How to use Editor GUI Table

Using the *[Table]* attribute (Full Version Only)

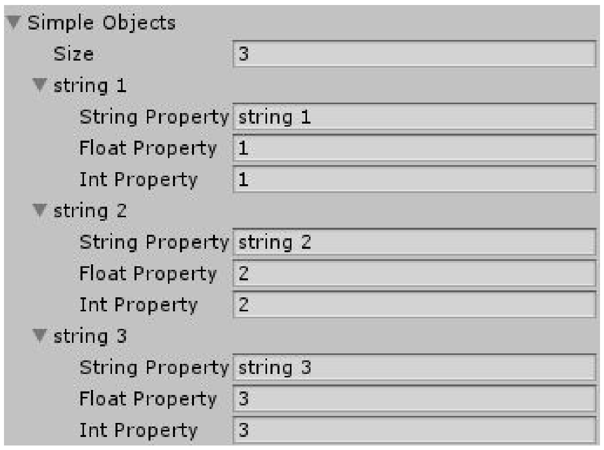
Draw all visible properties

Just use the *[Table]* attribute on a collection.

```
using EditorGUITable;

[System.Serializable]
public class SimpleObject
{
    public string stringProperty;
    public float floatProperty;
    public int intProperty;
}

public class TableAttributeExample : MonoBehaviour
{

    [Table]
    public List<SimpleObject>  simpleObjects;

}
```

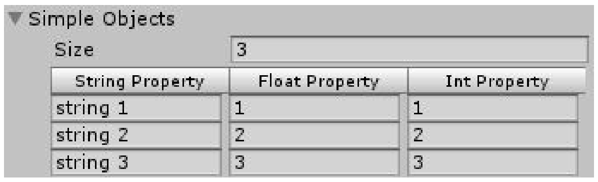
This will draw the collection in a table instead of the classic Unity display of collections.
This example uses a List, but any type of serializable collection can be used.

Without ***[Table]***



```
▼ Simple Objects
    Size              3
  ▼ string 1
      String Property string 1
      Float Property  1
      Int Property    1
  ▼ string 2
      String Property string 2
      Float Property  2
      Int Property    2
  ▼ string 3
      String Property string 3
      Float Property  3
      Int Property    3
```

With ***[Table]***



▼ Simple Objects

Size 3

String Property	Float Property	Int Property
string 1	1	1
string 2	2	2
string 3	3	3

Note all columns can be resized and sorted (assuming the values are sortable).

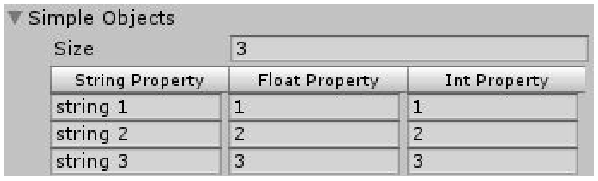
Draw selected properties

To display only some specific properties in the table, add parameters to the *Table* attribute.

```
[Table ("stringProperty", "intProperty")]
public List<SimpleObject>  simpleObjects;
```

[Table]

[Table ("prop1", "prop2")]



▼ Simple Objects

Size 3

String Property	Float Property	Int Property
string 1	1	1
string 2	2	2
string 3	3	3

Using in a custom editor / window

To display a table in a custom editor, call one of the DrawTable functions in GUITableLayout.

The table state

Note all the following functions use a GUITableState object as state parameter. This is the same functioning as other Unity GUI functions, like GUILayout.BeginScrollView for example. The calling code is responsible for holding, sending and receiving this state object so the table can be edited by the user.

Here is the basic usage:

```
using EditorGUITable;

[CustomEditor(typeof(SimpleExample))]
public class SimpleExampleEditor : Editor
{
  GUITableState tableState;

  public override void OnInspectorGUI ()
  {
    tableState = GUITableLayout.DrawTable (
      tableState,
      serializedObject.FindProperty("simpleObjects"));
  }
}
```

You might want the table state to be saved along sessions. To do this, build a table state with a key parameter. This will result in the table state being saved in EditorPrefs using this key. So the table will still look the same after reopening Unity.

Add:

```
void OnEnable ()
{
  tableState = new GUITableState("tableState");
}
```

Note: Be careful not to use the same key for different tables. This will result in unexpected behaviour.

Simply draw the table for the collection

Similar to the [Table] attribute with no parameters, this will display all visible properties of the collection elements in the table.

```
GUITableLayout.DrawTable (tableState, serializedObject.FindProperty("simpleObjects"));
```

Choose which properties to draw (Full Version Only)

Similar to the [Table (*properties*)] attribute, this will only display the chosen *properties* in the table.

```
GUITableLayout.DrawTable (
   tableState,
   serializedObject.FindProperty("simpleObjects"),
   new List<string>(){"floatProperty", "objectProperty"});
```

Customize the columns (Full Version Only)

Draw a table by defining the columns's settings and the path of the corresponding properties. This will automatically create Property Cells using these paths. Various column options are available. They are used in a similar way as GUILayoutOption.

```
List<SelectorColumn> propertyColumns = new List<SelectorColumn>()
{
   new SelectFromPropertyNameColumn("stringProperty",  "My String",  TableColumn.Width(60f)),
   new SelectFromPropertyNameColumn("floatProperty",    "My Float",    TableColumn.Width(50f), TableColumn.Optional()),
   new SelectFromPropertyNameColumn("objectProperty",  "My Object", TableColumn.Width(50f), TableColumn.Optional())
};

tableState = GUITableLayout.DrawTable (
   tableState,
   serializedObject.FindProperty("simpleObjects"),
   propertyColumns);
```

Create table cells from a function (Full Version Only)

Draw a table from the columns' settings, and a selector function that takes the corresponding row's element's SerializedProperty and returns the TableCell to put in the corresponding cell.

```
List<SelectorColumn> selectorColumns = new List<SelectorColumn>()
{
  new SelectFromFunctionColumn(
    prop => new LabelCell(prop.FindPropertyRelative("stringProperty") .stringValue),
    "My String Column",
    TableColumn.Width(60f)),
  new SelectFromFunctionColumn(
    prop => new LabelCell(prop.FindPropertyRelative("floatProperty") .floatValue.ToString()),
    "My Float Column",
    TableColumn.Width(50f), TableColumn.Optional(true)),
  new SelectFromFunctionColumn(
    prop => new LabelCell(prop.FindPropertyRelative("objectProperty") .objectReferenceValue.name),
    "My Object Column",
    TableColumn.Width(110f), TableColumn.EnabledTitle(false)),
};

tableState = GUITableLayout.DrawTable (
  tableState,
  serializedObject.FindProperty("simpleObjects"),
  selectorColumns);
```

Create each cells individually (Full Version Only)

Draw a table completely manually. Each cell has to be created and given as parameter in the *cells* parameter.

```
List<TableColumn> columns = new List<TableColumn>()
        {
            new TableColumn("String", 60f),
            new TableColumn("Float", 50f),
            new TableColumn("Object", 110f),
            new TableColumn("", TableColumn.Width(100f), TableColumn.EnabledTitle(false)),
        };

        List<List<TableCell>> rows = new List<List<TableCell>>();

        SimpleExample targetObject = (SimpleExample) serializedObject.targetObject;

        for (int i = 0 ; i < targetObject.simpleObjects.Count ; i++)
        {
            SimpleExample.SimpleObject entry = targetObject.simpleObjects[i];
            rows.Add (new List<TableCell>()
            {
                new LabelCell (entry.stringProperty),
                new PropertyCell (serializedObject, string.Format("simpleObjects.Array.data[{0}].floatProperty", i)),
                new PropertyCell (serializedObject, string.Format("simpleObjects.Array.data[{0}].objectProperty", i)),
                new ActionCell ("Reset", () => entry.Reset ()),
            });
        }

        tableState = GUITableLayout.DrawTable (tableState, columns, rows);
```

Note: For this function, it is not necessary to pass the collection’s serialized object to the function, unless you are using the Reorderable option.

The table cells

Various cell types are included in the package.

- *PropertyCell*: Displays the property using the very generic PropertyField function, which displays a property in a shape appropriate the data it contains.

- *ActionCell*: Displays a button which, when clicked, will trigger the callback given as parameter.

- *LabelCell*: Displays a string as a readonly label.

Create custom cell classes (Full Version Only)

In most cases the included types will be enough, but in some cases you want to display something in a specific way. In this case you can create a new class that inherits TableCell.

The *DrawCell* method needs to be overriden to draw the cell using GUI functions. Use this to customize the table look however needed.

CompareTo and *comparingValue* are used for sorting.
CompareTo is used to sort 2 entries of this type.
comparingValue is used as a fallback for sorting any types of entries, even of different types, using a string representing the data.

The table options (Full Version Only)

You can add options to the DrawTable function to customize the table.

Here are the current options:

- *AllowScrollView* (default value: *true*)
  Allows the tables to use internal scroll views to fit the containing view.
- *RowHeight* (default value: *EditorGUIUtility.singleLineHeight*)
  Defines the height of the rows in the table.
- *Reorderable* (default value: *false*)
  Enable reordering of the entries with Drag-and-Drop handles.
- *Filter* (default value: *null*)
  Defines a predicate that filters the entries to draw in the table.

Documentation

Detailed documentation available here.