Unity 3D UI Essentials

Unity is a powerful cross-platform development engine that provides rich framework to create 2D and 3D apps or games. Through Unity, users are able to master the complexities of game development with ease, including animations, physics, renderings, shaders, scripting and more. Unity has released a new and advanced system to cope with the UI demands for modern applications and games.

Unity 3D UI Essentials is a complete walk-through of the new UI system in Unity V4 and beyond (including Unity 5). This fast-paced practical tutorial provides you with in-depth details of the new UI controls and layouts that can be used to create stunning and multiresolution UI.

Venture forth into the highly componentized and advanced UI system, covering new features such as responsive designs and layouts. Also you get to deep-dive into the scripting side of the UI system as Unity has made the entire framework open source, giving free access to the source code of the new UI system to all developers.

So get ready for a wild ride into the new Unity UI system and set your course for awesome!

Who this book is written for
If you have a good understanding of Unity’s core functionality and a decent grasp of C# scripting in Unity (although not essential if you are just using the Editor with the new UI), you’ll be well placed to take advantage of the new UI feature set.

What you will learn from this book
- Get to grips with the Legacy Unity GUI fundamentals to better understand the path forward
- Explore the tools Unity introduced with the new UI system such as the Rect Transform layout tool
- Unwrap the new base Unity UI controls and what makes them tick
- Work with the layout features and take control in a multiresolution world
- Build stunning UI within the 3D as well as the traditional 2D world
- Understand the new Unity Event System and how it fits in to the new UI system and beyond

Simon Jackson


Content Type: Black & White
Paper Type: White
Page Count: 280
File Type: InDesign
Request ID: CSS1332515
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Looking Back, Looking Forward"
- A synopsis of the book’s content
- More information on Unity 3D UI Essentials

About the Author

Ever since my early years I have been a tinkerer, engineer, problem solver, and solution gatherer. In short, I love to break things apart, figure out how they work, and then put them back together, usually better than before.

I started way back when with my first computer, the Commodore Vic20. It was simple, used a tape deck, and forced you to write programs in basic or assembly; they were fun times. From there, I progressed through the ZX Spectrum +2 and the joyous days of modern graphics, but with the 30 minute load times from a trusty tape deck. Games were a passion of mine even then, which led to many requests for another gaming machine, but Santa brought me an Amstrad 1640, my first PC. From there, my tinkering and building exploded, and that machine ended up being a huge monstrosity with so many addons, tweaks, and fixes; I was Frankenstein, and this PC became my own personal monster crafted from so many parts. Good times.

This passion has led me down many paths, and I learned to help educate others on the tips and tricks I learned along the way; these skills have equipped me well for the future.

Today I would class myself as a game development generalist. I work with many different frameworks, each time digging down, ripping them apart, and then showing whoever would listen through my blog, videos, and speaking events how to build awesome frameworks and titles. This has been throughout many generations of C++, MDX, XNA (what a breath of fresh air that was), MonoGame, Unity 3D, The Sunburn Gaming Engine, HTML, and a bunch of other proprietary frameworks; I do them all. This gives a very balanced view of how to build and manage many different types of multiplatform titles.
I don't stop there, as I regularly contribute to the MonoGame project, adding new features and new samples before publishing it on NuGet. I also have several of my own open source projects and actively seek out any new and interesting ones to help with.

By day I am a lowly lead technical architect working in the healthcare software industry seeking to improve patients' health and care through better software (a challenge to be sure), but by night I truly soar! Building, tinkering, and educating whilst trying to push game titles of my own. One day they will pay the bills, but until then I still lead a double life.

More recently, I achieved the highly acclaimed reward of being a Microsoft MVP in the ID@Xbox program, for my evangelizing efforts in the game development space. This won't change much about me, but will give me additional tools to help game developers out there.

Lastly, you should check out my previous title, which has been one of Packt's best sellers since its release, especially if you want to learn more about Unity's 2D system. Check it out here: http://bit.ly/MasteringUnity2DGameDevelopment.

It truly has been a tormentous year but it looks to be getting better. Through it all, my wife Caroline has been my tiller, keeping me straight and true while I tend to my game development and writing efforts. Looking forward, I'll likely be crafting more of my own experiences with my kids pitching in; Caitlin, new to game development in school, being my sidekick, Jessica adding her colorful artistic talent to the mix, and the boys (Alexander and Nathan) no doubt trying to destroy my efforts through testing.

Additionally, a big thanks to my extended family (Mike and Marilyn) for helping out with the kids and keeping the writing area a kids-free zone for a few desperate hours. (It's amazing what a few hours respite can do.)

Also a big shout out to the PWSA (Prader-Willi Syndrome Association — http://pwsa.co.uk) for their help and support with Alexander, plus the Warrington Youth Club (http://www.warringtonyouthclub.co.uk/) for their exciting events to keep him entertained, especially in his more trying times. On that last thread, a very big thank you to Westland Drive respite (supported by Warrington council), who give us peace of mind and a night off from time to time; Alexander certainly loves his visits there.

Finally, thanks to the reviewers of this title (especially Simon W and Andrew D who joined me from my previous book); they kept me grounded and on target, although didn't help keeping the page count down. Thanks for your support guys!
Unity 3D UI Essentials

A new era has dawned, and Unity Technologies have taken a big, bold step. Not only have they delivered on some big promises for an all new and improved UI system for Unity projects, but they have also made the source for the new UI completely open source, giving everyday developers access to the inner workings of the new UI.

These are bold steps indeed. Many felt that the new UI wouldn't live up to the dream that was sold, as it had been years since they announced it was coming. Delays and rewrites made it look like it was never going to happen, leaving developers with either having to live with the existing legacy GUI or pay for some of the more advanced GUI systems on the asset store (such as NGUI).

Now, after a long and highly deliberated beta program, the new UI system is finally upon us. In some areas, it meets our expectations; in some, it falls a bit short (however, this is only the beginning). In other areas however, it has gone far beyond.

Throughout this title, we will peel back the layers of all this new technology to understand what each component does, how it fits together, and how to use it to build a fantastic new UI in our projects. Each chapter builds upon the last, to arm you (the reader) with all the knowledge required to assemble your UI within your projects. You will not just build on screen menus and options, but to embed UI elements within your 3D game world.

Not only have Unity released the new UI system, they have also given every developer access to the source that builds the UI, allowing you to better understand how things are built and enable you to extend the existing controls or even build your own. If you are feeling adventurous, you can even submit fixes or new features back to Unity for them to include within Unity itself.

Finally, we can now build what we want, how we want and best of all, it's completely free and available with the Free license for Unity. All hail and rejoice!

Now what are you waiting for? Pack up your towel, brew a freshly hot cup of tea, crack open this guide, and start exploring the all new universe of UI.

What This Book Covers

Chapter 1, Looking Back, Looking Forward, is a retrospective look at what Unity3D had to offer prior to 4.6 and an overview of what 4.6 and beyond brings to the table, including a high-level overview of all the new UI features.

Chapter 2, Building Layouts, covers the core elements of the new Unity UI system, the Canvas and Rect Transforms. These elements are the foundations of the new Unity UI system.
Chapter 3, Control, Control, You Must Learn Control, Unity UI introduces a heap-load of new UI controls to suit just about any UI need, from buttons and checkboxes to entire scrollable areas and layout masks. Here, we will delve deep into how to make the most of all the controls available.

Chapter 4, Anchors Away, provides a detailed walk-through of how to make the most of the new Unity UI anchor system and build responsive layouts/designs.

Chapter 5, Screen Space, World Space, and the Camera, Here we finally delve into one of the most highly anticipated parts of the new UI system: the ability to easily build perspective UI layouts and add UI elements as 3D objects within a scene.

Chapter 6, Working with the UI Source, looks at all the coding behind the UI framework and explores the new Event System and UnityEvent frameworks. The chapter finishes with a walk-through, the open source project for the UI system, allowing you to see just about every line of code Unity has written for the new UI.

Appendix, The 3D Scene Sample, talks about a flashy 3D demo scene, which was discussed in Chapter 5, Screen Space, World Space, and the Camera, to show off the UI. Because this wasn't the focus of the book, it was added as an optional appendix that you could follow if you wish. The instructions are also available online and as a downloadable package to enable developers of all levels to make use of it.
Looking Back, Looking Forward

The new Unity UI has long been sought by developers; it has been announced and re-announced over several years, and now it is finally here. The new UI system is truly awesome and (more importantly for a lot of developers on a shoestring budget) it's free.

As we start to look forward to the new UI system, it is very important to understand the legacy GUI system (which still exists for backwards compatibility) and all it has to offer, so you can fully understand just how powerful and useful the new system is. It's crucial to have this understanding, especially since most tutorials will still speak of the legacy GUI system (so you know what on earth they are talking about).

With an understanding of the legacy system, you will then peer over the diving board and walk through a 10,000-foot view of the new system, so you get a feel of what to expect from the rest of this book.

The following is the list of topics that will be covered in this chapter:

- A look back into what legacy Unity GUI is
- Tips, tricks, and an understanding of legacy GUI and what it has done for us
- A high level overview of the new UI features
State of play

You may not expect it, but the legacy Unity GUI has evolved over time, adding new features and improving performance. However, because it has kept evolving based on its original implementation, it has been hampered with many constraints and the ever pressing need to remain backwards compatible (just look at Windows, which even today has to cater for programs written in BASIC (http://en.wikipedia.org/wiki/BASIC)). Not to say the old system is bad, it's just not as evolved as some of the newer features being added to the Unity 4.x and Unity 5.x series, which are based on newer and more enhanced designs, and more importantly, a new core.

The main drawback of the legacy GUI system is that it is only drawn in screen space (drawn on the screen instead of within it) on top of any 3D elements or drawing in your scenes. This is fine if you want menus or overlays in your title but if you want to integrate it further within your 3D scene, then it is a lot more difficult.

For more information about world space and screen space, see this Unity Answers article (http://answers.unity3d.com/questions/256817/about-world-space-and-local-space.html).

So before we can understand how good the new system is, we first need to get to grips with where we are coming from. (If you are already familiar with the legacy GUI system, feel free to skip over this section.)
A point of reference
Throughout this book, we will refer to the legacy GUI simply as GUI. When we talk about the new system, it will be referred to as UI or Unity UI, just so you don't get mixed-up when reading. When looking around the Web (or even in the Unity support forums), you may hear about or see references to uGUI, which was the development codename for the new Unity UI system.

GUI controls
The legacy GUI controls provide basic and stylized controls for use in your titles.

All legacy GUI controls are drawn during the GUI rendering phase from the built-in OnGUI method. In the sample that accompanies this title, there are examples of all the controls in the Assets\BasicGUI.cs script.

For GUI controls to function, a camera in the scene must have the GUILayer component attached to it. It is there by default on any Camera in a scene, so for most of the time you won't notice it. However, if you have removed it, then you will have to add it back for GUI to work. The component is just the hook for the OnGUI delegate handler, to ensure it has called each frame.

Like the Update method in scripts, the OnGUI method can be called several times per frame if rendering is slowing things down. Keep this in mind when building your own legacy GUI scripts.

The controls that are available in the legacy GUI are:

- Label
- Texture
- Button
- Text fields (single/multiline and password variant)
- Box
- Toolbars
- Sliders
- ScrollView
- Window
Looking Back, Looking Forward

So let's go through them in more detail:

All the following code is implemented in the sample project in the basic GUI script located in the Assets\Scripts folder of the downloadable code.

To experiment yourself, create a new project, scene, and script, placing the code for each control in the script and attach the script to the camera (by dragging it from the project view on to the Main Camera GameObject in the scene hierarchy). You can then either run the project or adorn the class in the script with the [ExecuteInEditMode] attribute to see it in the game view.

The Label control

Most GUI systems start with a Label control; this simply provides a stylized control to display read-only text on the screen, it is initiated by including the following OnGUI method in your script:

```csharp
void OnGUI() {
    GUI.Label(new Rect(25, 15, 100, 30), "Label");
}
```

This results in the following on-screen display:

![Label Display]

The Label control supports altering its font settings through the use of the guiText GameObject property (guiText.font) or GUIStyles. (See the following section on GUIStyles for more detail.)

To set guiText.font in your script, you would simply apply the following in your script, either in the Awake/Start functions or before drawing the next section of text you want drawn in another font:

```csharp
public Font myFont = new Font("arial");
    guiText.font = myFont;
```

You can also set the myFont property in Inspector using an imported font.

The Label control forms the basis for all controls to display text, and as such, all other controls inherit from it and have the same behaviors for styling the displayed text.
The **Label** control also supports using a Texture for its contents, but not both text and a texture at the same time. However, you can layer Labels and other controls on top of each other to achieve the same effect (controls are drawn implicitly in the order they are called), for example:

```csharp
public Texture2D myTexture;
void Start() {
    myTexture = new Texture2D(125, 15);
}
void OnGUI() {
    // Draw a texture
    GUI.Label(new Rect(125, 15, 100, 30), myTexture);
    // Draw some text on top of the texture using a label
    GUI.Label(new Rect(125, 15, 100, 30), "Text overlay");
}
```

You can override the order in which controls are drawn by setting `GUI.depth = /*<depth number>*;*/` in between calls; however, I would advise against this unless you have a desperate need.

The texture will then be drawn to fit the dimensions of the **Label** field. By default it scales on the shortest dimension appropriately. This too can be altered using ` GUIStyle` to alter the fixed width and height or even its stretch characteristics.

```csharp
GUIStyles and GUI SKins are explained in the later GUI styles and skins section.
```

### Texture drawing

Not specifically a control in itself, the GUI framework also gives you the ability to simply draw a Texture to the screen. Granted there is little difference to using `DrawTexture` function instead of a **Label** with a texture or any other control. (Just another facet of the evolution of the legacy GUI). This is, in effect, the same as the previous **Label** control but instead of text it only draws a texture, for example:

```csharp
public Texture2D myTexture;
void Start() {
    myTexture = new Texture2D(125, 15);
}
void OnGUI() {
    // Draw a texture
    GUI.Label(new Rect(125, 15, 100, 30), myTexture);
    // Draw some text on top of the texture using a label
    GUI.Label(new Rect(125, 15, 100, 30), "Text overlay");
}
```
Looking Back, Looking Forward

```csharp
myTexture = new Texture2D(125, 15);
void OnGUI()
{
    GUI.DrawTexture(new Rect(325, 15, 100, 15), myTexture);
}
```

Note that in all the examples providing a Texture, I have provided a basic template to initialize an empty texture. In reality, you would be assigning a proper texture to be drawn.

You can also provide scaling and alpha blending values when drawing the texture to make it better fit in the scene, including the ability to control the aspect ratio that the texture is drawn in.

```csharp
A warning though, when you scale the image, it affects the rendering properties for that image under the legacy GUI system. Scaling the image can also affect its drawn position. You may have to offset the position it is drawn at sometimes.

For example:

```csharp
public Texture2D myTexture;
void Start()
{
    myTexture = new Texture2D(125, 15);
}
void OnGUI()
{
    GUI.DrawTexture(new Rect(325, 15, 100, 15), myTexture,
        ScaleMode.ScaleToFit, true, 0.5f);
}
```

This will do its best to draw the source texture with in the drawn area with alpha blending (true) and an aspect ratio of 0.5. Feel free to play with these settings to get your desired effect.

This is useful in certain scenarios in your game when you want a simple way to draw a full screen image on the screen on top of all the 3D/2D elements, such as pause or splash screen. However, like the Label control, it does not receive any input events (see the Button control for that).
There is also a variant of the DrawTexture function called DrawTextureWithTexCoords. This allows you to not only pick where you want the texture drawn on to the screen, but also from which part of the source texture you want to draw, for example:

```csharp
public Texture2D myTexture;
void Start() {
    myTexture = new Texture2D(125, 15);
}
void OnGUI() {
    GUI.DrawTextureWithTexCoords (new Rect(325, 15, 100, 15),
        myTexture,
        new Rect(10, 10, 50, 5));
}
```

This will pick a region from the source texture (myTexture) 50 pixels wide by 5 pixels high starting from position 10, 10 on the texture. It will then draw this to the Rect region specified.

However, the DrawTextureWithTexCoords function cannot perform scaling, it can only perform alpha blending! It will simply draw to fit the selected texture region to the size specified in the initial Rect.

DrawTextureWithTexCoords has also been used to draw individual sprites using the legacy GUI, which has a notion of what a sprite is.

### The Button control

Unity also provides a **Button** control, which comes in two variants. The basic Button control which only supports a single click, whereas **RepeatButton** supports holding down the button.

They are both instantiated the same way by using an if statement to capture when the button is clicked, as shown in the following script:

```csharp
void OnGUI() {
    if (GUI.Button(new Rect(25, 40, 120, 30), "Button"))
    {
        //The button has clicked, holding does nothing
    }
    if (GUI.RepeatButton(new Rect(170, 40, 170, 30),
        "RepeatButton")
    {
        //The button has been clicked or is held down
    }
}
```
Each will result in a simple button on screen as follows:

![Button]

The controls also support using a Texture for the button content as well by providing a texture value for the second parameter as follows:

```csharp
public Texture2D myTexture;
void Start() {
    myTexture = new Texture2D(125, 15);
}
void OnGUI() {
    if (GUI.Button(new Rect(25, 40, 120, 30), myTexture))
    {  }
}
```

Like the `Label`, the font of the text can be altered using `GUIStyle` or by setting the `guiText` property of the GameObject. It also supports using textures in the same way as the Label. The easiest way to look at this control is that it is a Label that can be clicked.

**The Text control**

Just as there is a need to display text, there is also a need for a user to be able to enter text, and the legacy GUI provides the following controls to do just that:

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextField</td>
<td>This is a basic text box, it supports a single line of text, no new lines (although, if the text contains end of line characters, it will draw the extra lines down).</td>
</tr>
<tr>
<td>TextArea</td>
<td>This is an extension of TextField that supports entering of multiple lines of text; new lines will be added when the user hits the enter key.</td>
</tr>
</tbody>
</table>
| PasswordField | This is a variant of TextField; however, it won't display the value entered, it will replace each character with a replacement character.  
Note, the password itself is still stored in text form and if you are storing users' passwords, you should encrypt/decrypt the actual password when using it. Never store characters in plain text. |
Using the **TextField** control is simple, as it returns the final state of the value that has been entered and you have to pass that same variable as a parameter for the current text to be displayed. To use the controls, you apply them in script as follows:

```csharp
string textString1 = "Some text here";
string textString2 = "Some more text here";
string textString3 = "Even more text here";
void OnGUI() {
    textString = GUI.TextField(new Rect(25, 100, 100, 30),
        textString1);
    textString = GUI.TextArea(new Rect(150, 100, 200, 75),
        textString2);
    textString = GUI.PasswordField(new Rect(375, 100, 90, 30),
        textString3, '*');
}
```

**A note about strings in Unity scripts**

Strings are immutable. Every time you change their value they create a new string in memory by having the `textString` variable declared at the class level it is a lot more memory efficient.

If you declare the `textString` variable in the `OnGUI` method, it will generate garbage (wasted memory) in each frame.

Worth keeping in mind.

When displayed, the textbox by default looks like this:

![Textbox](image)

As with the **Label** and **Button** controls, the font of the text displayed can be altered using either a `GUIStyle` or `guiText` GameObject property.

![Note](image)

Note that text will overflow within the field if it is too large for the display area, but it will not be drawn outside the `TextField` dimensions. The same goes for multiple lines.
The Box control

In the midst of all the controls is a generic purpose control that seemingly can be used for a variety of purposes. Generally, it's used for drawing a background/shaded area behind all other controls.

The Box control implements most of the features mentioned in the controls above controls (Label, Texture, and Text) in a single control with the same styling and layout options. It also supports text and images as the other controls do.

You can draw it with its own content as follows:

```csharp
void OnGUI() {
    GUI.Box (new Rect (350, 350, 100, 130), "Settings");
}
```

This gives you the following result:

![Box control example](image)

Alternatively, you can use it to decorate the background of other controls, for example:

```csharp
private string textString = "Some text here";
void OnGUI() {
    GUI.Box (new Rect (350, 350, 100, 130), "Settings");
    GUI.Label(new Rect(360, 370, 80, 30), "Label");
    textString = GUI.TextField(new Rect(360, 400, 80, 30), textString);
    if (GUI.Button (new Rect (360, 440, 80, 30), "Button")) {}      
}
```

Note that using the Box control does not affect any controls you draw upon it. It is drawn as a completely separate control. This statement will be made clearer when you look at the Layout controls later in this chapter.
The example will draw the box background and the **Label**, **Text**, and **Button** controls on top of the box area and looks like this:

![Box with controls](image)

The box control can be useful to highlight groups of controls or providing a simple background (alternatively, you can use an image instead of just text and color).

As with the other controls, the **Box** control supports styling using **GUIStyle**.

### The Toggle/checkbox control

If checking on / checking off is your thing, then the legacy GUI also has a checkbox control for you, useful for those times when you need to visualize the on/off state of an option.

Like the **TextField** control, you pass the variable that manages **Togglestate** as a parameter and it returns the new value (if it changes), so it is applied in code as follows:

```csharp
bool blnToggleState = false;
void OnGUI() {
    blnToggleState = GUI.Toggle(new Rect(25, 150, 250, 30),
                                blnToggleState, "Toggle");
}
```

This results in the following on-screen result:

![Toggle](image)

As with the **Label** and **Button** controls, the font of the text displayed can be altered using either a **GUIStyle** or guiText GameObject property.
Toolbar panels
The basic GUI controls also come with some very basic automatic layout panels: the first handles an arrangement of buttons, however these buttons are grouped and only one can be selected at a time.

As with other controls, the style of the button can be altered using a GUIstyles definition, including fixing the width of the buttons and spacing.

There are two layout options available, these are:

- The Toolbar control
- The Selection grid control

The Toolbar control
The Toolbar control arranges buttons in a horizontal pattern (vertical is not supported).

Note that it is possible to fake a vertical toolbar by using a selection grid with just one item per row. See the Selection grid section later in this chapter for more details.

As with other controls, the Toolbar returns the index of the currently selected button in the toolbar. The buttons are also the same as the base button control so it also offers options to support either text or images for the button content.

The RepeatButton control however is not supported.

To implement the toolbar, you specify an array of the content you wish to display in each button and the integer value that controls the selected button, as follows:

```csharp
private int toolbarInt;
private string[] toolbarStrings;
Void Start() {
    toolbarInt = 0;
    toolbarStrings = { "Toolbar1", "Toolbar2", "Toolbar3" };}
void OnGUI() {
    toolbarInt = GUI.Toolbar(new Rect(25, 200, 200, 30),
        toolbarInt, toolbarStrings);
}
```
The main difference between the preceding controls is that you have to pass the currently selected index value in to the control and it then returns the new value.

The Toolbar when drawn looks as follows:

![Toolbar](image)

As stated, you can also pass an array of textures as well and they will be displayed instead of the text content.

**The SelectionGrid control**

The SelectionGrid control is a customization of the previous standard Toolbar control, it is able to arrange the buttons in a grid layout and resize the buttons appropriately to fill the target area.

In code, SelectionGrid is used in a very similar format to the Toolbar code shown previously, for example:

```csharp
private int selectionGridInt;
private string[] selectionStrings;

Void Start() {
    selectionGridInt = 0;
    selectionStrings = { "Grid 1", "Grid 2", "Grid 3", "Grid 4" };
}

void OnGUI() {
    selectionGridInt = GUI.SelectionGrid(
        new Rect(250, 200, 200, 60), selectionGridInt, selectionStrings, 2);
}
```

The main difference between SelectionGrid and Toolbar in code is that with SelectionGrid you can specify the number of items in a single row and the control will automatically lay out the buttons accordingly (unless overridden using GUIStyle).

The preceding code will result in the following layout:

![SelectionGrid](image)

On their own, they provide a little more flexibility than just using buttons alone.
The Slider/Scrollbar controls

When you need to control a range in your games. GUI or add a handle to control moving properties between two values, like moving an object around in your scene, this is where the **Slider** and **Scrollbar** controls come in. They provide two similar *out-of-the-box* implementations that give a scrollable region and a handle to control the value behind the control.

Here, they are presented side by side:

The **slimmer Slider** and **chunkier Scrollbar** controls can both work in either horizontal or vertical modes and have presets for the minimum and maximum values allowed.

**Slider control code**

In code, the Slider control code is represented as follows:

```csharp
private float fltSliderValue = 0.5f;
void OnGUI() {
    fltSliderValue = GUI.HorizontalSlider(new Rect(25, 250, 100, 30), fltSliderValue, 0.0f, 10.0f);
    fltSliderValue = GUI.VerticalSlider(new Rect(150, 250, 25, 50), fltSliderValue, 10.0f, 0.0f);
}
```

**Scrollbar control code**

In code, the Scrollbar control code is represented as follows:

```csharp
private float fltScrollerValue = 0.5f;
void OnGUI() {
    fltScrollerValue = GUI.HorizontalScrollbar(new Rect(25, 285, 100, 30), fltScrollerValue, 1.0f, 0.0f, 10.0f);
    fltScrollerValue = GUI.VerticalScrollbar(new Rect(200, 250, 25, 50), fltScrollerValue, 1.0f, 10.0f, 0.0f);
}
```
Like Toolbar and SelectionGrid, you are required to pass in the current value and it will return the updated value. However, unlike all the other controls, you actually have two style points, so you can style the bar and the handle separately, giving you a little more freedom and control over the look and feel of the sliders.

Normally, you would only use the Slider control; The main reason the Scrollbar is available is that it forms the basis for the next control, the ScrollView control.

**The ScrollView control**
The last of the displayable controls is the ScrollView control, which gives you masking abilities over GUI elements with optional horizontal and vertical Scrollbars. Simply put, it allows you to define an area larger for controls behind a smaller window on the screen, for example:

![Example list implementation using a ScrollView control](image)

Here we have a list that has many items that go beyond the drawable area of the ScrollView control. We then have the two scrollbars to move the content of the scroll viewer up/down and left/right to change the view. The background content is hidden behind a viewable mask that is the width and height of the ScrollView control’s main window.

Styling the control is a little different as there is no base style for it; it relies on the currently set default GUISkin (see the following GUIStyles section). You can however set separate GUIStyles for each of the sliders but only over the whole slider, not its individual parts (bar and handle). If you don't specify styles for each slider, it will also revert to the base GUIStyle.
Implementing a `ScrollView` is fairly easy, for example:

1. Define the visible area along with a virtual background layer where the controls will be drawn using a `BeginScrollView` function.

2. Draw your controls in the virtual area. Any GUI drawing between the `ScrollView` calls is drawn within the scroll area.

   ![Note](image)
   
   Note that 0,0 in the `ScrollView` is from the top-left of the `ScrollView` area and not the top-left-hand corner of the screen.

3. Complete it by closing the control with the `EndScrollView` function. For example, the previous example view was created with the following code:

   ```csharp
   private Vector2 scrollPosition = Vector2.zero;
   private bool blnToggleState = false;
   void OnGUI()
   {
   scrollPosition = GUI.BeginScrollView(
   new Rect(25, 325, 300, 200),
   scrollPosition,
   new Rect(0, 0, 400, 400));

   for (int i = 0; i < 20; i++)
   {
   //Add new line items to the background
   addScrollViewListItem(i, "I'm listItem number " + i);
   }
   GUI.EndScrollView();
   }
   
   //Simple function to draw each list item, a label and checkbox
   void addScrollViewListItem(int i, string strItemName)
   {
   GUI.Label(new Rect(25, 25 + (i * 25), 150, 25), strItemName);
   blnToggleState = GUI.Toggle(
   new Rect(175, 25 + (i * 25), 100, 25),
   blnToggleState, ":");
   }
   ```
In this code, we define a simple function (addScrollViewListItem) to draw a list item (consisting of a label and checkbox). We then begin the ScrollView control by passing the visible area (the first Rect parameter), the starting scroll position, and finally the virtual area behind the control (the second Rect parameter). Then we use that to draw 20 list items inside the virtual area of the ScrollView control using our helper function before finishing off and closing the control with the EndScrollView command.

If you want to, you can also nest ScrollView controls within each other.

The ScrollView control also has some actions to control its use like the ScrollTo command. This command will move the visible area to the coordinates of the virtual layer, bringing it into focus. (The coordinates for this are based on the top-left position of the virtual layer; 0,0 being top-left.)

To use the ScrollTo function on ScrollView, you must use it within the Begin and End ScrollView commands. For example, we can add a button in ScrollView to jump to the top-left of the virtual area, for example:

```csharp
public Vector2 scrollPosition = Vector2.zero;
void OnGUI()
{
    scrollPosition = GUI.BeginScrollView(
        new Rect(10, 10, 100, 50),
        scrollPosition,
        new Rect(0, 0, 220, 10));

    if (GUI.Button(new Rect(120, 0, 100, 20), "Go to Top Left"))
        GUI.ScrollTo(new Rect(0, 0, 100, 20));

    GUI.EndScrollView();
}
```

You can also additionally turn on/off the sliders on either side of the control by specifying the BeginScrollView statement using the alwayShowHorizontal and alwayShowVertical properties; these are highlighted here in an updated GUI.BeginScrollView call:

```csharp
Vector2 scrollPosition = Vector2.zero;
bool ShowVertical = false; // turn off vertical scrollbar
```
bool ShowHorizontal = false; // turn off horizontal scrollbar
void OnGUI() {
    scrollPosition = GUI.BeginScrollView(
        new Rect(25, 325, 300, 200),
        scrollPosition,
        new Rect(0, 0, 400, 400),
        ShowHorizontal,
        ShowVertical);
    GUI.EndScrollView();
}

Rich Text Formatting

Now having just plain text everywhere would not look that great and would likely force developers to create images for all the text on their screens (granted a fair few still do so for effect). However, Unity does provide a way to enable richer text display using a style akin to HTML wherever you specify text on a control (only for label and display purposes; getting it to work with input fields is not recommended).

In this HTML style of writing text, we have the following tags we can use to liven up the text displayed.

<b></b>
This gives text a Bold format, for example:

The <b>quick</b> brown <b>Fox</b> jumped over the <b>lazy Frog</b>

This would result in:

The quick brown Fox jumped over the lazy Frog

<i></i>
Using this tag will give text an Italic format, for example:

The <i>quick</i> brown <i>Fox</i> jumped over the <i>lazy Frog</i>

This would result in:

The quick brown Fox jumped over the lazy Frog

<size></size>
As you can probably guess, this tag will alter the Size of the text it surrounds.

For reference, the default size for the font is set by the font itself.
For example:

The `<b>`<i>quick</i>`</b>` `<size=50>`brown` `<b>`Fox` `<i>`jumped` `<i>`over the `<b>`lazy` `<b> Frog`<b>`</b>`

This would result in:

The **quick** brown **Fox** jumped over the **lazy** Frog

**<color></color>**

Lastly, you can specify different colors for text surrounded by the Color tag. The color itself is denoted using its 8-digit RGBA hex value, for example:

The `<b>`<i>quick</i>`</b>` `<size=50>`<color=#a52a2aff>`brown` `<b>`Fox` `<i>`jumped` `<i>`over the `<b>`lazy` `<b> Frog`<b>`</b>`

Note that the color is defined using normal RGBA color space notation (http://en.wikipedia.org/wiki/RGBA_color_space) in hexadecimal form with two characters per color, for example, RRGGBBAA. Although the color property does also support the shorter RGB color space, which is the same notation but without the A (Alpha) component, for example, RRGGBB.

The preceding code would result in:

The **quick** brown **Fox** jumped over the **lazy** Frog

(If you are reading this in print, the previous word brown is in the color brown.)

You can also use a color name to reference it but the pallet is quite limited; for more details, see the Rich Text manual reference page at http://docs.unity3d.com/Manual/StyledText.html.

For text meshes, there are two additional tags:

- `<material></material>`
- `<quad></quad>`

These only apply when associated to an existing mesh. The material is one of the materials assigned to the mesh, which is accessed using the mesh index number (the array of materials applied to the mesh). When applied to a quad, you can also specify a size, position (x, y), width, and height to the text.
Looking Back, Looking Forward

The text mesh isn't well documented and is only here for reference; as we delve deeper into the new UI system, we will find much better ways of achieving this.

Common control features

The legacy GUI system does also have some features for controlling flow, control selection, and targeted behavior. When it was introduced in Unity V2, it was a huge step up from the previous component-based system.

Grouping controls

The legacy GUI allow you to group controls together on the screen, making all positions for the group relative to the group's position. This means that if you started a group at position X 50 and Y 50, then all child control positions within the group would start at 50,50 when they define their position as 0,0.

Like the ScrollView control, each group has a beginning and an end to define the scope of all the controls within the group, for example:

```csharp
void OnGUI() {
  //Start a group at position 50, 50. 150 width and 60 height
  GUI.BeginGroup(new Rect (50,50,150,60));
  //Draw a label with a 10, 10 offset in the group
  GUI.Label(new Rect (10, 10, 100, 30), "Label in a Group");
  GUI.EndGroup();
}
```

Here the label is drawn within the group bounds, and as the group starts at X 50, the Labels screen position will be at X 60 (50 + 10). Anything positioned or overflowing the group's bounds will not be drawn.

The group, like other controls, can also specify content within the group as text or a texture with appropriate GUIStyles.

What is odd is that unlike the rest of the controls, if you specify text content in the function, the default color of text is white, whereas if you specify text in the content parameter for the BeginGroup function, the text in the group is black by default. It's also left justified instead of centered.

Additionally, by default the group does not support Rich Text Formatting unlike the other controls, so you will need to apply GUIStyle to change that.
Naming controls

With each control that you implement through script, you can name them as you place them; this is essential if you want to control flow and access to each field from the keyboard or to derive logic based on the currently selected/focused control.

Now unlike most other Unity functionality, you cannot directly name controls, there is no Name field on the properties of the controls as they are just commands to the GUI system to draw things to the screen, kind of like a rendering pipeline.

So to name GUI controls in Unity, we simply need to tell the GUI system that the next control we are going to draw has a name, as follows:

```csharp
string login = "Would you like to play a game?";
void OnGUI() {
    GUI.SetNextControlName("MyAwesomeField");
    login = GUI.TextField(new Rect(10, 10, 200, 20), login);
}
```

Getting in focus

With names defined on controls, you could then define which control you were focusing on. To focus on a specific control, you would simply need to call:

```csharp
GUI.FocusControl("MyAwesomeField");
```

This would then change the user's input focus or selection to the specific GUI control with that name.

Once you have a control in focus, you then discover the name of the specific control in focus by calling:

```csharp
string selectedControl = GUI.GetNameOfFocusedControl();
```

If the control in focus has a name, it will return the name you set for that control. If no control is in focus or the control in focus has no name, it will return an empty string.

The logon example

As an example of using the previous naming and focus capabilities, you can build a simple logon GUI for a user to enter with validation behavior and some usability features.

To demonstrate, we will create a user registration form where the user can enter a username and password to register with your game. The password however will have to be more than six characters long for security reasons (no weak passwords here).
To start, create a new script called IntermediateGUI in your project (the full sample can be found in the project available with this book in the code download) and replace its contents with the following:

```csharp
using UnityEngine;

[ExecuteInEditMode]
public class IntermediateGUI : MonoBehaviour {

    public string username = "Enter username";
    public string password = "Enter password";
    private bool passwordInError = false;
    private string passwordErrorMessage = "<color=red>Password too short</color>";
}
```

This gives a basic class with some of the parameters you might expect in a logon or registration form.

To this we'll add a simple function to validate the password entered to ensure it meets our stringent security policy:

```csharp
void CheckUserPasswordAndRegister() {
    if (password.Length < 6) {
        // If the password is not long enough, mark it in error
        // and focus on the password field
        passwordInError = true;
        GUI.FocusControl("PasswordField");
    } else {
        passwordInError = false;
        GUI.FocusControl("");
        // Register User
    }
}
```

With that in place, now we can add our GUI controls:

```csharp
void OnGUI() {
    // A tidy group for our fields and a box to decorate it
    GUI.BeginGroup(new Rect(Screen.width / 2 - 75,
        Screen.height / 2 - 80, 150, 160));
    GUI.Box(new Rect(0, 0, 150, 160), "User registration form");
}
```
GUI.SetNextControlName("UsernameField");
username = GUI.TextField(new Rect(10, 40, 130, 20), username);
GUI.SetNextControlName("PasswordField");
password = GUI.PasswordField(new Rect(10, 70, 130, 20),
password,'**');
if (passwordInError)
{
    GUI.Label (new Rect (10, 100, 200, 20),
    passwordErrorMessage);
}
if (Event.current.isKey &&
    Event.current.keyCode == KeyCode.Return &&
    GUI.GetNameOfFocusedControl() == "PasswordField")
{
    CheckUserPasswordAndRegister();
}
if (GUI.Button(new Rect(80, 130, 65, 20), "Register")
{
    CheckUserPasswordAndRegister();
}
GUI.EndGroup();

Note that the Event keyword here relates to the legacy GUI event system for handling user input. See the Event section later in this chapter for more information.  This is NOT to be confused with the UnityEvent system introduced with the new UI system.

These results are shown in the following GUI screen:
Looking Back, Looking Forward

In this example, we draw a box, a text field, and a password field together with a simple button within a group, which is then centered on the screen.

We check whether the user hits the Enter key and whether they are on the password field (checked using the GUI.GetNameOfFocusedControl() function) and we try to register them. The same happens if the user clicks on the Register button.

If the user's password is longer than six characters, then they are registered; if not, then the passwordInError flag is set to True, which causes the additional label to be drawn, this then warns the user that their password could be broken easily by a 6-year-old.

Don't forget to add the IntermediateGUI script to an active GameObject in a scene or Main Camera to see the result!

Tooltips

Each of the GUI controls can also have a tooltip associated with it to display some additional text when it is either in focus or the mouse is hovering over the control.

Adding a tooltip is simple; you just need to replace the content of the control when it is being drawn using the GUIContent class. For example, we can update the Register button in the previous script as follows:

```csharp
if (GUI.Button(new Rect(80, 130, 65, 20),
                new GUIContent("Register", "My Tooltip")))
{
    CheckUserPasswordAndRegister();
}
```

With the tooltip defined, we just then need to display the current tooltip somewhere on the screen, usually as a label, but it can be any control that can display text (input fields are not recommended however), so add the following after the button block but before EndGroup():

```csharp
GUI.Label (new Rect (10, 120, 65, 20), GUI.tooltip);
```

This simply gets the content of the current tooltip in focus and returns the tooltip text for that control.

GUIContent also has several other options for displaying text and texture variants, so it's worth checking out some more.
The Window control

The last weapon in the legacy GUI arsenal is the Window control. As the name suggests, this defines a separate drawable window for your controls.

- The window behavior is similar to ScrollView; however, it is just one layer. Any controls drawn outside the bounds of the window are simply not drawn.

- But there is nothing to stop you using a ScrollView control inside a Window to achieve the same thing however.

With this separate Window, we can control many things, including:

- The modal nature of the Window
  
  Modal means that this window is the only one you can control; non-modal means it is a side-by-side window

- The drag state of Window; as in, the window can be dragged by holding on with a mouse or touch

- The draw order of each Window; this allows sorting of draw windows on top of each other

- The specific Window in focus, if there are multiple side-by-side windows or a modal window

This opens lots of possibilities with a GUI Window.

- The full Window example can be found in the BasicGUI script in the sample project, displaying all the same controls as before but within a single separate Window control.

To create a Window control, you first need to define a new method callback for the Window using the following signature:

```csharp
void DoMyWindow(int windowID)
{
}
```
Looking Back, Looking Forward

This method is where you will add all your GUI code using the previous examples; each control is positioned is based off the top-left position of the window when it is displayed (same as the Group and ScrollView controls described earlier).

Additionally, you can specify any of the previous options for the window, for example:

```csharp
void DoMyWindow(int windowID)
{
    GUI.Label(new Rect(25, 15, 100, 30), "Label");
    // Make the window Draggable
    GUI.DragWindow();
}
```

With your Window method in place, you just need to call the GUI.Window function to open it along with the property to track the Window's location:

```csharp
private Rect rctWindow1;
void OnGUI()
{
    Rect rctWindow1;
    rctWindow1 = GUI.Window(0, rctWindow1, DoMyWindow, "Controls Window");
}
```

This calls Window control into view using:

- An ID for the window
- The Rect position for where Window will open
- The delegate method for the GUI contents of Window
- A name/title for the window

If you want a modal window, then you would need to instantiate the window with the GUI.ModalWindow function instead of the Window function:

```csharp
rctWindow1 = GUI.ModalWindow(0, rctWindow1, DoMyWindow, "Modal Controls Window");
```

If we take all the controls together (that we have created so far in this chapter), it would create a Window view, as shown in the following screenshot:

For a complete end-to-end example, please see the code download package, which has all this defined.
GUI styles and skins
Recognizing that not everyone likes plain backgrounds and the same font throughout their projects, Unity provided options to define a style for the layout and general look and feel of the legacy GUI system, these are defined as **GUIStyles**.

![GUI styles and skins screenshot]

These styles can either be applied globally using a **GUISkin** (see **GUISkin** in the following section), or they can be applied individually to each control (as detailed in the previous screenshot).
Each style has options to define:

- A Name
- A texture or text color for the different states of the control it's attached to (Normal, Hover, Active, and Focused)
- The border, margin, padding, and overflow sizes for the control (for each edge)
- A Font (with suitable size, style, alignment, word wrapping, and rich text support options)
- A text clipping size
- Image position within the control
- Offset settings for the content within the control
- A fixed width and height
- Stretching options for the width and height

I recommended configuring a public GUIStyle property in a class and then modifying it in Editor Inspector when setting up a GUIStyle, for example:

```csharp
using UnityEngine;

[ExecuteInEditMode]
public class GUIStyles : MonoBehaviour {
    public GUIStyle;
    void OnGUI() {
        //Create a label using the GUIStyle property above
        GUI.Label(new Rect(25, 15, 100, 30), "Label", myGUIStyle);
    }
}
```

You can also configure a GUIStyle in code, however it's not recommended as the editor is just better at it.

It is worth noting that having too many different GUIStyles all over the place can become very inefficient and hard to maintain. If you find you are using a lot of GUIStyles then I'd recommend you create a single script attached to a common object (say Main Camera) in your scene with all your GUIStyle's defined and have each script take GUIStyle references from there.
When you attach the preceding script with the `GUIStyle` property to a GameObject in your scene, it will look like this in Inspector:

![Inspector window showing GUIStyle properties](image)

Note that the first time you open it in the editor you will get `NullReferenceException` in the console window; this is just because you haven’t configured `GUIStyle` yet.

If you don't want to apply a style to each and every control directly, you can then optionally create `GUISkin`, which contains all the styles for each control type. This is then applied using the `GUI` class prior to drawing any controls.

A `GUISkin` also has some additional options that apply to the GUI, which include:

- Setting whether a double-click action selects
- Setting whether a triple-click action selects
- The color of the cursor
- The cursor flash speed
- The default selection color
- Custom styles (an array of `GUIStyle` properties you can then reuse on controls)
To demonstrate, click on the Create button in the project folder view and select **GUISkin**, which will give you a new **GUISkin** asset in the project view. By selecting it, you will see the following window in Inspector:

![Inspector window](image)

As you can see, it contains all the options for altering the style globally for each control. To use **GUISkin**, create a new script called **GUISkins**, then replace its contents with the following:

```csharp
using UnityEngine;

public class GUISkins : MonoBehaviour {
    public GUISkin MySkin;
    void OnGUI()
    {
        GUI.skin = MySkin;
        GUI.Label(new Rect(25, 15, 100, 30), "Label");
        //Draw the rest of your controls
    }
}
```

Then attach the **GUISkins** script to **Main Camera** in your current scene (disabling any other scripts currently attached) and drag the **GUISkin** you have created and apply it to the **My Skin** property of the script in the inspector.
By setting the skin at the beginning of any GUI drawing, any and all controls drawn will now use your custom skin instead of the Unity default. If you wish, you can use several skins by just changing the skin before drawing more controls.

For some of the best examples of **GUISkins**, try installing the Unity Extra GUI Skins asset ([http://bit.ly/UnityExtraGUISkins](http://bit.ly/UnityExtraGUISkins)), which is a collection of skin samples built by Unity themselves (and it’s free).

[Note that if you want to reuse your own skins in other projects (or sell more skins through the asset store), then you can export them using Unity's Export Package option under Assets in the menu. For more details, check out [http://docs.unity3d.com/ Manual/HOWTO-exportpackage.html](http://docs.unity3d.com/ Manual/HOWTO-exportpackage.html)]

Here's an example of what the GUISkins asset gives you:
GUI events and properties

To support events in the legacy GUI area, Unity added an entire event handler specifically for GUI interactions. This class is aptly named the Event class.

Remember, this section refers to the legacy GUI Event classes, which has nothing to do with the new UnityEvent system introduced in the new Unity UI system. See Chapter 6, Working with the UI Source, for more details of the UnityEvent system.

These events center on user and device input, varying from:

- **Event types**: Is it a key event, mouse event, and so on
- **Event values**: Which key pressed, mouse button pressed, and so on
- **Event summary information**: Modifier keys, mouse movement delta, and so on

To access the events you simply need to query the Event.current property to get the current Event state. (The Event state updates when there is a change, until then you will get the last/previous state.)

The logon example earlier shows an example for using events, where we detect if the user has pressed a key and if that key is the Enter key as shown in this script:

```csharp
if (Event.current.isKey &&
    Event.current.keyCode == KeyCode.Return &&
    GUI.GetNameOfFocusedControl() == "PasswordField")
{
    CheckUserPasswordAndRegister();
}
```

Along with the events, the GUI class also provides some additional properties you can query or set in the OnGUI method, namely:

- **enabled**: Is the GUI enabled and displayed on the screen. Can it be used to turn on or off controls that are to be drawn to the screen.
- **changed**: This returns true if any controls' values have changed since the last call of OnGUI.
- **color**: This is the global color tint for the GUI layout.
- **contentColor**: This is the global text color tint for the GUI.
• `backgroundColor`: This is the global background color tint.
• `depth`: This is the depth order for the current `GUI` script. This is useful if you have `GUI` elements in more than one script and want to layer them.
• `matrix`: The 3D transformation matrix for the current `GUI`.

All of these elements can be used to override all controls or individual controls by setting them in between controls.

**Layout controls**

If you prefer not to hand draw the position of every single control in your `GUI`, Unity does offer some automatic layout controls from the `GUILayout` class.

The Layout controls (using `GUILayout` instead of just `GUI`) have the same set of controls as the normal `GUI` class (hence I'm not going to describe them all over again), the main difference is that you do not need to specify a `Rect` area to draw the control, as it will just be drawn at the first suitable location; any further controls added will be laid out appropriately with enough spacing between the controls.

You can also control the spacing and even if you want, any empty space between the controls using the `Width`, `Height`, and `Spacing` properties (`Space/FlexibleSpace`) of `GUILayout`, following the same rules as for `GUI` controls (setting up the `GUILayout` before drawing a control).

If you don't want the layout to take up the maximum space for a control, you also have the settings for Width (`MaxWidth/MinWidth`) and Height (`MaxHeight/MinHeight`).

The main differences are as follows:

• `BeginGroup` becomes `BeginArea`
• Horizontal and vertical groups (sub groups)

**BeginArea**

Instead of defining `Groups`, you define `Areas`. Apart from the name, they behave exactly the same. This is the only layout control that takes a `Rect` parameter to specify where you want to draw the controls (excluding Windows of course); all `GUILayout` controls are then aligned to `Area` in the same way they are in `Group`.

[It's recommended that when using `GUILayout` controls that you place them in an Area for the best effect.]
**Horizontal and Vertical layout groups**

To control the layout of controls, you can define a set of controls to draw in a particular direction, either horizontally or vertically. You start these in the same way as areas by setting a `GUILayout.BeginHorizontal()` and `GUILayout.EndHorizontal()` command. Like the Area, you can specify additional content for the new sub-area such as text or textures.

**The Asset Store**

Several packages on the Unity asset store have tried to build a more fluent UI creation system. They have met with varying success and all suffer from one underlying issue, they weren't built by Unity and don't have access to some of the underlying runtime and rendering components of the Unity editor and player. This results in some performance issues (but in some cases they are actually faster than the legacy GUI, especially on mobile platforms) and (in some cases) hacky workarounds. All in all, most have done incredibly well without access to Unity's innards.

With the release of the new Unity UI system however, I would recommend checking the state of many of the GUI assets out there as several (quite understandably) have bowed their heads and are dropping off the store. The main package is still ongoing is NGUI (http://bit.ly/UnityAssetStore-NGUI), but it does have a hefty price tag to it. With it though are a multitude of supporting assets to make its adoption easier and offer integration into several other assets.

**Enter Thunderdome**

Now that you've seen what Unity has had available for so long with the legacy GUI (and if you have experienced it you will undoubtedly shudder at this point), it is a very welcome relief that the UI system has received such an overhaul in Unity 4.6. It has been a long time coming and very much anticipated!

Note that this section is just a preliminary overview so you know what's coming. Each section will be described in depth in the following chapters.

Recognizing the need for change, Unity set upon the path of redesigning the GUI system from the ground up. They have looked at what games have needed, what the community has built (even with the limitations and restrictions of not having access to Unity's core) and then sprinkled some fairy dust and hey presto, the new Unity UI system was born.
It has been a long, hard road with many bumps in the way, but the journey has begun (I say begun because the Unity UI journey does not end with Unity 4.6; it will continue to evolve into Unity 5 and beyond like many other components).

With a keen eye on the future, the new Unity UI system delivers on several core points, namely:

- **Extensibility**: Each and every control is a script and you can create new scripts to derive from them and create your own controls.
- **Uses sprites**: The new Unity UI system is predominately built on top of the new sprite system introduced in Unity 4.3. It has also however extended the sprite system with some new features as well.
- **Tight integration with GameObjects**: Each control is a GameObject in its own right with the same capabilities as any other game object including the ability to add further components and/or scripts.
- **Exposed events**: Each control has its own events which you can attach to and extend upon.
- **Tight integration with rendering and update loops**: Because they are GameObjects, you can even override the default rendering and updating of a control.
- **Animation**: Each control can be fully animated using the new Animator dope sheet and Mecanim. Some controls (such as the button) leverage Mecanim to do state-driven control. There are even animation specific events.
- **Screenspace AND WorldSpace**: Finally UI can be drawn in 3D in a performant way that doesn't involve hacking the project together or having to use PRO-only features.
- **FREE**: Unity UI comes as standard as part of the free license version of Unity (indie developers rejoice).
- **Open source**: Unity has made the source for the UI system open source and available for anyone to look at and even offer fixes / new suggestions, to coders delight.

**New layouts**

The layout features begin the story of the new Unity UI; they set the stage and ultimately define where controls will be drawn, in which direction they will be drawn, and also how they fit within a certain area.
Rect Transform

Introduced in 4.3 with the new Sprite functionality, the Rect Transform component provides a defining region for any 2D element within the Unity system. However, like most things in 4.6, it has received significant updates allowing more control over the area it manages as shown here:

It also sports a new button in the editor (called the Rect Tool) to edit and manage Rect Transform from the scene view, as shown in the following screenshot:

The Canvas

At the core of all Unity UI drawings, is the new Canvas control. It acts as the paint board for your Unity UI controls (and even other canvases), once rendered the canvas is drawn to the relative point in your scene.

Thanks to vast improvements this canvas can be drawn in basic 2D overlay mode (same as the legacy GUI system), in 2D camera space (adding perspective), or even directly in 3D world space like as any other traditional 3D object (such as a render target for UI) as shown here:
Groups
In the legacy GUI, groups were defined by the controls themselves; if you wanted to orientate multiple controls together in a particular fashion, you simply couldn't.

With the new Unity UI system, you can define several layout groups.

**Horizontal Layout Group**
A **Horizontal Layout Group** displays items in a horizontal line:

**Vertical Layout Group**
A **Vertical Layout Group** displays controls in a vertical line:

**Grid Layout Group**
A **Grid Layout Group** lays out controls according to a grid-based pattern of rows and columns:
Looking Back, Looking Forward

**Toggle Group**
A Toggle Group manages arranges toggle controls in to a group where only one can be active at a time (Like a Multiple choice selection where only one option can be chosen).

**Canvas Group**
A Canvas Group allows you to generically group child UI controls together and affect several common properties from the group, for example, the Alpha value of all child items:

![Canvas Group](image)

We'll cover these in more detail in Chapter 2, Building Layouts.

**Masking**
Recognizing the need for generic masking capabilities within the new Unity UI system (the ability to hide portions of UI within a certain region), they created a new Mask component. This can be added to any GameObject and any child objects outside the bounds of the parent GameObject would not be drawn, either partially or fully.

The Image control (highlighted later in the chapter) also includes an additional masking feature, when the Image Type property of an image is set to Filled; it gives several additional masking options to gradually bring the image into view. Just for reference now, we'll go into this in a lot more detail later.

**New controls**
Something old, something new, something borrowed, something blue.

Obviously when we look at a GUI, there is only so much that is really needed and Unity has recognized this. Starting with a fresh slate, they have looked at what it means to create a stunning UI and what you need to build one.
So with this fresh start, here are the new **Unity UI** controls:

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selectable</td>
<td>The selectable component is the base object for anything that needs interaction, and it basically turns any other component into a button. Note that it cannot be used on the same GameObject with something that is already selectable (button, toggle, and sliderbar).</td>
</tr>
<tr>
<td>Panel</td>
<td>This is a highlighted region, similar to the old <strong>Box</strong> control, mainly used to define groups when additional components are added. <strong>When Rect Transform</strong> alone is not enough. (Actually it’s an Image control with a preset image and set to full screen)</td>
</tr>
<tr>
<td>Text</td>
<td>If text is your aim, then this control is for you. It gives you text options like font, color, font style, and overflow. Can also be set to resize text to fit the container it is in.</td>
</tr>
<tr>
<td>Image</td>
<td>This is a basic image that can either use a sprite or material to display to the screen along with an optional color tint.</td>
</tr>
<tr>
<td>Raw Image</td>
<td>This is an alternate image display component that takes a texture instead of a sprite or a material and can define a color tint. Additionally has options to set the UV coordinates for the displayed image.</td>
</tr>
<tr>
<td>Button</td>
<td>If you need a big red (color optional) button to press, then this control is for you. The button has received the biggest overhaul by far with so many options it will likely take a chapter on its own to explain.</td>
</tr>
<tr>
<td></td>
<td>It also includes a new <strong>UnityEvent</strong> framework that allows you to create behaviors that can affect other objects or scripts directly from the editor.</td>
</tr>
<tr>
<td></td>
<td>You can even set different colors for different states of the button, swap out images, or if you wish, use Mecanim and the new animation system to animate the button between states.</td>
</tr>
<tr>
<td>Toggle</td>
<td>Switch it on, switch it off. The toggle takes the button behavior and extends it as a prime example of what extensibility features are in the new UI framework. It adds additional properties to the button framework to identify the checkbox graphic and a grouping option should you want to group checkboxes (using a Toggle group).</td>
</tr>
<tr>
<td>Scrollbar</td>
<td>It slices, it dices, it even slides as well. Your typical scroll bar with a handle, fully customizable with options to control the direction, minimum and maximum values, step size, and number of steps to slide between. Also includes the event system used for buttons for when the value changes.</td>
</tr>
<tr>
<td>Slider</td>
<td>This is a more advanced version of <strong>Scrollbar</strong> with fill options (for a filling cereal bar) so you can build the fanciest health bar with ease.</td>
</tr>
</tbody>
</table>
New UnityEvent system

Unity has always lacked a good and robust event system. Sure there are the SendMessage and BroadcastMessage functions, but these are really slow and can be expensive.

The new UnityEvent System is built around providing and handling all events with a scene, primarily for the new UI system; but like everything else in Unity 4.x, it is built to be extensible, and you can enhance your own components and scripts to expose themselves automatically to the event system and derive new events when things happen.

A note about UI events in the new Unity UI system: the interaction events all rely on raycasting to detect clicks, touches, hovering, and so on. It is very fast and efficient. However, if you build a new UI component that cannot react to raycasting, then it won't be recognized or respond to such interaction events.

Control extensibility

One very cool feature of the new UI system is that practically every component is a script, meaning it can be used as the base for any new scripts you create. You can either use them as is or even extend them further.

In Chapter 6, Working with the UI Source, we will cover the coding behind all these components including a walk through the open source library. I'll even throw in a load of examples and reusable components from the community and myself.

Animation

A core tenant of the new UI system (since it is built upon the core of the new 2D sprite system) was animation. Every single control or component of the new Unity UI system is able to be fully animated.

Not only that, but it also gives different modes for how properties can be animated, from static fixed values to dynamic values that will alter and update with the controls behavior.
Even the Asset Store has you covered

Through the dark of the beta process, which many developers participated in. Several worthy asset creators worked feverishly to update their craft to make use of the new UI system ready for release. Some existing projects were greatly enhanced and even some new toolkits.

Note that these recommendations are from my own experiences through the beta evolution of the new UI system. I've worked with (and against at times) a lot of things described in this section. I most certainly am not being paid to highlight these, they are just the best out there solving very unique gaps or limitations of the new UI system to date (however, Unity isn't sitting still, so these teams better keep up!).

No doubt I'll mention more of them on my blog as I find them.

I'll point these out again in some of the chapters where you will get the most use out of them or where they are relevant.

The most notable assets at the time of writing are the following.

**TextMeshPro ($65)**

This is a fantastic text management system that helps to bridge the gaps and the limitations in Unity's aging text rendering system. Unity themselves have noted on several occasions they want to rip out the text system and replace it with something better, until that happens, the best asset to help with text generation from within Unity (not just getting an artist to build lots of text assets) is *TextMeshPro*.

TextMeshPro has been around for quite a while now and has run with the leaders to get it updated for the new UI system but the author didn't stop there. TextMeshPro has gone far and beyond its humble text rendering beginnings to add such features as improved alignment/indentation and rich text support, even adding vertex animation for the generated text! (Just check out [http://bit.ly/TextMeshProAnimation](http://bit.ly/TextMeshProAnimation).)

GUI Generator ($40)

Building clean and efficient UIs in Unity have always been a strain (just look at this chapter!), this is where GUI Generator comes in, starting with the old GUI, then adapting to also handle NGUI and now the new Unity UI system. It's a quick and advanced tool to clean up the look and feel of your UI (no, it doesn't build your UI for you, it styles it!).

It also has many built-in effects you can add to your UI to make it stunning. Like TextMeshPro, the author has been working hard through the beta to get this great tool updated to handle the new UI elements and skin them to great effect.


MenuPage ($10)

Not every asset for the new UI has to be a big beast of a tool meant to save you hours, sometimes you just need something to get you off the ground and using the new UI with great effect. This is where MenuPage comes in.

Put simply it's a new asset on the store aimed at building menu systems using the new UI system automatically. They are fully configured, laid out effectively and offer some advanced features such as fading, transitions, and much more.

What's even better is that all the source code is there and is fully documented/commented, so you can learn from some of the best coders out there purveying their wares.


Summary

So now we have a deep appreciation of the past and a glimpse into the future. One thing you might realize is that there is no one stop shop when it comes to Unity; each feature has its pros and cons and each has its uses. There may still be cases when it is just easier to use the Unity legacy GUI to achieve an effect (still using the basic GUI.DrawTexture for a splash screen for example, although even that is very easy with the new UI system) provided you take the performance concerns on board with mobile devices. It all comes down to what you want to achieve and the way you want to achieve it; never dismiss anything just because it's old (except for GUIText and GUITexture, they are just really really old…)
In this chapter, we covered the following topics:

- The history of Unity legacy GUI
- Detailed walkthrough of the legacy Unity GUI
- Whistlestop tour of the new **Unity UI** system and what we can expect from this title

In the next chapter, we will start with the underlying framework and guts behind the new GUI system, namely:

- The new **Rect Transform** control (not just for Unity UI)
- The new **Rect Transform** component (and why it’s great)
- The **Canvas** control
- What Unity have done for scaling and resolution for UIs
- The all new and improved Event messaging system for the Unity UI system, complete with new shiny raycasting features and helpers

It promises to be a fun ride, and once we are through that, then in **chapter 3, Control, Control, You Must Learn Control** we can move on to actually building some UI and then placing it in your game scenes in weird and wonderful ways.

Now stop reading this and turn the page already!!
Where to buy this book

You can buy Unity 3D UI Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.