Like these tutorials? Want More?
Become a patron!

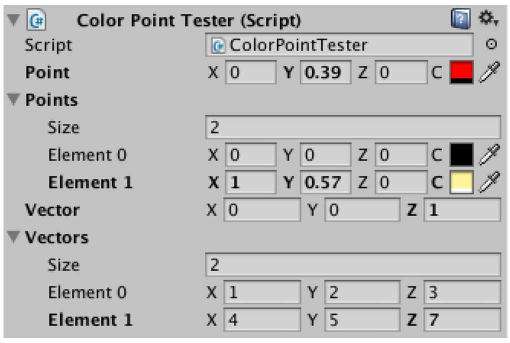
Custom Data, an introduction to serialized classes and property drawers

In this Unity C# tutorial you will create a simple data structure and write your own property drawer for it. You will learn to

- use a serialized class
- create a custom property drawer
- use `SerializedProperty`
- use Unity's immidiate-mode GUI in the editor

You're assumed to know your way around Unity's editor and know the basics of Unity C# scripting. If you've completed some of the other tutorials then you're good to go.

This tutorial is for Unity version 4.3 and above.



```
Color Point Tester (Script)
Script          ColorPointTester
Point           X 0   Y 0.39  Z 0   C
Points
  Size          2
  Element 0     X 0   Y 0     Z 0   C
  Element 1     X 1   Y 0.57  Z 0   C
Vector          X 0   Y 0     Z 1
Vectors
  Size          2
  Element 0     X 1   Y 2     Z 3
  Element 1     X 4   Y 5     Z 7
```

Compact color points.

Colored Points

Unity has a variety of data types and you can make all kinds of custom components to contain data. But sometimes you need a small custom set of data that's used in multiple places. Instead of writing the same code repeatedly, we can use a simple class to encapsulate this data and reuse that as if it were a built-in data type.

We will create a colored point, a data structure that contains both a color and a position.

We start by creating a new empty project and adding a new C# script named *ColorPoint* with the required variables.

What about classes and performance?

```
using UnityEngine;

public class ColorPoint {

	public Color color;
	public Vector3 position;
}
```

Then we create a new script named *ColorPointTester* for the sole purpose of testing our new data type. We give it a single point and an array of points, as well as a single vector and an array of vectors for comparison. Then we create an empty game object and add the component to it.

Will it work with lists as well?

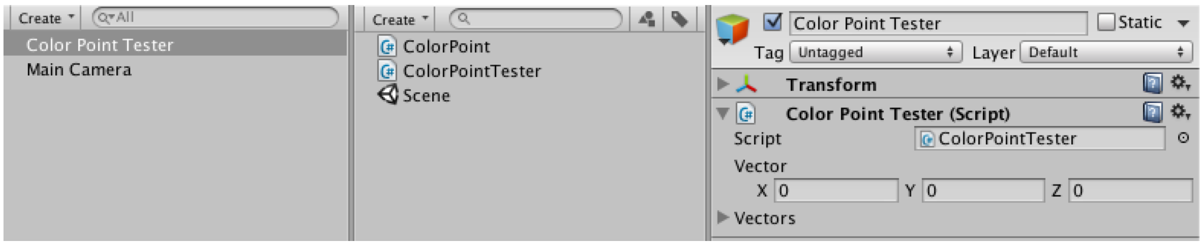
```
using UnityEngine;

public class ColorPointTester : MonoBehaviour {

	public ColorPoint point;

	public ColorPoint[] points;
	public Vector3 vector;

	public Vector3[] vectors;
}
```



```
Color Point Tester          Main Camera
ColorPoint    ColorPointTester    Scene

Color Point Tester     Static
Tag Untagged    Layer Default
Transform
Color Point Tester (Script)
Script    ColorPointTester
Vector
  X 0    Y 0    Z 0
Vectors
```

Color point and empty tester.

Our new type fails to show in the inspector, because its contents currently cannot be saved. We can solve this problem by adding the `System.Serializable` attribute to our class. By doing this it becomes possible to serialize all public fields of the class to a data stream, which allows it to be stored.

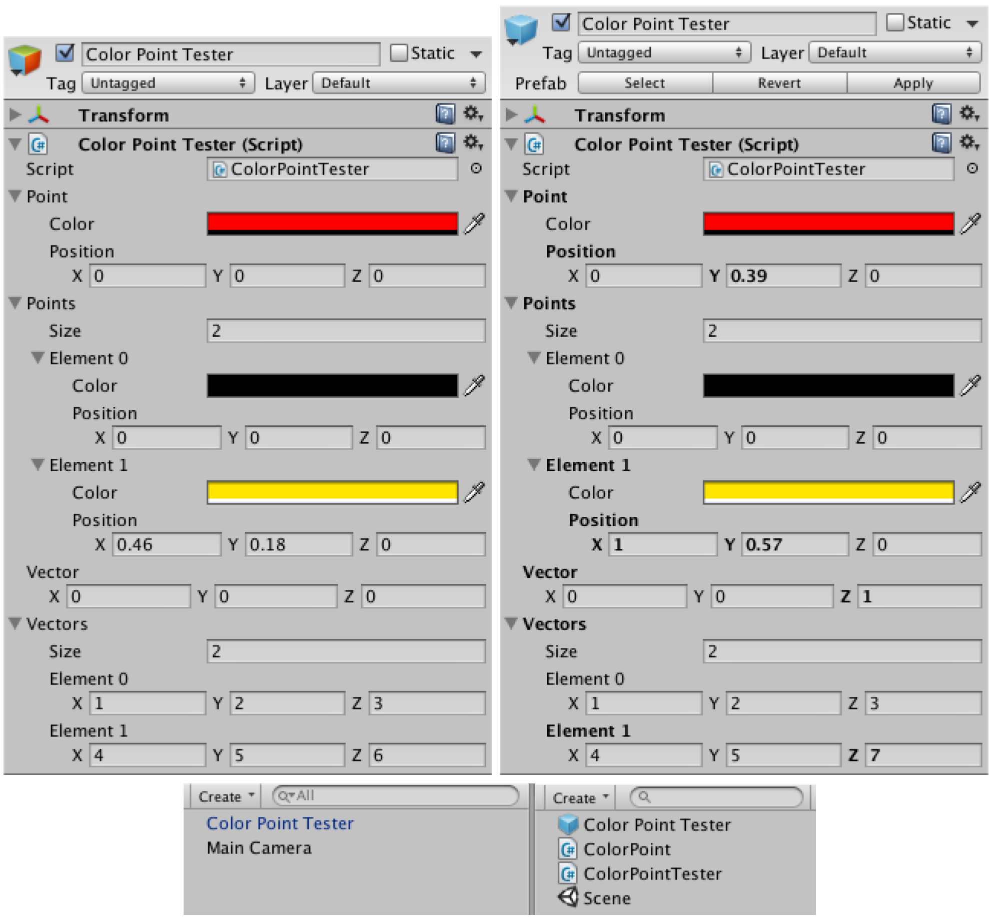
How does serialization work?

```
using UnityEngine;
using System;

[Serializable]
public class ColorPoint {

	public Color color;
	public Vector3 position;
}
```

Our data now shows up in the inspector, we can edit it as we please, and it can be saved. Also, turn our test object into a prefab by dragging it into the project view, then change some values of the instance in the scene. This will demonstrate that the type works correctly with prefabs.



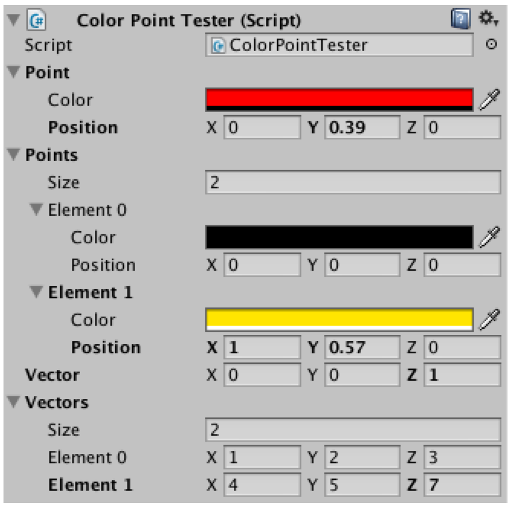
```
Color Point Tester     Static
Tag Untagged    Layer Default
Transform
Color Point Tester (Script)
Script    ColorPointTester
Point
  Color    🟥
  Position
    X 0    Y 0    Z 0
Points
  Size    2
  Element 0
    Color    ⬛
    Position
      X 0    Y 0    Z 0
  Element 1
    Color    🟨
    Position
      X 0.46    Y 0.18    Z 0
Vector
  X 0    Y 0    Z 0
Vectors
  Size    2
  Element 0
    X 1    Y 2    Z 3
  Element 1
    X 4    Y 5    Z 6
```

```
Color Point Tester     Static
Tag Untagged    Layer Default
Prefab    Select    Revert    Apply
Transform
Color Point Tester (Script)
Script    ColorPointTester
Point
  Color    🟥
  Position
    X 0    Y 0.39    Z 0
Points
  Size    2
  Element 0
    Color    ⬛
    Position
      X 0    Y 0    Z 0
  Element 1
    Color    🟨
    Position
      X 1    Y 0.57    Z 0
Vector
  X 0    Y 0    Z 1
Vectors
  Size    2
  Element 0
    X 1    Y 2    Z 3
  Element 1
    X 4    Y 5    Z 7
```

```
Color Point Tester          Color Point Tester
Main Camera                 ColorPoint
                            ColorPointTester
                            Scene
```

Normal object and a tweaked prefab instance.

The inspector looks quite messy. This can be amended somewhat by making it a bit wider by dragging, as the vectors will collapse to a single line once it's considered wide enough.



```
Color Point Tester (Script)
Script          ColorPointTester
Point
  Color
  Position      X 0    Y 0.39   Z 0
Points
  Size          2
  Element 0
    Color
    Position    X 0    Y 0      Z 0
  Element 1
    Color
    Position    X 1    Y 0.57   Z 0
Vector          X 0    Y 0      Z 1
Vectors
  Size          2
  Element 0     X 1    Y 2      Z 3
  Element 1     X 4    Y 5      Z 7
```

A wider inspector.

Drawing the Property

Unfortunately, even with a wider inspector our color point still requires multiple lines. It would be better if it needed less space.

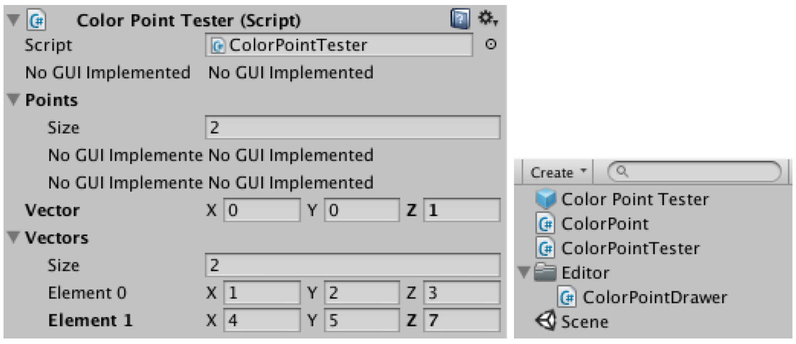
Fortunately, we can replace Unity's default way of drawing properties in the editor with our own variant. This is done by creating a class that extends from `UnityEditor.PropertyDrawer` and using the `UnityEditor.CustomPropertyDrawer` attribute to associate it with the type we want it to do the drawing for. We will name this class *ColorPointDrawer* and because this is an editor class we place it in a new folder named *Editor*.

What does `typeof` do?

Is the Editor folder required?

```
using UnityEditor;
using UnityEngine;

[CustomPropertyDrawer(typeof(ColorPoint))]
public class ColorPointDrawer : PropertyDrawer {
}
```



```
Color Point Tester (Script)
Script                ColorPointTester
No GUI Implemented    No GUI Implemented
Points
  Size                2
  No GUI Implemente   No GUI Implemented
  No GUI Implemente   No GUI Implemented
Vector                X 0    Y 0    Z 1
Vectors
  Size                2
  Element 0           X 1    Y 2    Z 3
  Element 1           X 4    Y 5    Z 7
```

```
Create
Color Point Tester
ColorPoint
ColorPointTester
Editor
  ColorPointDrawer
Scene
```

Property drawer that does nothing.

Now the inspector doesn't show anything useful anymore, but we'll change that by overriding the default `OnGUI` method of `PropertyDrawer` with our own version.

This `OnGUI` method has three parameters. First is a `Rect` that tells us what area of the window we should use to draw our property. Second is the property itself, represented by a `SerializedProperty`. Third is a `GUIContent` that defines the label we should use for the property.

Let's start by drawing just the label using the `GUIEditor.PrefixLabel` method followed by the positions using the `GUIEditor.PropertyField` method.

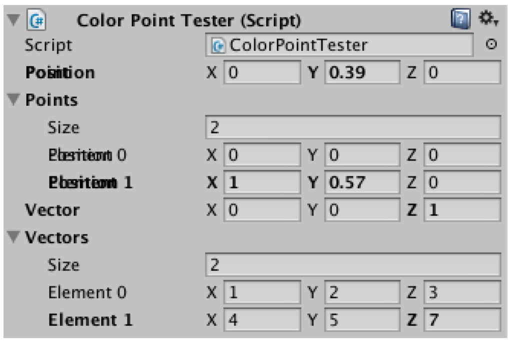
What does `override` do?

Isn't *position* a misnomer?

How does `SerializedProperty` work?

What's a `GUIContent`?

```
public override void OnGUI (Rect position, SerializedProperty property, GUIContent label) {
        EditorGUI.PrefixLabel(position, label);
        EditorGUI.PropertyField(position, property.FindPropertyRelative("position"));
}
```

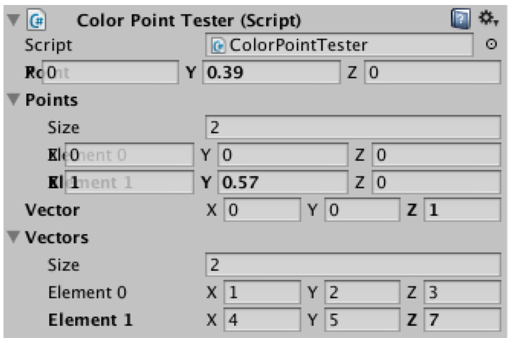


```
Color Point Tester (Script)
Script      ColorPointTester
Position    X 0      Y 0.39   Z 0
Points
  Size      2
  Element 0 X 0      Y 0      Z 0
  Element 1 X 1      Y 0.57   Z 0
Vector      X 0      Y 0      Z 1
Vectors
  Size      2
  Element 0 X 1      Y 2      Z 3
  Element 1 X 4      Y 5      Z 7
```

Property drawer with overlapping labels.

While we do get our position, its label now overlaps with the label of our color points. Let's get rid of the position's label by overriding it with `GUIContent.none`.

```
public override void OnGUI (Rect position, SerializedProperty property, GUIContent label) {
        EditorGUI.PrefixLabel(position, label);
        EditorGUI.PropertyField(position, property.FindPropertyRelative("position"), GUIContent.none);
}
```

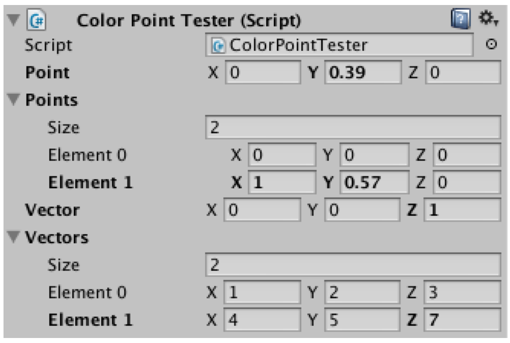


```
Color Point Tester (Script)
Script      ColorPointTester
Point  X 0        Y 0.39        Z 0
Points
  Size      2
  Element 0  X 0      Y 0        Z 0
  Element 1  X 1      Y 0.57     Z 0
Vector      X 0      Y 0      Z 1
Vectors
  Size      2
  Element 0 X 1      Y 2      Z 3
  Element 1 X 4      Y 5      Z 7
```

One label, still overlapping.

The vector is still overlapping the label, because we are using the exact same position rectangle for it. Fortunately, the `PrefixLabel` method returns an adjusted rectangle for the space to the right of it. So we will use this rect instead.

```
public override void OnGUI (Rect position, SerializedProperty property, GUIContent label) {
        Rect contentPosition = EditorGUI.PrefixLabel(position, label);
        EditorGUI.PropertyField(contentPosition, property.FindPropertyRelative("position"), GUIContent.none);
}
```



```
Color Point Tester (Script)
Script      ColorPointTester
Point       X 0      Y 0.39   Z 0
Points
  Size      2
  Element 0    X 0      Y 0      Z 0
  Element 1    X 1      Y 0.57   Z 0
Vector      X 0      Y 0      Z 1
Vectors
  Size      2
  Element 0 X 1      Y 2      Z 3
  Element 1 X 4      Y 5      Z 7
```

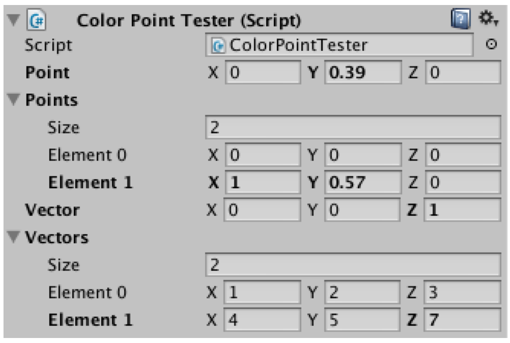
No longer overlapping, though incorrectly positioned.

This looks a lot better, but the position vector of the array elements is placed too far to the right. That happens because the `PropertyField` method adjusts for the current editor indent level. While this is usually convenient, we don't want any automatic adjustments in this case.

The intent level is set via the static int `EditorGUI.indentLevel`. To temporarily eliminate automatic indenting, we simply set it to zero.

Do we need to restore `indentLevel`?

```
	public override void OnGUI (Rect position, SerializedProperty property, GUIContent label) {
		Rect contentPosition = EditorGUI.PrefixLabel(position, label);
		EditorGUI.indentLevel = 0;
		EditorGUI.PropertyField(contentPosition, property.FindPropertyRelative("position"), GUIContent.none);
	}
```



Color Point Tester (Script)			
Script	ColorPointTester		
Point	X 0	Y 0.39	Z 0
Points			
Size	2		
Element 0	X 0	Y 0	Z 0
Element 1	X 1	Y 0.57	Z 0
Vector	X 0	Y 0	Z 1
Vectors			
Size	2		
Element 0	X 1	Y 2	Z 3
Element 1	X 4	Y 5	Z 7

Position correctly placed.

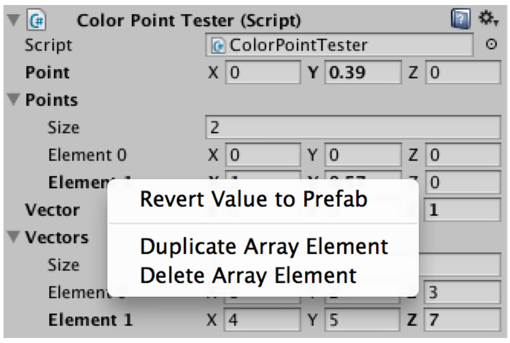
Fixing the Prefix

While our prefix label turns bold to signal that it is a modified prefab value, it doesn't allow any actions. So we cannot revert our entire color point at once and neither can we easily delete or duplicate array elements of it.

We need to tell the editor where our property starts and where it ends, because right now we are only showing part of its contents. We can use the `EditorGUI.BeginProperty` method to construct a new label and signal the start of a property, and use the `EditorGUI.EndProperty` method to signal when we are done. This will give us a label that provides the expected functionality via its context menu.

Prefab revert doesn't work with undo?

```
	public override void OnGUI (Rect position, SerializedProperty property, GUIContent label) {
		label = EditorGUI.BeginProperty(position, label, property);
		Rect contentPosition = EditorGUI.PrefixLabel(position, label);
		EditorGUI.indentLevel = 0;
		EditorGUI.PropertyField(contentPosition, property.FindPropertyRelative("position"), GUIContent.none);
		EditorGUI.EndProperty();
	}
```



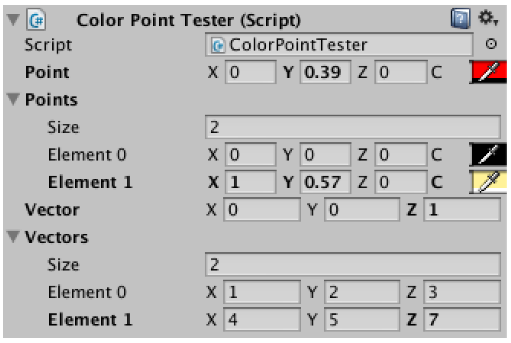
Color Point Tester (Script)			
Script	ColorPointTester		
Point	X 0	Y 0.39	Z 0
Points			
Size	2		
Element 0	X 0	Y 0	Z 0
Elemen			0
Vector			1
Vectors			
Size			
Element			3
Element 1	X 4	Y 5	Z 7

Revert and array functionality support.

Adding the Color

Now it's time to also draw the color property. To make it fit on the same line, we'll have to reduce the space that the vector uses. As the vector has three parts and the color is the fourth part, we'll give the vector the fist 75% of the horizontal space and place the color in the remaining 25%. We also use a single letter for the color label.

```
	public override void OnGUI (Rect position, SerializedProperty property, GUIContent label) {
		label = EditorGUI.BeginProperty(position, label, property);
		Rect contentPosition = EditorGUI.PrefixLabel(position, label);
		contentPosition.width *= 0.75f;
		EditorGUI.indentLevel = 0;
		EditorGUI.PropertyField(contentPosition, property.FindPropertyRelative("position"), GUIContent.none);
		contentPosition.x += contentPosition.width;
		contentPosition.width /= 3f;
		EditorGUI.PropertyField(contentPosition, property.FindPropertyRelative("color"), new GUIContent("C"));
		EditorGUI.EndProperty();
	}
```

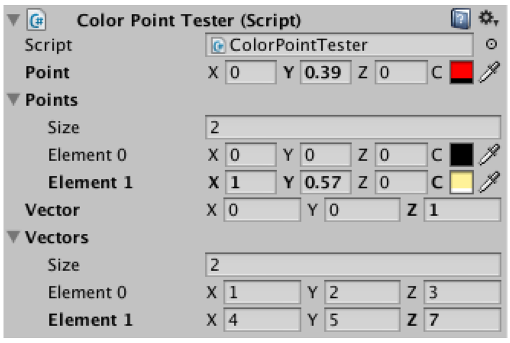


Color Point Tester (Script)
Script ColorPointTester
Point X 0 Y 0.39 Z 0 C
Points
Size 2
Element 0 X 0 Y 0 Z 0 C
Element 1 X 1 Y 0.57 Z 0 C
Vector X 0 Y 0 Z 1
Vectors
Size 2
Element 0 X 1 Y 2 Z 3
Element 1 X 4 Y 5 Z 7

With color, but wrong.

Even though we're using a short label, it claims too much space and pushes the color data to the right. This is because the label width is fixed, regardless of its contents. You can tweak the label width by adjusting `EditorGUIUtility.labelWidth`. Using a width of 14 pixels works fine.

```
	public override void OnGUI (Rect position, SerializedProperty property, GUIContent label) {
		label = EditorGUI.BeginProperty(position, label, property);
		Rect contentPosition = EditorGUI.PrefixLabel(position, label);
		contentPosition.width *= 0.75f;
		EditorGUI.indentLevel = 0;
		EditorGUI.PropertyField(contentPosition, property.FindPropertyRelative("position"), GUIContent.none);
		contentPosition.x += contentPosition.width;
		contentPosition.width /= 3f;
		EditorGUIUtility.labelWidth = 14f;
		EditorGUI.PropertyField(contentPosition, property.FindPropertyRelative("color"), new GUIContent("C"));
		EditorGUI.EndProperty();
	}
```



Color Point Tester (Script)
Script ColorPointTester
Point X 0 Y 0.39 Z 0 C
Points
Size 2
Element 0 X 0 Y 0 Z 0 C
Element 1 X 1 Y 0.57 Z 0 C
Vector X 0 Y 0 Z 1
Vectors
Size 2
Element 0 X 1 Y 2 Z 3
Element 1 X 4 Y 5 Z 7

Correctly sized color label.

Claiming an Extra Line

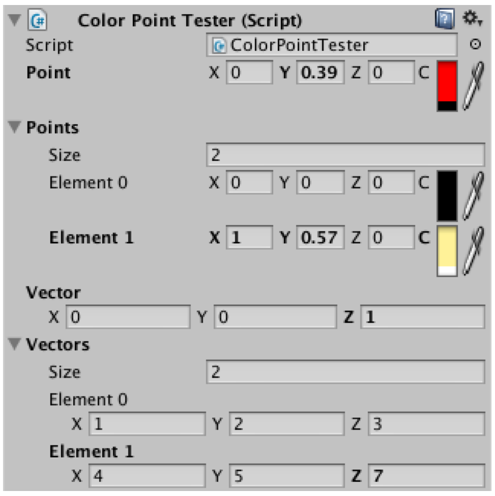
The default vectors switch between a single line and a double line depending on the width of the inspector. We can do so as well.

We have to override the `GetPropertyHeight` method to specify how much vertical space we need. The default for one line is 16 pixels. Adding a second line requires 18 additional pixels, 16 for the second line plus a margin of 2 between then.

It turns out that `Screen.width` contains the width of our inspector panel when we need it, so we can use that. Vectors switch to multiple lines when this width drops below 333, so we will do this as well.

Why a separate `GetPropertyHeight` method?

```
public override float GetPropertyHeight (SerializedProperty property, GUIContent label) {
        return Screen.width < 333 ? (16f + 18f) : 16f;
}
```



Color Point Tester (Script)
Script ColorPointTester
Point X 0 Y 0.39 Z 0 C
Points
Size 2
Element 0 X 0 Y 0 Z 0 C
Element 1 X 1 Y 0.57 Z 0 C
Vector
X 0 Y 0 Z 1
Vectors
Size 2
Element 0
X 1 Y 2 Z 3
Element 1
X 4 Y 5 Z 7

Claiming more space.

Now we claim more vertical space when we make the inspector narrow enough. However, we're not yet using it. To do so we have to take care of four things.

First, we can detect that we're using two lines by checking the height of our position rect. Second, we need to set the height back to 16 so the color property stays on one line. Third, we have to move down a line after we've drawn the property label. Fourth, we have to increase the indent level by one and apply it to our position, using the `EditorGUI.IndentedRect` method.

```
public override void OnGUI (Rect position, SerializedProperty property, GUIContent label) {
        label = EditorGUI.BeginProperty(position, label, property);
        Rect contentPosition = EditorGUI.PrefixLabel(position, label);
        if (position.height > 16f) {
                position.height = 16f;
                EditorGUI.indentLevel += 1;
                contentPosition = EditorGUI.IndentedRect(position);
                contentPosition.y += 18f;
        }
        contentPosition.width *= 0.75f;
        EditorGUI.indentLevel = 0;
        EditorGUI.PropertyField(contentPosition, property.FindPropertyRelative("position"), GUIContent.none);
        contentPosition.x += contentPosition.width;
        contentPosition.width /= 3f;
        EditorGUIUtility.labelWidth = 14f;
        EditorGUI.PropertyField(contentPosition, property.FindPropertyRelative("color"), new GUIContent("C"));
        EditorGUI.EndProperty();
}
```

Using more space.

We now have a nice compact representation of our `ColorPoint`. It supports undo, redo, prefabs, and multi-object editing like normal. It uses only one line if the inspector is wide enough, otherwise it uses two.

The next editor tutorial is Custom List.

Downloads

custom-data.unitypackage
The finished project.

Questions & Answers

What about class performance?

There are trade-offs between using a class or a struct as containers for data. If your data is very small, short-lived, treated like an atomic value, or you iterate through many instances, usually you'd use a struct. Otherwise, a class is usually better.

Before Unity 4.5, if you want your data to show up in the inspector and be saved, you had no choice but to use a class.

Our `ColorPoint` data might look small, but it consists of seven floats, a total of 28 bytes. That's almost twice the maximum size recommended for structs.

So just use classes, unless the criteria for using a struct are met, you don't need to save or show them in the editor, you're using thousands of them, and you're running into performance issues.

Will it work with lists as well?

Yes. You can use either a `ColorPoint[]` or a `System.Collections.Generic.List<ColorPoint>`. The Unity editor treats both exactly the same.

Only bother using a list when you're modifying its element structure a lot – especially its size – while in play mode. If you don't, then there's no reason to introduce the slight overhead of using a list.

How does serialization work?

Serialization is the process of converting a collection of data in memory into a stream of data that can be stored in a persistent state or transmitted over a network. It's what Unity does when it saves your scene and asset data. Deserialization is its complement, constructing data in memory from a stream.

This functionality is part of .NET and you can read more about it on MSDN, though it's not required.

What does "typeof" do?

The `typeof` operator is used to get the type object of something, usually a class. You cannot use it with variables, only with explicit type names.

Why not just write down the class name? Because that results in a compiler error! The extra step is needed because you're converting a type into a variable.

Is the Editor folder required?

Unity splits projects into multiple parts that are compiled in a specific order. The Editor folder is used to separate everything that's about the editor from everything that's not. It's not included in game builds, and code outside of it cannot access it. For example, while `ColorPointDrawer` knows about `ColorPoint`, the reverse is not true.

Technically, it is possible to place `ColorPointDrawer` outside of the Editor folder and everything will still work. However, it's best not to mix editor and non-editor code. Also, some editor specific stuff won't work if not placed in an Editor folder.

What does `override` do?

Overriding is a mechanic which you use to replace a method from the class you are extending. You can only `override` methods that have been marked as `virtual`. In this case, it allows Unity to call any property drawer's `OnGUI` method without having to know it's exact type.

So why don't Unity event methods like `Start` and `Update` require the `override` keyword? Because Unity uses a different approach to make such event methods from outside the .NET framework.

Isn't *position* a misnomer?

It's not just a position, it has a width and height as well. It defines the bounds to use for drawing the property. So why is it named *position* and not *bounds* or just *rect*?

The answer is consistency. All Unity code and examples that use such a rect name it *position*.

How does `SerializedProperty` work?

A `SerializedObject` object acts as a wrapper or proxy for Unity objects that can be edited. You can use it to extract data from an object, even if you don't have a clue what's inside it. This is how the Unity inspector can show default inspectors for anything you create yourself.

The fields of these objects are in turn represented by `SerializedProperty` objects. You can drill down the data hierarchy through use of the `FindPropertyRelative` method.

Using these objects makes it possible to support undo, redo, prefabs, and multi-object editing without much effort.

What's a `GUIContent`?

`GUIContent` is a wrapper object for text, textures, and tooltips that you typically use as labels.

Do we need to restore `indentLevel`?

Because `EditorGUI.indentLevel` lives outside the scope of the method, changes to it will persist after the method invocation is finished. However, Unity takes care of setting it back to the old value for us afterwards.

Prefab revert doesn't work with undo?

Unity version 4.3 and above have some editor bugs, and this is one of them. Some bugs have been partially or completely fixed in updates, but others still remain.

Why a separate `GetPropertyHeight` method?

The method is separate so that the height of a drawer can be queried before it is drawn. For example, it can be used to determine whether a scrollbar is needed because the content won't fit in the available space.

About, Contact, Tutorials

© Catlike Coding

Twitter, Facebook, Google+