Learning NGUI for Unity

NGUI is a robust UI system both powerful and optimized. It is an effective plugin for Unity, which gives you the power to create beautiful and complex user interfaces while reducing performance costs.

This book is a practical tutorial that will guide you through creating a fully functional and localized main menu along with 2D and 3D in-game user interfaces. The book starts by teaching you about NGUI's workflow and creating a basic UI, before gradually moving on to building widgets and enhancing your UI. You will then switch to the Android platform to take care of different issues mobile devices may encounter. By the end of this book, you will have the knowledge to create ergonomic user interfaces for your existing and future PC or mobile games and applications developed with Unity 3D and NGUI.

Who this book is written for
If you are a Unity 3D developer who wants to create an effective and user-friendly GUI using NGUI for Unity, then this book is for you. Prior knowledge of C# scripting is expected; however, no knowledge of NGUI is required.

What you will learn from this book
- Build customizable 2D and 3D in-game user interfaces
- Use NGUI to save options, implement volume modification, launch animations, and change variables
- Enhance your UI using draggable panels, animations, content alignment, and localization
- Implement your own sprites and fonts to customize your user interface
- Move elements through code and handle collisions to create a game
- Create an effective and user-friendly cross-platform main menu
- Deal with different screen sizes and aspect ratios

Charles Pearson

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started with NGUI'
- A synopsis of the book’s content
- More information on *Learning NGUI for Unity*

**About the Author**

**Charles Pearson** has a Bachelor's degree in Game Design and Level Design from ISART Digital, a video-game school located in Paris, France.

This book is both an update and an upgrade of *NGUI for Unity*, the first book he wrote concerning this plugin.

With 4 years of experience as a game designer, level designer, and Unity developer at Cyanide Studio, Playsoft, and Airbus, Charles has worked on the PC versions of *Blood Bowl, Dungeonbowl*, and *Confrontation*. He has also worked on mobile games, such as *Space Run 3D, Oggy, and Psycho Gnomes*, and is currently working on PlayStation 4 and Xbox One projects as a freelance Unity 3D developer and game designer.

I would like to thank my friends and family for the precious support they provided me with during this project.

Special thanks to Amélie Beaudroit, User eXperience Specialist at Airbus, for the fabulous work she delivered for the last chapter of this book.

Finally, a big thank you to the editing team (Arun and Subho) and reviewers (Faris, Nicki, Ray, and Philip) for their efficient and precise feedback, which greatly helped bring this book's overall quality to the maximum.
Learning NGUI for Unity

This book is a beginner's guide to the Next-Gen UI kit, also known as NGUI. You might have heard about this Unity 3D plugin; it is popular among developers for its easy-to-use and effective What You See Is What You Get workflow.

NGUI provides built-in components and scripts to create beautiful 2D and 3D UIs for your projects, with most of the work done inside the editor.

With this book, you will gather the necessary knowledge to create functional UIs. The 10 practical chapters of this book will guide you through the creation of both a main menu and a simple 3D game, which are compatible with both PC and mobile platforms.

What This Book Covers

Chapter 1, Getting Started with NGUI, describes NGUI's functionalities and workflow. You will then import the plugin, create your first UI system, and study its structure.

Chapter 2, Creating NGUI Widgets, introduces you to your first widget and explains how to configure it. Then, it covers how to create a main menu.

Chapter 3, Enhancing Your UI, explains how to create draggable windows and scrollable text and use anchors intelligently. It also covers animations and localization with NGUI.

Chapter 4, C# with NGUI, introduces C# event methods used to create tooltips, tweens, and more through code, along with making the UI persistent through different scenes.

Chapter 5, Atlas and Font Customization, explains how to customize your UI using your own sprites and fonts, and we will modify the entire main menu's appearance.

Chapter 6, The In-game User Interface, introduces in-game 2D interface elements, such as displaying life gauges and player names above them, along with interactive buttons.

Chapter 7, 3D User Interface, covers how to add 3D widgets in the environment, such as a pause menu painted on the ground affected by the world's lighting effects.

Chapter 8, Going Mobile, teaches you how to prepare the project and fix mobile platform-specific issues so that our game runs perfectly on Android devices.

Chapter 9, Screen Sizes and Aspect Ratios, explains how you can ensure you'll have a beautiful user interface on all mobile devices even if they have different screen sizes.

Chapter 10, User Experience and Best Practices, discusses user interface design guidelines, user experience, and testing to help you improve your system's efficiency.
Getting Started with NGUI

The Next-Gen User Interface kit is a plugin for Unity 3D. It has the great advantage of being easy to use, very powerful, and optimized compared to Unity's built-in GUI system called UnityGUI.

Using this plugin, you can create main menus and in-game user interfaces. It also comes with useful generic methods and an event system that can come in handy.

In this first chapter, we'll import the plugin in a new project and create our first UI. After displaying texts and sprites, we'll take a look under the hood to understand NGUI's global structure and review its important parameters.

Overview
Before we actually start working with this plugin, let's have a quick overview of what its main functionalities are.

Licenses
NGUI is available in three different licenses:

- The NGUI Standard License costs $95. With this, you will have useful example scenes included. I recommend this license for a comfortable start—a free evaluation version is available, but it is limited, outdated, and not recommended.
- The NGUI Professional License priced at $200 gives you access to NGUI's GIT repository to access the latest beta features and releases in advance.
- A $2000 Site License is available for an unlimited number of developers within the same studio.

Now that we know the different licenses, let's make a quick comparison between NGUI and Unity's default UI system.
UnityGUI versus NGUI

With Unity's GUI, you must create the entire UI in code by adding lines that display labels, textures, or any other UI element on the screen. These lines have to be written inside a special function called every frame: `OnGUI()`. This is no longer necessary—with NGUI, *UI elements are simple GameObjects!*

You can create **widgets**—this is what NGUI calls labels, sprites, input fields, and so on—move them, rotate them, and change their dimensions using handles or the Inspector. Copy, paste, create prefabs, and every other useful feature of Unity's workflow are available.

These widgets are viewed by a camera and rendered on a layer you can specify. Most of the parameters are accessible through Unity's Inspector, and you can see what your UI looks like directly in the game window, without having to hit Unity's **play** button.

**Atlases**

Sprites and fonts are all contained in a large texture called **atlas**. With only a few clicks, you can easily create and edit your atlases. If you don't have any images to create your own UI assets, simple default atlases come with the plugin.

Using this system ensures that, for a complex UI window composed of different textures and fonts, the same material and texture will be used when rendering.

This results in **only one draw call** for the entire window. That, along with other optimizations, makes NGUI the perfect tool for working on mobile platforms.

**Events**

There also is an easy-to-use **event framework** written in C#. The plugin comes with a large number of additional components you can attach to any GameObject. These components can perform advanced tasks depending on which events are triggered: hover, click, input, and so on. Therefore, you might enhance your UI experience while keeping it simple to configure. **Code less, get more.**

**Localization**

NGUI comes with its own localization system, enabling you to easily set up and change your UI's language with the push of a button. All your **strings** are located in a .txt file.
Chapter 1

Shaders
Lighting, normal mapping, and refraction shaders are supported, which can give you beautiful results. Clipping is also a shader-controlled feature with NGUI, used for showing or hiding specific areas of your UI.

We’ve now covered what NGUI’s main features are; let’s see what we’ll create using this book.

Final build
You can download the final build of the Unity project we’ll create using this book, so that you can have an idea of the final result:

- Windows: http://goo.gl/dj9Lps
- Mac: http://goo.gl/4FnBC2

Launch the LearninNGUI.exe or .app file within the downloaded .zip archive; you’ll see that we have a localized main menu with different options, draggable windows, checkboxes, and so on.

By clicking on the main menu’s Play button, you’ll see that we have in-game 2D UI elements, such as the player’s, and a 3D score counter, and a Pause button. These UI widgets are customizable; after a long left-click on them, you can drag them around.

You can move your character with a left-click. The colored cubes are power sources, draggable with a long left-click. Right-click on them to display the elemental switch user interface; you can now select an element for it to switch to—either Fire, Ice, Lightning, or Water.

There also is an in-game 3D pause menu that lets you go back to the main menu or resume the game. This game will be fully compatible with mobile devices.

Once you have played around with the build, you can continue to the next section and see how you can import the NGUI plugin in Unity and start working to create this!

Importing NGUI
First of all, create a new Unity project with the name LearnNGUI.

Now that we have our new project, there are two different ways to import NGUI. Follow the Importing from the Asset Store section if you buy the plugin directly from the Asset Store.
If you already have the package on your disk, move on to the Importing from disk section.

At the time I am writing these lines, the latest version of NGUI is V3.7.7 with Unity 4.6.1. Depending on the version you work with, slight changes might occur.

Importing from the Asset Store

After buying the product from the Asset Store, download and import it by performing the following steps:

1. Navigate to Window | AssetStore.
2. Select your Downloads library. Once your downloads library is open, find the plugin within the list of packages. The result is shown in the following screenshot:

   ![NGUI: Next-Gen UI Tasharen Entertainment](image)

   - Download
   - Import

3. Now, click on the Download button (1) next to NGUI: Next-Gen UI.
4. Click on the Import button (2), and wait for a pop-up window to appear.

Ok. Once the Importing package pop-up window appears, skip the following section (reserved for users who already have the NGUI package on disk) and move on to the Package Importing section of the book.

Importing from a disk

If you already have NGUI's Unity package on your disk, follow these steps:

1. Navigate to Assets | Import Package | Custom Package...
2. Browse to your NGUI's .unitypackage file location.
3. Double-click on the .unitypackage file to import it.

When the Importing package window is displayed, continue to the next section.
Package importing

You have opted to import NGUI's package for Unity, and the Importing package window should be displayed. Follow these steps to finalize the importation:

1. Make sure everything is checked, and click on the Import button (3), as shown in the following screenshot:

![Importing package window]

2. Click anywhere on the toolbar (4) to update it. The NGUI menu appears as shown in the following screenshot:

![NGUI menu]

If the NGUI menu doesn't appear even after clicking on the toolbar, restart Unity, and it will appear.

We have now successfully imported the plugin into our project. Let's create our first UI!
Creating the UI Root
In order to display UI elements, we need a 2D UI system on the scene, called a UI Root.

Before we continue, please save your currently empty scene as Menu.unity.

Within our new empty scene that holds only the default Main Camera, navigate to NGUI | Create | 2D UI, as shown in the following screenshot:

By doing the above manipulation, NGUI has created the necessary components to display UI elements. Now, take a look at your Hierarchy view; you'll discover two new GameObjects as shown in the following screenshot:

We can briefly summarize their respective roles like this:

- **UI Root**: This scales itself to make child widgets at a manageable size in the scene view. It also handles the UI's scaling style and size.
• **Camera**: This is the camera object that views our UI elements and displays them over the 3D scene. It also handles widget interactions with the help of its attached **UI Camera** component.

We'll see how they work in detail later. Let's first display some text on the screen!

**Displaying text**

A **label** is used to display text on screen. Let's create one and see its parameters.

**Creating the Label widget**

In order to create a label, navigate to **NGUI | Create | Label**, as shown in the following screenshot:

You can also use the shortcut **Alt + Shift + L** to create a new label.

Look at your **Hierarchy** view; you'll see a new GameObject called **Label**, as shown in the following screenshot:

Great, we have a label in the scene. But nothing is displayed. Why? Because we have to choose with which **font** we want to render the text. Let's do that!
Selecting a font

Select this new Label GameObject. Now, in your Inspector view, you'll see that it has a UILabel component attached to it as shown in the following screenshot:

Currently, the Font field is set to None (UIFont). This means that no font is selected for this label. That's why most of its parameters are grayed out.

In order to select a font, click on the Font button (1) as shown in the preceding screenshot. By doing so, a new pop-up window appears, letting you choose a font by clicking on the Select button (1) in the next screenshot. For now, select the Arimo20 font, as shown in the following screenshot:

If no fonts are listed, or if you want to try other default fonts, hit the Show All button (2). This will search for fonts inside your Assets folder.

Once the font is selected, switch to the Game scene. You should see text like this:

It's great that we have our first NGUI text label. Now, we'll see how we can configure it.
UILabel parameters

The UILabel component attached to our new Label GameObject has some parameters. Let's see what they are and what they affect. The parameters are shown in the following screenshot:

1. You can choose between two font types:
   - **NGUI (Bitmap):** This gives the best performance. The font can be embedded in an atlas, thus saving a draw call. Compatible with pre-rendered effects. It is necessary to have different font sizes to avoid pixelation with large text.
   - **Unity (Dynamic):** This gives the best quality. These are flat, simple fonts. Increasing or reducing font size will never pixelate; the text will stay crispy. Will add one draw call for every different font displayed onscreen.

2. **Font Size:** Select your font size here.

   Increasing the font size now won't work since the **Overflow** (4) parameter is set to **ShrinkContent,** preventing it from enlarging the label's size. If you want to increase the font size, set **Overflow** to **ResizeFreely.**
3. **Text**: Enter the text to be displayed here.

4. **Overflow**: This shows the following four different behaviors when text exceeds the allowed space:
   - **ShrinkContent**: Text size is shrunk to fit the label's dimensions
   - **ClampContent**: Hide text that doesn’t fit the label’s dimensions
   - **ResizeFreely**: Label's dimensions adapt to the text
   - **ResizeHeight**: Adjust label's height only if text overflows

5. **Alignment**: Select the text alignment out of the following types:
   - **Left, Center, Right, or Justified**
   - **Automatic**: Takes the horizontal parameter of the Pivot

6. **Gradient**: This checkbox enables/disables the gradient effect on this label. **Top** and **Bottom** parameters let you choose the two colors for the gradient.

7. **Effect**: From the following, add a nice effect to your text label:
   - **None**: No effect—best performance.
   - **Shadow**: This has a shadow effect. This increases performance cost slightly by doubling the amount of triangles required to display the label.
   - **Outline**: This has an outline effect. This impacts performance by multiplying the number of triangles for this label by five; use with caution.
   - **Color**: This is the color of the selected effect.
   - **X, Y**: This shows how far in pixels the effect is from the text. These offsets are relative from the top-left corner of the label; positive values for X and Y will place the effect on the right and bottom respectively, while negative values will set it on the left and top of the label.
   - **Outline 8**: Outline effect that looks better on very small fonts. Impacts performance by multiplying the number of triangles for this label by 8; use only if Normal outline doesn’t look good.
8. **Float Spacing**: By default, a label's spacing is defined by integers to make sure the characters can be snapped to pixels and remain pixel-perfect. This option overrides the spacing to be defined with float values to let you have smaller steps. If you wish to create an expansion text animation by changing the X spacing from -10 to 0, it will be smoother with this option enabled.

9. **Spacing**: This is the distance in pixels between each character for both X and Y.

10. **MaxLines**: This is the maximum number of lines. Set it to 0 for unlimited.

11. **BBCode**: This enables/disables BBCode in labels. For example:

    - **[FF0000]**Red [-]Back to previous color
    - **[b]**Bold[/b], **[s]**Strikethrough[/s], **[u]**Underline[/u]
    - **[url=http://google.fr/]**[u]**Link[/u][/url]

    ![Icons]
    You can also configure **Symbols** (smileys). The **Colored** value simply makes them receptive to BBCode colors.

12. **Color Tint**: Set here a global color tint for the entire label's characters.

    ![Icons]
    In order to have a nice result with gradients, I recommend that you always leave the gradient colors as grayscale colors and change the label's color tint to give color to it. If specific words must ignore the label's color tint defined in the Inspector, you can simply use **[c]**keyword[/c]** as BBCode.

We've reviewed the label's parameters. Let's configure them so they look good!
Configuring the label

Before we study more parameters, let’s configure this label correctly for later! Please set its parameters as shown in the following screenshot (changes are highlighted in the screenshot):

1. In order to find the Coalition font, click on the Font button next to the font field, and then click on the Show All button in the font selection window.

2. **Effect**: Set a Shadow effect with X and Y values of 3. Leave the shadow’s color black.

3. Set these values for the **Color Tint**: (R: 255, G: 170, B: 70, A: 255).

Ok, our label is configured. Below these parameters, you might have noticed more **Widget** parameters. We are now ready to see what they are.
Widget parameters

You can see more parameters below the Color Tint field. These are the widget's global parameters from the UIWidget class—they are common to most NGUI widget components, such as UILabel, UISprite, and UScrollbar, since they are derived from it:

1. **Pivot**: This is the widget's pivot point. This represents the static point from which the widget will resize itself:
   - **Left set of buttons**: Horizontal pivot point
   - **Right set of buttons**: Vertical pivot point

   As you change the pivot point using the sets of buttons, its Transform position updates relatively to maintain its current world position.

2. **Depth**: Change this value to render a widget over another. The widget with the highest value is rendered on top. You can use the Back and Forward buttons or simply enter a number. Negative values are accepted.

   If you need a widget over another, make sure they don't have the same Depth value; you might not have the one you want rendered over the other.

3. **Size**: This shows the widget's dimensions in pixels. It is grayed out here because the label's Overflow parameter is set to ResizeFreely and is automatic.

   You should use the Size(3) parameter and keep the widget's scale at (1, 1, 1) if you want to keep your UI pixel-perfect and avoid bad surprises!
4. **Aspect**: Select any of the following desired aspect ratio lock behaviors:
   - **Free**: You are free to modify the widget's height and width as you wish—no aspect ratio is imposed
   - **BasedOnWidth**: You are free to modify the width only—the height will adjust automatically to keep the current aspect ratio
   - **BasedOnHeight**: You are free to modify the widget's height only—the width will adjust to keep the current aspect ratio

Below these parameters, you can see the **Anchors** group. We'll deal with them later since we need to have more widgets to really understand how they work.

We can now see how to display images!

**Displaying a sprite**

Now that we have displayed text, let's display a **sprite**.

**Creating the sprite widget**

Make sure you have nothing selected, and navigate to **NGUI | Create | Sprite**, as follows:

The newly created widget will be nested as a child of the current selection. To avoid having our new sprite as a child of our **Label**, make sure you have nothing selected—**UI Root** will be the default parent.

You can also use the shortcut **Alt + Shift + S** to create a new sprite.
Now, take a look at the following **Hierarchy** view; you’ll see a new GameObject called **Sprite**:

![Hierarchy view](image)

Great, we have a sprite in the scene. Let's see its Inspector parameters.

**UISprite parameters**

Let's define some terms as we review **UISprite**'s parameters:

![UISprite parameters](image)

1. **Atlas**: An atlas is an image file containing our sprites and bitmap fonts. Assembling them all together in one large texture enables NGUI to render an entire UI in only one single draw call! That field determines which atlas is going to be used to display this sprite.

2. **Sprite Type**: Select one of these five different behaviors:
   - **Simple**: This displays a simple image.
   - **Sliced**: This is used for **nine-slice** sprites, letting you resize sprites without stretching the corners. We'll detail and try this shortly.
   - **Tiled**: This is used for **tiled** sprites for patterns that can be repeated indefinitely.
Getting Started with NGUI

- **Filled**: This is used for progress bars. This hides the sprite partially, depending on a fill amount value and direction.
- **Advanced**: This allows you to choose a different sprite type for each border and center.

3. **Flip**: This flips the sprite in one of the following directions you define:
   - Nothing, Horizontally, Vertically or Both

4. **Color Tint**: This is the sprite's color tint.

Right now, nothing is displayed because we haven't chosen which atlas and sprite we want to use.

**Configuring the sprite**
We have reviewed the parameters of UISprite. Let's configure it to display something.

**Selecting an atlas**
We must select an atlas. We'll use an example atlas that comes with NGUI.

Select our Sprite GameObject, and click on the Atlas (1) button showed in the previous screenshot. The following window appears:

![Select an Atlas Window](image)

Select **Fantasy Atlas (2)**. If the atlas is not listed, click on the Show All (3) button of the atlas selection window, and it will appear along with other example atlases.
Selecting a sprite

Now that we have selected an atlas, we can select which sprite to render, as shown in the following screenshot:

Perform the following steps:

1. Click on the Sprite (1) button to open the sprite selection window. Within this new window, select the Dark sprite by double-clicking on it.
2. Change the sprite Type (2) to Sliced.
3. Click on Color Tint (3) and set: \{R: 255, G: 180, B: 0, A: 255\}.

Your Game view should now look like this:

Ok. We now have a square sprite. But we have a problem: it's rendered over our text label. That's because the sprite's Depth value is higher than the label's.

The new sprite's Depth value is higher than the label's because NGUI automatically sets a new widget's Depth to the highest to make sure it is displayed over previous ones.
Set the following widget parameters for our new sprite in order to have this result, as follows:

1. **Depth**: The label's depth is at 0. Set this one to -1 so that it displays behind.
2. **Size**: Set it to 400 x 80 to create a box around our **Main Menu** label.

Great, our first sprite is correctly configured! Before we move on, we can try out the different sprite types.

**Sprite types**

With NGUI, we have four basic types of sprites:

- Sliced
- Simple
- Tiled
- Filled

Now that we have a sprite onscreen, let's try them and see their parameters.

**Sliced**

Let's start with the **Sliced** type of sprites, since our sprite is already configured like that.
Select our UI Root | Sprite GameObject in the Hierarchy view. Look at the Preview window at the bottom of Inspector:

You can see that the dark sprite's size is only 17 x 17 pixels. We have stretched it up to 400 x 80, and it still looks good — no pixelation or stretched corners!

That's because it is a nine-slice sprite: the four dotted lines in the preceding screenshot represent the slicing lines. Corners always remain at the same size, while other parts of the sprite stretch to cover the demanded area. Sliced sprites are thus very useful to create windows, borders, title bars, and more.

**Simple**

A Simple sprite is used to display an icon or a specific image that does not require slicing. In order to avoid pixelation, you shouldn't resize it too much over its original file size.

If you switch our sprite's Type from Sliced to Simple, slicing parameters are ignored, and our 17 x 17 sprite is stretched and looks terrible, as shown in the following screenshot:

Ok. We can now move on to tiled sprites.
Getting Started with NGUI

Tiled

A Tiled sprite is a pattern that can be repeated indefinitely. In order to illustrate it effectively, select our Sprite GameObject and follow these steps:

2. Set the Type parameter of UISprite to Tiled.
3. Change its Sprite to the Stripes 1 sprite.

Within the Game view, you might notice that our background is now diagonal orange stripes. The pattern is repeated and it looks good compared to a simple stretched sprite.

Here's a comparison:

![Simple vs Tiled](image)

In the preceding image, the same sprite is used. On the left-hand side, the sprite Type is set to Simple: the sprite is simply stretched and pixelated. On the right, the sprite Type is set to Tiled: the pattern is repeated and not stretched.

Now that we've seen how tiled sprites work, let's talk about filled sprites.

Filled

A Filled sprite is displayed partially depending on a FillAmount value between 0 and 1. Set the Type parameter of our UISprite to Filled. Ok. Now, we can review the three new parameters, as follows:
The following parameters are specific to filled sprites:

1. **FillDir**: Select one of the following five fill directions:
   - **Horizontal**: The sprite is filled horizontally. This is usually used for progress bars and life gauges.
   - **Vertical**: The sprite is filled vertically.
   - **Radial90**: The sprite is filled around a 90-degree angle.
   - **Radial180**: The sprite is filled around a 180-degree angle, resulting in a car windshield wiper effect.
   - **Radial360**: The sprite is filled around a 360-degree angle, resulting in a clock or timer effect.

2. **FillAmount**: This floats between 0 and 1 and defines how much of the sprite is visible. The mask's shape depends on the selected FillDir (1).

3. **InvertFill**: Check it to invert the sprite's fill direction.

You can visualize the fill effect by modifying the value of Fill Amount(2). If Fill Dir is set to Radial360 with Fill Amount around 0.55, the output will look like this:

![MAIN MENU](image)

You can try out the different fill directions before we explain how UICamera and UIPanel work.

Before we continue to the next section, please set back our sprite's Type parameter to Sliced, with the Dark sprite from Fantasy Atlas.

**Under the hood**

When we added UI Root at the beginning of the chapter, it created both UIRoot and Camera GameObjects.

Now that we have a sprite and a label within our scene, we can properly understand what the purposes of these two mysterious GameObjects are.

Select our UI Root GameObject in the Hierarchy view. You can see that the UIRoot and UIPanel components are attached to it. Let's see what UIRoot's purpose is.
UIRoot

The `UIRoot` component scales widgets down to keep them at a manageable size inside the scene view. It also handles the different scaling styles.

Your Game view's aspect ratio might be set to Free Aspect or another value. For us to have the same results during the explanations to come, set it to 16:9, like this:

1. Display your Game view.
2. Click on the current aspect ratio (1), which is Free Aspect here.
   
   Select the 16:9 (2) aspect ratio. Ok. Now that we have a 16:9 aspect ratio for our Game view, we can continue.

Scaling styles

Select our UI Root GameObject in the Hierarchy view. The first parameter of its attached `UIRoot` component is Scaling Style. It defines how your UI reacts when the screen resolution is changed. There are three different scaling styles available:

1. Flexible: The UI is in pixels and remains pixel-perfect. A 300 x 300-pixel image always takes 300 x 300 pixels onscreen regardless of the resolution. No stretching or scaling occurs.
2. Constrained: The UI is not pixel-perfect. An image taking 30 percent of the screen will always take 30 percent of the screen, regardless of the resolution. The UI is scaled up or down to fit the screen's height, width, or both, depending on the parameters you choose.
3. ConstrainedOnMobiles: This is the Flexible mode on the desktop and Constrained mode everywhere else.

The `UIRoot` component has different parameters depending on the selected Scaling Style. Let's start with the default Flexible scaling style.
Flexible
This mode ensures that your UI elements remain at the same size in pixels (pixel-perfect) regardless of the resolution or Dots Per Inch (DPI). Here's an illustration of that:

![Illustration of Flexible mode](image)

The 320 x 240 widget shown in the preceding figure (W1) appears small on a 1920 x 1080 screen (S1) and large on a 640 x 480 screen (S2) because it's in Flexible (pixel-perfect) mode.

With the preceding illustration, if your screen is smaller than 320 x 240, your widget is not entirely visible. That's where the other parameters shown in the following screenshot come in:

![Screen parameters](image)

The parameters are as follows:

1. **Minimum Height**: This is an important parameter to keep our UI from being cropped on small screens. When the screen's height is below this value, your UI is scaled down to fit. It will then be as if Scaling Style was set to Constrained with Content Height set to the screen's height.

2. **Maximum Height**: This is similar to Minimum Height, but for the screen's maximum height. When the screen's height is above this value, your UI is scaled up to fit.
3. **Shrink Portrait UI**: If your game or app can change orientation (landscape or portrait), check this option—the UI will be scaled down to fit the screen.

4. **Adjust by DPI**: Enabling this option will make your pixel-perfect UI take the screen's DPI into account. In practice, NGUI will believe that these two screen configurations are identical:
   - Resolution: 1280 x 720—200 DPI
   - Resolution: 1920 x 1080—400 DPI

Here's a practical example to illustrate the pixel-perfect flexible behavior:

1. Select our Sprite GameObject.
   - Change its Size to 425 x 240
2. Select our UI Root GameObject.
   - Set the attached **Minimum Height** parameter of **UIRoot** to 240

Now, resize your Game view window to a larger screen: when it's more than 240 pixels high, the sprite's always displayed at 320 x 240 pixels. It's pixel-perfect and crisp.

Resize the Game view to a small size: when it's less than 240 pixels high, the UI is scaled down to avoid it being cropped. That's the purpose of the **Minimum Height** parameter.

The same principle applies to **MaximumHeight**.

**Constrained**

The **Constrained** scaling style is the opposite of pixel-perfect: you set **Content Width** and **Content Height** for the virtual screen, and your UI will rescale itself up or down to fit.

In practice, it means that a widget taking 100 percent of the screen will always take 100 percent of the screen regardless of the resolution. Let's try it now. The following screenshot shows the **Constrained** scaling style:
Select our **UI Root** GameObject in the **Hierarchy** view. For its attached **UIRoot**:

1. Set **Scaling Style** (1) to **Constrained**.
2. Change **Content Width** (2) to **320**.
3. Set **Content Height** (3) to **240**.
4. Check the **Fit** option for **Content Height**.

Our screen is now always considered as having a height of 240, which is our sprite's height; it is now taking all the screen.

Resize the **Game** view to a large view. You'll see that, at all screen sizes and resolutions, our sprite always takes 100 percent of the screen's height (and width, if it's a 16:9 resolution). That's the **Constrained** mode: the UI is scaled proportionally to the screen's height based on the referenced **Content Height**.

### ConstrainedOnMobiles

By setting your scaling style to **ConstrainedOnMobiles**, your UI will be **Flexible** (pixel-perfect) on desktop and **Constrained** everywhere else.

### Configuration

For the purpose of this book, we'll set up a simple configuration that will enable us to have the same result on all screens. We'll set our scaling style to **Constrained**, with **Content Height** of **1080**.

This configuration ensures our UI looks beautiful on 1920 x 1080 screens; it is scaled up or down on higher or lower resolutions and still look great.

Select our **UI Root** GameObject in the **Hierarchy** view. For its attached **UIRoot** component, perform the following steps:

1. Set **Scaling Style** to **Constrained**.
2. Change **Content Width** to **1920**.
3. Set **Content Height** to **1080**.
4. Check the **Fit** option for **Content Height**.

The 1920 x 1080 resolution has a 16:9 aspect ratio. We'll make sure we only authorize this aspect ratio to avoid having cropped screen sides. Perform the following steps:

1. Navigate to **Edit | Project Settings | Player**.
2. In the **Player Settings** (Inspector view), click on **Resolution and Presentation**.
3. Click on **Supported Aspect Ratios** (the last one, at the bottom).
4. Uncheck all aspect ratios, except the **16:9**.

Now that we have understood and configured our **UIRoot**, let's talk about **UIPanel**.

### UIPanel

Select our **UI Root** GameObject. Below the **UIRoot** component, you'll find **UIPanel**.

**UIPanel** acts like a widget container—it creates the geometry. All child widgets will be rendered by this panel in one single draw call, if they all use the same atlas.

You might have more than one panel if you want to separate your UI, but keep in mind that a new panel equals a supplementary draw call.

Wherever **UIPanel** is attached, a kinematic **Rigidbody** is also automatically added; don't worry, it's an optimization trick from NGUI.

The **UIPanel** component has three basic parameters:

![UIPanel (Script) parameters](image)

These parameters are as follows:

1. **Alpha**: This is the panel's global alpha value. It affects all child widgets and panels.
2. **Depth**: This is used to display an entire panel (and its child widgets) in front or behind another. It works in the same way as the other widgets' **Depth** parameter.

Keep in mind that panel depth takes precedence over widget depth. In order to understand this clearly, consider the following situation:

- PanelA has a **Depth** of 0 and holds WidgetA with a **Depth** of 10.
- PanelB has a **Depth** of 1 and holds WidgetB with a **Depth** of 0.

In the previous situation, even though WidgetA has a higher **Depth** value, it will be displayed behind WidgetB because PanelB has a higher **Depth** value than PanelA.
3. **Clipping**: Clipping consists in displaying only part of the panel using a mask that can be either a rectangle or a texture. Choose one of the four available clipping options:

- **None**: No clipping occurs—the entire panel is visible.
- **Texture Mask**: You can select a texture to use as a mask to clip the panel. The Offset and Center parameters are available to adjust the texture's position, and the Size parameter lets you define a width and height for the texture mask.
- **SoftClip**: Four new parameters appear to customize a clipping rectangle. Content outside this rectangle is not displayed. The Softness parameter is an area that smoothly fades out its contents.
- **ConstrainButDon'tClip**: You can define a clipping rectangle, but no clipping occurs. Can be useful if you need to delimit an area without hiding its contents.

You can set the **Clipping** to **None** and leave default values. We'll talk about the advanced parameters when we need them. Now, let's talk about the camera.

**The camera system**

Select our **UI Root | Camera GameObject**. You can see that it has both a Unity orthographic **Camera** and a **UICamera** component.

**Orthographic Camera**

The orthographic (2D) **Camera** is used to view our widgets and render them over the **Main Camera** (3D) used to render the actual scene.

This is achieved by setting the **rendering depth** of **Main Camera** to -1, while the camera used to render NGUI widgets has a depth of 0 with the **Clear Flags** parameter set to **Depth Only**.

All these steps were executed automatically on the creation of a new 2D UI. Hence, we can see the widgets we created, rendered over **Main Camera**.

Here, we're talking about camera rendering depth, not NGUI widget **Depth** value. The camera's depth defines its rendering order. The camera with the highest depth will be rendered over the others.
The **culling mask** of the orthographic camera must be set to only display our 2D UI layer. Let's create one right now.

Click on the button to change our GameObject's **Layer (1)**, and select **Add Layer... (2)**. This is shown in the following screenshot:

![Inspector view showing Layer menu](image)

The Inspector view now displays the **Layer** menu. Next to **User Layer 8**, type in our new layer's name: **2DUI**.

Now, reselect our **UI Root** GameObject in the **Hierarchy** view. In the Inspector view, you can notice that our **UIRoot** component and all its children are on the **2DUI** layer, and the **Culling Mask** parameter of **Camera** is set to **2DUI** only.

Now that we have our separate layer to hold our 2DUI, let's see what **UICamera** is.

**UICamera**

We'll now explain the purpose of **UICamera**, review its parameters, and configure it.

**Purpose**

This component sends out messages concerning events triggered on UI elements viewed by the camera it's attached to. For example, events such as **OnClick()** and **OnHover()** can be triggered on a button when the player clicks or hovers it.

You might have multiple cameras if needed. Here's an example with three different cameras:

1. 3D perspective main camera that renders the game. Layer: Default.
2. Orthographic camera for in-game 2D UI elements. Layer: 2DUI.
3. Separate 3D perspective camera used for 3D menus. Layer: 3DUI.
The afore mentioned cameras used for UI (cameras 2 and 3) need the UICamera component to interact with UI elements. The component can be added to the game's main camera (camera 1) if you want your 3D in-game objects to receive NGUI events too—they must have a Collider component attached to them.

The in-game objects you wish to also receive NGUI events require Collider because these events are triggered by raycasts. Therefore, if the object has no Collider attached to it, the raycast will simply go through it.

Ok, now that we know the purpose of UICamera, we can review its parameters.

**Parameters**

These are the 11 parameters of UICamera:

- **Event Type**: 3D UI
- **Event Mask**: Everything
- **Events go to...**: NGUI/ Canvas
- **Debug**: False
- **Allow Multi Touch**: False
- **Sticky Tooltip**: False
- **Tooltip Delay**: 1 seconds
- **Raycast Range**: Unlimited
- **Mouse**: True
- **Touch**: True
- **Keyboard**: True
- **Controller**: True
- **Mouse Drag**: 4 pixels
- **Mouse Click**: 10 pixels
- **Touch Drag**: 40 pixels
- **Touch Tap**: 40 pixels
- **Horizontal**: Horizontal
- **Vertical**: Vertical
- **Scroll**: Mouse ScrollWheel
- **Submit 1**: Return
- **Submit 2**: Joystick Button 3
- **Cancel 1**: Escape
Let's see what the above parameters correspond to:

1. **Event Type**: Select which event type this camera will send. Types that concern the UI sort events depending on their **Depth** value, whereas in World types events are sorted depending on their distance from the Camera component:
   - 3D World: This is used to interact with 3D-world GameObjects. Select it if this is your 3D game's main camera.
   - 3D UI: Option used for interaction with 3D user interfaces.
   - 2D World: Select this option if this is your 2D game's main camera.
   - 2DUI: This is used for interacting with the 2D UI.

2. **Event Mask**: Select the layer on which the event raycast is triggered. For example, setting it to **Everything** will result in all objects with Colliders receiving these events.

3. **Events go to...**: Here, you can select whether the objects you wish to receive events require either Colliders or Rigidbodies.

4. **Debug**: This enables or disables debug mode. This option is useful when you have unwanted behavior. When enabled, the currently hovered object is displayed in the top-left corner of the screen.

5. **Allow Multi Touch**: This enables or disables simultaneous touches. This is mandatory if you want to use pinch-to-zoom or other such gestures on mobile platforms.

6. **Sticky Tooltip**: This enables or disables the sticky tooltip option:
   - Enabled: The tooltip disappears when the mouse moves out of the widget's collider
   - Disabled: The tooltip disappears as soon as the mouse moves

7. **Tooltip Delay**: This defines the required stationary time in seconds before the widget's tooltip is displayed.

8. **Raycast Range**: A raycast is an invisible ray that is cast from one point towards a specific direction and is stopped if it encounters another object. The UICamera uses raycasts from the mouse or touch position towards the forward direction of Camera to detect collisions and handle events. You might set the range of this raycast if you need to limit the interaction to a certain range. The default -1 value implies that the raycast's range will be as far as Camera can see, defined by its **Far Clipping Plane** parameter.
9. **Event Sources**: You can specify which events this camera listens to:
   - **Mouse**: This is used for mouse movements: left/right/middle click, and scroll wheel
   - **Touch**: This is used for touch-enabled devices
   - **Keyboard**: This enables keyboard input. It uses the `OnKey()` event
   - **Controller**: This enables support for joysticks or game controllers

10. **Thresholds**: These values come in handy when you want to specify the minimum values before a particular event is triggered:
   - **Mouse Drag**: When a mouse button is pressed (the `OnPress()` event is triggered), this value determines how far in pixels the mouse must move before it is considered a drag, and sends `OnDrag()` events to the dragged object
   - **Mouse Click**: When a mouse button is pressed (the `OnPress()` event is triggered), this value determines how far in pixels the mouse can travel before the button release has no effect (the `OnClick()` event is not triggered)
   - **Touch Drag**: This is the same as **Mouse Drag**, but for touch-based devices
   - **Touch Tap**: This is the same as **Mouse Click**, but for touch-based devices

11. **Axes and Keys**: These parameters let you assign Unity input axes and keys to NGUI's input system.
   - **Horizontal**: This is the input axis for horizontal movement (the left and right key events)
   - **Vertical**: This is the input axis for vertical movement (the up and down key events)
   - **Scroll**: This is the input axis for scrolling
   - **Submit 1**: This is the primary keycode for validation
   - **Submit 2**: This is the secondary keycode for validation
   - **Cancel 1**: This is the primary keycode for cancel
   - **Cancel 2**: This is the secondary keycode for cancel

You can edit Unity input axes at any time by navigating to **Edit | Project Settings | Input**.
Now that we have reviewed the parameters of UICamera, we can configure it for our project.

**Configuration**
We will now configure our UICamera component for this specific camera so that it suits our project and our future UI. Select our UI Root | Camera GameObject, and then:

1. Set the **Event Type** parameter to **3DUI** because this camera will be used for both 2D and 3D UI interactions.
2. Set the **Event Mask** to the **2DUI** layer only since our UI will reside on it.
3. Set **Events go to...** to the **Colliders** value because our widgets will have Colliders attached to them.

Good. We are now ready to create more interactive user interface elements.

**Summary**
During this first chapter, we understood the basics of NGUI. After importing the NGUI plugin, we created our first label and sprite and reviewed their parameters. We also created our own **2DUI** layer for our UI to reside on.

Finally, we analyzed the elements that were created automatically for us by NGUI. After reviewing their parameters, we can summarize their roles as follows:

- **UI Root**: This holds our UI and handles ScalingStyle: Flexible (pixel-perfect) or Constrained Constrained (stretched UI).
- **UIPanel**: This renders our widget's actual geometry, with or without clipping.
- **Camera**: This views the widgets and renders them over Main Camera. Its attached UICamera sends event messages to widgets using raycasts.

We are now ready to move on to *Chapter 2, Creating NGUI Widgets*, in which we'll create buttons, pop-up lists, input fields, checkboxes, and more!
Where to buy this book
You can buy Learning NGUI for Unity 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.