKey( nodeId ) )  
 return m\_nodesDict[ nodeId ];  
 return null;  
 }  
 public void ForceReOrder()  
 {  
 m\_nodes.Sort( ( x, y ) => x.Depth.CompareTo( y.Depth ) );  
 }  
 public bool Draw( DrawInfo drawInfo )  
 {  
 MasterNode masterNode = GetNode( m\_masterNodeId ) as MasterNode;  
 if( m\_forceCategoryRefresh && masterNode != null )  
 {  
 masterNode.RefreshAvailableCategories();  
 m\_forceCategoryRefresh = false;  
 }  
 SaveIsDirty = false;  
 if( m\_afterDeserializeFlag )  
 {  
 CleanCorruptedNodes();  
 if( m\_nodes.Count == 0 )  
 {  
 NodeAvailability cachedCanvas = CurrentCanvasMode;  
 ParentWindow.CreateNewGraph( "Empty" );  
 CurrentCanvasMode = cachedCanvas;  
 if( OnEmptyGraphDetectedEvt != null )  
 {  
 OnEmptyGraphDetectedEvt( this );  
 SaveIsDirty = false;  
 }  
 else  
 {  
 SaveIsDirty = true;  
 }  
 }  
 }  
 if( drawInfo.CurrentEventType == EventType.Repaint )  
 {  
 if( m\_markedToDeSelect )  
 DeSelectAll();  
 if( m\_markToSelect > -1 )  
 {  
 AddToSelectedNodes( GetNode( m\_markToSelect ) );  
 m\_markToSelect = -1;  
 }  
 if( m\_markToReOrder )  
 {  
 m\_markToReOrder = false;  
 int nodesCount = m\_nodes.Count;  
 for( int i = 0; i < nodesCount; i++ )  
 {  
 m\_nodes[ i ].Depth = i;  
 }  
 }  
 }  
 if( drawInfo.CurrentEventType == EventType.Repaint )  
 {  
 NodeLOD newLevel = NodeLOD.LOD0;  
 float referenceValue;  
 if( drawInfo.InvertedZoom > 0.5f )  
 {  
 newLevel = NodeLOD.LOD0;  
 referenceValue = 4;  
 }  
 else if( drawInfo.InvertedZoom > 0.25f )  
 {  
 newLevel = NodeLOD.LOD1;  
 referenceValue = 2;  
 }  
 else if( drawInfo.InvertedZoom > 0.15f )  
 {  
 newLevel = NodeLOD.LOD2;  
 referenceValue = 1;  
 }  
 else if( drawInfo.InvertedZoom > 0.1f )  
 {  
 newLevel = NodeLOD.LOD3;  
 referenceValue = 0;  
 }  
 else if( drawInfo.InvertedZoom > 0.07f )  
 {  
 newLevel = NodeLOD.LOD4;  
 referenceValue = 0;  
 }  
 else  
 {  
 newLevel = NodeLOD.LOD5;  
 referenceValue = 0;  
 }  
 nodeStyleOff = UIUtils.GetCustomStyle( CustomStyle.NodeWindowOff );  
 nodeStyleOn = UIUtils.GetCustomStyle( CustomStyle.NodeWindowOn );//= UIUtils.GetCustomStyle( CustomStyle.NodeWindowOn );  
 nodeTitle = UIUtils.GetCustomStyle( CustomStyle.NodeHeader );  
 commentaryBackground = UIUtils.GetCustomStyle( CustomStyle.CommentaryBackground );  
 if( newLevel != m\_lodLevel || ( UIUtils.MainSkin != null && UIUtils.MainSkin.textField.border.left != referenceValue ) )  
 {  
 m\_lodLevel = newLevel;  
 switch( m\_lodLevel )  
 {  
 default:  
 case NodeLOD.LOD0:  
 {  
 UIUtils.MainSkin.textField.border = UIUtils.RectOffsetFour;  
 nodeStyleOff.border = UIUtils.RectOffsetSix;  
 UIUtils.NodeWindowOffSquare.border = UIUtils.RectOffsetFour;  
 nodeStyleOn.border = UIUtils.RectOffsetSix;  
 UIUtils.NodeWindowOnSquare.border = UIUtils.RectOffsetSix;  
 nodeTitle.border.left = 6;  
 nodeTitle.border.right = 6;  
 nodeTitle.border.top = 6;  
 nodeTitle.border.bottom = 4;  
 UIUtils.NodeHeaderSquare.border = UIUtils.RectOffsetFour;  
 commentaryBackground.border = UIUtils.RectOffsetSix;  
 }  
 break;  
 case NodeLOD.LOD1:  
 {  
 UIUtils.MainSkin.textField.border = UIUtils.RectOffsetTwo;  
 nodeStyleOff.border = UIUtils.RectOffsetFive;  
 UIUtils.NodeWindowOffSquare.border = UIUtils.RectOffsetFive;  
 nodeStyleOn.border = UIUtils.RectOffsetFive;  
 UIUtils.NodeWindowOnSquare.border = UIUtils.RectOffsetFour;  
 nodeTitle.border.left = 5;  
 nodeTitle.border.right = 5;  
 nodeTitle.border.top = 5;  
 nodeTitle.border.bottom = 2;  
 UIUtils.NodeHeaderSquare.border = UIUtils.RectOffsetThree;  
 commentaryBackground.border = UIUtils.RectOffsetFive;  
 }  
 break;  
 case NodeLOD.LOD2:  
 {  
 UIUtils.MainSkin.textField.border = UIUtils.RectOffsetOne;  
 nodeStyleOff.border.left = 2;  
 nodeStyleOff.border.right = 2;  
 nodeStyleOff.border.top = 2;  
 nodeStyleOff.border.bottom = 3;  
 UIUtils.NodeWindowOffSquare.border = UIUtils.RectOffsetThree;  
 nodeStyleOn.border.left = 4;  
 nodeStyleOn.border.right = 4;  
 nodeStyleOn.border.top = 4;  
 nodeStyleOn.border.bottom = 3;  
 UIUtils.NodeWindowOnSquare.border = UIUtils.RectOffsetThree;  
 nodeTitle.border = UIUtils.RectOffsetTwo;  
 UIUtils.NodeHeaderSquare.border = UIUtils.RectOffsetTwo;  
 commentaryBackground.border.left = 2;  
 commentaryBackground.border.right = 2;  
 commentaryBackground.border.top = 2;  
 commentaryBackground.border.bottom = 3;  
 }  
 break;  
 case NodeLOD.LOD3:  
 case NodeLOD.LOD4:  
 case NodeLOD.LOD5:  
 {  
 UIUtils.MainSkin.textField.border = UIUtils.RectOffsetZero;  
 nodeStyleOff.border.left = 1;  
 nodeStyleOff.border.right = 1;  
 nodeStyleOff.border.top = 1;  
 nodeStyleOff.border.bottom = 2;  
 UIUtils.NodeWindowOffSquare.border = UIUtils.RectOffsetTwo;  
 nodeStyleOn.border = UIUtils.RectOffsetTwo;  
 UIUtils.NodeWindowOnSquare.border = UIUtils.RectOffsetTwo;  
 nodeTitle.border = UIUtils.RectOffsetOne;  
 UIUtils.NodeHeaderSquare.border = UIUtils.RectOffsetOne;  
 commentaryBackground.border.left = 1;  
 commentaryBackground.border.right = 1;  
 commentaryBackground.border.top = 1;  
 commentaryBackground.border.bottom = 2;  
 }  
 break;  
 }  
 }  
 }  
 m\_hasUnConnectedNodes = false;  
 bool repaint = false;  
 Material currentMaterial = masterNode != null ? masterNode.CurrentMaterial : null;  
 EditorGUI.BeginChangeCheck();  
 bool repaintMaterialInspector = false;  
 int nodeCount = m\_nodes.Count;  
 for( int i = 0; i < nodeCount; i++ )  
 {  
 m\_nodes[ i ].OnNodeLogicUpdate( drawInfo );  
 }  
 if( m\_afterDeserializeFlag || m\_lateOptionsRefresh )  
 {  
 m\_afterDeserializeFlag = false;  
 m\_lateOptionsRefresh = false;  
 if( CurrentCanvasMode == NodeAvailability.TemplateShader )  
 {  
 RefreshLinkedMasterNodes( true );  
 OnRefreshLinkedPortsComplete();  
 if( m\_parentWindow.ClipboardInstance.HasCachedMasterNodes )  
 {  
 m\_parentWindow.ClipboardInstance.AddMultiPassNodesToClipboard( MultiPassMasterNodes.NodesList,true,-1 );  
 for( int i = 0; i < m\_lodMultiPassMasterNodes.Count; i++ )  
 {  
 if( m\_lodMultiPassMasterNodes[ i ].Count > 0 )  
 m\_parentWindow.ClipboardInstance.AddMultiPassNodesToClipboard( m\_lodMultiPassMasterNodes[ i ].NodesList, false, i );  
 }  
 }  
 }  
 }  
 if( m\_forceRepositionCheck )  
 {  
 RepositionTemplateNodes( CurrentMasterNode );  
 }  
 nodeCount = m\_nodes.Count;  
 ParentNode node = null;  
 for( int i = 0; i < nodeCount; i++ )  
 {  
 node = m\_nodes[ i ];  
 if( !node.IsOnGrid )  
 {  
 m\_nodeGrid.AddNodeToGrid( node );  
 }  
 node.MovingInFrame = false;  
 if( drawInfo.CurrentEventType == EventType.Repaint )  
 node.OnNodeLayout( drawInfo );  
 m\_hasUnConnectedNodes = m\_hasUnConnectedNodes ||  
 ( node.ConnStatus != NodeConnectionStatus.Connected && node.ConnStatus != NodeConnectionStatus.Island );  
 if( node.RequireMaterialUpdate && currentMaterial != null )  
 {  
 node.UpdateMaterial( currentMaterial );  
 repaintMaterialInspector = true;  
 }  
 IsDirty = ( m\_isDirty || node.IsDirty );  
 SaveIsDirty = ( m\_saveIsDirty || node.SaveIsDirty );  
 }  
 nodeCount = m\_nodes.Count;  
 for( int i = nodeCount - 1; i >= 0; i-- )  
 {  
 node = m\_nodes[ i ];  
 bool restoreMouse = false;  
 if( drawInfo.CurrentEventType == EventType.MouseDown && m\_nodeClicked > -1 && node.UniqueId != m\_nodeClicked )  
 {  
 restoreMouse = true;  
 drawInfo.CurrentEventType = EventType.Ignore;  
 }  
 node.DrawGUIControls( drawInfo );  
 if( restoreMouse )  
 {  
 drawInfo.CurrentEventType = EventType.MouseDown;  
 }  
 }  
 if( drawInfo.CurrentEventType == EventType.Repaint )  
 DrawWires( ParentWindow.WireTexture, drawInfo, ParentWindow.WindowContextPallete.IsActive, ParentWindow.WindowContextPallete.CurrentPosition );  
 nodeCount = m\_nodes.Count;  
 for( int i = 0; i < nodeCount; i++ )  
 {  
 node = m\_nodes[ i ];  
 bool restoreMouse = false;  
 if( drawInfo.CurrentEventType == EventType.MouseDown && m\_nodeClicked > -1 && node.UniqueId != m\_nodeClicked )  
 {  
 restoreMouse = true;  
 drawInfo.CurrentEventType = EventType.Ignore;  
 }  
 node.Draw( drawInfo );  
 if( restoreMouse )  
 {  
 drawInfo.CurrentEventType = EventType.MouseDown;  
 }  
 }  
 if( drawInfo.CurrentEventType == EventType.Repaint || drawInfo.CurrentEventType == EventType.MouseDown )  
 {  
 nodeCount = m\_nodes.Count;  
 for( int i = nodeCount - 1; i >= 0; i-- )  
 {  
 node = m\_nodes[ i ];  
 if( node.IsVisible && !node.IsMoving )  
 {  
 bool showing = node.ShowTooltip( drawInfo );  
 if( showing )  
 break;  
 }  
 }  
 }  
 if( repaintMaterialInspector )  
 {  
 if( ASEMaterialInspector.Instance != null )  
 {  
 ASEMaterialInspector.Instance.Repaint();  
 }  
 }  
 if( m\_checkSelectedWireHighlights )  
 {  
 m\_checkSelectedWireHighlights = false;  
 ResetHighlightedWires();  
 for( int i = 0; i < m\_selectedNodes.Count; i++ )  
 {  
 HighlightWiresStartingNode( m\_selectedNodes[ i ] );  
 }  
 }  
 if( EditorGUI.EndChangeCheck() )  
 {  
 SaveIsDirty = true;  
 repaint = true;  
 }  
 if( drawInfo.CurrentEventType == EventType.Repaint )  
 {  
 if( UIUtils.MainSkin.textField.border.left != 4 )  
 {  
 UIUtils.MainSkin.textField.border = UIUtils.RectOffsetFour;  
 nodeStyleOff.border = UIUtils.RectOffsetSix;  
 UIUtils.NodeWindowOffSquare.border = UIUtils.RectOffsetFour;  
 nodeStyleOn.border = UIUtils.RectOffsetSix;  
 UIUtils.NodeWindowOnSquare.border = UIUtils.RectOffsetSix;  
 nodeTitle.border.left = 6;  
 nodeTitle.border.right = 6;  
 nodeTitle.border.top = 6;  
 nodeTitle.border.bottom = 4;  
 UIUtils.NodeHeaderSquare.border = UIUtils.RectOffsetFour;  
 commentaryBackground.border = UIUtils.RectOffsetSix;  
 }  
 }  
 ChangedLightingModel = false;  
 return repaint;  
 }  
 public bool UpdateMarkForDeletion()  
 {  
 if( m\_markedForDeletion.Count != 0 )  
 {  
 DeleteMarkedForDeletionNodes();  
 return true;  
 }  
 return false;  
 }  
 public void DrawWires( Texture2D wireTex, DrawInfo drawInfo, bool contextPaletteActive, Vector3 contextPalettePos )  
 {  
 m\_wireBezierCount = 0;  
 for( int nodeIdx = 0; nodeIdx < m\_nodes.Count; nodeIdx++ )  
 {  
 ParentNode node = m\_nodes[ nodeIdx ];  
 if( (object)node == null )  
 return;  
 for( int inputPortIdx = 0; inputPortIdx < node.InputPorts.Count; inputPortIdx++ )  
 {  
 InputPort inputPort = node.InputPorts[ inputPortIdx ];  
 if( inputPort.ExternalReferences.Count > 0 && inputPort.Visible )  
 {  
 bool cleanInvalidConnections = false;  
 for( int wireIdx = 0; wireIdx < inputPort.ExternalReferences.Count; wireIdx++ )  
 {  
 WireReference reference = inputPort.ExternalReferences[ wireIdx ];  
 if( reference.NodeId != -1 && reference.PortId != -1 )  
 {  
 ParentNode outputNode = GetNode( reference.NodeId );  
 if( outputNode != null )  
 {  
 OutputPort outputPort = outputNode.GetOutputPortByUniqueId( reference.PortId );  
 Vector3 endPos = new Vector3( inputPort.Position.x, inputPort.Position.y );  
 Vector3 startPos = new Vector3( outputPort.Position.x, outputPort.Position.y );  
 float x = ( startPos.x < endPos.x ) ? startPos.x : endPos.x;  
 float y = ( startPos.y < endPos.y ) ? startPos.y : endPos.y;  
 float width = Mathf.Abs( startPos.x - endPos.x ) + outputPort.Position.width;  
 float height = Mathf.Abs( startPos.y - endPos.y ) + outputPort.Position.height;  
 Rect portsBoundingBox = new Rect( x, y, width, height );  
 bool isVisible = node.IsVisible || outputNode.IsVisible;  
 if( !isVisible )  
 {  
 isVisible = drawInfo.TransformedCameraArea.Overlaps( portsBoundingBox );  
 }  
 if( isVisible )  
 {  
 Rect bezierBB = DrawBezier( drawInfo.InvertedZoom, startPos, endPos, inputPort.DataType, outputPort.DataType, node.GetInputPortVisualDataTypeByArrayIdx( inputPortIdx ), outputNode.GetOutputPortVisualDataTypeById( reference.PortId ), reference.WireStatus, wireTex, node, outputNode );  
 bezierBB.x -= Constants.OUTSIDE\_WIRE\_MARGIN;  
 bezierBB.y -= Constants.OUTSIDE\_WIRE\_MARGIN;  
 bezierBB.width += Constants.OUTSIDE\_WIRE\_MARGIN \* 2;  
 bezierBB.height += Constants.OUTSIDE\_WIRE\_MARGIN \* 2;  
 if( m\_wireBezierCount < m\_bezierReferences.Count )  
 {  
 m\_bezierReferences[ m\_wireBezierCount ].UpdateInfo( ref bezierBB, inputPort.NodeId, inputPort.PortId, outputPort.NodeId, outputPort.PortId );  
 }  
 else  
 {  
 m\_bezierReferences.Add( new WireBezierReference( ref bezierBB, inputPort.NodeId, inputPort.PortId, outputPort.NodeId, outputPort.PortId ) );  
 }  
 m\_wireBezierCount++;  
 }  
 }  
 else  
 {  
 if( DebugConsoleWindow.DeveloperMode )  
 UIUtils.ShowMessage( "Detected Invalid connection from node " + node.UniqueId + " port " + inputPortIdx + " to Node " + reference.NodeId + " port " + reference.PortId, MessageSeverity.Error );  
 cleanInvalidConnections = true;  
 inputPort.ExternalReferences[ wireIdx ].Invalidate();  
 }  
 }  
 }  
 if( cleanInvalidConnections )  
 {  
 inputPort.RemoveInvalidConnections();  
 }  
 }  
 }  
 }  
 if( m\_parentWindow.WireReferenceUtils.ValidReferences() )  
 {  
 if( m\_parentWindow.WireReferenceUtils.InputPortReference.IsValid )  
 {  
 InputPort inputPort = GetNode( m\_parentWindow.WireReferenceUtils.InputPortReference.NodeId ).GetInputPortByUniqueId( m\_parentWindow.WireReferenceUtils.InputPortReference.PortId );  
 Vector3 endPos = Vector3.zero;  
 if( m\_parentWindow.WireReferenceUtils.SnapEnabled )  
 {  
 Vector2 pos = ( m\_parentWindow.WireReferenceUtils.SnapPosition + drawInfo.CameraOffset ) \* drawInfo.InvertedZoom;  
 endPos = new Vector3( pos.x, pos.y ) + UIUtils.ScaledPortsDelta;  
 }  
 else  
 {  
 endPos = contextPaletteActive ? contextPalettePos : new Vector3( Event.current.mousePosition.x, Event.current.mousePosition.y );  
 }  
 Vector3 startPos = new Vector3( inputPort.Position.x, inputPort.Position.y );  
 DrawBezier( drawInfo.InvertedZoom, endPos, startPos, inputPort.DataType, inputPort.DataType, inputPort.DataType, inputPort.DataType, WireStatus.Default, wireTex );  
 }  
 if( m\_parentWindow.WireReferenceUtils.OutputPortReference.IsValid )  
 {  
 OutputPort outputPort = GetNode( m\_parentWindow.WireReferenceUtils.OutputPortReference.NodeId ).GetOutputPortByUniqueId( m\_parentWindow.WireReferenceUtils.OutputPortReference.PortId );  
 Vector3 endPos = Vector3.zero;  
 if( m\_parentWindow.WireReferenceUtils.SnapEnabled )  
 {  
 Vector2 pos = ( m\_parentWindow.WireReferenceUtils.SnapPosition + drawInfo.CameraOffset ) \* drawInfo.InvertedZoom;  
 endPos = new Vector3( pos.x, pos.y ) + UIUtils.ScaledPortsDelta;  
 }  
 else  
 {  
 endPos = contextPaletteActive ? contextPalettePos : new Vector3( Event.current.mousePosition.x, Event.current.mousePosition.y );  
 }  
 Vector3 startPos = new Vector3( outputPort.Position.x, outputPort.Position.y );  
 DrawBezier( drawInfo.InvertedZoom, startPos, endPos, outputPort.DataType, outputPort.DataType, outputPort.DataType, outputPort.DataType, WireStatus.Default, wireTex );  
 }  
 }  
 }  
 Rect DrawBezier( float invertedZoom, Vector3 startPos, Vector3 endPos, WirePortDataType inputDataType, WirePortDataType outputDataType, WirePortDataType inputVisualDataType, WirePortDataType outputVisualDataType, WireStatus wireStatus, Texture2D wireTex, ParentNode inputNode = null, ParentNode outputNode = null )  
 {  
 startPos += UIUtils.ScaledPortsDelta;  
 endPos += UIUtils.ScaledPortsDelta;  
 float mag = ( endPos - startPos ).magnitude;  
 float resizedMag = Mathf.Min( mag \* 0.66f, Constants.HORIZONTAL\_TANGENT\_SIZE \* invertedZoom );  
 Vector3 startTangent = new Vector3( startPos.x + resizedMag, startPos.y );  
 Vector3 endTangent = new Vector3( endPos.x - resizedMag, endPos.y );  
 if( (object)inputNode != null && inputNode.GetType() == typeof( WireNode ) )  
 endTangent = endPos + ( ( inputNode as WireNode ).TangentDirection ) \* mag \* 0.33f;  
 if( (object)outputNode != null && outputNode.GetType() == typeof( WireNode ) )  
 startTangent = startPos - ( ( outputNode as WireNode ).TangentDirection ) \* mag \* 0.33f;  
 int ty = 1;  
 float wireThickness = 0;  
 if( ParentWindow.Options.MultiLinePorts )  
 {  
 GLDraw.MultiLine = true;  
 Shader.SetGlobalFloat( "\_InvertedZoom", invertedZoom );  
 WirePortDataType smallest = ( (int)outputDataType < (int)inputDataType ? outputDataType : inputDataType );  
 smallest = ( (int)smallest < (int)outputVisualDataType ? smallest : outputVisualDataType );  
 smallest = ( (int)smallest < (int)inputVisualDataType ? smallest : inputVisualDataType );  
 switch( smallest )  
 {  
 case WirePortDataType.FLOAT2: ty = 2; break;  
 case WirePortDataType.FLOAT3: ty = 3; break;  
 case WirePortDataType.FLOAT4:  
 case WirePortDataType.COLOR:  
 {  
 ty = 4;  
 }  
 break;  
 default: ty = 1; break;  
 }  
 wireThickness = Mathf.Lerp( Constants.WIRE\_WIDTH \* ( ty \* invertedZoom \* -0.05f + 0.15f ), Constants.WIRE\_WIDTH \* ( ty \* invertedZoom \* 0.175f + 0.3f ), invertedZoom + 0.4f );  
 }  
 else  
 {  
 GLDraw.MultiLine = false;  
 wireThickness = Mathf.Lerp( Constants.WIRE\_WIDTH \* ( invertedZoom \* -0.05f + 0.15f ), Constants.WIRE\_WIDTH \* ( invertedZoom \* 0.175f + 0.3f ), invertedZoom + 0.4f );  
 }  
 Rect boundBox = new Rect();  
 int segments = 11;  
 if( LodLevel <= ParentGraph.NodeLOD.LOD4 )  
 segments = Mathf.Clamp( Mathf.FloorToInt( mag \* 0.2f \* invertedZoom ), 11, 35 );  
 else  
 segments = (int)( invertedZoom \* 14.28f \* 11 );  
 if( ParentWindow.Options.ColoredPorts && wireStatus != WireStatus.Highlighted )  
 boundBox = GLDraw.DrawBezier( startPos, startTangent, endPos, endTangent, UIUtils.GetColorForDataType( outputVisualDataType, false, false ), UIUtils.GetColorForDataType( inputVisualDataType, false, false ), wireThickness, segments, ty );  
 else  
 boundBox = GLDraw.DrawBezier( startPos, startTangent, endPos, endTangent, UIUtils.GetColorFromWireStatus( wireStatus ), wireThickness, segments, ty );  
 float extraBound = 30 \* invertedZoom;  
 boundBox.xMin -= extraBound;  
 boundBox.xMax += extraBound;  
 boundBox.yMin -= extraBound;  
 boundBox.yMax += extraBound;  
 return boundBox;  
 }  
 public void DrawBezierBoundingBox()  
 {  
 for( int i = 0; i < m\_wireBezierCount; i++ )  
 {  
 m\_bezierReferences[ i ].DebugDraw();  
 }  
 }  
 public WireBezierReference GetWireBezierInPos( Vector2 position )  
 {  
 for( int i = 0; i < m\_wireBezierCount; i++ )  
 {  
 if( m\_bezierReferences[ i ].Contains( position ) )  
 return m\_bezierReferences[ i ];  
 }  
 return null;  
 }  
 public List<WireBezierReference> GetWireBezierListInPos( Vector2 position )  
 {  
 List<WireBezierReference> list = new List<WireBezierReference>();  
 for( int i = 0; i < m\_wireBezierCount; i++ )  
 {  
 if( m\_bezierReferences[ i ].Contains( position ) )  
 list.Add( m\_bezierReferences[ i ] );  
 }  
 return list;  
 }  
 public void MoveSelectedNodes( Vector2 delta, bool snap = false )  
 {  
 bool performUndo = delta.magnitude > 0.01f;  
 if( performUndo )  
 {  
 Undo.RegisterCompleteObjectUndo( ParentWindow, Constants.UndoMoveNodesId );  
 Undo.RegisterCompleteObjectUndo( this, Constants.UndoMoveNodesId );  
 }  
 for( int i = 0; i < m\_selectedNodes.Count; i++ )  
 {  
 if( !m\_selectedNodes[ i ].MovingInFrame )  
 {  
 if( performUndo )  
 m\_selectedNodes[ i ].RecordObject( Constants.UndoMoveNodesId );  
 m\_selectedNodes[ i ].Move( delta, snap );  
 }  
 }  
 IsDirty = true;  
 }  
 public void SetConnection( int InNodeId, int InPortId, int OutNodeId, int OutPortId )  
 {  
 ParentNode inNode = GetNode( InNodeId );  
 ParentNode outNode = GetNode( OutNodeId );  
 InputPort inputPort = null;  
 OutputPort outputPort = null;  
 if( inNode != null && outNode != null )  
 {  
 inputPort = inNode.GetInputPortByUniqueId( InPortId );  
 outputPort = outNode.GetOutputPortByUniqueId( OutPortId );  
 if( inputPort != null && outputPort != null )  
 {  
 if( inputPort.IsConnectedTo( OutNodeId, OutPortId ) || outputPort.IsConnectedTo( InNodeId, InPortId ) )  
 {  
 if( DebugConsoleWindow.DeveloperMode )  
 UIUtils.ShowMessage( "Node/Port already connected " + InNodeId, MessageSeverity.Error );  
 return;  
 }  
 if( !inputPort.CheckValidType( outputPort.DataType ) )  
 {  
 if( DebugConsoleWindow.DeveloperMode )  
 UIUtils.ShowIncompatiblePortMessage( true, inNode, inputPort, outNode, outputPort );  
 return;  
 }  
 if( !outputPort.CheckValidType( inputPort.DataType ) )  
 {  
 if( DebugConsoleWindow.DeveloperMode )  
 UIUtils.ShowIncompatiblePortMessage( false, outNode, outputPort, inNode, inputPort );  
 return;  
 }  
 if( !inputPort.Available || !outputPort.Available )  
 {  
 if( DebugConsoleWindow.DeveloperMode )  
 UIUtils.ShowMessage( "Ports not available to connection", MessageSeverity.Warning );  
 return;  
 }  
 if( inputPort.ConnectTo( OutNodeId, OutPortId, outputPort.DataType, false ) )  
 {  
 inNode.OnInputPortConnected( InPortId, OutNodeId, OutPortId );  
 }  
 if( outputPort.ConnectTo( InNodeId, InPortId, inputPort.DataType, inputPort.TypeLocked ) )  
 {  
 outNode.OnOutputPortConnected( OutPortId, InNodeId, InPortId );  
 }  
 }  
 else if( (object)inputPort == null )  
 {  
 if( DebugConsoleWindow.DeveloperMode )  
 UIUtils.ShowMessage( "Input Port " + InPortId + " doesn't exist on node " + InNodeId, MessageSeverity.Error );  
 }  
 else  
 {  
 if( DebugConsoleWindow.DeveloperMode )  
 UIUtils.ShowMessage( "Output Port " + OutPortId + " doesn't exist on node " + OutNodeId, MessageSeverity.Error );  
 }  
 }  
 else if( (object)inNode == null )  
 {  
 if( DebugConsoleWindow.DeveloperMode )  
 UIUtils.ShowMessage( "Input node " + InNodeId + " doesn't exist", MessageSeverity.Error );  
 }  
 else  
 {  
 if( DebugConsoleWindow.DeveloperMode )  
 UIUtils.ShowMessage( "Output node " + OutNodeId + " doesn't exist", MessageSeverity.Error );  
 }  
 }  
 public void CreateConnection( int inNodeId, int inPortId, int outNodeId, int outPortId, bool registerUndo = true )  
 {  
 ParentNode outputNode = GetNode( outNodeId );  
 if( outputNode != null )  
 {  
 OutputPort outputPort = outputNode.GetOutputPortByUniqueId( outPortId );  
 if( outputPort != null )  
 {  
 ParentNode inputNode = GetNode( inNodeId );  
 InputPort inputPort = inputNode.GetInputPortByUniqueId( inPortId );  
 if( !inputPort.CheckValidType( outputPort.DataType ) )  
 {  
 UIUtils.ShowIncompatiblePortMessage( true, inputNode, inputPort, outputNode, outputPort );  
 return;  
 }  
 if( !outputPort.CheckValidType( inputPort.DataType ) )  
 {  
 UIUtils.ShowIncompatiblePortMessage( false, outputNode, outputPort, inputNode, inputPort );  
 return;  
 }  
 inputPort.DummyAdd( outputPort.NodeId, outputPort.PortId );  
 outputPort.DummyAdd( inNodeId, inPortId );  
 if( UIUtils.DetectNodeLoopsFrom( inputNode, new Dictionary<int, int>() ) )  
 {  
 inputPort.DummyRemove();  
 outputPort.DummyRemove();  
 m\_parentWindow.WireReferenceUtils.InvalidateReferences();  
 UIUtils.ShowMessage( "Infinite Loop detected" );  
 Event.current.Use();  
 return;  
 }  
 inputPort.DummyRemove();  
 outputPort.DummyRemove();  
 if( inputPort.IsConnected )  
 {  
 DeleteConnection( true, inNodeId, inPortId, true, false, registerUndo );  
 }  
 if( outputPort.ConnectTo( inNodeId, inPortId, inputPort.DataType, inputPort.TypeLocked ) )  
 outputNode.OnOutputPortConnected( outputPort.PortId, inNodeId, inPortId );  
 if( inputPort.ConnectTo( outputPort.NodeId, outputPort.PortId, outputPort.DataType, inputPort.TypeLocked ) )  
 inputNode.OnInputPortConnected( inPortId, outputNode.UniqueId, outputPort.PortId );  
 MarkWireHighlights();  
 }  
 SaveIsDirty = true;  
 }  
 }  
 public void DeleteInvalidConnections()  
 {  
 int count = m\_nodes.Count;  
 for( int nodeIdx = 0; nodeIdx < count; nodeIdx++ )  
 {  
 {  
 int inputCount = m\_nodes[ nodeIdx ].InputPorts.Count;  
 for( int inputIdx = 0; inputIdx < inputCount; inputIdx++ )  
 {  
 if( !m\_nodes[ nodeIdx ].InputPorts[ inputIdx ].Visible &&  
 m\_nodes[ nodeIdx ].InputPorts[ inputIdx ].IsConnected &&  
 !m\_nodes[ nodeIdx ].InputPorts[ inputIdx ].IsDummy )  
 {  
 DeleteConnection( true, m\_nodes[ nodeIdx ].UniqueId, m\_nodes[ nodeIdx ].InputPorts[ inputIdx ].PortId, true, true );  
 }  
 }  
 }  
 {  
 int outputCount = m\_nodes[ nodeIdx ].OutputPorts.Count;  
 for( int outputIdx = 0; outputIdx < outputCount; outputIdx++ )  
 {  
 if( !m\_nodes[ nodeIdx ].OutputPorts[ outputIdx ].Visible && m\_nodes[ nodeIdx ].OutputPorts[ outputIdx ].IsConnected )  
 {  
 DeleteConnection( false, m\_nodes[ nodeIdx ].UniqueId, m\_nodes[ nodeIdx ].OutputPorts[ outputIdx ].PortId, true, true );  
 }  
 }  
 }  
 }  
 }  
 public void DeleteAllConnectionFromNode( int nodeId, bool registerOnLog, bool propagateCallback, bool registerUndo )  
 {  
 ParentNode node = GetNode( nodeId );  
 if( (object)node == null )  
 return;  
 DeleteAllConnectionFromNode( node, registerOnLog, propagateCallback, registerUndo );  
 }  
 public void DeleteAllConnectionFromNode( ParentNode node, bool registerOnLog, bool propagateCallback, bool registerUndo )  
 {  
 for( int i = 0; i < node.InputPorts.Count; i++ )  
 {  
 if( node.InputPorts[ i ].IsConnected )  
 DeleteConnection( true, node.UniqueId, node.InputPorts[ i ].PortId, registerOnLog, propagateCallback, registerUndo );  
 }  
 for( int i = 0; i < node.OutputPorts.Count; i++ )  
 {  
 if( node.OutputPorts[ i ].IsConnected )  
 DeleteConnection( false, node.UniqueId, node.OutputPorts[ i ].PortId, registerOnLog, propagateCallback, registerUndo );  
 }  
 }  
 public void DeleteConnection( bool isInput, int nodeId, int portId, bool registerOnLog, bool propagateCallback, bool registerUndo = true )  
 {  
 ParentNode node = GetNode( nodeId );  
 if( (object)node == null )  
 return;  
 if( registerUndo )  
 {  
 UIUtils.MarkUndoAction();  
 Undo.RegisterCompleteObjectUndo( ParentWindow, Constants.UndoDeleteConnectionId );  
 Undo.RegisterCompleteObjectUndo( this, Constants.UndoDeleteConnectionId );  
 node.RecordObject( Constants.UndoDeleteConnectionId );  
 }  
 if( isInput )  
 {  
 InputPort inputPort = node.GetInputPortByUniqueId( portId );  
 if( inputPort != null && inputPort.IsConnected )  
 {  
 if( node.ConnStatus == NodeConnectionStatus.Connected )  
 {  
 node.DeactivateInputPortNode( portId, false );  
 m\_checkSelectedWireHighlights = true;  
 }  
 for( int i = 0; i < inputPort.ExternalReferences.Count; i++ )  
 {  
 WireReference inputReference = inputPort.ExternalReferences[ i ];  
 ParentNode outputNode = GetNode( inputReference.NodeId );  
 if( registerUndo )  
 outputNode.RecordObject( Constants.UndoDeleteConnectionId );  
 outputNode.GetOutputPortByUniqueId( inputReference.PortId ).InvalidateConnection( inputPort.NodeId, inputPort.PortId );  
 if( propagateCallback )  
 outputNode.OnOutputPortDisconnected( inputReference.PortId );  
 }  
 inputPort.InvalidateAllConnections();  
 if( propagateCallback )  
 node.OnInputPortDisconnected( portId );  
 }  
 }  
 else  
 {  
 OutputPort outputPort = node.GetOutputPortByUniqueId( portId );  
 if( outputPort != null && outputPort.IsConnected )  
 {  
 if( propagateCallback )  
 node.OnOutputPortDisconnected( portId );  
 for( int i = 0; i < outputPort.ExternalReferences.Count; i++ )  
 {  
 WireReference outputReference = outputPort.ExternalReferences[ i ];  
 ParentNode inputNode = GetNode( outputReference.NodeId );  
 if( registerUndo )  
 inputNode.RecordObject( Constants.UndoDeleteConnectionId );  
 if( inputNode.ConnStatus == NodeConnectionStatus.Connected )  
 {  
 node.DeactivateNode( portId, false );  
 m\_checkSelectedWireHighlights = true;  
 }  
 inputNode.GetInputPortByUniqueId( outputReference.PortId ).InvalidateConnection( outputPort.NodeId, outputPort.PortId );  
 if( propagateCallback )  
 {  
 if( !inputNode.GetInputPortByUniqueId( outputReference.PortId ).IsConnected )  
 inputNode.OnInputPortDisconnected( outputReference.PortId );  
 }  
 }  
 outputPort.InvalidateAllConnections();  
 }  
 }  
 IsDirty = true;  
 SaveIsDirty = true;  
 }  
 public void DeleteNodesOnArray( ref ParentNode[] nodeArray )  
 {  
 bool invalidateMasterNode = false;  
 for( int nodeIdx = 0; nodeIdx < nodeArray.Length; nodeIdx++ )  
 {  
 ParentNode node = nodeArray[ nodeIdx ];  
 if( node.UniqueId == m\_masterNodeId )  
 {  
 FunctionOutput fout = node as FunctionOutput;  
 if( fout != null )  
 {  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 FunctionOutput secondfout = m\_nodes[ i ] as FunctionOutput;  
 if( secondfout != null && secondfout != fout )  
 {  
 secondfout.Function = fout.Function;  
 AssignMasterNode( secondfout, false );  
 DeselectNode( fout );  
 DestroyNode( fout );  
 break;  
 }  
 }  
 }  
 invalidateMasterNode = true;  
 }  
 else  
 {  
 DeselectNode( node );  
 DestroyNode( node );  
 }  
 nodeArray[ nodeIdx ] = null;  
 }  
 if( invalidateMasterNode && CurrentMasterNode != null )  
 {  
 CurrentMasterNode.Selected = false;  
 }  
 nodeArray = null;  
 IsDirty = true;  
 }  
 public void MarkWireNodeSequence( WireNode node, bool isInput )  
 {  
 if( node == null )  
 {  
 return;  
 }  
 if( m\_markedForDeletion.Contains( node ) )  
 return;  
 m\_markedForDeletion.Add( node );  
 if( isInput && node.InputPorts[ 0 ].IsConnected )  
 {  
 MarkWireNodeSequence( GetNode( node.InputPorts[ 0 ].ExternalReferences[ 0 ].NodeId ) as WireNode, isInput );  
 }  
 else if( !isInput && node.OutputPorts[ 0 ].IsConnected )  
 {  
 MarkWireNodeSequence( GetNode( node.OutputPorts[ 0 ].ExternalReferences[ 0 ].NodeId ) as WireNode, isInput );  
 }  
 }  
 public void UndoableDeleteSelectedNodes( List<ParentNode> nodeList )  
 {  
 if( nodeList.Count == 0 )  
 return;  
 List<ParentNode> validNode = new List<ParentNode>();  
 for( int i = 0; i < nodeList.Count; i++ )  
 {  
 if( nodeList[ i ] != null && nodeList[ i ].UniqueId != m\_masterNodeId )  
 {  
 validNode.Add( nodeList[ i ] );  
 }  
 }  
 UIUtils.ClearUndoHelper();  
 ParentNode[] selectedNodes = new ParentNode[ validNode.Count ];  
 for( int i = 0; i < selectedNodes.Length; i++ )  
 {  
 if( validNode[ i ] != null )  
 {  
 selectedNodes[ i ] = validNode[ i ];  
 UIUtils.CheckUndoNode( selectedNodes[ i ] );  
 }  
 }  
 List<ParentNode> extraNodes = new List<ParentNode>();  
 for( int selectedNodeIdx = 0; selectedNodeIdx < selectedNodes.Length; selectedNodeIdx++ )  
 {  
 if( selectedNodes[ selectedNodeIdx ] != null )  
 {  
 int inputIdxCount = selectedNodes[ selectedNodeIdx ].InputPorts.Count;  
 if( inputIdxCount > 0 )  
 {  
 for( int inputIdx = 0; inputIdx < inputIdxCount; inputIdx++ )  
 {  
 if( selectedNodes[ selectedNodeIdx ].InputPorts[ inputIdx ].IsConnected )  
 {  
 int nodeIdx = selectedNodes[ selectedNodeIdx ].InputPorts[ inputIdx ].ExternalReferences[ 0 ].NodeId;  
 if( nodeIdx > -1 )  
 {  
 ParentNode node = GetNode( nodeIdx );  
 if( node != null && UIUtils.CheckUndoNode( node ) )  
 {  
 extraNodes.Add( node );  
 }  
 }  
 }  
 }  
 }  
 }  
 if( selectedNodes[ selectedNodeIdx ] != null )  
 {  
 int outputIdxCount = selectedNodes[ selectedNodeIdx ].OutputPorts.Count;  
 if( outputIdxCount > 0 )  
 {  
 for( int outputIdx = 0; outputIdx < outputIdxCount; outputIdx++ )  
 {  
 int inputIdxCount = selectedNodes[ selectedNodeIdx ].OutputPorts[ outputIdx ].ExternalReferences.Count;  
 if( inputIdxCount > 0 )  
 {  
 for( int inputIdx = 0; inputIdx < inputIdxCount; inputIdx++ )  
 {  
 int nodeIdx = selectedNodes[ selectedNodeIdx ].OutputPorts[ outputIdx ].ExternalReferences[ inputIdx ].NodeId;  
 if( nodeIdx > -1 )  
 {  
 ParentNode node = GetNode( nodeIdx );  
 if( UIUtils.CheckUndoNode( node ) )  
 {  
 extraNodes.Add( node );  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
 UIUtils.ClearUndoHelper();  
 UIUtils.MarkUndoAction();  
 Undo.RegisterCompleteObjectUndo( ParentWindow, Constants.UndoDeleteNodeId );  
 Undo.RegisterCompleteObjectUndo( this, Constants.UndoDeleteNodeId );  
 Undo.RecordObjects( selectedNodes, Constants.UndoDeleteNodeId );  
 Undo.RecordObjects( extraNodes.ToArray(), Constants.UndoDeleteNodeId );  
 for( int i = 0; i < selectedNodes.Length; i++ )  
 {  
 CurrentOutputNode.Selected = false;  
 selectedNodes[ i ].Alive = false;  
 DeleteAllConnectionFromNode( selectedNodes[ i ], false, true, true );  
 }  
 DeleteNodesOnArray( ref selectedNodes );  
 extraNodes.Clear();  
 extraNodes = null;  
 EditorUtility.SetDirty( ParentWindow );  
 ParentWindow.ForceRepaint();  
 }  
 public void DeleteMarkedForDeletionNodes()  
 {  
 UndoableDeleteSelectedNodes( m\_markedForDeletion );  
 m\_markedForDeletion.Clear();  
 IsDirty = true;  
 }  
 public void DestroyNode( int nodeId )  
 {  
 ParentNode node = GetNode( nodeId );  
 DestroyNode( node );  
 }  
 public void DestroyNode( ParentNode node, bool registerUndo = true, bool destroyMasterNode = false )  
 {  
 if( node == null )  
 {  
 UIUtils.ShowMessage( "Attempting to destroying a inexistant node ", MessageSeverity.Warning );  
 return;  
 }  
 if( node.ConnStatus == NodeConnectionStatus.Connected && !m\_checkSelectedWireHighlights )  
 {  
 ResetHighlightedWires();  
 m\_checkSelectedWireHighlights = true;  
 }  
 if( destroyMasterNode || ( node.UniqueId != m\_masterNodeId && !( node is TemplateMultiPassMasterNode )/\*!m\_multiPassMasterNodes.HasNode( node.UniqueId )\*/ ) )  
 {  
 m\_nodeGrid.RemoveNodeFromGrid( node, false );  
 if( node.ConnStatus == NodeConnectionStatus.Connected )  
 {  
 node.DeactivateNode( -1, true );  
 }  
 for( int inputPortIdx = 0; inputPortIdx < node.InputPorts.Count; inputPortIdx++ )  
 {  
 InputPort inputPort = node.InputPorts[ inputPortIdx ];  
 if( inputPort.IsConnected )  
 {  
 for( int wireIdx = 0; wireIdx < inputPort.ExternalReferences.Count; wireIdx++ )  
 {  
 WireReference inputReference = inputPort.ExternalReferences[ wireIdx ];  
 ParentNode outputNode = GetNode( inputReference.NodeId );  
 outputNode.GetOutputPortByUniqueId( inputReference.PortId ).InvalidateConnection( inputPort.NodeId, inputPort.PortId );  
 outputNode.OnOutputPortDisconnected( inputReference.PortId );  
 }  
 inputPort.InvalidateAllConnections();  
 }  
 }  
 for( int outputPortIdx = 0; outputPortIdx < node.OutputPorts.Count; outputPortIdx++ )  
 {  
 OutputPort outputPort = node.OutputPorts[ outputPortIdx ];  
 if( outputPort.IsConnected )  
 {  
 for( int wireIdx = 0; wireIdx < outputPort.ExternalReferences.Count; wireIdx++ )  
 {  
 WireReference outputReference = outputPort.ExternalReferences[ wireIdx ];  
 ParentNode outnode = GetNode( outputReference.NodeId );  
 if( outnode != null )  
 {  
 outnode.GetInputPortByUniqueId( outputReference.PortId ).InvalidateConnection( outputPort.NodeId, outputPort.PortId );  
 outnode.OnInputPortDisconnected( outputReference.PortId );  
 }  
 }  
 outputPort.InvalidateAllConnections();  
 }  
 }  
 if( registerUndo )  
 {  
 UIUtils.MarkUndoAction();  
 Undo.RegisterCompleteObjectUndo( ParentWindow, Constants.UndoDeleteNodeId );  
 Undo.RegisterCompleteObjectUndo( this, Constants.UndoDeleteNodeId );  
 node.RecordObjectOnDestroy( Constants.UndoDeleteNodeId );  
 }  
 if( OnNodeRemovedEvent != null )  
 OnNodeRemovedEvent( node );  
 m\_nodes.Remove( node );  
 m\_nodesDict.Remove( node.UniqueId );  
 node.Destroy();  
 if( registerUndo )  
 Undo.DestroyObjectImmediate( node );  
 else  
 DestroyImmediate( node );  
 IsDirty = true;  
 m\_markToReOrder = true;  
 }  
 else  
 {  
 TemplateMultiPassMasterNode templateMasterNode = node as TemplateMultiPassMasterNode;  
 if( templateMasterNode != null && templateMasterNode.InvalidNode )  
 {  
 DestroyNode( node, false, true );  
 return;  
 }  
 DeselectNode( node );  
 UIUtils.ShowMessage( "Attempting to destroy a master node" );  
 }  
 }  
 void AddToSelectedNodes( ParentNode node )  
 {  
 node.Selected = true;  
 m\_selectedNodes.Add( node );  
 node.OnNodeStoppedMovingEvent += OnNodeFinishMoving;  
 if( node.ConnStatus == NodeConnectionStatus.Connected )  
 {  
 HighlightWiresStartingNode( node );  
 }  
 }  
 void RemoveFromSelectedNodes( ParentNode node )  
 {  
 node.Selected = false;  
 m\_selectedNodes.Remove( node );  
 node.OnNodeStoppedMovingEvent -= OnNodeFinishMoving;  
 }  
 public void SelectNode( ParentNode node, bool append, bool reorder )  
 {  
 if( node == null )  
 return;  
 if( append )  
 {  
 if( !m\_selectedNodes.Contains( node ) )  
 {  
 AddToSelectedNodes( node );  
 }  
 }  
 else  
 {  
 DeSelectAll();  
 AddToSelectedNodes( node );  
 }  
 if( reorder && !node.ReorderLocked )  
 {  
 m\_nodes.Remove( node );  
 m\_nodes.Add( node );  
 m\_markToReOrder = true;  
 }  
 }  
 public void MultipleSelection( Rect selectionArea, bool appendSelection = true )  
 {  
 if( !appendSelection )  
 {  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 if( selectionArea.Overlaps( m\_nodes[ i ].Position, true ) )  
 {  
 RemoveFromSelectedNodes( m\_nodes[ i ] );  
 }  
 }  
 m\_markedToDeSelect = false;  
 ResetHighlightedWires();  
 }  
 else  
 {  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 if( !m\_nodes[ i ].Selected && selectionArea.Overlaps( m\_nodes[ i ].Position, true ) )  
 {  
 AddToSelectedNodes( m\_nodes[ i ] );  
 }  
 }  
 }  
 for( int i = 0; i < m\_selectedNodes.Count; i++ )  
 {  
 if( !m\_selectedNodes[ i ].ReorderLocked )  
 {  
 m\_nodes.Remove( m\_selectedNodes[ i ] );  
 m\_nodes.Add( m\_selectedNodes[ i ] );  
 m\_markToReOrder = true;  
 if( m\_selectedNodes[ i ].ConnStatus == NodeConnectionStatus.Connected )  
 {  
 HighlightWiresStartingNode( m\_selectedNodes[ i ] );  
 }  
 }  
 }  
 }  
 public void SelectAll()  
 {  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 if( !m\_nodes[ i ].Selected )  
 AddToSelectedNodes( m\_nodes[ i ] );  
 }  
 }  
 public void SelectMasterNode()  
 {  
 if( m\_masterNodeId != Constants.INVALID\_NODE\_ID )  
 {  
 SelectNode( CurrentMasterNode, false, false );  
 }  
 }  
 public void SelectOutputNode()  
 {  
 if( m\_masterNodeId != Constants.INVALID\_NODE\_ID )  
 {  
 SelectNode( CurrentOutputNode, false, false );  
 }  
 }  
 public void DeselectNode( int nodeId )  
 {  
 ParentNode node = GetNode( nodeId );  
 if( node )  
 {  
 m\_selectedNodes.Remove( node );  
 node.Selected = false;  
 }  
 }  
 public void DeselectNode( ParentNode node )  
 {  
 m\_selectedNodes.Remove( node );  
 node.Selected = false;  
 PropagateHighlightDeselection( node );  
 }  
 public void DeSelectAll()  
 {  
 m\_markedToDeSelect = false;  
 for( int i = 0; i < m\_selectedNodes.Count; i++ )  
 {  
 m\_selectedNodes[ i ].Selected = false;  
 m\_selectedNodes[ i ].OnNodeStoppedMovingEvent -= OnNodeFinishMoving;  
 }  
 m\_selectedNodes.Clear();  
 ResetHighlightedWires();  
 }  
 public void AssignMasterNode()  
 {  
 if( m\_selectedNodes.Count == 1 )  
 {  
 OutputNode newOutputNode = m\_selectedNodes[ 0 ] as OutputNode;  
 MasterNode newMasterNode = newOutputNode as MasterNode;  
 if( newOutputNode != null )  
 {  
 if( m\_masterNodeId != Constants.INVALID\_NODE\_ID && m\_masterNodeId != newOutputNode.UniqueId )  
 {  
 OutputNode oldOutputNode = GetNode( m\_masterNodeId ) as OutputNode;  
 MasterNode oldMasterNode = oldOutputNode as MasterNode;  
 if( oldOutputNode != null )  
 {  
 oldOutputNode.IsMainOutputNode = false;  
 if( oldMasterNode != null )  
 {  
 oldMasterNode.ClearUpdateEvents();  
 }  
 }  
 }  
 m\_masterNodeId = newOutputNode.UniqueId;  
 newOutputNode.IsMainOutputNode = true;  
 if( newMasterNode != null )  
 {  
 newMasterNode.OnMaterialUpdatedEvent += OnMaterialUpdatedEvent;  
 newMasterNode.OnShaderUpdatedEvent += OnShaderUpdatedEvent;  
 }  
 }  
 }  
 IsDirty = true;  
 }  
 public void AssignMasterNode( OutputNode node, bool onlyUpdateGraphId )  
 {  
 AssignMasterNode( node.UniqueId, onlyUpdateGraphId );  
 MasterNode masterNode = node as MasterNode;  
 if( masterNode != null )  
 {  
 masterNode.OnMaterialUpdatedEvent += OnMaterialUpdatedEvent;  
 masterNode.OnShaderUpdatedEvent += OnShaderUpdatedEvent;  
 }  
 }  
 public void AssignMasterNode( int nodeId, bool onlyUpdateGraphId )  
 {  
 if( nodeId < 0 || m\_masterNodeId == nodeId )  
 return;  
 if( m\_masterNodeId > Constants.INVALID\_NODE\_ID )  
 {  
 OutputNode oldOutputNode = ( GetNode( nodeId ) as OutputNode );  
 MasterNode oldMasterNode = oldOutputNode as MasterNode;  
 if( oldOutputNode != null )  
 {  
 oldOutputNode.IsMainOutputNode = false;  
 if( oldMasterNode != null )  
 {  
 oldMasterNode.ClearUpdateEvents();  
 }  
 }  
 }  
 if( onlyUpdateGraphId )  
 {  
 m\_masterNodeId = nodeId;  
 }  
 else  
 {  
 OutputNode outputNode = ( GetNode( nodeId ) as OutputNode );  
 if( outputNode != null )  
 {  
 outputNode.IsMainOutputNode = true;  
 m\_masterNodeId = nodeId;  
 }  
 }  
 IsDirty = true;  
 }  
 public void RefreshOnUndo()  
 {  
 if( m\_nodes != null )  
 {  
 int count = m\_nodes.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 if( m\_nodes[ i ] != null )  
 {  
 m\_nodes[ i ].RefreshOnUndo();  
 }  
 }  
 }  
 }  
 public void DrawGrid( DrawInfo drawInfo )  
 {  
 m\_nodeGrid.DrawGrid( drawInfo );  
 }  
 public float MaxNodeDist  
 {  
 get { return m\_nodeGrid.MaxNodeDist; }  
 }  
 public List<ParentNode> GetNodesInGrid( Vector2 transformedMousePos )  
 {  
 return m\_nodeGrid.GetNodesOn( transformedMousePos );  
 }  
 public void FireMasterNode( Shader selectedShader )  
 {  
 ( GetNode( m\_masterNodeId ) as MasterNode ).Execute( selectedShader );  
 }  
 public Shader FireMasterNode( string pathname, bool isFullPath )  
 {  
 return ( GetNode( m\_masterNodeId ) as MasterNode ).Execute( pathname, isFullPath );  
 }  
 private void ForceSignalPropagationOnMasterNodeInternal( UsageListTemplateMultiPassMasterNodes masterNodes )  
 {  
 int mpCount = masterNodes.Count;  
 for( int i = 0; i < mpCount; i++ )  
 {  
 masterNodes.NodesList[ i ].GenerateSignalPropagation();  
 }  
 }  
 public void ForceSignalPropagationOnMasterNode()  
 {  
 if( m\_multiPassMasterNodes.Count > 0 )  
 {  
 ForceSignalPropagationOnMasterNodeInternal( m\_multiPassMasterNodes );  
 for( int i = 0; i < m\_lodMultiPassMasterNodes.Count; i++ )  
 {  
 ForceSignalPropagationOnMasterNodeInternal( m\_lodMultiPassMasterNodes[ i ] );  
 }  
 }  
 else if( CurrentOutputNode != null )  
 CurrentOutputNode.GenerateSignalPropagation();  
 List<FunctionOutput> allOutputs = m\_functionOutputNodes.NodesList;  
 for( int i = 0; i < allOutputs.Count; i++ )  
 {  
 allOutputs[ i ].GenerateSignalPropagation();  
 }  
 }  
 public void UpdateShaderOnMasterNode( Shader newShader )  
 {  
 MasterNode mainMasterNode = ( GetNode( m\_masterNodeId ) as MasterNode );  
 if( mainMasterNode == null )  
 {  
 Debug.LogError( "No Master Node was detected. Aborting update!" );  
 return;  
 }  
 mainMasterNode.UpdateFromShader( newShader );  
 if( HasLODs )  
 {  
 int passIdx = ( (TemplateMultiPassMasterNode)mainMasterNode ).PassIdx;  
 for( int i = 0; i < m\_lodMultiPassMasterNodes.Count; i++ )  
 {  
 if( m\_lodMultiPassMasterNodes.Count != 0 && m\_lodMultiPassMasterNodes[ i ].NodesList.Count > 0 )  
 {  
 if( m\_lodMultiPassMasterNodes[ i ].NodesList[ passIdx ] != null )  
 {  
 m\_lodMultiPassMasterNodes[ i ].NodesList[ passIdx ].UpdateFromShader( newShader );  
 }  
 else  
 {  
 Debug.LogError( "Null master node detected. Aborting update!" );  
 return;  
 }  
 }  
 else break;  
 }  
 }  
 }  
 public void CopyValuesFromMaterial( Material material )  
 {  
 Material currMaterial = CurrentMaterial;  
 if( currMaterial == material )  
 {  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 m\_nodes[ i ].ForceUpdateFromMaterial( material );  
 }  
 }  
 }  
 public void UpdateMaterialOnMasterNode( Material material )  
 {  
 MasterNode mainMasterNode = ( GetNode( m\_masterNodeId ) as MasterNode );  
 mainMasterNode.UpdateMasterNodeMaterial( material );  
 if( HasLODs )  
 {  
 int passIdx = ( (TemplateMultiPassMasterNode)mainMasterNode ).PassIdx;  
 for( int i = 0; i < m\_lodMultiPassMasterNodes.Count; i++ )  
 {  
 if( m\_lodMultiPassMasterNodes.Count != 0 && m\_lodMultiPassMasterNodes[ i ].NodesList.Count > 0 )  
 {  
 m\_lodMultiPassMasterNodes[ i ].NodesList[ passIdx ].UpdateMasterNodeMaterial( material );  
 }  
 else break;  
 }  
 }  
 }  
 public void UpdateMaterialOnPropertyNodes( Material material )  
 {  
 int propertyCount = m\_propertyNodes.Count;  
 for(int i = 0;i< propertyCount;i++ )  
 {  
 m\_propertyNodes.NodesList[i].UpdateMaterial( material );  
 }  
 }  
 public void SetMaterialModeOnGraph( Material mat, bool fetchMaterialValues = true )  
 {  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 m\_nodes[ i ].SetMaterialMode( mat, fetchMaterialValues );  
 }  
 }  
 public ParentNode CheckNodeAt( Vector3 pos, bool checkForRMBIgnore = false )  
 {  
 ParentNode selectedNode = null;  
 for( int i = m\_nodes.Count - 1; i > -1; i-- )  
 {  
 if( m\_nodes[ i ].Contains( pos ) )  
 {  
 if( checkForRMBIgnore )  
 {  
 if( !m\_nodes[ i ].RMBIgnore )  
 {  
 selectedNode = m\_nodes[ i ];  
 break;  
 }  
 }  
 else  
 {  
 selectedNode = m\_nodes[ i ];  
 break;  
 }  
 }  
 }  
 return selectedNode;  
 }  
 public void ResetNodesLocalVariables()  
 {  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 m\_nodes[ i ].Reset();  
 m\_nodes[ i ].ResetOutputLocals();  
 FunctionNode fnode = m\_nodes[ i ] as FunctionNode;  
 if( fnode != null )  
 {  
 if( fnode.Function != null )  
 fnode.FunctionGraph.ResetNodesLocalVariables();  
 }  
 }  
 }  
 public void ResetNodesLocalVariablesIfNot( MasterNodePortCategory category )  
 {  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 m\_nodes[ i ].Reset();  
 m\_nodes[ i ].ResetOutputLocalsIfNot( category );  
 FunctionNode fnode = m\_nodes[ i ] as FunctionNode;  
 if( fnode != null )  
 {  
 if( fnode.Function != null )  
 fnode.FunctionGraph.ResetNodesLocalVariablesIfNot( category );  
 }  
 }  
 }  
 public void ResetNodesLocalVariables( ParentNode node )  
 {  
 if( node is GetLocalVarNode )  
 {  
 GetLocalVarNode localVarNode = node as GetLocalVarNode;  
 if( localVarNode.CurrentSelected != null )  
 {  
 node = localVarNode.CurrentSelected;  
 }  
 }  
 node.Reset();  
 node.ResetOutputLocals();  
 int count = node.InputPorts.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 if( node.InputPorts[ i ].IsConnected )  
 {  
 ResetNodesLocalVariables( m\_nodesDict[ node.InputPorts[ i ].GetConnection().NodeId ] );  
 }  
 }  
 }  
 public void ResetNodesLocalVariablesIfNot( ParentNode node, MasterNodePortCategory category )  
 {  
 if( node is GetLocalVarNode )  
 {  
 GetLocalVarNode localVarNode = node as GetLocalVarNode;  
 if( localVarNode.CurrentSelected != null )  
 {  
 node = localVarNode.CurrentSelected;  
 }  
 }  
 node.Reset();  
 node.ResetOutputLocalsIfNot( category );  
 int count = node.InputPorts.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 if( node.InputPorts[ i ].IsConnected )  
 {  
 ResetNodesLocalVariablesIfNot( m\_nodesDict[ node.InputPorts[ i ].GetConnection().NodeId ], category );  
 }  
 }  
 }  
 public override string ToString()  
 {  
 string dump = ( "Parent Graph \n" );  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 dump += ( m\_nodes[ i ] + "\n" );  
 }  
 return dump;  
 }  
 public void OrderNodesByGraphDepth()  
 {  
 if( CurrentMasterNode != null )  
 {  
 int count = m\_nodes.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 if( m\_nodes[ i ].ConnStatus == NodeConnectionStatus.Island )  
 {  
 m\_nodes[ i ].CalculateCustomGraphDepth();  
 }  
 }  
 }  
 else  
 {  
 List<OutputNode> allOutputs = new List<OutputNode>();  
 for( int i = 0; i < AllNodes.Count; i++ )  
 {  
 OutputNode temp = AllNodes[ i ] as OutputNode;  
 if( temp != null )  
 allOutputs.Add( temp );  
 }  
 for( int j = 0; j < allOutputs.Count; j++ )  
 {  
 allOutputs[ j ].SetupNodeCategories();  
 int count = m\_nodes.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 if( m\_nodes[ i ].ConnStatus == NodeConnectionStatus.Island )  
 {  
 m\_nodes[ i ].CalculateCustomGraphDepth();  
 }  
 }  
 }  
 }  
 m\_nodes.Sort( ( x, y ) => { return y.GraphDepth.CompareTo( x.GraphDepth ); } );  
 }  
 public void WriteToString( ref string nodesInfo, ref string connectionsInfo )  
 {  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 m\_nodes[ i ].FullWriteToString( ref nodesInfo, ref connectionsInfo );  
 IOUtils.AddLineTerminator( ref nodesInfo );  
 }  
 }  
 public void Reset()  
 {  
 SaveIsDirty = false;  
 IsDirty = false;  
 }  
 public void OnBeforeSerialize()  
 {  
 }  
 public void OnAfterDeserialize()  
 {  
 m\_afterDeserializeFlag = true;  
 }  
 public void CleanCorruptedNodes()  
 {  
 for( int i = 0; i < m\_nodes.Count; i++ )  
 {  
 if( (object)m\_nodes[ i ] == null )  
 {  
 m\_nodes.RemoveAt( i );  
 CleanCorruptedNodes();  
 }  
 }  
 }  
 public void OnDuplicateEventWrapper()  
 {  
 if( OnDuplicateEvent != null )  
 {  
 AmplifyShaderEditorWindow temp = UIUtils.CurrentWindow;  
 UIUtils.CurrentWindow = ParentWindow;  
 OnDuplicateEvent();  
 UIUtils.CurrentWindow = temp;  
 }  
 }  
 public ParentNode CreateNode( AmplifyShaderFunction shaderFunction, bool registerUndo, int nodeId = -1, bool addLast = true )  
 {  
 FunctionNode newNode = ScriptableObject.CreateInstance<FunctionNode>();  
 if( newNode )  
 {  
 newNode.ContainerGraph = this;  
 newNode.CommonInit( shaderFunction, nodeId );  
 newNode.UniqueId = nodeId;  
 AddNode( newNode, nodeId < 0, addLast, registerUndo );  
 }  
 return newNode;  
 }  
 public ParentNode CreateNode( AmplifyShaderFunction shaderFunction, bool registerUndo, Vector2 pos, int nodeId = -1, bool addLast = true )  
 {  
 ParentNode newNode = CreateNode( shaderFunction, registerUndo, nodeId, addLast );  
 if( newNode )  
 {  
 newNode.Vec2Position = pos;  
 }  
 return newNode;  
 }  
 public TemplateMultiPassMasterNode CreateMultipassMasterNode( int lodId, bool registerUndo, int nodeId = -1, bool addLast = true )  
 {  
 TemplateMultiPassMasterNode newNode = ScriptableObject.CreateInstance<TemplateMultiPassMasterNode>();  
 if( newNode )  
 {  
 newNode.LODIndex = lodId;  
 newNode.ContainerGraph = this;  
 if( newNode.IsStubNode )  
 {  
 TemplateMultiPassMasterNode stubNode = newNode.ExecuteStubCode() as TemplateMultiPassMasterNode;  
 ScriptableObject.DestroyImmediate( newNode, true );  
 newNode = stubNode;  
 }  
 else  
 {  
 newNode.UniqueId = nodeId;  
 AddNode( newNode, nodeId < 0, addLast, registerUndo );  
 }  
 }  
 return newNode;  
 }  
 public ParentNode CreateNode( System.Type type, bool registerUndo, int nodeId = -1, bool addLast = true )  
 {  
 ParentNode newNode = ScriptableObject.CreateInstance( type ) as ParentNode;  
 if( newNode )  
 {  
 newNode.ContainerGraph = this;  
 if( newNode.IsStubNode )  
 {  
 ParentNode stubNode = newNode.ExecuteStubCode();  
 ScriptableObject.DestroyImmediate( newNode, true );  
 newNode = stubNode;  
 }  
 else  
 {  
 newNode.UniqueId = nodeId;  
 AddNode( newNode, nodeId < 0, addLast, registerUndo );  
 }  
 }  
 return newNode;  
 }  
 public ParentNode CreateNode( System.Type type, bool registerUndo, Vector2 pos, int nodeId = -1, bool addLast = true )  
 {  
 ParentNode newNode = CreateNode( type, registerUndo, nodeId, addLast );  
 if( newNode )  
 {  
 newNode.Vec2Position = pos;  
 }  
 return newNode;  
 }  
 public void FireMasterNodeReplacedEvent()  
 {  
 MasterNode masterNode = CurrentMasterNode;  
 int count = m\_nodes.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 if( m\_nodes[ i ].UniqueId != m\_masterNodeId )  
 {  
 m\_nodes[ i ].OnMasterNodeReplaced( masterNode );  
 }  
 }  
 }  
 public void FireMasterNodeReplacedEvent( MasterNode masterNode )  
 {  
 int count = m\_nodes.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 if( m\_nodes[ i ].UniqueId != masterNode.UniqueId )  
 {  
 m\_nodes[ i ].OnMasterNodeReplaced( masterNode );  
 }  
 }  
 }  
 public void CrossCheckTemplateNodes( TemplateDataParent templateData , List<TemplateMultiPassMasterNode> mpNodesList , int lodId )  
 {  
 DeSelectAll();  
 TemplateMultiPassMasterNode newMasterNode = null;  
 Dictionary<string, TemplateReplaceHelper> nodesDict = new Dictionary<string, TemplateReplaceHelper>();  
 int mpNodeCount = mpNodesList.Count;  
 for( int i = 0; i < mpNodeCount; i++ )  
 {  
 string masterNodeId = mpNodesList[ i ].InvalidNode ? mpNodesList[ i ].OriginalPassName + "ASEInvalidMasterNode" + i : mpNodesList[ i ].OriginalPassName;  
 nodesDict.Add( masterNodeId, new TemplateReplaceHelper( mpNodesList[ i ] ) );  
 }  
 TemplateMultiPassMasterNode currMasterNode = GetNode( m\_masterNodeId ) as TemplateMultiPassMasterNode;  
 TemplateMultiPass multipassData = templateData as TemplateMultiPass;  
 m\_currentSRPType = multipassData.SubShaders[ 0 ].Modules.SRPType;  
 bool sortTemplatesNodes = false;  
 Vector2 currentPosition = currMasterNode.Vec2Position;  
 for( int subShaderIdx = 0; subShaderIdx < multipassData.SubShaders.Count; subShaderIdx++ )  
 {  
 for( int passIdx = 0; passIdx < multipassData.SubShaders[ subShaderIdx ].Passes.Count; passIdx++ )  
 {  
 string currPassName = multipassData.SubShaders[ subShaderIdx ].Passes[ passIdx ].PassNameContainer.Data;  
 if( nodesDict.ContainsKey( currPassName ) )  
 {  
 bool wasMainNode = nodesDict[ currPassName ].MasterNode.IsMainOutputNode;  
 currentPosition.y += nodesDict[ currPassName ].MasterNode.Position.height + 10;  
 nodesDict[ currPassName ].Used = true;  
 nodesDict[ currPassName ].MasterNode.SetTemplate( multipassData, false, false, subShaderIdx, passIdx, SetTemplateSource.NewShader );  
 if( wasMainNode && !nodesDict[ currPassName ].MasterNode.IsMainOutputNode )  
 {  
 nodesDict[ currPassName ].MasterNode.ReleaseResources();  
 }  
 else if( !wasMainNode && nodesDict[ currPassName ].MasterNode.IsMainOutputNode )  
 {  
 newMasterNode = nodesDict[ currPassName ].MasterNode;  
 }  
 }  
 else  
 {  
 sortTemplatesNodes = true;  
 TemplateMultiPassMasterNode masterNode = CreateMultipassMasterNode( lodId, false );  
 if( multipassData.SubShaders[ subShaderIdx ].Passes[ passIdx ].IsMainPass )  
 {  
 newMasterNode = masterNode;  
 currMasterNode.ReleaseResources();  
 }  
 masterNode.Vec2Position = currentPosition;  
 masterNode.SetTemplate( multipassData, true, true, subShaderIdx, passIdx, SetTemplateSource.NewShader );  
 }  
 }  
 }  
 foreach( KeyValuePair<string, TemplateReplaceHelper> kvp in nodesDict )  
 {  
 if( !kvp.Value.Used )  
 DestroyNode( kvp.Value.MasterNode, false, true );  
 }  
 nodesDict.Clear();  
 if( newMasterNode != null )  
 {  
 if( lodId == -1 )  
 {  
 m\_masterNodeId = newMasterNode.UniqueId;  
 }  
 newMasterNode.OnMaterialUpdatedEvent += OnMaterialUpdatedEvent;  
 newMasterNode.OnShaderUpdatedEvent += OnShaderUpdatedEvent;  
 newMasterNode.IsMainOutputNode = true;  
 }  
 if( sortTemplatesNodes )  
 {  
 mpNodesList.Sort( ( x, y ) => ( x.PassIdx.CompareTo( y.PassIdx ) ) );  
 }  
 }  
 public void OnRefreshLinkedPortsComplete()  
 {  
 OnRefreshLinkedPortsCompleteInternal( m\_multiPassMasterNodes );  
 for( int i = 0; i < m\_lodMultiPassMasterNodes.Count; i++ )  
 {  
 OnRefreshLinkedPortsCompleteInternal( m\_lodMultiPassMasterNodes[ i ] );  
 }  
 }  
 private void OnRefreshLinkedPortsCompleteInternal( UsageListTemplateMultiPassMasterNodes masterNodes )  
 {  
 int mpCount = masterNodes.Count;  
 for( int i = 0; i < mpCount; i++ )  
 {  
 masterNodes.NodesList[ i ].OnRefreshLinkedPortsComplete();  
 }  
 }  
 public void RefreshLinkedMasterNodes( bool optionsUpdate = false )  
 {  
 if( DebugConsoleWindow.DeveloperMode )  
 Debug.Log( "Refresh linked master nodes" );  
 RefreshLinkedMasterNodesInternal( m\_multiPassMasterNodes, optionsUpdate );  
 for( int i = 0; i < m\_lodMultiPassMasterNodes.Count; i++ )  
 {  
 RefreshLinkedMasterNodesInternal( m\_lodMultiPassMasterNodes[i], optionsUpdate );  
 }  
 }  
 private void RefreshLinkedMasterNodesInternal( UsageListTemplateMultiPassMasterNodes masterNodes, bool optionsUpdate )  
 {  
 int mpCount = masterNodes.Count;  
 if( mpCount > 1 )  
 {  
 Dictionary<string, List<InputPort>> registeredLinks = new Dictionary<string, List<InputPort>>();  
 for( int i = 0; i < mpCount; i++ )  
 {  
 CheckLinkedPorts( ref registeredLinks, masterNodes.NodesList[ mpCount - 1 - i ] );  
 }  
 foreach( KeyValuePair<string, List<InputPort>> kvp in registeredLinks )  
 {  
 int linkCount = kvp.Value.Count;  
 if( linkCount == 1 )  
 {  
 kvp.Value[ 0 ].Visible = true;  
 }  
 else  
 {  
 kvp.Value[ 0 ].Visible = true;  
 for( int i = 1; i < linkCount; i++ )  
 {  
 kvp.Value[ i ].SetExternalLink( kvp.Value[ 0 ].NodeId, kvp.Value[ 0 ].PortId );  
 kvp.Value[ i ].Visible = false;  
 }  
 }  
 kvp.Value.Clear();  
 }  
 registeredLinks.Clear();  
 registeredLinks = null;  
 }  
 masterNodes.NodesList.Sort( ( x, y ) => ( x.SubShaderIdx \* 1000 + x.PassIdx ).CompareTo( y.SubShaderIdx \* 1000 + y.PassIdx ) );  
 masterNodes.UpdateNodeArr();  
 m\_parentWindow.TemplatesManagerInstance.ResetOptionsSetupData();  
 for( int i = 0; i < mpCount; i++ )  
 {  
 int visiblePorts = 0;  
 for( int j = 0; j < masterNodes.NodesList[ i ].InputPorts.Count; j++ )  
 {  
 if( masterNodes.NodesList[ i ].InputPorts[ j ].Visible )  
 {  
 visiblePorts++;  
 }  
 }  
 if( masterNodes.NodesList[ i ].VisiblePorts != visiblePorts )  
 {  
 masterNodes.NodesList[ i ].VisiblePorts = visiblePorts;  
 ForceRepositionCheck = true;  
 }  
 masterNodes.NodesList[ i ].Docking = visiblePorts <= 0;  
 if( optionsUpdate )  
 {  
 masterNodes.NodesList[ i ].ForceOptionsRefresh();  
 }  
 }  
 }  
 void CheckLinkedPorts( ref Dictionary<string, List<InputPort>> registeredLinks, TemplateMultiPassMasterNode masterNode )  
 {  
 if( masterNode.HasLinkPorts )  
 {  
 int inputCount = masterNode.InputPorts.Count;  
 for( int i = 0; i < inputCount; i++ )  
 {  
 if( !string.IsNullOrEmpty( masterNode.InputPorts[ i ].ExternalLinkId ) )  
 {  
 string linkId = masterNode.InputPorts[ i ].ExternalLinkId;  
 if( !registeredLinks.ContainsKey( masterNode.InputPorts[ i ].ExternalLinkId ) )  
 {  
 registeredLinks.Add( linkId, new List<InputPort>() );  
 }  
 if( masterNode.IsMainOutputNode )  
 {  
 registeredLinks[ linkId ].Insert( 0, masterNode.InputPorts[ i ] );  
 }  
 else  
 {  
 registeredLinks[ linkId ].Add( masterNode.InputPorts[ i ] );  
 }  
 }  
 else  
 {  
 masterNode.InputPorts[ i ].Visible = true;  
 }  
 }  
 }  
 else  
 {  
 int inputCount = masterNode.InputPorts.Count;  
 for( int i = 0; i < inputCount; i++ )  
 {  
 masterNode.InputPorts[ i ].Visible = true;  
 }  
 }  
 }  
 public MasterNode ReplaceMasterNode( AvailableShaderTypes newType, bool writeDefaultData = false, TemplateDataParent templateData = null )  
 {  
 DeSelectAll();  
 ResetNodeConnStatus();  
 MasterNode newMasterNode = null;  
 List<TemplateMultiPassMasterNode> nodesToDelete = null;  
 int mpNodeCount = m\_multiPassMasterNodes.NodesList.Count;  
 if( mpNodeCount > 0 )  
 {  
 nodesToDelete = new List<TemplateMultiPassMasterNode>();  
 for( int i = 0; i < mpNodeCount; i++ )  
 {  
 if( m\_multiPassMasterNodes.NodesList[ i ].UniqueId != m\_masterNodeId )  
 {  
 nodesToDelete.Add( m\_multiPassMasterNodes.NodesList[ i ] );  
 }  
 }  
 for( int lod = 0; lod < m\_lodMultiPassMasterNodes.Count; lod++ )  
 {  
 int lodNodeCount = m\_lodMultiPassMasterNodes[ lod ].Count;  
 for( int i = 0; i < lodNodeCount; i++ )  
 {  
 nodesToDelete.Add( m\_lodMultiPassMasterNodes[ lod ].NodesList[ i ] );  
 }  
 }  
 }  
 MasterNode currMasterNode = GetNode( m\_masterNodeId ) as MasterNode;  
 if( currMasterNode != null )  
 {  
 currMasterNode.ReleaseResources();  
 }  
 bool refreshLinkedMasterNodes = false;  
 switch( newType )  
 {  
 default:  
 case AvailableShaderTypes.SurfaceShader:  
 {  
 CurrentCanvasMode = NodeAvailability.SurfaceShader;  
 m\_currentSRPType = TemplateSRPType.BuiltIn;  
 newMasterNode = CreateNode( typeof( StandardSurfaceOutputNode ), false ) as MasterNode;  
 }  
 break;  
 case AvailableShaderTypes.Template:  
 {  
 CurrentCanvasMode = NodeAvailability.TemplateShader;  
 if( templateData.TemplateType == TemplateDataType.LegacySinglePass )  
 {  
 newMasterNode = CreateNode( typeof( TemplateMasterNode ), false ) as MasterNode;  
 ( newMasterNode as TemplateMasterNode ).SetTemplate( templateData as TemplateData, writeDefaultData, false );  
 m\_currentSRPType = TemplateSRPType.BuiltIn;  
 }  
 else  
 {  
 TemplateMultiPass multipassData = templateData as TemplateMultiPass;  
 m\_currentSRPType = multipassData.SubShaders[ 0 ].Modules.SRPType;  
 Vector2 currentPosition = currMasterNode.Vec2Position;  
 for( int subShaderIdx = 0; subShaderIdx < multipassData.SubShaders.Count; subShaderIdx++ )  
 {  
 for( int passIdx = 0; passIdx < multipassData.SubShaders[ subShaderIdx ].Passes.Count; passIdx++ )  
 {  
 TemplateMultiPassMasterNode masterNode = CreateNode( typeof( TemplateMultiPassMasterNode ), false ) as TemplateMultiPassMasterNode;  
 if( multipassData.SubShaders[ subShaderIdx ].Passes[ passIdx ].IsMainPass )  
 {  
 newMasterNode = masterNode;  
 ParentWindow.IsShaderFunctionWindow = false;  
 CurrentCanvasMode = NodeAvailability.TemplateShader;  
 }  
 masterNode.Vec2Position = currentPosition;  
 masterNode.SetTemplate( multipassData, true, true, subShaderIdx, passIdx, SetTemplateSource.NewShader );  
 }  
 }  
 refreshLinkedMasterNodes = true;  
 }  
 }  
 break;  
 }  
 if( currMasterNode != null )  
 {  
 newMasterNode.CopyFrom( currMasterNode );  
 m\_masterNodeId = -1;  
 DestroyNode( currMasterNode, false, true );  
 }  
 if( nodesToDelete != null )  
 {  
 for( int i = 0; i < nodesToDelete.Count; i++ )  
 {  
 DestroyNode( nodesToDelete[ i ], false, true );  
 }  
 nodesToDelete.Clear();  
 }  
 m\_masterNodeId = newMasterNode.UniqueId;  
 if( refreshLinkedMasterNodes )  
 RefreshLinkedMasterNodes( true );  
 newMasterNode.OnMaterialUpdatedEvent += OnMaterialUpdatedEvent;  
 newMasterNode.OnShaderUpdatedEvent += OnShaderUpdatedEvent;  
 newMasterNode.IsMainOutputNode = true;  
 OnRefreshLinkedPortsComplete();  
 FullCleanUndoStack();  
 return newMasterNode;  
 }  
 private void RepositionTemplateNodes( MasterNode newMasterNode )  
 {  
 m\_forceRepositionCheck = false;  
 int dockedElementsBefore = 0;  
 int dockedElementsAfter = 0;  
 int masterIndex = 0;  
 bool foundMaster = false;  
 for( int i = 0; i < MultiPassMasterNodes.Count; i++ )  
 {  
 if( MultiPassMasterNodes.NodesList[ i ].UniqueId == m\_masterNodeId )  
 {  
 foundMaster = true;  
 masterIndex = i;  
 }  
 if( !MultiPassMasterNodes.NodesList[ i ].IsInvisible && MultiPassMasterNodes.NodesList[ i ].Docking )  
 {  
 if( foundMaster )  
 dockedElementsAfter++;  
 else  
 dockedElementsBefore++;  
 }  
 }  
 if( dockedElementsBefore > 0 )  
 {  
 newMasterNode.UseSquareNodeTitle = true;  
 }  
 for( int i = masterIndex - 1; i >= 0; i-- )  
 {  
 float forwardTracking = 0;  
 for( int j = i + 1; j <= masterIndex; j++ )  
 {  
 if( !MultiPassMasterNodes.NodesList[ i ].IsInvisible && !MultiPassMasterNodes.NodesList[ j ].Docking )  
 {  
 forwardTracking += MultiPassMasterNodes.NodesList[ j ].HeightEstimate + 10;  
 }  
 }  
 MasterNode node = MultiPassMasterNodes.NodesList[ i ];  
 node.Vec2Position = new Vector2( node.Vec2Position.x, newMasterNode.Position.y - forwardTracking - 33 \* ( dockedElementsBefore ) );  
 }  
 for( int i = masterIndex + 1; i < MultiPassMasterNodes.Count; i++ )  
 {  
 if( MultiPassMasterNodes.NodesList[ i ].UniqueId == newMasterNode.UniqueId || MultiPassMasterNodes.NodesList[ i ].Docking )  
 continue;  
 float backTracking = 0;  
 for( int j = i - 1; j >= masterIndex; j-- )  
 {  
 if( !MultiPassMasterNodes.NodesList[ i ].IsInvisible && !MultiPassMasterNodes.NodesList[ j ].Docking )  
 {  
 backTracking += MultiPassMasterNodes.NodesList[ j ].HeightEstimate + 10;  
 }  
 }  
 MasterNode node = MultiPassMasterNodes.NodesList[ i ];  
 node.Vec2Position = new Vector2( node.Vec2Position.x, newMasterNode.Position.y + backTracking + 33 \* ( dockedElementsAfter ) );  
 }  
 }  
 public void CreateNewEmpty( string name )  
 {  
 CleanNodes();  
 if( m\_masterNodeDefaultType == null )  
 m\_masterNodeDefaultType = typeof( StandardSurfaceOutputNode );  
 MasterNode newMasterNode = CreateNode( m\_masterNodeDefaultType, false ) as MasterNode;  
 newMasterNode.SetName( name );  
 m\_masterNodeId = newMasterNode.UniqueId;  
 ParentWindow.IsShaderFunctionWindow = false;  
 CurrentCanvasMode = NodeAvailability.SurfaceShader;  
 newMasterNode.OnMaterialUpdatedEvent += OnMaterialUpdatedEvent;  
 newMasterNode.OnShaderUpdatedEvent += OnShaderUpdatedEvent;  
 newMasterNode.IsMainOutputNode = true;  
 LoadedShaderVersion = VersionInfo.FullNumber;  
 }  
 public void CreateNewEmptyTemplate( string templateGUID )  
 {  
 CleanNodes();  
 TemplateDataParent templateData = m\_parentWindow.TemplatesManagerInstance.GetTemplate( templateGUID );  
 if( templateData.TemplateType == TemplateDataType.LegacySinglePass )  
 {  
 TemplateMasterNode newMasterNode = CreateNode( typeof( TemplateMasterNode ), false ) as TemplateMasterNode;  
 m\_masterNodeId = newMasterNode.UniqueId;  
 ParentWindow.IsShaderFunctionWindow = false;  
 CurrentCanvasMode = NodeAvailability.TemplateShader;  
 m\_currentSRPType = TemplateSRPType.BuiltIn;  
 newMasterNode.OnMaterialUpdatedEvent += OnMaterialUpdatedEvent;  
 newMasterNode.OnShaderUpdatedEvent += OnShaderUpdatedEvent;  
 newMasterNode.IsMainOutputNode = true;  
 newMasterNode.SetTemplate( templateData as TemplateData, true, true );  
 }  
 else  
 {  
 TemplateMultiPass multipassData = templateData as TemplateMultiPass;  
 m\_currentSRPType = multipassData.SubShaders[ 0 ].Modules.SRPType;  
 Vector2 currentPosition = Vector2.zero;  
 for( int subShaderIdx = 0; subShaderIdx < multipassData.SubShaders.Count; subShaderIdx++ )  
 {  
 for( int passIdx = 0; passIdx < multipassData.SubShaders[ subShaderIdx ].Passes.Count; passIdx++ )  
 {  
 TemplateMultiPassMasterNode newMasterNode = CreateNode( typeof( TemplateMultiPassMasterNode ), false ) as TemplateMultiPassMasterNode;  
 if( multipassData.SubShaders[ subShaderIdx ].Passes[ passIdx ].IsMainPass )  
 {  
 m\_masterNodeId = newMasterNode.UniqueId;  
 ParentWindow.IsShaderFunctionWindow = false;  
 CurrentCanvasMode = NodeAvailability.TemplateShader;  
 newMasterNode.OnMaterialUpdatedEvent += OnMaterialUpdatedEvent;  
 newMasterNode.OnShaderUpdatedEvent += OnShaderUpdatedEvent;  
 newMasterNode.IsMainOutputNode = true;  
 }  
 newMasterNode.Vec2Position = currentPosition;  
 newMasterNode.SetTemplate( multipassData, true, true, subShaderIdx, passIdx, SetTemplateSource.NewShader );  
 }  
 }  
 RefreshLinkedMasterNodes( false );  
 OnRefreshLinkedPortsComplete();  
 }  
 LoadedShaderVersion = VersionInfo.FullNumber;  
 }  
 public void CreateNewEmptyFunction( AmplifyShaderFunction shaderFunction )  
 {  
 CleanNodes();  
 FunctionOutput newOutputNode = CreateNode( typeof( FunctionOutput ), false ) as FunctionOutput;  
 m\_masterNodeId = newOutputNode.UniqueId;  
 ParentWindow.IsShaderFunctionWindow = true;  
 CurrentCanvasMode = NodeAvailability.ShaderFunction;  
 newOutputNode.IsMainOutputNode = true;  
 }  
 public void ForceCategoryRefresh() { m\_forceCategoryRefresh = true; }  
 public void RefreshExternalReferences()  
 {  
 int count = m\_nodes.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 m\_nodes[ i ].RefreshExternalReferences();  
 }  
 }  
 public Vector2 SelectedNodesCentroid  
 {  
 get  
 {  
 if( m\_selectedNodes.Count == 0 )  
 return Vector2.zero;  
 Vector2 pos = new Vector2( 0, 0 );  
 for( int i = 0; i < m\_selectedNodes.Count; i++ )  
 {  
 pos += m\_selectedNodes[ i ].Vec2Position;  
 }  
 pos /= m\_selectedNodes.Count;  
 return pos;  
 }  
 }  
 public void AddVirtualTextureCount()  
 {  
 m\_virtualTextureCount += 1;  
 }  
 public void RemoveVirtualTextureCount()  
 {  
 m\_virtualTextureCount -= 1;  
 if( m\_virtualTextureCount < 0 )  
 {  
 Debug.LogWarning( "Invalid virtual texture count" );  
 }  
 }  
 public bool HasVirtualTexture { get { return m\_virtualTextureCount > 0; } }  
 public void AddInstancePropertyCount()  
 {  
 m\_instancePropertyCount += 1;  
 }  
 public void RemoveInstancePropertyCount()  
 {  
 m\_instancePropertyCount -= 1;  
 if( m\_instancePropertyCount < 0 )  
 {  
 Debug.LogWarning( "Invalid property instance count" );  
 }  
 }  
 public int InstancePropertyCount { get { return m\_instancePropertyCount; } set { m\_instancePropertyCount = value; } }  
 public bool IsInstancedShader { get { return m\_instancePropertyCount > 0; } }  
 public void AddNormalDependentCount() { m\_normalDependentCount += 1; }  
 public void RemoveNormalDependentCount()  
 {  
 m\_normalDependentCount -= 1;  
 if( m\_normalDependentCount < 0 )  
 {  
 Debug.LogWarning( "Invalid normal dependentCount count" );  
 }  
 }  
 public void SetModeFromMasterNode()  
 {  
 MasterNode masterNode = CurrentMasterNode;  
 if( masterNode != null )  
 {  
 switch( masterNode.CurrentMasterNodeCategory )  
 {  
 default:  
 case AvailableShaderTypes.SurfaceShader:  
 {  
 if( masterNode is StandardSurfaceOutputNode )  
 CurrentCanvasMode = ParentWindow.CurrentNodeAvailability;  
 else  
 CurrentCanvasMode = NodeAvailability.SurfaceShader;  
 }  
 break;  
 case AvailableShaderTypes.Template:  
 {  
 CurrentCanvasMode = NodeAvailability.TemplateShader;  
 }  
 break;  
 }  
 }  
 else  
 {  
 CurrentCanvasMode = NodeAvailability.SurfaceShader;  
 }  
 }  
 public void MarkToDelete( ParentNode node )  
 {  
 m\_markedForDeletion.Add( node );  
 }  
 public bool IsMasterNode( ParentNode node )  
 {  
 return ( node.UniqueId == m\_masterNodeId ) ||  
 m\_multiPassMasterNodes.HasNode( node.UniqueId );  
 }  
 public TemplateMultiPassMasterNode GetMainMasterNodeOfLOD( int lod )  
 {  
 if( lod == -1 )  
 return CurrentMasterNode as TemplateMultiPassMasterNode;  
 return m\_lodMultiPassMasterNodes[ lod ].NodesList.Find( x => x.IsMainOutputNode );  
 }  
 public TemplateMultiPassMasterNode GetMasterNodeOfPass( string passName, int lod )  
 {  
 if( lod == -1 )  
 return m\_multiPassMasterNodes.NodesList.Find( x => x.PassName.Equals( passName ) );  
 return m\_lodMultiPassMasterNodes[lod].NodesList.Find( x => x.PassName.Equals( passName ) );  
 }  
 public void ForceMultiPassMasterNodesRefresh()  
 {  
 int mainOutputId = 0;  
 int count = m\_multiPassMasterNodes.Count;  
 for( int i = 0; i < count; i++ )  
 {  
 m\_multiPassMasterNodes.NodesList[ i ].ForceTemplateRefresh();  
 if( m\_multiPassMasterNodes.NodesList[ i ].IsMainOutputNode )  
 mainOutputId = i;  
 }  
 int lodCount = m\_lodMultiPassMasterNodes.Count;  
 for( int i = 0; i < lodCount; i++ )  
 {  
 if( m\_lodMultiPassMasterNodes[ i ] != null )  
 {  
 count = m\_lodMultiPassMasterNodes[ i ].Count;  
 for( int j = 0; j < count; j++ )  
 {  
 m\_lodMultiPassMasterNodes[ i ].NodesList[ j ].ForceTemplateRefresh();  
 }  
 }  
 }  
 m\_multiPassMasterNodes.NodesList[ mainOutputId ].CheckTemplateChanges();  
 }  
 public void SetLateOptionsRefresh()  
 {  
 m\_lateOptionsRefresh = true;  
 }  
 public void CreateLodMasterNodes( TemplateMultiPass templateMultiPass,int index, Vector2 initialPosition )  
 {  
 for( int lod = 0; lod < m\_lodMultiPassMasterNodes.Count; lod++ )  
 {  
 if( m\_lodMultiPassMasterNodes[ lod ].Count == 0 )  
 {  
 TemplateMultiPassMasterNode reference = CurrentMasterNode as TemplateMultiPassMasterNode;  
 int shaderLod = -1;  
 if( lod == 0 )  
 {  
 shaderLod = reference.ShaderLOD - MasterNodeLODIncrement;  
 }  
 else  
 {  
 if( index == -2 )  
 {  
 shaderLod = m\_lodMultiPassMasterNodes[ lod - 1 ].NodesList[ reference.PassIdx ].ShaderLOD - MasterNodeLODIncrement;  
 }  
 else if( index == -1 )  
 {  
 int mainShaderLOD = m\_lodMultiPassMasterNodes[ 0 ].NodesList[ reference.PassIdx ].ShaderLOD;  
 shaderLod = ( reference.ShaderLOD + mainShaderLOD )/2;  
 }  
 else  
 {  
 if( m\_lodMultiPassMasterNodes[ index ].Count > 0 )  
 {  
 if( m\_lodMultiPassMasterNodes[ index + 1 ].Count > 0 )  
 {  
 shaderLod = (m\_lodMultiPassMasterNodes[ index ].NodesList[ reference.PassIdx ].ShaderLOD +  
 m\_lodMultiPassMasterNodes[ index + 1 ].NodesList[ reference.PassIdx ].ShaderLOD )/2;  
 }  
 else  
 {  
 shaderLod = m\_lodMultiPassMasterNodes[ index ].NodesList[ reference.PassIdx ].ShaderLOD - MasterNodeLODIncrement;  
 }  
 }  
 }  
 }  
 int nodeId = 0;  
 TemplateMultiPassMasterNode mainMasterNode = null;  
 for( int subShaderIdx = 0; subShaderIdx < templateMultiPass.SubShaders.Count; subShaderIdx++ )  
 {  
 for( int passIdx = 0; passIdx < templateMultiPass.SubShaders[ subShaderIdx ].Passes.Count; passIdx++ )  
 {  
 TemplateMultiPassMasterNode masterNode = ScriptableObject.CreateInstance( typeof( TemplateMultiPassMasterNode ) ) as TemplateMultiPassMasterNode;  
 masterNode.LODIndex = lod;  
 masterNode.ContainerGraph = this;  
 masterNode.Vec2Position = initialPosition;  
 AddNode( masterNode, true );  
 masterNode.SetTemplate( templateMultiPass, true, true, subShaderIdx, passIdx, SetTemplateSource.NewShader );  
 masterNode.CopyOptionsFrom( m\_multiPassMasterNodes.NodesList[ nodeId++ ] );  
 if( masterNode.IsMainOutputNode || ( subShaderIdx == 0 && passIdx == 0 ) )  
 {  
 masterNode.SetShaderLODValueAndLabel( shaderLod );  
 mainMasterNode = masterNode;  
 }  
 }  
 }  
 mainMasterNode.ForceOptionsRefresh();  
 SortLODMasterNodes();  
 if( OnLODMasterNodesAddedEvent != null )  
 {  
 OnLODMasterNodesAddedEvent( lod );  
 }  
 TemplateMultiPassMasterNode lodMainMasterNode = CurrentMasterNode as TemplateMultiPassMasterNode;  
 lodMainMasterNode.SetShaderLODValueAndLabel( lodMainMasterNode.ShaderLOD );  
 return;  
 }  
 }  
 }  
 public void DestroyLodMasterNodes( int index )  
 {  
 if( index < 0 )  
 {  
 for( int lod = m\_lodMultiPassMasterNodes.Count - 1; lod >= 0; lod-- )  
 {  
 if( m\_lodMultiPassMasterNodes[ lod ].Count > 0 )  
 {  
 while( m\_lodMultiPassMasterNodes[ lod ].Count > 0 )  
 {  
 DestroyNode( m\_lodMultiPassMasterNodes[ lod ].NodesList[ 0 ], false, true );  
 }  
 break;  
 }  
 }  
 }  
 else  
 {  
 while( m\_lodMultiPassMasterNodes[ index ].Count > 0 )  
 {  
 DestroyNode( m\_lodMultiPassMasterNodes[ index ].NodesList[ 0 ], false, true );  
 }  
 }  
 SortLODMasterNodes();  
 TemplateMultiPassMasterNode lodMainMasterNode = CurrentMasterNode as TemplateMultiPassMasterNode;  
 lodMainMasterNode.SetShaderLODValueAndLabel( lodMainMasterNode.ShaderLOD );  
 }  
 public void SortLODMasterNodes()  
 {  
 int idx = (CurrentMasterNode as TemplateMultiPassMasterNode).PassIdx;  
 m\_lodMultiPassMasterNodes.Sort( ( x, y ) =>  
 {  
 if( x.Count > 0 )  
 {  
 if( y.Count > 0 )  
 {  
 return -x.NodesList[ idx ].ShaderLOD.CompareTo( y.NodesList[ idx ].ShaderLOD );  
 }  
 else  
 {  
 return -1;  
 }  
 }  
 else  
 {  
 if( y.Count > 0 )  
 {  
 return 1;  
 }  
 }  
 return 0;  
 });  
 for( int lodIdx = 0; lodIdx < m\_lodMultiPassMasterNodes.Count; lodIdx++ )  
 {  
 for( int nodeIdx = 0; nodeIdx < m\_lodMultiPassMasterNodes[ lodIdx ].Count; nodeIdx++ )  
 {  
 m\_lodMultiPassMasterNodes[ lodIdx ].NodesList[ nodeIdx ].LODIndex = lodIdx;  
 }   
 }  
 }  
 public List<TemplateMultiPassMasterNode> GetMultiPassMasterNodes( int lod )  
 {  
 if( lod == -1 )  
 return m\_multiPassMasterNodes.NodesList;  
 return m\_lodMultiPassMasterNodes[ lod ].NodesList;  
 }  
 public bool IsNormalDependent { get { return m\_normalDependentCount > 0; } }  
 public void MarkToDeselect() { m\_markedToDeSelect = true; }  
 public void MarkToSelect( int nodeId ) { m\_markToSelect = nodeId; }  
 public void MarkWireHighlights() { m\_checkSelectedWireHighlights = true; }  
 public List<ParentNode> SelectedNodes { get { return m\_selectedNodes; } }  
 public List<ParentNode> MarkedForDeletionNodes { get { return m\_markedForDeletion; } }  
 public int CurrentMasterNodeId { get { return m\_masterNodeId; } set { m\_masterNodeId = value; } }  
 public Shader CurrentShader  
 {  
 get  
 {  
 MasterNode masterNode = GetNode( m\_masterNodeId ) as MasterNode;  
 if( masterNode != null )  
 return masterNode.CurrentShader;  
 return null;  
 }  
 }  
 public Material CurrentMaterial  
 {  
 get  
 {  
 MasterNode masterNode = GetNode( m\_masterNodeId ) as MasterNode;  
 if( masterNode != null )  
 return masterNode.CurrentMaterial;  
 return null;  
 }  
 }  
 public NodeAvailability CurrentCanvasMode { get { return m\_currentCanvasMode; } set { m\_currentCanvasMode = value; ParentWindow.LateRefreshAvailableNodes(); } }  
 public OutputNode CurrentOutputNode { get { return GetNode( m\_masterNodeId ) as OutputNode; } }  
 public FunctionOutput CurrentFunctionOutput { get { return GetNode( m\_masterNodeId ) as FunctionOutput; } }  
 public MasterNode CurrentMasterNode { get { return GetNode( m\_masterNodeId ) as MasterNode; } }  
 public StandardSurfaceOutputNode CurrentStandardSurface { get { return GetNode( m\_masterNodeId ) as StandardSurfaceOutputNode; } }  
 public List<ParentNode> AllNodes { get { return m\_nodes; } }  
 public int NodeCount { get { return m\_nodes.Count; } }  
 public int NodeClicked  
 {  
 set { m\_nodeClicked = value; }  
 get { return m\_nodeClicked; }  
 }  
 public bool IsDirty  
 {  
 set { m\_isDirty = value && UIUtils.DirtyMask; }  
 get  
 {  
 bool value = m\_isDirty;  
 m\_isDirty = false;  
 return value;  
 }  
 }  
 public bool SaveIsDirty  
 {  
 set { m\_saveIsDirty = value && UIUtils.DirtyMask; }  
 get { return m\_saveIsDirty; }  
 }  
 public int LoadedShaderVersion  
 {  
 get { return m\_loadedShaderVersion; }  
 set { m\_loadedShaderVersion = value; }  
 }  
 public AmplifyShaderFunction CurrentShaderFunction  
 {  
 get { if( CurrentFunctionOutput != null ) return CurrentFunctionOutput.Function; else return null; }  
 set { if( CurrentFunctionOutput != null ) CurrentFunctionOutput.Function = value; }  
 }  
 public bool HasUnConnectedNodes { get { return m\_hasUnConnectedNodes; } }  
 public UsageListSamplerNodes SamplerNodes { get { return m\_samplerNodes; } }  
 public UsageListFloatIntNodes FloatIntNodes { get { return m\_floatNodes; } }  
 public UsageListTexturePropertyNodes TexturePropertyNodes { get { return m\_texturePropertyNodes; } }  
 public UsageListTextureArrayNodes TextureArrayNodes { get { return m\_textureArrayNodes; } }  
 public UsageListPropertyNodes PropertyNodes { get { return m\_propertyNodes; } }  
 public UsageListPropertyNodes RawPropertyNodes { get { return m\_rawPropertyNodes; } }  
 public UsageListCustomExpressionsOnFunctionMode CustomExpressionOnFunctionMode { get { return m\_customExpressionsOnFunctionMode; } }  
 public UsageListStaticSwitchNodes StaticSwitchNodes { get { return m\_staticSwitchNodes; } }  
 public UsageListScreenColorNodes ScreenColorNodes { get { return m\_screenColorNodes; } }  
 public UsageListRegisterLocalVarNodes LocalVarNodes { get { return m\_localVarNodes; } }  
 public UsageListGlobalArrayNodes GlobalArrayNodes { get { return m\_globalArrayNodes; } }  
 public UsageListFunctionInputNodes FunctionInputNodes { get { return m\_functionInputNodes; } }  
 public UsageListFunctionNodes FunctionNodes { get { return m\_functionNodes; } }  
 public UsageListFunctionOutputNodes FunctionOutputNodes { get { return m\_functionOutputNodes; } }  
 public UsageListFunctionSwitchNodes FunctionSwitchNodes { get { return m\_functionSwitchNodes; } }  
 public UsageListFunctionSwitchCopyNodes FunctionSwitchCopyNodes { get { return m\_functionSwitchCopyNodes; } }  
 public UsageListTemplateMultiPassMasterNodes MultiPassMasterNodes { get { return m\_multiPassMasterNodes; } set { m\_multiPassMasterNodes = value; } }  
 public List<UsageListTemplateMultiPassMasterNodes> LodMultiPassMasternodes { get { return m\_lodMultiPassMasterNodes; } }  
 public PrecisionType CurrentPrecision  
 {  
 get { return m\_currentPrecision; }  
 set { m\_currentPrecision = value; }  
 }  
 public NodeLOD LodLevel  
 {  
 get { return m\_lodLevel; }  
 }  
 public List<ParentNode> NodePreviewList { get { return m\_nodePreviewList; } set { m\_nodePreviewList = value; } }  
 public void SetGraphId( int id )  
 {  
 m\_graphId = id;  
 }  
 public int GraphId  
 {  
 get { return m\_graphId; }  
 }  
 public AmplifyShaderEditorWindow ParentWindow  
 {  
 get { return m\_parentWindow; }  
 set { m\_parentWindow = value; }  
 }  
 public bool ChangedLightingModel  
 {  
 get { return m\_changedLightingModel; }  
 set { m\_changedLightingModel = value; }  
 }  
 public bool ForceRepositionCheck  
 {  
 get { return m\_forceRepositionCheck; }  
 set { m\_forceRepositionCheck = value; }  
 }  
 public bool IsLoading { get { return m\_isLoading; } set { m\_isLoading = value; } }  
 public bool IsDuplicating { get { return m\_isDuplicating; } set { m\_isDuplicating = value; } }  
 public TemplateSRPType CurrentSRPType { get { return m\_currentSRPType; }set { m\_currentSRPType = value; } }  
 public bool IsSRP { get { return m\_currentSRPType == TemplateSRPType.Lightweight || m\_currentSRPType == TemplateSRPType.HD; } }  
 public bool IsHDRP { get { return m\_currentSRPType == TemplateSRPType.HD; } }  
 public bool IsLWRP { get { return m\_currentSRPType == TemplateSRPType.Lightweight; } }  
 public bool IsStandardSurface { get { return GetNode( m\_masterNodeId ) is StandardSurfaceOutputNode; } }  
 public bool SamplingMacros {   
 get { return m\_samplingThroughMacros; }  
 set { m\_samplingThroughMacros = value; }   
 }  
 public bool HasLODs { get { return m\_lodMultiPassMasterNodes[ 0 ].Count > 0; } }  
 }  
}