Anatomy of a View

Introduction

This tutorial goes through all you need to know to create basic to advanced views. When you've gone through it you'll be able to utilize the MarkUX API to its fullest.

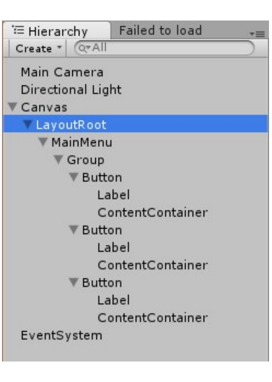
Visual Hierarchy

Before we break down how views are constructed I'd like to mention briefly how the system represents the views once they've been imported and processed. When the views are processed they are transformed into view objects (unity GameObjects) and put into a visual hierarchy that resides under *LayoutRoot* in the scene Canvas. The following XML:

MainMenu.xml

```
<MainMenu>
    <Group Spacing="10px">
        <Button Text="Play" />
        <Button Text="Options" />
        <Button Text="Quit" />
    </Group>
</MainMenu>
```

Results in the following visual hierarchy:



The view-models are added as components to the view-objects.

View-Model Declaration

A view-model is declared as a class that inherits from the **View** base class or any of its subclasses. The name of the class is the view name you use to refer to the view in XML.

Label.cs

```
[InternalView]
[RemoveComponent(typeof(UnityEngine.UI.Image))]
[AddComponent(typeof(Text))]
public class Label : View
```

As shown above, when declaring the view-model class you have the option to associate some information with it using the following class attributes:

Attribute	Description
InternalView	Indicates that the view shouldn't be used as a main view and hides it from the view presenter's drop-down selection of views.
AddComponent(type)	Tells the view processor to add the component *type* to the view object when it is created.
RemoveComponent(type)	Tells the view processor to remove the component *type* from the view object when it is created.
CreatesView(type)	Indiciates that the view dynamically creates other views of the specified *type*. A prefab is created for *type*. The prefab is used to dynamically instanciate the view.

View Field Declaration

View fields are declared as class fields on the view-model. The name of the class field is the name you use to refer to the field in the view XML.

Group.cs

```
[ChangeHandler("UpdateLayouts")]
public Orientation Orientation;

[ChangeHandler("UpdateLayouts")]
public ElementSize Spacing;

[ChangeHandler("UpdateLayout")]
public Alignment ContentAlignment;
public bool ContentAlignmentSet;
```

Like with the view-model class declaration the field declaration has a number of attributes that can be used:

Attribute	Description
NotSetFromXml	Tells the view processor that this field isn't allowed to be set from XML.
ChangeHandler(name)	Sets a view field change handler. The parameter *name* is the name of the change handler method residing in the view-model.
ValueConverter	Used to override the default value converter used for the field.

To understand how to effectively use these attributes there are two concepts you need to be aware of when dealing with view fields: **Value Converters** and **Change Handlers**. We'll go over Change Handlers first as it's something you'll want to utilize in most views.

Change Handlers

Change handlers are methods within the view-model that are invoked when fields are changed. The standard views utilizes three change handlers called *UpdateLayout*, *UpdateLayouts* and *UpdateBehavior*.

View.cs

```
public virtual void UpdateLayout()
public virtual void UpdateLayouts()
public virtual void UpdateBehavior()
```

UpdateLayout	Invoked when any field that has to do with layout of the view is changed: Width, Height, Margin, Alignment, Offset and OffsetFromParent.
UpdateLayouts	Calls UpdateLayout as well as informs the parent views that the layout has been changed.
UpdateBehavior	Invoked when fields are changed that impact behavior/visual appearance (but not layout) of the view: Alpha, BackgroundColor, etc.

These methods can be overridden within your view-model in order to respond to field changes. By using the ChangeHandler attribute you can apply field change handlers to your custom fields. Here is an example on how to utilize the change handlers within your view-model:

CustomView.cs

```
[ChangeHandler("UpdateLayouts")]
public int MyInt;

public override UpdateLayout()
{
    // called when MyInt changes or layout fields such as Width

    // important that you call base.UpdateLayout if
    // you don't want to override the default layout behavior
    base.UpdateLayout();
}
```

Notice that we've applied the change handler UpdateLayouts (notice plural 's') but overridden UpdateLayout, this is because UpdateLayout is invoked by UpdateLayouts (and there will be no need to override it). We've been using the default change handlers but you can use any method you want as a change handler.

Setting Field Values

In order to utilize the change handlers and binding systems you need to set field values using a special generic method called **SetValue()**.

CustomView.cs

```
SetValue(() => MyInt, 7); // same as MyInt = 7

// same as ChildView.Orientation = Orientation.Horizontal;
SetValue(() => ChildView.Orientation, Orientation.Horizontal);
```

If you're changing internal values and don't want to trigger change handlers or utilize the binding system you can set the values as you'd normally do.

Value Converters

Value Converters are used by the system to convert between values of different types. Field values are often set in the view XML as string values:

Group.xml

```
<Group Spacing="10px" Orientation="Horizontal" ContentAlignment="Top">
```

In order to convert the string values to the view field value type the system utilizes **Value Converters**. Value Converters are classes that inherit from the base class **ValueConverter**.

ColorValueConverter.cs

```
public class ColorValueConverter : ValueConverter
{
    public ColorValueConverter()
    {
        _type = typeof(Color);
    }

    public override ConversionResult Convert(object value,
        ValueConverterContext context)
    {
        // conversion of value to Color happens here
    }
}
```

The above code creates a *ColorValueConverter* that is associated with the type *Color*. If the system encounters a field of type *Color* it will automatically use the ColorValueConverter for value conversions.

The MarkUX API provides value converters for the types used within the standard views: bool, int, float, ElementSize, Vector3, Sprite, Font, Margin, Orientation, Color, Alignment, etc. You'll only need to create custom value converters if you introduce a new view field type (the most common being enums).

View References

View fields may be references to views that reside within the view XML. To do this you simply need to add an Id to the view you want to reference within the XML:

CustomView.xml

```
<CustomView>
    <Group Id="MyGroup" Spacing="10px">
        <Button Id="PlayButton" Text="Play" />
        <Button Text="Options" />
        <Button Text="Quit" />
    </Group>
</CustomView>
```

Add a class field with the same name and type as the view you want to reference:

CustomView.cs

```
public class CustomView : View
{
    public Group MyGroup;
    public Button PlayButton;
}
```

You may also reference their view objects in the same way by changing the type to GameObject.

Set-fields

Set-fields are special boolean fields that are used to indicate if the value has been set (programatically or in XML). This is useful in cases where you want to have certain logic applied when a value hasn't been set by the user. The set-field is simply the field-name you want to track with the postfix *Set*:

Group.cs

```
[ChangeHandler("UpdateLayout")]
public Alignment ContentAlignment;
public bool ContentAlignmentSet;
```

The set-field ContentAlignmentSet will be True only if ContentAlignment is set.

View Action Declaration

View Actions are ways to bind operational logic between view-models. View actions are declared as class field of the type **ViewAction**. The name of the class field is the name you use to refer to the action in XML.

CustomView.cs

```
public ViewAction MyCustomAction;
```

You now have an action that can be referenced in XML:

AnotherView.xml

```
<AnotherView>
    <CustomView MyCustomAction="MyActionHandler" />
</AnotherView>
```

In order to trigger an action you simply call:

CustomView.cs

```
MyCustomAction.Trigger();
MyCustomAction.Trigger(eventData); // if you want to pass event data to the handlers
```

View Action Handlers

View action handlers are methods that will be invoked when an action is triggered. The action handler can have certain optional parameters that will be injected when the handler is invoked:

CustomView.cs

```
public void MyActionHandler(View parent, View source, BaseEventData eventData)
public void MyActionHandler(GameObject parent, GameObject source)
```

parent	The parent View or GameObject handling the action.
source	The source View or GameObject that triggered the View Action.
eventData	Any event data inheriting from the class BaseEventData.

A button click handler might look like this:

CustomView.cs

```
public void MyButtonClickHandler(PointerEventData eventData, Button source)
```

Event-System Actions

You can let the unity event-system trigger your actions by naming your actions one of the following:

BeginDrag
Cancel
Deselect
Drag
Drop
EndDrag
InitializePotentialDrag
Move
Click
PointerClick
MouseClick
PointerDown
MouseDown
PointerEnter
MouseEnter
PointerExit
MouseExit
PointerUp
MouseUp
Scroll
Select
Submit
UpdateSelected

E.g. to add a Click action to your view simply name the action *Click* and it will be automatically triggered when the user clicks on the view.

Content Views

If you want your view to allow content you simply need to inherit from the **ContentView** class.

MyCustomContentView.cs

```
public class MyCustomContentView : ContentView
```

In the view XML you can control where the content will be put by using the special <ContentContainer /> view.

MyCustomContentView.xml

```
<MyCustomContentView>
    <Region>
        <ContentContainer />
    </Region>
</MyCustomContentView>
```

Any content put in your view will be added as children to your content container.

Embedded XML

You might have noticed that none of the standard views comes with separate XML files. This is because the XML is embedded in the view-models. This might be useful if you want to distribute views and not have to provide separate files (we encourage you to share your views in any way you please). In order to embed your XML you simply need to override the method **GetEmbeddedXml()**:

Frame.cs

```
public override string GetEmbeddedXml()
{
    return @"<Frame ResizeToContent=""True"" ContentMargin=""0"">
              <ContentContainer Id=""Content"" Margin=""{ContentMargin}""
                                ResizeToContent=""{ResizeToContent}"" />
            </Frame>";
}
```

We've broken down the major building blocks and components of the view. This should be enough information to get you started building advanced views.

Comments, questions, suggestions? Discuss this tutorial at the MarkUX developer subreddit.