Getting Started

Introduction

MarkUX is a framework for designers and programmers who want more power to create rich user-experiences in Unity. Before we dive into building our first UIs I'd like to explain the main idea behind MarkUX.

MarkUX offers a language (**XML**) in which you can express UI design and its relationship with application logic:

MainMenu.xml

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

The above example shows a simple main menu **view** created in XML. Programmers that are comfortable with code can express designs in a structured language. Designers should be somewhat used to the idea if they've worked with HTML before.

As the designer works on the view the programmer can work on the view's logic in what is called the **view-model**:

MainMenu.cs

```
public class MainMenu : View
{
    public void StartGame()
    {
        // called when user clicks on "Play" button
    }
}
```

The view-model is optional but needed if the view is to have any kind of logic of its own. Together the view and its view-model forms a cohesive UI element (simply referred to as the view) that can be combined with other views:

AnotherView.xml

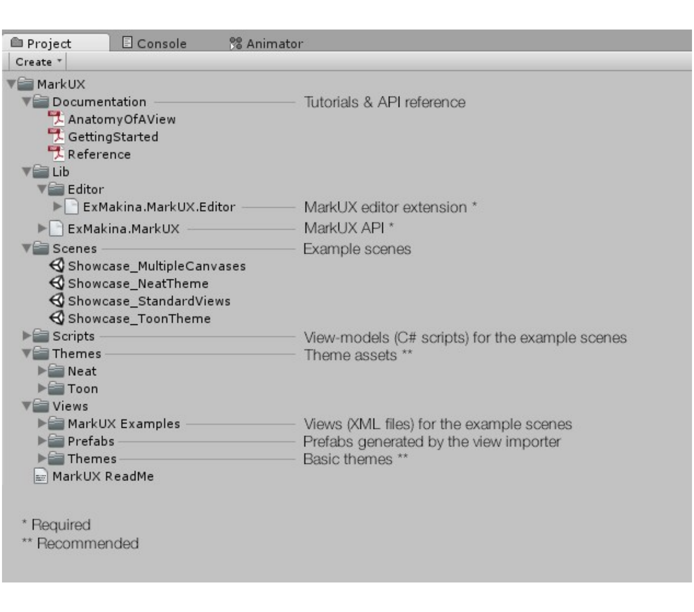
```
<AnotherView>
    <MainMenu Width="50%" Alignment="Left" />
    <MainMenu Width="50%" Alignment="Right" />
</AnotherView>
```

This pattern of separating application logic from UI design is called **MVVM** (model view view-model). The *model* part of the pattern is not relevant to this tutorial. Just know that the model refers to actual application data that is completely independent of the UI.

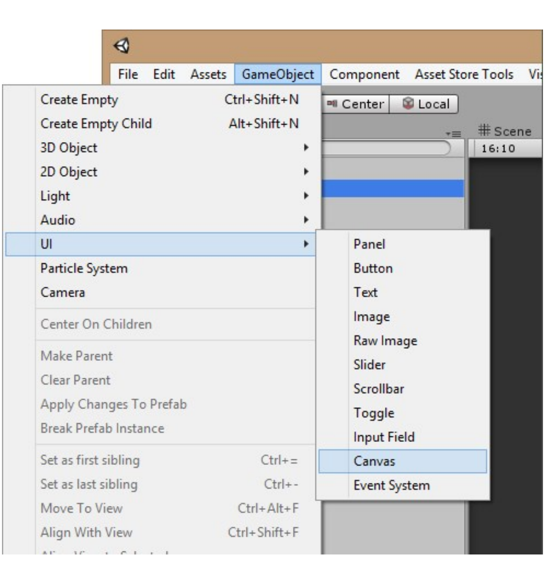
The views and view-models interacts using **data-binding** and **view actions**. I'll get into those later. Let's get started building our first UI with MarkUX from scratch.

Installation

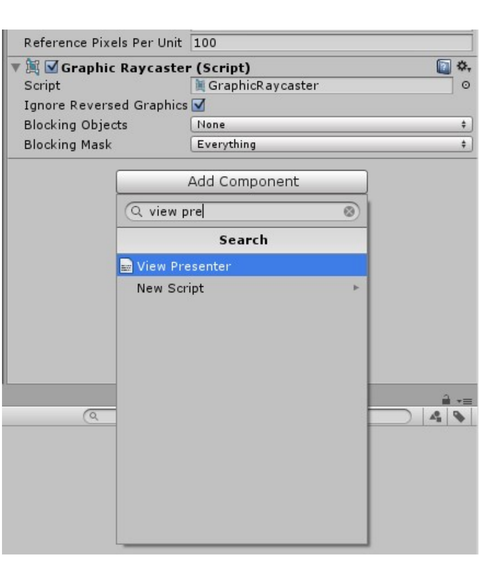
Start a new unity project and import the MarkUX package (downloadable from the asset store).



Make sure there is a standard UI **Canvas** in the scene:



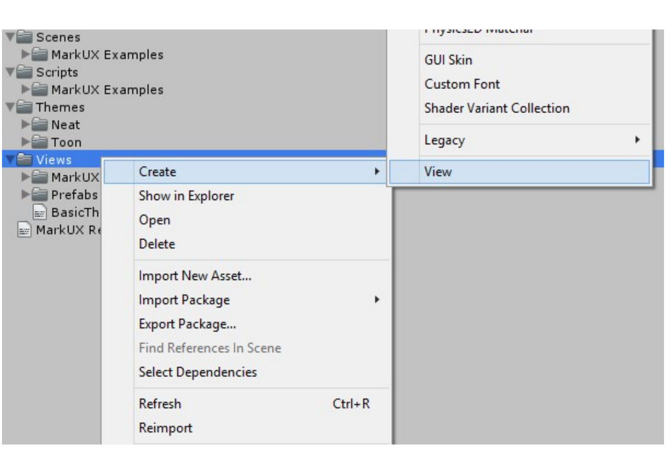
Add a **View Presenter** component to the canvas.



The view presenter will render views on the canvas.

Creating a View

Views are simply XML files located under the project's views-folder (*Assets/MarkUX/Views*). To create a new view, right-click on the folder and select *Create->View*.



The file *NewView.xml* should now be in the */Views* folder. Rename the file *MainMenu.xml*.

> Views are automatically imported and processed by the MarkUX editor extension when XML files under the */Views* folder gets added, deleted or changed. Views are also processed if you click on the button **Reload Views** on the View Presenter's inspector window.

Edit the view *MainMenu.xml* so that it contains the following:

MainMenu.xml

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

> When editing XML I recommend you use an editor that supports XML validation and syntax highlighting. It will save you a lot of time by catching common formatting mistakes.

The name of the root tag *<MainMenu>* is the name we've designated the view. The view contains three *Button* views that are arranged vertically by a *Group* view. The *Group* and *Button* are standard views that are included with the MarkUX API. The *Group* is a **Content View** which means it can have content that it manages in its own way.

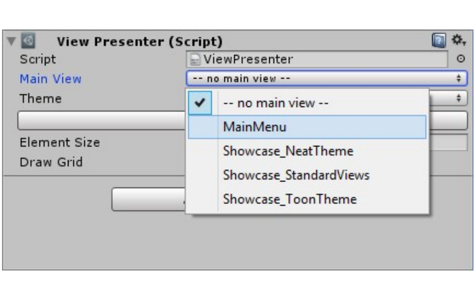
Spacing and *Text* are **View Fields** which are values that can be set on the views that changes their layout and behavior. All views have the following fields:

View Field	Description
Width	Width of the view. Can be specified in pixels, percentage or elements. Default value: "100%".
Height	Height of the view. Can be specified in pixels, percentage or elements. Default value: "100%".
Alignment	Alignment of the view: TopLeft, Top, TopRight, Left, Center, Right, BottomLeft, Bottom, BottomRight. Default value: "Center".
Margin	Specifies the view's margin from left, top, right and bottom. Default value: "0,0,0,0".
Offset	Specifies the view's offset from left, top, right and bottom. Default value: "0,0,0,0".
BackgroundColor	Background color overlay or tint of the view. Color values can be specified by name or hexcode (#aarrggbb or #rrggbb). Default value: (not set).
BackgroundImage	The background image of the view. The value is the path to the sprite asset. Default value: (not set)
BackgroundImageType	The image type of the background image: Simple, Sliced, Tiled or Filled. Default value: "Simple".

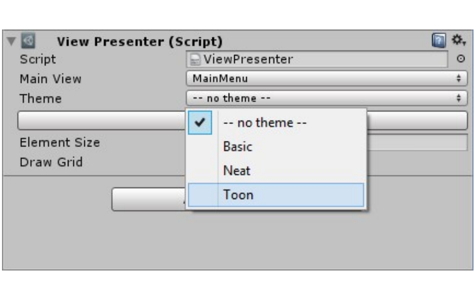
Views also has their own custom fields like the *Text* field on the button. For information on what fields a certain view has check the API docs. We'll also go over how to add custom fields to our view in the Data-Binding section.

Presenting the View

Go to the inspector window for the View Presenter component on the Canvas. Change the *Main View* value to be the **MainMenu** view we've just created.



The scene window should now show the view being rendered. In order to make the standard views (the buttons in this case) more presentable, pick a theme.



The *Toon* theme has the most visual flair while the *Basic* theme has the minimal amount of styling needed for the views.

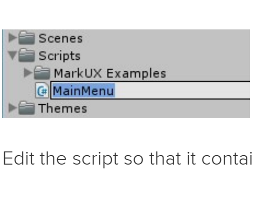
> Themes are XML files that modify the look & feel of views similar to style-sheets in HTML. More information on creating custom themes will be available in other tutorials.

The scene window should now show our view:

Next up is adding some logic to our view.

Creating a View-Model

In order to handle button clicks we first need to create a view-model for the view. To do this we create a new script anywhere in the project (e.g. in the */Scripts* folder). Name the script *MainMenu*.



Edit the script so that it contains the following:

MainMenu.cs

```
public class MainMenu : View
{
    public void StartGame()
    {
        Debug.Log("StartGame() called.");
    }

    public void Options()
    {
        Debug.Log("Options() called.");
    }

    public void Quit()
    {
        Debug.Log("Quit() called.");
    }
}
```

Make sure the class has the name *MainMenu* and inherits from the class *View* that resides in the MarkUX namespace. Note that all views must inherit from *View* (or a subclass of). We now have a view-model called *MainMenu* containing three methods. MarkUX will automatically connect the view-model to the view with the same name.

> The MarkUX API relies heavily on *naming conventions* to simplify the workflow by removing the need to add configurations and "plumbing" logic.

Next we'll look into adding some interaction logic to the view.

View Actions

In order to add the methods in the view-model as click handlers we need to add the following to the main menu view:

MainMenu.xml

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

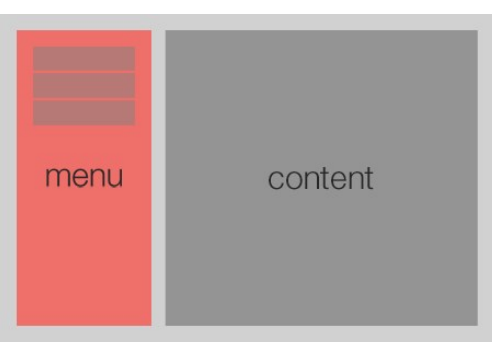
If you run the application and click on the Play button you should see the following in the console window:

View Actions are ways to bind operational logic between view-models. In this case Button exposes a View Action called *Click* that is triggered when the user clicks on the button. When an action is triggered all its handlers (also called view action entries) are invoked.

Before we continue adding logic to our view-model I want to go over some fundamentals on UI layout.

Layout

This is the kind of layout we are going for:



The MarkUX API includes various standard views that can be used to do layout. We've seen the *Group* that is used to group views vertically or horizontally with spacing between. A more basic layout view is the *Region* which is simply a blank content view used to spacially arrange views. Let's edit our view MainMenu to make us of regions to get the layout we want:

MainMenu.xml

```
<MainMenu>
    <Region Width="25%" Alignment="Left" Margin="30,30,15,30"
        BackgroundColor="#ef706a">
        <Group Spacing="10px">
            <Button Text="Play" Click="StartGame" />
            <Button Text="Options" Click="Options" />
            <Button Text="Quit" Click="Quit" />
        </Group>
    </Region>
    <Region Width="75%" Alignment="Right" Margin="15,30,30,30"
        BackgroundColor="#949494">
        <!-- content area -->
    </Region>
</MainMenu>
```

We've added background colors to the regions just to make it easier to see the space they cover. When you've seen the result, remove the background colors and we'll move on to adding a view switcher to our main menu.

View Switcher

The View Switcher is a content view that provides functionality for switching its content. We can use it to switch between menus. Let's add a view switcher to our view:

MainMenu.xml

```
<MainMenu>
    <Region Width="25%" Alignment="Left" Margin="30,30,15,30">
        <Group Spacing="10px" Alignment="Top">
            <Button Text="Play" Click="StartGame" />
            <Button Text="Options" Click="Options" />
            <Button Text="Quit" Click="Quit" />
        </Group>
    </Region>
    <Region Width="75%" Alignment="Right" Margin="15,30,30,30">
        <ViewSwitcher Id="ContentViewSwitcher">
            <Region />
            <Region BackgroundColor="Red" />
            <Region BackgroundColor="Blue" />
        </ViewSwitcher>
    </Region>
</MainMenu>
```

Note that we've given *ViewSwitcher* an Id which we need to be able to reference it in our view-model. Currently it contains three empty regions that it will switch between. It will show the first region by default.

Next update the view-model to make use of the view switcher:

MainMenu.cs

```
public class MainMenu : View
{
    public ViewSwitcher ContentViewSwitcher;

    public void StartGame()
    {
        ContentViewSwitcher.SwitchTo(1);
    }

    public void Options()
    {
        ContentViewSwitcher.SwitchTo(2);
    }

    public void Quit()
    {
        Application.Quit();
    }
}
```

We've added the field *ContentViewSwitcher* which will automatically be set to reference the view with the same *Id*. The ViewSwitcher contains the method *SwitchTo(int zeroBasedIndex)* that we use to switch between the views.

If you run the scene and click on "Play" and "Options" you'll see the view switcher in action. In order to demonstrate the data-binding capabilities we'll go on to implement a very simple options menu.

Data-Binding

For the options menu we'll create a separate view that we then will put inside the main menu view. This time we'll begin by creating the view-model. Create a new script called *Options* and edit it so it contains the following:

Options.cs

```
public class Options : View
{
    public float Volume;
    public bool EasyMode;
    public string PlayerName = "";
}
```

Our options view will contain three **View Fields** that we can reference in code and in the XML. Next we'll create a options view that contains various views that we will bind the fields we've just created to. Create a new view *Options.xml*.

Options.xml

```
<Options>
    <Group Spacing="10px" Alignment="TopLeft" ContentAlignment="Left">
      <Label Text="Volume: " />
      <Slider Value="{Volume}" Min="0" Max="100" Width="400px" />

      <Label Text="Easy mode: " />
      <Button Style="CheckBox" IsToggleButton="True" ToggleValue="{EasyMode}" />

      <Label Text="Player: " />
      <InputField Text="{PlayerName}" />
    </Group>
</Options>
```

Data-binding is a key feature in MarkUX and it enables our views to connect data between view-models. E.g. in this example we've bound the field *Value* of the slider view:

```
      <Slider Value="{Volume}" />
```

...to the field *Volume* on the options view. If we interact with the slider it will change its *Value* field and its value will automatically propagate to the Volume field. Likewise if we change the Volume field the value will propagate to the slider's Value field.

> Data-binding in XML is done using the **{FieldPath}** notation. Most of the time you'd bind directly to a field on the parent view-model. However, MarkUX does support more advanced field path bindings, including nested fields: *{field1.field2}*, array indexors: *{arrayField[index].field}*, ID-selectors *{viewID.field}* and format-strings *"Hello, {name}!"*.

Let's edit the contents of our view switcher in our main menu view to make use the options view.

MainMenu.xml

```
        ...
        <ViewSwitcher Id="ContentViewSwitcher">
            <Region />
            <Region BackgroundColor="Red" />
            <Options Volume="75" PlayerName="Player" EasyMode="True" />
        </ViewSwitcher>
        ...
```

Run the scene and click on the Options button to see the options view with default values being set.

The image above shows how you can inspect the view fields on the options view using the inspector window. Next we'll add some animations to create smoother transitions between the views.

Animations

The view switcher can work with animations to make smooth animated transitions between views. Animations are special views that animates view fields using specified parameters. For this example we will create a fade-in animation. Add the following to the main menu view:

MainMenu.xml

```
...
        <ViewSwitcher Id="ContentViewSwitcher" TransitionIn="FadeIn">
            <Region />
            <Region BackgroundColor="Red" />
            <Options Volume="75" PlayerName="Player" EasyMode="True" />
        </ViewSwitcher>
    </Region>

    <ViewAnimation Id="FadeIn">
        <Animate Field="Alpha" From="0" To="1" Duration="0.2s"
                 EasingFunction="QuadraticEaseIn" />
    </ViewAnimation>

</MainMenu>
```

We've created a view animation with the Id **FadeIn** and told the view switcher to apply the animation on views being transitioned to. If you run the scene you should see the views being faded in when clicking on "Options" and "Play".

This concludes the introductory guide. There are much more topics to delve into but this should be enough to get you started using MarkUX. I hope you've at least caught a glimpse of the capabilities of the framework. Good luck and have fun making awesome UIs.

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