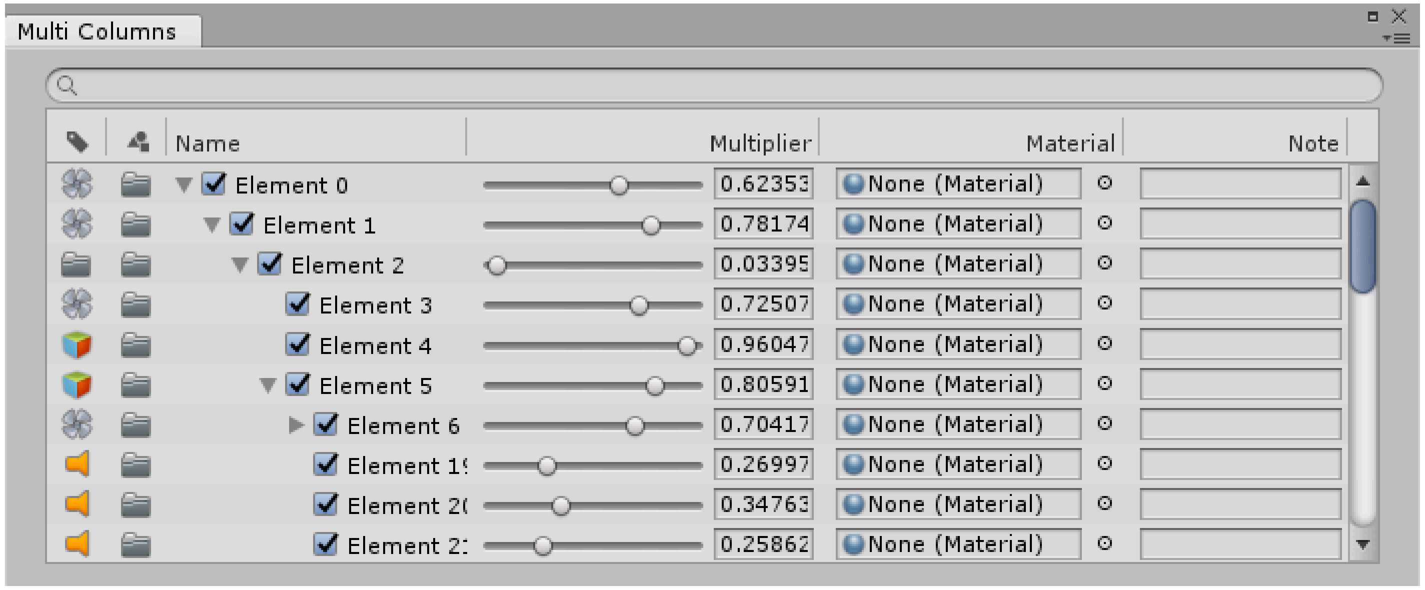
# TreeView Manual

With the addition of the TreeView API to the IMGUI toolset we seek to help the people that write Unity Editor Extensions to take advantage of the tools we use internally at Unity. The information in the following assumes the reader has basic knowledge of IMGUI concepts. Otherwise a good place to start is here and here.

The TreeView is an IMGUI control that can display hierarchical data that can be expanded and collapsed. But it also lets you create list views and multi-column tables for Editor tools. It is highly customizable and can be composed with other IMGUI controls and components. The TreeView API is somewhat different from our 'traditional' IMGUI controls which mainly are static method calls. The TreeView is a control you allocate an instance of up front and that can have internal state (selection, expanded items etc).



| Name | Multiplier | Material | Note |
|---|---|---|---|
| Element 0 | 0.62353 | None (Material) | |
| Element 1 | 0.78174 | None (Material) | |
| Element 2 | 0.03395 | None (Material) | |
| Element 3 | 0.72507 | None (Material) | |
| Element 4 | 0.96047 | None (Material) | |
| Element 5 | 0.80591 | None (Material) | |
| Element 6 | 0.70417 | None (Material) | |
| Element 1! | 0.26997 | None (Material) | |
| Element 2( | 0.34763 | None (Material) | |
| Element 2: | 0.25862 | None (Material) | |

*Example of a TreeView with a MultiColumnHeader.*

Note that the TreeView is not a Tree Data Model. Therefore you can construct the TreeView using any back-end tree data-structure you prefer. This can be a C# tree model or a Unity based tree structure like the Transform hierarchy. What you should keep in mind is how you persist the data you are modifying with the TreeView.

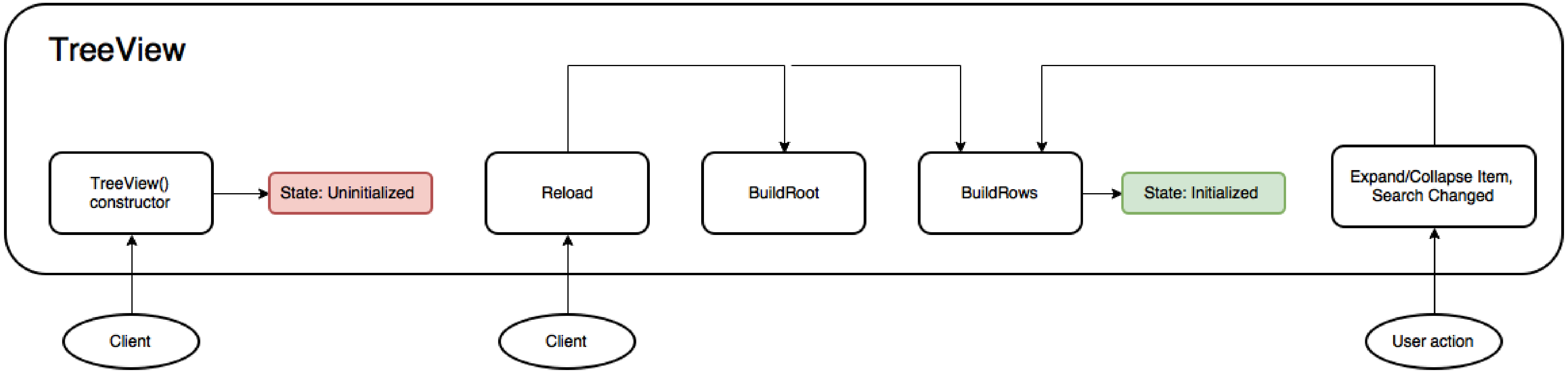
The project with the source code for the examples shown below can be downloaded here.

## Important classes and methods

The most important classes beside the TreeView itself are the TreeViewItem and the TreeViewState classes.
**TreeViewState** contains state information that the can be changed by interaction with the TreeView, for example: selection state, expanded state, navigation state and scroll state. TreeViewState is the only state that is serializable. The TreeView itself is not serializable; it is reconstructed from the backend data it is representing. The TreeViewState should be persisted across domain reloading to ensure user changed state is not lost (domain reloading happens when going into playmode or after script recompiling). The TreeViewState can be persisted by adding it as a field in the EditorWindow class, see the examples project.
**TreeViewItem** contains data about an individual item and is used to build the representation of the tree structure in the Editor. Each TreeViewItem should be constructed with a unique integer ID (unique among all the items in the TreeView). The ID is used for finding items in the tree, for selection state, expanded state and navigation. If the tree is representing Unity objects it is a good idea to use the GetInstanceID() for each object as ID for the TreeViewItem. The IDs are used in the TreeViewState to persist the user changed state across domain reloading.
**Depth:** All TreeViewItems have a *depth* property which indicates the visual indentation. Because we want to accommodate many different tree back-ends we have ensured multiple ways to create the tree of TreeViewItems. See also *Initializing a TreeView* section for more information on the ways a TreeView can be initialized.
**Root:** The TreeView has a single root TreeViewItem which is hidden (never shown). This item is the root of all other items. Using the children properties of the items the tree can be traversed. When this root is created it is required to have depth = -1. The first visible item is at depth = 0.
**Rows:** While the root item is the root of the logical tree, the rendering of the TreeView is handled by determining a list of expanded items. We call this linear list the *rows*. This list is cached for performance. Each row represents one TreeViewItem. TreeViewItem can be derived from to add custom data that can be used when rendering the item in the TreeView. Each TreeViewItem has parent and children info which is used by the TreeView to determine navigation and other internal functionality.

**The BuildRoot method:** Is the single abstract method of the TreeView that is required to be implemented to create a TreeView. This method should handle creating the root item of the tree. This is called once every time Reload() is called on the tree. By default you should create the

entire tree structure under the root in this method. For very large trees it might not be optimal to create the entire tree on every reload. In this situation just create the root and then override the BuildRows method to only create items for the currently rows.
**The BuildRows method:** This is a virtual method where the default implementation handles building the rows list based on the full tree created in BuildRoot. If only the root was created in BuildRoot then this method should be overridden to handle the expanded rows. See also *Initializing a TreeView*.



*The diagram above summarises the ordering and repetition of BuildRoot and BuildRows event methods during a TreeView's lifetime. Note that the BuildRoot method is called once every time Reload() is called. BuildRows is called more often because it is called once on reload (right after BuildRoot) and every time TreeViewItems are being expanded or collapsed.*

## Initializing a TreeView

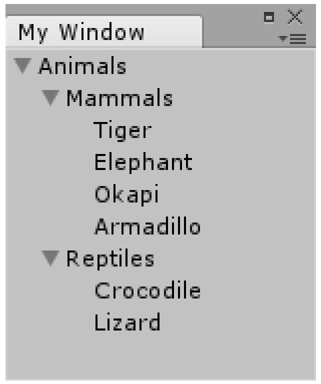
The TreeView is being initialized when calling its Reload() method. Its tree structure can be initialized in different ways to best accommodate different back-end tree model APIs. Some tree model APIs are parent - children based while others are depths based etc.

There are two main approaches to setup the TreeView:
1) Create the full tree: create TreeViewItems for all items in the tree model data. This is the default and requires less code to setup: just implement BuildRoot to build the full tree.
2) Create only the expanded items. This approach scales best with large data sets or data that changes often. This approach requires you to override BuildRows to manually control the rows being shown and BuildRoot should just create the root TreeViewItem.

Each of these have pros and cons depending on the amount of data. For large data sets or data that changes often we recommend to use approach 2) as this is faster to initialize than a full tree.
No matter what approach is chosen above, there are also three ways of actual creating the TreeViewItems:

- Create TreeViewItems with children, parent and depths initialized from start.
- Create TreeViewItems with parent and children and then use SetupDepthsFromParentsAndChildren to set the depths.
- Create TreeViewItems only with depth information and then use SetupParentsAndChildrenFromDepths to set the the parent and children references.

## Example 1: A simple TreeView



To create a TreeView you basically have to derive from TreeView and implement the abstract method BuildRoot(). The following example creates a simple TreeView.

```
class SimpleTreeView : TreeView
{
	public SimpleTreeView(TreeViewState treeViewState)
		: base(treeViewState)
	{
		Reload();
	}

	protected override TreeViewItem BuildRoot ()
	{
		// BuildRoot is called every time Reload is called to ensure that TreeViewItems
		// are created from data. Here we just create a fixed set of items, in a real world example
		// a data model should be passed into the TreeView and the items created from the model.

		// This section illustrates that IDs should be unique and that the root item is required to
		// have a depth of -1 and the rest of the items increment from that.
		var root = new TreeViewItem {id = 0, depth = -1, displayName = "Root"};
		var allItems = new List<TreeViewItem>
		{
			new TreeViewItem {id = 1, depth = 0, displayName = "Animals"},
			new TreeViewItem {id = 2, depth = 1, displayName = "Mammals"},
			new TreeViewItem {id = 3, depth = 2, displayName = "Tiger"},
			new TreeViewItem {id = 4, depth = 2, displayName = "Elephant"},
			new TreeViewItem {id = 5, depth = 2, displayName = "Okapi"},
			new TreeViewItem {id = 6, depth = 2, displayName = "Armadillo"},
			new TreeViewItem {id = 7, depth = 1, displayName = "Reptiles"},
			new TreeViewItem {id = 8, depth = 2, displayName = "Crocodile"},
			new TreeViewItem {id = 9, depth = 2, displayName = "Lizard"},
		};

		// Utility method that initializes the TreeViewItem.children and .parent for all items.
		SetupParentsAndChildrenFromDepths (root, allItems);

		// Return root of the tree
		return root;
	}
}
```

In this example the depth information is used to build the tree view. Finally, a call to `SetupParentsAndChildrenUsingDepth` sets the parent and children data of the TreeViewItems. These are used by the TreeView for navigation, dragging etc.
Note that if you prefer, you can set up the TreeViewItems by setting the parent and children directly or use the AddChild method, see the example below.

```
protected override TreeViewItem BuildRoot()
{
       var root = new TreeViewItem      { id = 0, depth = -1, displayName = "Root" };
       var animals = new TreeViewItem   { id = 1, displayName = "Animals" };
       var mammals = new TreeViewItem   { id = 2, displayName = "Mammals" };
       var tiger = new TreeViewItem     { id = 3, displayName = "Tiger" };
       var elephant = new TreeViewItem  { id = 4, displayName = "Elephant" };
       var okapi = new TreeViewItem     { id = 5, displayName = "Okapi" };
       var armadillo = new TreeViewItem { id = 6, displayName = "Armadillo" };
       var reptiles = new TreeViewItem  { id = 7, displayName = "Reptiles" };
       var croco = new TreeViewItem     { id = 8, displayName = "Crocodile" };
       var lizard = new TreeViewItem    { id = 9, displayName = "Lizard" };

       root.AddChild(animals);
       animals.AddChild(mammals);
       animals.AddChild(reptiles);
       mammals.AddChild(tiger);
       mammals.AddChild(elephant);
       mammals.AddChild(okapi);
       mammals.AddChild(armadillo);
       reptiles.AddChild(croco);
       reptiles.AddChild(lizard);

       SetupDepthsFromParentsAndChildren(root);

       return root;
}
```
*Alternative BuildRoot method for the SimpleTreeView class above*

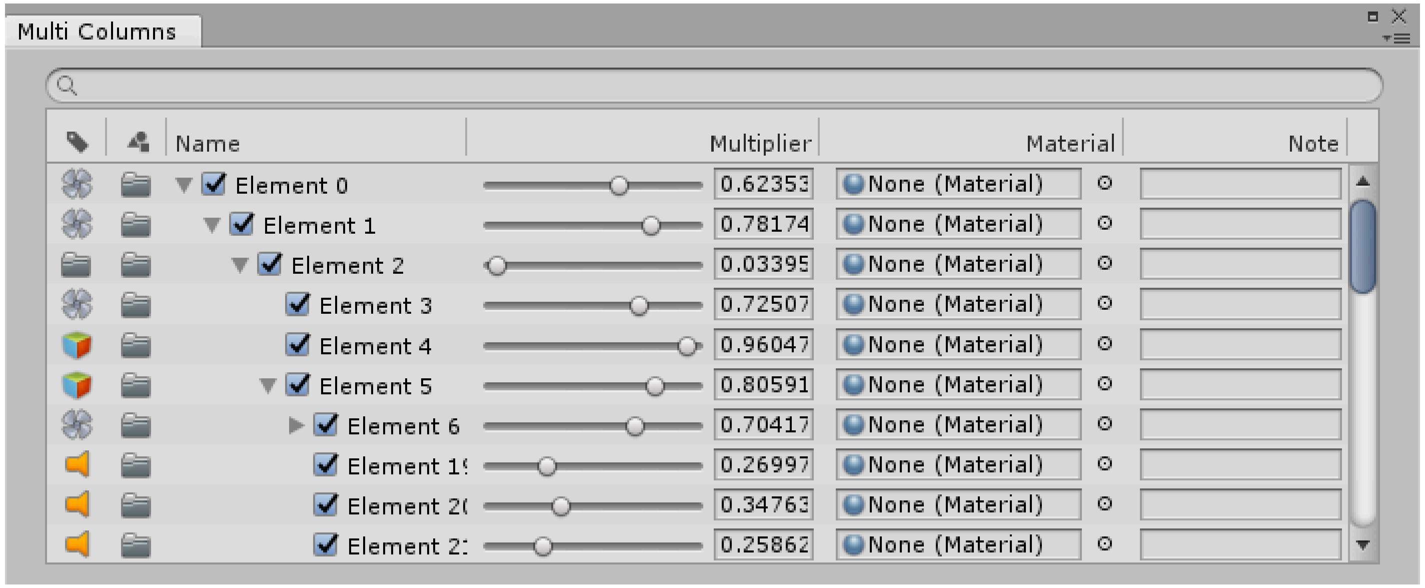
The following code snippet shows the EditorWindow that contains the SimpleTreeView. TreeViews are constructed with a TreeViewState instance. The client of the TreeView should determine how this view state should be handled: whether its state should persist until the next session of Unity, or whether it should only preserve its state after a assembly reload (either entering Play Mode or recompiling scripts). In this example, the TreeViewState is serialized in the EditorWindow, ensuring the TreeView preserves its state between consecutive sessions of the Unity Editor and as long the window is not closed.

```
using System.Collections.Generic;
using UnityEngine;
using UnityEditor.IMGUI.Controls;

class SimpleTreeViewWindow : EditorWindow
{
	// We are using SerializeField here to make sure view state is written to the window
	// layout file. This means that the state survives restarting Unity as long as the window
	// is not closed. If omitting the attribute then the state just survives assembly reloading
	// (i.e. it still gets serialized/deserialized)
	[SerializeField] TreeViewState m_TreeViewState;

	// The TreeView is not serializable it should be reconstructed from the tree data.
	SimpleTreeView m_SimpleTreeView;

	void OnEnable ()
	{
		// Check if we already had a serialized view state (state
		// that survived assembly reloading)
		if (m_TreeViewState == null)
			m_TreeViewState = new TreeViewState ();

		m_SimpleTreeView = new SimpleTreeView(m_TreeViewState);
	}

	void OnGUI ()
	{
		m_SimpleTreeView.OnGUI(new Rect(0, 0, position.width, position.height));
	}

	// Add menu named "My Window" to the Window menu
	[MenuItem ("TreeView Examples/Simple Tree Window")]
	static void ShowWindow ()
```

```
        {
                // Get existing open window or if none, make a new one:
                var window = GetWindow<SimpleTreeViewWindow> ();
                window.titleContent = new GUIContent ("My Window");
                window.Show ();
        }
}
```

## Example 2: A multi-column TreeView

For a more comprehensive example of using and customizing a TreeView, the following example illustrates a multi-column TreeView that uses the MultiColumnHeader class. It is used to illustrate the feature set of the TreeView and supports the following functionality: renaming items; multi-selection; reordering items and custom row content using normal IMGUI controls (such as sliders and object fields) for cell contents; sorting of columns; and filtering/searching of rows. The example also shows how the tree structure is serialized to a ScriptableObject and saved in an asset for persistence.


Multi Columns

| Name | Multiplier | Material | Note |
|---|---|---|---|
| Element 0 | 0.62353 | None (Material) | |
| Element 1 | 0.78174 | None (Material) | |
| Element 2 | 0.03395 | None (Material) | |
| Element 3 | 0.72507 | None (Material) | |
| Element 4 | 0.96047 | None (Material) | |
| Element 5 | 0.80591 | None (Material) | |
| Element 6 | 0.70417 | None (Material) | |
| Element 1 | 0.26997 | None (Material) | |
| Element 2 | 0.34763 | None (Material) | |
| Element 2 | 0.25862 | None (Material) | |

## Tree Model used for the multi-column example

For our multi column example we create a back-end data model using the classes: TreeElement and TreeModel. Data is fetched by the TreeView from this back-end TreeModel. The TreeElement and TreeModel classes have been built in to show the features of the TreeView class. They are included in the ‘TreeView Examples’ project and we will not go in depth with them here.

We start by extending the TreeElement data class to hold extra data that we can show and manipulate in the front-end TreeView.

```
[Serializable]
internal class MyTreeElement : TreeElement
{
	public float floatValue1, floatValue2, floatValue3;
	public Material material;
	public string text = "";
```

```
        public bool enabled = true;

        public MyTreeElement (string name, int depth, int id) : base (name, depth, id)
        {
               floatValue1 = Random.value;
               floatValue2 = Random.value;
               floatValue3 = Random.value;
        }
}
```

We use the following ScriptableObject class to persist our data in an asset.

```
[CreateAssetMenu (fileName = "TreeDataAsset", menuName = "Tree Asset", order = 1)]
public class MyTreeAsset : ScriptableObject
{
	[SerializeField] List<MyTreeElement> m_TreeElements = new List<MyTreeElement> ();

	internal List<MyTreeElement> treeElements
	{
		get { return m_TreeElements; }
		set { m_TreeElements = value; }
	}
}
```

## MultiColumnTreeView construction

In the following we will just show snippets of the MultiColumnTreeView that illustrate how the multi column GUI is achieved. For full source code look at the TreeView Examples project.

First we take a look at the MultiColumnTreeView constructor. Here we tweak the properties of the TreeView to our needs:

- *rowHeight = 20*: Change the default height (which is based on EditorGUIUtility.singleLineHeight’s 16 points) to 20 to make more vertical room for showing GUI controls.
- *columnIndexForTreeFoldouts = 2*: We do not show the foldout arrows in the default first column so change this to 2 since we have icons in the first two columns.

- *showAlternatingRowBackgrounds = true*:  For a table like this we enable the alternating row background colors to visually aid the contents that belong together on each row.
- *showBorder = true*: Our TreeView is ‘floating’ in the window so show a thin border to delimit it from the rest of the content
- *customFoldoutYOffset = (kRowHeights - EditorGUIUtility.singleLineHeight) * 0.5f*: Center foldout vertically in the row since we also center content. See RowGUI below
- *extraSpaceBeforeIconAndLabel = 20*: Make space before to the tree labels so we can show the toggle button
- *multicolumnHeader.sortingChanged += OnSortingChanged*: Assign a method to the event to get notified when the sorting changes in the header component (when the header column is clicked). So we can change the rows of the TreeView to reflect the sorting state.

```
public MultiColumnTreeView (TreeViewState state,
                            MultiColumnHeader multicolumnHeader,
                            TreeModel<MyTreeElement> model)
                            : base (state, multicolumnHeader, model)
{
       // Custom setup
       rowHeight = 20;
       columnIndexForTreeFoldouts = 2;
       showAlternatingRowBackgrounds = true;
       showBorder = true;
       customFoldoutYOffset = (kRowHeights - EditorGUIUtility.singleLineHeight) * 0.5f;
       extraSpaceBeforeIconAndLabel = kToggleWidth;
       multicolumnHeader.sortingChanged += OnSortingChanged;

       Reload();
}
```

## Customizing the GUI

If we use the default RowGUI handling the TreeView would look like the SimpleTreeView example above: with just foldouts and a label. But now that we have multiple data values for each item we override the method RowGUI(RowGUIArgs args) to be able to visualize this.

```
protected override void RowGUI (RowGUIArgs args)
```

Lets look at at the argument struct first to see what is available to us:

```
protected struct RowGUIArgs
{
	public TreeViewItem item;
	public string label;
	public Rect rowRect;
	public int row;
	public bool selected;
	public bool focused;
	public bool isRenaming;

	public int GetNumVisibleColumns ()
	public int GetColumn (int visibleColumnIndex)
	public Rect GetCellRect (int visibleColumnIndex)
}
```

First we see the *TreeViewItem item,* remember each item is represented by a row. This can be derived from to add additonal user data that can be used for GUI handling in the row. We therefore cast the item to our own type first before using. We will see this get used in CellGUI below.

Apart from the different UI states the item can be in (selected, being renamed etc) we also see there are three methods that are related to column handling. These are only valid to call when the TreeVies is constructed with a MultiColumHeader (otherwise an exception is thrown). Using the information from these column helper methods we can now start to add GUI elements for each column.

```
protected override void RowGUI (RowGUIArgs args)
{
	var item = (TreeViewItem<MyTreeElement>) args.item;

	for (int i = 0; i < args.GetNumVisibleColumns (); ++i)
	{
		CellGUI(args.GetCellRect(i), item, (MyColumns)args.GetColumn(i), ref args);
	}
}
```

```
void CellGUI (Rect cellRect, TreeViewItem<MyTreeElement> item, MyColumns column, ref RowGUIArgs args)
{
	// Center the cell rect vertically using EditorGUIUtility.singleLineHeight.
	// This makes it easier to place controls, icons etc in the cells.
	CenterRectUsingSingleLineHeight(ref cellRect);

	switch (column)
	{

		case MyColumns.Icon1:

			// Draw custom texture
			GUI.DrawTexture(cellRect, s_TestIcons[GetIcon1Index(item)], ScaleMode.ScaleToFit);
			break;

		case MyColumns.Icon2:

			//Draw custom texture
			GUI.DrawTexture(cellRect, s_TestIcons[GetIcon2Index(item)], ScaleMode.ScaleToFit);
			break;
```

```
              case MyColumns.Name:

                     // Make a toggle button to the left of the label text
                     Rect toggleRect = cellRect;
                     toggleRect.x += GetContentIndent(item);
                     toggleRect.width = kToggleWidth;
                     if (toggleRect.xMax < cellRect.xMax)
                            item.data.enabled = EditorGUI.Toggle(toggleRect, item.data.enabled);

                     // Default icon and label
                     args.rowRect = cellRect;
                     base.RowGUI(args);
                     break;

              case MyColumns.Value1:

                     // Show a Slider control for value 1
                     item.data.floatValue1 = EditorGUI.Slider(cellRect, GUIContent.none, item.data.floatValue1, 0f, 1f);
                     break;

              case MyColumns.Value2:

                     // Show an ObjectField for materials
                     item.data.material = (Material)EditorGUI.ObjectField(cellRect, GUIContent.none, item.data.material,
                                          typeof(Material), false);
                     break;

              case MyColumns.Value3:

                     // Show a TextField for the data text string
                     item.data.text = GUI.TextField(cellRect, item.data.text);
                     break;
       }
}
```

## TreeView FAQ

Q: In my class that subclasses the TreeView I have the functions, BuildRoot and RowGUI. Is RowGUI called for every TreeViewItem that got added in the build function, or really only for items that are visible on screen in the scroll view?
A: RowGUI is only called for the items visible on screen. Say you have 10.000 items then only the 20 visible items on screen have their RowGUI called.

Q: Can I get the indicies of the rows that are visible on the screen.
A: Yes. Use the method: GetFirstAndLastVisibleRows().

Q: Can I get the rows list I built in BuildRows?
A: Yes. Use GetRows()

Q: Is it mandatory for any of the overridden functions to call base.Method()
A: No, only if the method has a default behavior you want to extend.

Q: In my case I just want to make list of items (not a tree). Do I have to create the root?
A: Yes you should always have a root. You can create the root item and set root.children = rows for fast setup.

Q: If I add a Toggle to my row and click on it, the selection doesn't jump to that row.
A: No by default the row is only selected if the mouse down is not consumed by the contents of the row. Here your Toggle consumes the event. To fix this you can use method SelectionClick before your Toggle button is called.

Q: Is there methods I can use before and/or after all RowGUI methods are called?
A: Yes. See BeforeRowsGUI and AfterRowsGUI

Q: Is there a simple way to return key focus to the treeview from API? If I select a FloatField in my row the row selection becomes grey. How do I make it blue again?

A: The blue color indicates what has key focus, because the FloatField have focus the TreeView lost focus so this is intended behavoir. From API you can set GUIUtility.keyboardControl = treeViewControlID when needed.