```
        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; } }
		}
}
```