UNITY CANVAS AND SCENE MANAGEMENT

MODULE 3

3.1

Summary

User interface – it's a complex thing but in the video it is demonstrated how to add a button on the screen.

We open Scenes folder under assets and add button from UI section. Then we move to editing our UI. At first we select it in Canvas Scaler to move with screen size, otherwise, when we change the size of the screen our UI button will not adjust to new dimensions. With the option turned on, our button scales properly.

We can edit the UI and adjust its size, and drag it in the screen. We can create objects for our UI's for example we can add text to our button so it will move along with the button itself.

Key Points

- User interface is a complex part of the project.
- UI elements can be added under assets tab.
- We should make our UI elements such as a button move with screen size, so turn on Canvas Scaler option.
- We can add text to a button and make it move with the button itself.

3.1.1 UI

Unity provides the following user interface (UI) toolkits for creating in either the Unity Editor or in a game or application:

UIElements: User Interface Elements (UIElements) is a retained-mode UI toolkit for developing user interfaces in the Unity Editor. UIElements is based on recognized web technologies and supports stylesheets, dynamic and contextual event handling, and data persistence.

Unity UI: Unity User Interface (Unity UI) is a simple UI toolkit for developing user interfaces for games and applications. Unity UI is a GameObject-based UI system that uses components and the Game View to arrange, position, and style the user interface. You cannot use Unity UI for user interfaces within the Unity Editor.

IMGUI: Immediate Mode Graphical User Interface is a code-driven UI toolkit that is mainly intended as a tool for developers. IMGUI uses the OnGUI function, and scripts that implement the OnGUI function, to draw and manage its user interface. IMGUI is used for creating in-game debugging displays, custom inspectors for script components, and editor windows or tools that extend the Unity Editor. It is not recommended for game or application user interfaces.

Selecting a UI toolkit

You should select a UI toolkit based on your answers to the following questions: * Are you developing for a game or application, or a tool or extension to the Unity Editor? * If you are developing for a game or application, are you shipping the UI with the game or application?

	Runtime dev UI	Runtime game UI	Unity Editor
UIElements	TBD	TBD	✓
Unity UI	✓	✓	not available
IMGUI	for debugging	not recommmended	✓

UIElements is still in active development. It is poised to become the recommended UI toolkit for both in-game and Unity Editor UI development. Until then, there are some features that are available in Unity UI and IMGUI that have yet to be added to UIElements.

In addition, changes to UIElements might not be backported to previous versions of Unity. If you upgrade, you will also need to upgrade your interface from previous Unity versions.

3.1.2 UIElements Developer Guide

The goal of this guide is to help you take advantage of UIElements by describing the concepts behind the framework and by providing you with a clear explanation of how to build an interactive user interface with UIElements.

The UIElements Developer Guide is split into the following sections:

The Visual Tree: Holds all the visual elements in a window. The visual tree is an object graph made of lightweight nodes referred to as visual elements. See this topic for information on the visual tree, visual elements, connectivity, drawing order, and more.

The Layout Engine: Positions visual elements based on layout and styling property. See this topic for more onthe layout engine.

The UXML format: Defines the structure of the user interface. See this topic for details on writing, loading, anddefining Unity eXtensible Markup Language (UXML) templates.

Styles and Unity style sheets (USS): Defines the style properties that set the dimensions and appearance of visual elements. See this topic for details on USS, its syntax, and its differences compared to Cascading Style Sheets (CSS).

The Event system: Communicates user interactions to visual elements. See this topic for details on how to use the event dispatcher, event handler, event synthesizer, and event types for handling user interactions with UIElements.

Built-in controls: See this topic for a list of standard controls that are built into UIElements.

Binding: Links a property to the visual control that modifies the value of the property. See this topic for more information on how to bind a property to a control.

Supporting IMGUI: See this topic for information on how to use IMGUI code with UIElements.

ViewData persistence: Persists UI-specific state data. See this topic for information on how to store and retrieve state data after a domain reloads or when the Editor restarts.

3.1.3 The Layout Engine

UIElements includes a layout engine that positions visual elements based on layout and styling properties. The layout engine is the Yoga open source project that implements a subset of Flexbox: a HTML/CSS layout system.

To get started with Yoga and Flexbox, consult the following external resources:

Yoga official documentation: the mapping of properties is almost 1-to–1
CSS-Tricks guide to Flexbox: most properties are supported with some minor differences

By default, all visual elements are part of the layout. The layout has the following default behaviours:

- A container distributes its children vertically.
- The position of a container rectangle includes its children rectangles. This behaviour can be restricted by other layout properties.
- A visual element with text uses the text size in its size calculations. This behaviour can be restricted by other layout properties.

UIElements include built-in controls for standard UI controls such as button, toggle, text field, or label. These built-in controls have styles that affect their layout.

The following list provides tips on how to use the layout engine:

- Set the width and height to define the size of an element.
- Use the flexGrow property (in USS: flex-grow: <value>;) to assign a flexible size to an element. The value of the flexGrowproperty acts as weighting when the size of an element is determined by its siblings.
- Set the flexDirection property to row (in USS: flex-direction: row;) to switch to a horizontal layout.
- Use relative positioning to offset an element based on its original layout position.
- Set the position property to absolute to place an element relative to its parent position rectangle. In this case, it does not affect the layout of its siblings or parent.

3.1.4 Styles and Unity Style Sheets

Each VisualElement includes style properties that set the dimensions of the element and how the element is drawn on screen, such as backgroundColor or borderColor.

Style properties are either set in C# or from a style sheet. Style properties are regrouped in their own data structure (IStyleinterface).

UIElements supports style sheets written in USS (Unity style sheet). USS files are text files inspired by Cascading Style Sheets (CSS) from HTML. The USS format is similar to CSS, but USS includes overrides and customizations to work better with Unity.

This section includes details on USS, its syntax, and its differences when compared to CSS.

Definition of a Unity style sheet

The fundamental building blocks of a Unity style sheet (USS) are as follows:

- A USS is a text file recognized as an asset. The text file must have the .uss extension.
- A USS only supports style rules.
- A style rule is composed of a selector and a declaration block.
- The selector identifies which visual element the style rule affects.
- The declaration block, enclosed by curly braces, contains one or more style declarations. Each style declaration is comprised of a property and a value. Each style declaration ends with a semi-colon.
- The value for each style property is a literal which, when parsed, must match the target property name.

The general syntax of a style rule is :

```
selector {
  property1:value;
  property2:value;
}
```

Attaching USS to visual elements

You can attach a Unity style sheet (USS) to any visual element. Style rules apply to the visual element and all of its descendants. Style sheets are also re-applied automatically when necessary.

Load StyleSheet objects with standard Unity APIs such as AssetDatabase.Load() or Resources.Load(). Use the VisualElement.styleSheets.Add() method to attach style sheets to visual elements.

If you modify a USS file while the EditorWindow is running, style changes are applied immediately.

The process of style application is transparent to a developer using UIElements. Style sheets are re-applied automatically when needed (hierarchy changes, stylesheet reload).

Style matching with rules

Once a style sheet is defined, it can be applied to a UIElements tree of visual elements.

During this process, selectors are matched against elements to resolve which properties are applied from the USS file. If a selector matches an element, the style declarations are applied to the element.

For example, the following rule matches any Button object:

```
Button {
  width: 200px;
}
```

VisualElement matching

UIElements use the following criteria to match a visual element with its style rule:

- Its C# class name (always the most derived class)
- A name property that is a string
- A class list represented as a set of strings
- The ancestry and position of the VisualElement in the visual tree.

These traits can be used in selectors in the style sheet.

If you are familiar with CSS you can see the similarity with the HTML tag name, the id attribute and class attribute.

3.1.5 Canvas

The Canvas is the area that all UI elements should be inside. The Canvas is a Game Object with a Canvas component on it, and all UI elements must be children of such a Canvas.

Creating a new UI element, such as an Image using the menu GameObject > UI > Image, automatically creates a Canvas, if there isn't already a Canvas in the scene. The UI element is created as a child to this Canvas.

The Canvas area is shown as a rectangle in the Scene View. This makes it easy to position UI elements without needing to have the Game View visible at all times.

Canvas uses the EventSystem object to help the Messaging System.

Draw order of elements

UI elements in the Canvas are drawn in the same order they appear in the Hierarchy. The first child is drawn first, the second child next, and so on. If two UI elements overlap, the later one will appear on top of the earlier one.

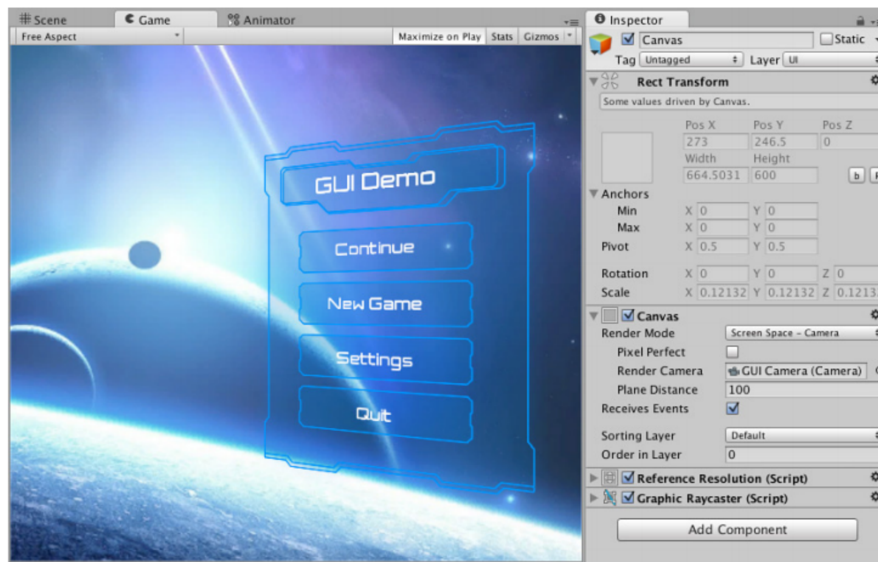
To change which element appear on top of other elements, simply reorder the elements in the Hierarchy by dragging them. The order can also be controlled from scripting by using these methods on the Transform component: SetAsFirstSibling, SetAsLastSibling, and SetSiblingIndex.

Render Modes

The Canvas has a Render Mode setting which can be used to make it render in screen space or world space.

Screen Space - Overlay

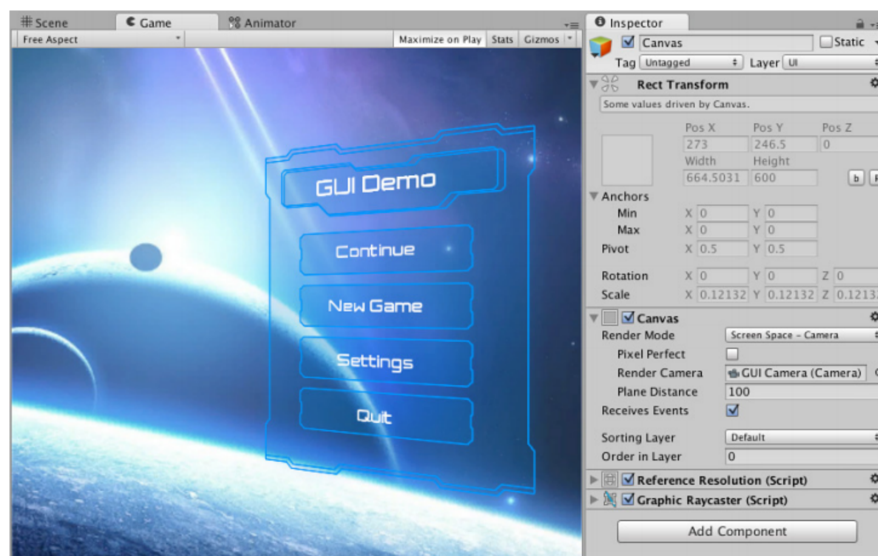
This render mode places UI elements on the screen rendered on top of the scene. If the screen is resized or changes resolution, the Canvas will automatically change size to match this.



UI in screen space overlay canvas

Screen Space - Camera

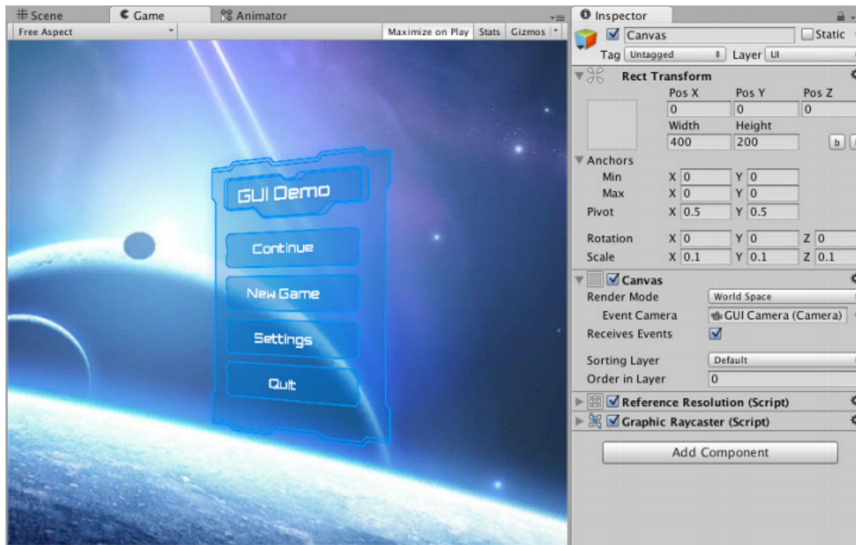
This is similar to Screen Space - Overlay, but in this render mode the Canvas is placed a given distance in front of a specified Camera. The UI elements are rendered by this camera, which means that the Camera settings affect the appearance of the UI. If the Camera is set to Perspective, the UI elements will be rendered with perspective, and the amount of perspective distortion can be controlled by the Camera Field of View. If the screen is resized, changes resolution, or the camera frustum changes, the Canvas will automatically change size to match as well.



UI in screen space camera canvas

World Space

In this render mode, the Canvas will behave as any other object in the scene. The size of the Canvas can be set manually using its Rect Transform, and UI elements will render in front of or behind other objects in the scene based on 3D placement. This is useful for UIs that are meant to be a part of the world. This is also known as a “diegetic interface”.



UI in screen space camera canvas

3.1.6 Built-in Controls

The following standard controls are built into UIElements:

- Button
- Contextual menu
- EditorTextField
- Label
- ScrollView
- TextField
- Toggle

Contextual menus

Contextual menus can present a set of choices or actions to the user, depending on the context. This context is usually the current selection, but the context can be anything.

This topic demonstrates how to add a contextual menu, explains its callbacks, and shows how to respond to user selection.

Enabling contextual menus

To enable contextual menus, attach the ContextualMenuManipulator manipulator to a visual element. This manipulator will display a contextual menu after either a right button mouse up event, or a menu key up event. The ContextualMenuManipulator manipulator also adds a callback that responds to a ContextualMenuPopulateEvent. The following code example shows how to do this:

```
void InstallManipulator(VisualElement element)
{
    ContextualMenuManipulator m = new ContextualMenuManipulator(MyDelegate);
    m.target = element;
}

void MyDelegate(ContextualMenuPopulateEvent event)
{
    // Modify event.menu
    event.menu.AppendAction("Properties", DisplayProperties, DropdownMenu.MenuAction.AlwaysEnabled);
}

void DisplayProperties(DropdownMenu.MenuAction menuItem)
{
    // ...
}
```

The callback given to the ContextualMenuManipulator constructor will be invoked last to allow children elements to populate the menu.

Internally the manipulator sends a ContextualMenuPopulateEventevent that is propagated to the target element hierarchy, along the propagation path: from the root of the visual tree to the event target, then back up the visual tree to the root. Along the propagation path, the elements with a callback for the ContextualMenuPopulateEvent event can add, remove, or modify items in the contextual menu.

Responding to the user selection

When an element receives a ContextualMenuPopulateEvent, it adds menu items to the contextual menu by calling either DropdownMenu.InsertAction()or DropdownMenu.AppendAction().

Each of these functions take two callbacks as parameters. The first callback is executed when the user selects the item in the menu. The second callback is executed before displaying the menu. The second callback also checks whether the menu item is enabled.

Both callbacks receive a MenuAction as a parameter. The MenuAction represents the menu item and has the following other useful properties:

- MenuAction.userData contains a reference to user data that might have been used with AppendAction() or InsertAction().
- MenuAction.eventInfo contains information about the event that triggered the display of the contextual menu. Use MenuAction.eventInfo in the action that responds to the event. For example, you can use the mouse position to create and place an object based on the selected contextual menu item.

Bindings

The purpose of bindings is to synchronize properties within objects to the visible UI. A binding refers to the link between the property and the visual control that modifies it.

Binding is done between an object and any UIElement that either derives from BindableElement or implements the IBindable interface.

From the UnityEditor.UIElements namespace:

Base Class:

- BaseCompositeField
- BasePopupField
- CompoundFields
- TextValueField

Controls:

- InspectorElement
- ProgressBar
- BoundsField
- BoundsIntField
- ColorField
- CurveField
- DoubleField
- EnumField
- FloatField
- GradientField
- IntegerField
- LayerField
- LayerMaskField
- LongField
- MaskField
- ObjectField
- PopupField
- PropertyControl
- RectField
- RectIntField
- TagField
- Vector2Field
- Vector2IntField
- Vector3Field
- Vector3IntField
- Vector4Field

From the UnityEngine.UIElements namespace:

Base Class:

- BaseField
- BaseSlider
- TextInputBaseField
- TemplateContainer

Controls:

- Foldout
- MinMaxSlider
- Slider
- SliderInt
- TextField
- Toggle

Binding is done by following these steps while using a Control from one of the namespaces listed above.

1. In the Control, specify the bindingPath from the IBindable interface so the UI knows which property to bind. You can do this in C# or in UXML. An example of each is provided later in this topic.
2. Create a SerializedObject for the object being bound.
3. Bind this object to the UIElements Control or one of its parents.

Binding with C#

The following code snippet shows how to create a binding with C# code. To use this snippet, save this example as a C# file in an editor folder, in your project. Name the C# file SimpleBindingExample.cs.

The contents of SimpleBindingExample.cs:

```
using UnityEditor;
using UnityEngine;
using UnityEditor.UIElements;
using UnityEngine.UIElements;

namespace UIElementsExamples
{
   public class SimpleBindingExample : EditorWindow
   {
       TextField m_ObjectNameBinding;

       [MenuItem("Window/UIElementsExamples/Simple Binding Example")]
       public static void ShowDefaultWindow()
       {
           var wnd = GetWindow<SimpleBindingExample>();
           wnd.titleContent = new GUIContent("Simple Binding");
       }

       public void OnEnable()
       {
           var root = this.rootVisualElement;
           m_ObjectNameBinding = new TextField("Object Name Binding");
           m_ObjectNameBinding.bindingPath = "m_Name";
           root.Add(m_ObjectNameBinding);
           OnSelectionChange();
       }

       public void OnSelectionChange()
       {
           GameObject selectedObject = Selection.activeObject as GameObject;
           if (selectedObject != null)
           {
               // Create serialization object
               SerializedObject so = new SerializedObject(selectedObject);
               // Bind it to the root of the hierarchy. It will find the right object to bind to...
               rootVisualElement.Bind(so);
               // ... or alternatively you can also bind it to the TextField itself.
               // m_ObjectNameBinding.Bind(so);
           }
           else
           {
               // Unbind the object from the actual visual element
               rootVisualElement.Unbind();

               // m_ObjectNameBinding.Unbind();

               // Clear the TextField after the binding is removed
               m_ObjectNameBinding.value = "";
           }
       }
   }
}
```

In Unity, select Window > UIElementsExamples > Simple Binding Example. You can use this window to select any GameObject in your scene and modify its name with the TextField shown.

Binding with UXML

This section shows how to use binding through the UXML hierarchy set-up.

In UXML, the attribute binding-path is defined in the TextField control. This is what binds the control to the effective property of the object.

The contents of SimpleBindingExample.uxml:

```
<UXML xmlns:ui="UnityEngine.UIElements">
 <ui:VisualElement name="top-element">
   <ui:Label name="top-label" text="UXML-Defined Simple Binding"/>
   <ui:TextField name="GameObjectName" label="Name" text="" binding-path="m_Name"/>
 </ui:VisualElement>
</UXML>
```

The contents of SimpleBindingExample.cs:

```
using UnityEditor;
using UnityEngine;
using UnityEditor.UIElements;
using UnityEngine.UIElements;


namespace UIElementsExamples
{
   public class SimpleBindingExampleUXML : EditorWindow
   {
       [MenuItem("Window/UIElementsExamples/Simple Binding Example UXML")]
       public static void ShowDefaultWindow()
       {
           var wnd = GetWindow<SimpleBindingExampleUXML>();
           wnd.titleContent = new GUIContent("Simple Binding UXML");
       }

       public void OnEnable()
       {
           var root = this.rootVisualElement;
           var visualTree = AssetDatabase.LoadAssetAtPath<VisualTreeAsset>("Assets/Editor/SimpleBindingEx
           visualTree.CloneTree(root);
           OnSelectionChange();
       }

       public void OnSelectionChange()
       {
           GameObject selectedObject = Selection.activeObject as GameObject;
           if (selectedObject != null)
           {
               // Create serialization object
               SerializedObject so = new SerializedObject(selectedObject);
               // Bind it to the root of the hierarchy. It will find the right object to bind to...
               rootVisualElement.Bind(so);
           }
           else
           {
               // Unbind the object from the actual visual element
               rootVisualElement.Unbind();

               // Clear the TextField after the binding is removed
               // (this code is not safe if the Q() returns null)
               rootVisualElement.Q<TextField>("GameObjectName").value = "";
           }
       }
   }
}
```

Using bindings within InspectorElement

An InspectorElement is the UIElement counterpart of an inspector that is meant a specific type of Unity object. Usingan InspectorElement to inspect objects gives the following advantages :

• Creates the UI.
• Automatically binds objects and the UI.

Another simple binding example, found under Assets/Editor/SimpleBindingExample.cs, provides a usage example and an overview of the process.

The contents of Assets/Editor/SimpleBindingExample.cs:

```
using UnityEditor;
using UnityEngine;
using UnityEditor.UIElements;

namespace UIElementsExamples
{
   public class SimpleBindingExampleUXML : EditorWindow
   {
       [MenuItem("Window/UIElementsExamples/Simple Binding Example UXML")]
       public static void ShowDefaultWindow()
       {
           var wnd = GetWindow<SimpleBindingExampleUXML>();
           wnd.titleContent = new GUIContent("Simple Binding UXML");
       }

       TankScript m_Tank;
       public void OnEnable()
       {
           m_Tank = GameObject.FindObjectOfType<TankScript>();
           if (m_Tank == null)
               return;

           var inspector = new InspectorElement(m_Tank);
           rootVisualElement.Add(inspector);
       }
   }
}
```

This code references the TankScript script and uses the InspectorElement. The TankScript script is an example of a MonoBehaviour assigned to a GameObject.

The contents of Assets/TankScript.cs:

```
using UnityEngine;

[ExecuteInEditMode]
public class TankScript : MonoBehaviour
{
   public string tankName = "Tank";
   public float tankSize = 1;

   private void Update()
   {
       gameObject.name = tankName;
       gameObject.transform.localScale = new Vector3(tankSize, tankSize, tankSize);
   }
}
```

The InspectorElement is customized with a specific UI. This is done with the TankEditor script. The TankEditorscript defines a custom editor for the TankScript type. The TankEditorscript also uses a UXML file for the hierarchy and a USS file to style the inspector.

The contents of Assets/Editor/TankEditor.cs:

```
using UnityEditor;
using UnityEngine;
using UnityEngine.UIElements;

[CustomEditor(typeof(TankScript))]
public class TankEditor : Editor
{
   public override VisualElement CreateInspectorGUI()
   {
       var visualTree = Resources.Load("tank_inspector_uxml") as VisualTreeAsset;
       var uxmlVE = visualTree.CloneTree();
uxmlVE.styleSheets.Add(AssetDatabase.LoadAssetAtPath<StyleSheet>("Assets/Resources/tank_inspector_styles.
      return uxmlVE;
   }
}
```

The contents of Assets/Resources/tank_inspector_uxml.uxml:

```
<UXML xmlns:ui="UnityEngine.UIElements" xmlns:ue="UnityEditor.UIElements">
   <ui:VisualElement name="row" class="container">
       <ui:Label text="Tank Script - Custom Inspector" />
       <ue:PropertyField binding-path="tankName" name="tank-name-field" />
       <ue:PropertyField binding-path="tankSize" name="tank-size-field" />
   </ui:VisualElement>
</UXML>
```

The UXML file, tank_inspector_uxml.uxml specifies the binding. Specifically, each binding-path attribute, for each the PropertyFields tag, is set to the property to bind. What element is shown in the UI is based on the type of each bound property.

The contents of Assets/Resources/tank_inspector_styles.uss:

```
.container {
   background-color: rgb(80, 80, 80);
   flex-direction: column;
}
Label {
   background-color: rgb(80, 80, 80);
}
TextField:hover {
   background-color: rgb(255, 255, 0);
}
FloatField {
   background-color: rgb(0, 0, 255);
}
```

The USS file, tank_inspector_styles.uss, defines the style of each element.

The following table lists the fields supported by the PropertyField. Each field includes its data type.

Field	Type
BoundsField	Bounds
BoundsIntField	BoundsInt
ColorField	Color
CurveField	AnimationCurve
FloatField	float
GradientField	Gradient
IntegerField	int
IntegerField	int, for the ArraySize
LayerMaskField	unit
ObjectField	UnityEngine.Object
PopupField<string>	Enum
RectField	Rect
RectIntField	RecInt
TextField	string
TextField, with a maxLength=1	char
Toggle	bool
Vector2Field	Vector2
Vector2IntField	Vector2Int
Vector3Field	Vector3
Vector3IntField	Vector3Int
Vector4Field	Vector4

UNITY CANVAS AND SCENE MANAGEMENT

MODULE 3

3.2

Summary

How to switch scenes by a code.

We can add a new level of scene to our already existing scene by pressing ctrl+n. We add some objects and save the new scene.

Now we need to connect the button from our main scene so when we click on it a new level scene will be opened.

We press ctrl+shift, which opens a new window. The window controls the scenes which are connected to each other and will be affected by a script.

So we add in this window all the scenes that we have and want to be related.

We want our button to initiate a new level scene, so we click on the button and add component, then we add a code in visual studio.

At first, we add SceneManagement function to visual studio using section. Then we create our own function by typing= public void PlayButtonPressed() {SceneManager.Loadscene(“level_1”) } and we save.

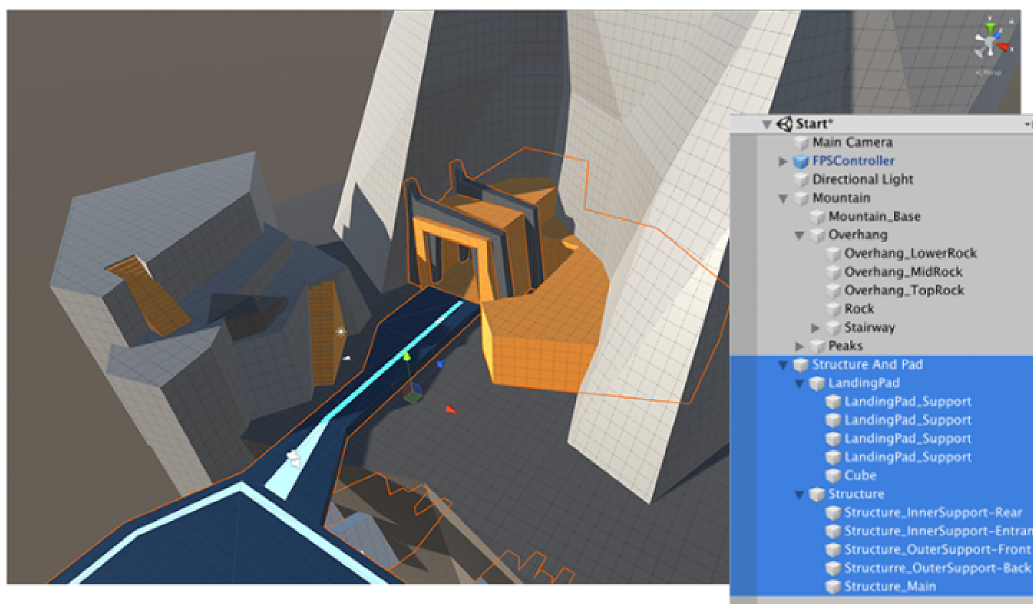
Then we have to open on-click section and drag our script there. Open menu and add PlayButtonPressed() function that we have created.

Key Points

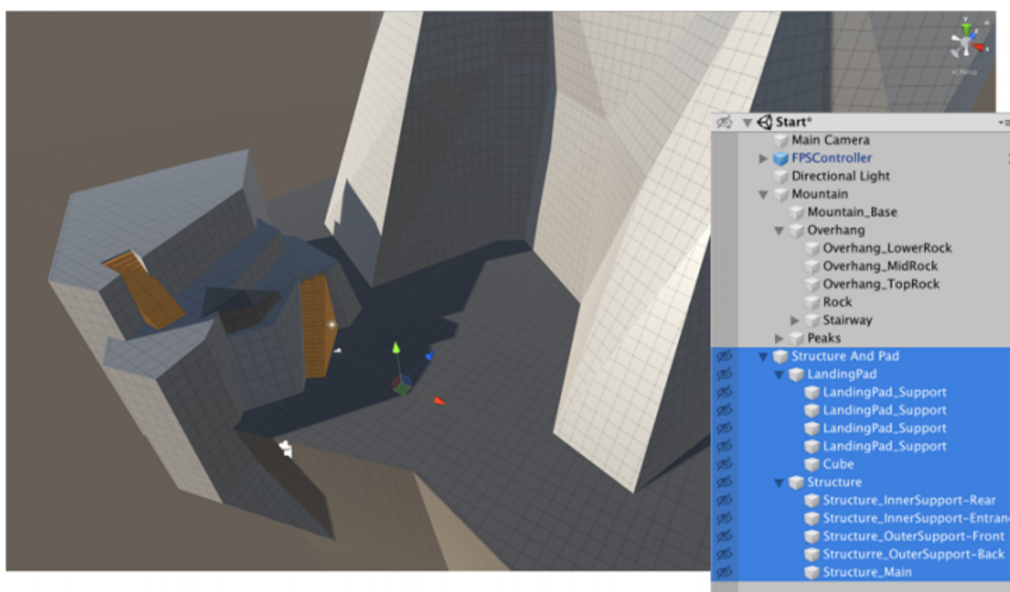
- Switch between a scene levels requires several steps.
- Ctrl+shift opens up a new window where we should add scene levels that will be affected by our switching code
- We add a script to a component of a scene that will initiate opening of a new scene
- We devise a new function to initiate scene switch
- SceneManagement function must be added to visual studio using section.
- We open on-click section of the element which we want to initiate opening of a new scene and add our script.

3.2.1 Scene Visibility

Unity's SceneView visibility controls allow you to quickly hide and show GameObjects in the Scene viewwithout changing their in-game visibility. This is useful for working with large or complex Scenes where it can be difficult to view and select specific GameObjects.



Selected GameObjects are highlighted in orange



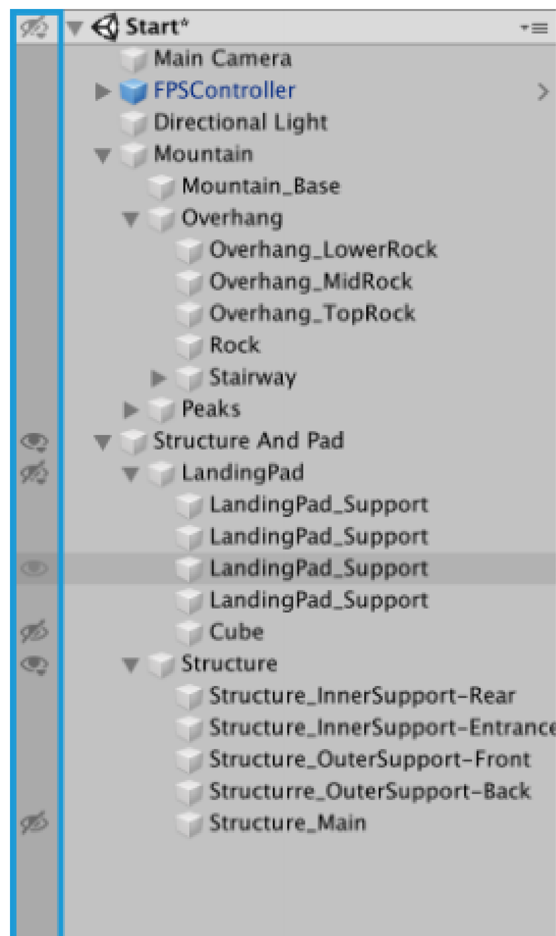
Changing the Scene visibility settings hides the selected GameObjects in the scene view

Using visibility options is safer than deactivating GameObjectsbecause visibility options only affect the Scene view. This means you can't accidentally remove GameObjects from the rendered scene, or trigger unnecessary bake jobs for lighting, occlusion, and other systems.

Unity saves Scene visibility settings to a file called SceneVisibilityState.asset in the Project's Library folder. The scene reads from this file and updates it automatically whenever you change the visibility settings. This makes it possible for your settings to persist from one session to the next. Because source control setups for Unity typically ignore the Library folder, changing visibility settings should not create source control conflicts.

Setting Scene visibility for GameObjects and their children

You control Scene visibility for individual GameObjects from the Hierarchy window.

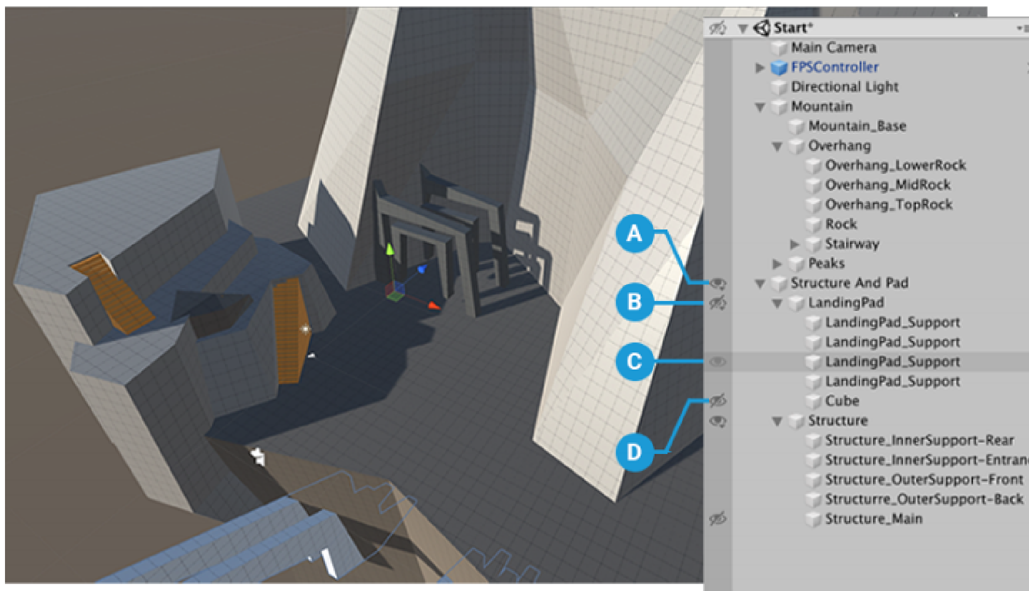


Every GameObject has a Scene visibility icon/toggle

To toggle Scene visibility:

- Click a GameObject's visibility icon in the Hierarchy window, or press H, to hide/show the GameObject and its children.
- Toggling visibility for an object and its children affects all child objects, from the "target" object all the way down to the bottom of the hierarchy.
- Alt + Click a GameObject's visibility icon in the Hierarchy window to hide/show the GameObject only.
- Toggling visibility for a single object does not affect its children. They retain whatever visibility status they had previously.
- Click the scene's visibility icon to hide/show everything in the scene.

Because you can toggle visibility for a whole branch or a single GameObject, you can end up with GameObjects that are visible, but have hidden children or parents. To help you track what's going on, the visibility icon changes to indicate each GameObject's status.

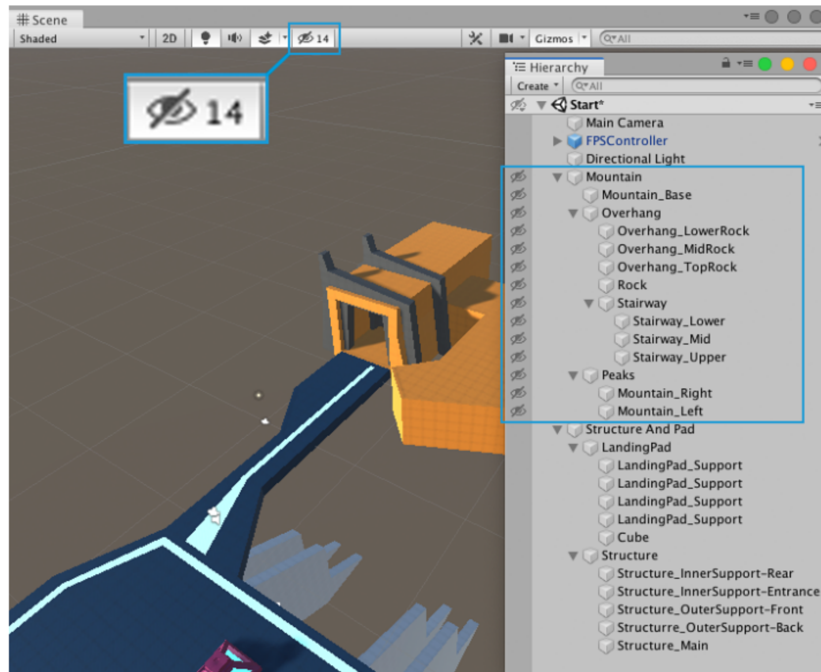


A		The GameObject is visible, but some of its children are hidden.
B		The GameObject is hidden, but some of its children are visible.
C		The GameObject and its children are visible. This icon only appears when you hover over the GameObject.
D		The GameObject and its children are hidden.

Scene visibility changes you make in the Hierarchy window are persistent. Unity re-applies them whenever you toggle Scene visibility off and on again in the Scene view, close and re-open the Scene, and so on.

Turning Scene visibility on and off

The Scene visibility switch in the Scene view control bar displays the number of hidden GameObjects in the scene. Click it to toggle Scene visibility on and off.



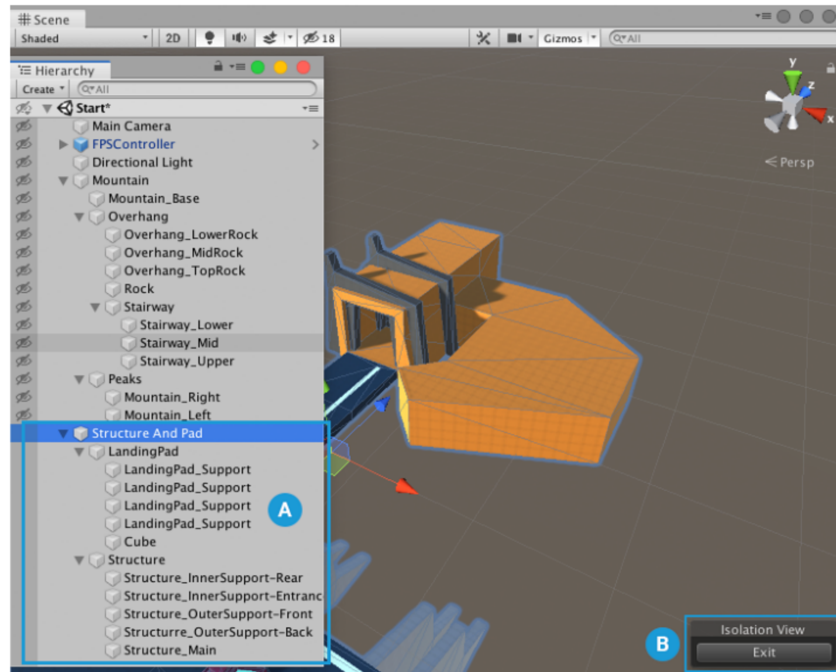
The Scene visibility counter shows 14 hidden GameObjects in this scene

Turning Scene visibility off essentially mutes the Scene visibility settings you set in the Hierarchy window, but doesn't delete or change them. All hidden GameObjects are temporarily visible.

Turning Scene visibility back on re-applies the visibility settings set in the Hierarchy window.

Isolating selected GameObjects

The Isolation view temporarily overrides the Scene visibility settings so that only the selected GameObjects are visible, and everything else is hidden.



Isolation view overrides Scene visibility settings so only the selection and its children (A) are visible.
Clicking the Exit button (B) reverts to the previous Scene visibility settings.

To enter the Isolation view, press Shift + H.

This isolates all selected GameObjects and their children. Isolating hidden GameObjects makes them visible until you exit the Isolation view.

While in the Isolation view, you can continue to change Scene visibility settings, but any changes you make are lost on exit.

To exit the Isolation view, press Shift + H again, or click the Exitbutton in the Scene view.

Exiting the Isolation view reverts back to your original Scene visibility settings.

3.2.2 Multi-Scene Editing

Multi Scene Editing allows you to have multiple scenes open in the editor simultaneously, and makes it easier to manage scenes at runtime.

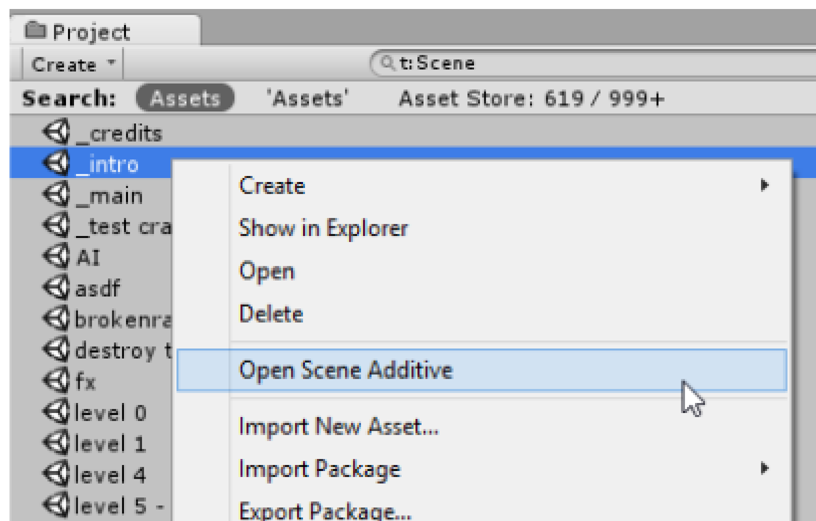
The ability to have multiple scenes open in the editor allows you to create large streaming worlds and improves the workflow when collaborating on scene editing.

This page describes:

- The multi scene editing integration in the Editor
- The Editor scripting and the Runtime scripting APIs
- Current known issues

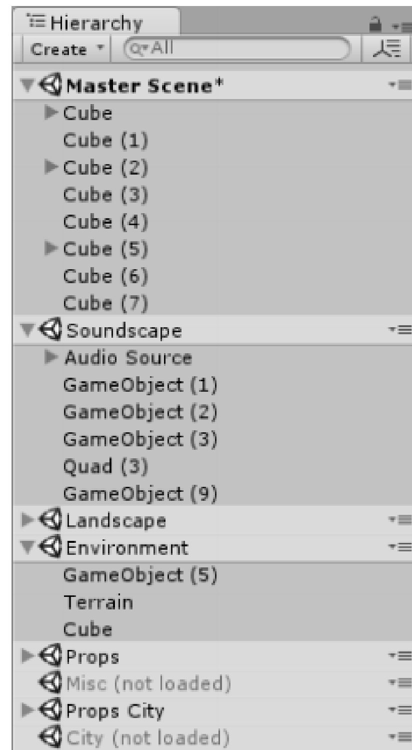
In the editor

To open a new scene and add it to the current list of scenes in the Hierarchy, either select Open Scene Additive in the context menu for a scene asset, or drag one or more scenes from the Project window into the Hierarchy Window.



Open Scene Additive will add the selected scene asset to the current scenes shown in the hierarchy

When you have multiple scenes open in the editor, each scene's contents are displayed separately in the hierarchy window. Each scene's contents appears below a scene divider bar which shows the scene's name and its save state

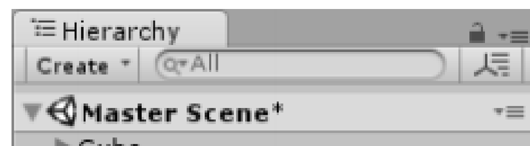


The Hierarchy window showing multiple scenes open simultaneously

While present in the hierarchy, scenes can be loaded or unloaded to reveal or hide the gameobjects contained within each scene. This is different to adding and removing them from the hierarchy window.

The scene dividers can be collapsed in the hierarchy to the scene's contents which may help you to navigate your hierarchy if you have lots of scenes loaded.

When working on multiple scenes, each scene that is modified will need its changes saved, so it is possible to have multiple unsaved scenes open at the same time. Scenes with unsaved changes will have an asterisk shown next to the name in the scene divider bar.

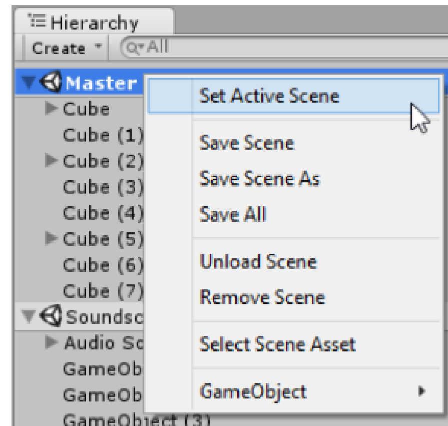


An asterisk in the scene divider indicating this scene has unsaved changes

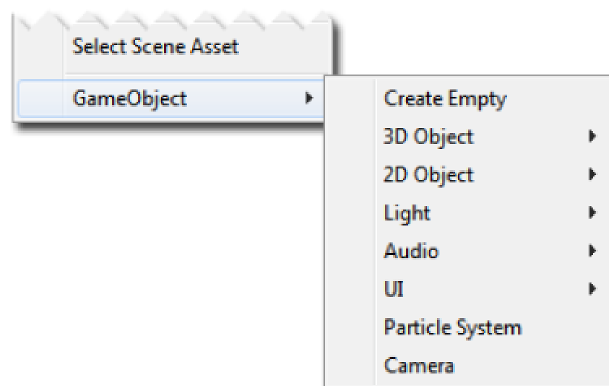
Each Scene can be saved separately via the context menu in the divider bar. Selecting "Save Scene" from the file menu or pressing Ctrl/Cmd + S will save changes to all open scenes.

The context menu in the scene divider bars allow you to perform other actions on the selected scene.

The Scene divider menu for loaded Scenes

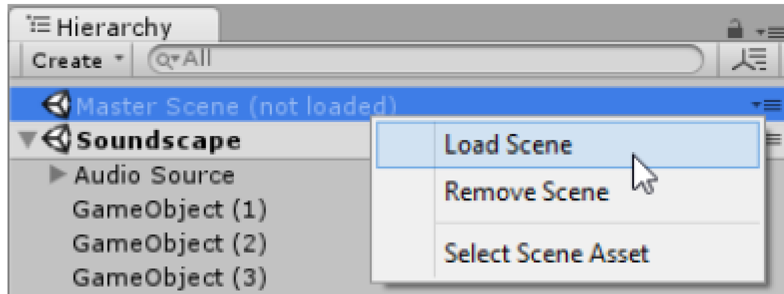


Set Active Scene	This allows you to specify which scene new Game Objects are created/instantiated in. There must always be one scene marked as the active scene.
Save Scene	Saves the changes to the selected scene only.
Save Scene As	Saves the selected scene (along with any current modifications) as a new Scene asset.
Save All	Saves changes to all scenes.
Unload Scene	Unloads the scene, but keeps the scene in the Hierarchy window.
Remove Scene	Unloads and removes the scene from the Hierarchy window.
Select Scene Asset	Selects the scene's asset in the Project window.
GameObject	Provides a sub-menu allowing you to create GameObjects in the selected scene. The menu mirrors the creatable items available in Unity's main GameObject menu. (shown below)



The GameObject sub-menu in the Scene divider bar menu

The Scene divider menu for unloaded Scenes



Load Scene	Loads the scene's contents
Remove Scene	Remove the scene from the Hierarchy window.
Select Scene Asset	Selects the scene's asset in the Project window.

Baking Lightmaps with multiple Scenes

To bake Lightmap data for multiple scenes at once, you should open the scenes that you want to bake, turn off "Auto" mode in the Lighting Window, and click the Build button to build the lighting.

The input to the lighting calculations is the static geometry and lights from all scenes. Therefore shadows and GI light bounces will work across all scenes. However, the lightmaps and realtime GI data are separated out into data that is loaded / unloaded separately for each scene. The lightmaps and realtime GI data atlases are split between scenes. This means lightmaps between scenes are never shared and they can be unloaded safely when unloading a scene.

Lightprobe data is currently always shared and all lightprobes for all scenes baked together are loaded at the same time. Alternatively, you can automate building lightmaps for multiple scenes by using the Lightmapping.BakeMultipleScenes function in an editor script.

Baking Navmesh data with multiple Scenes

To make Navmesh ata for multiple scenes at once, you should open the scenes that you want to bake, and click the Bake button in the Navigation Window. The navmesh data will be baked into a single asset, shared by all loaded scenes.

The data is saved into the folder matching the name of the current active scene (e.g. ActiveSceneName/NavMesh.asset). All loaded scenes will share this navmesh asset. After baking the navmesh, the scenes affected should be saved to make the scene-to-navmesh reference persistent.

Alternatively, you can automate building navmesh data for multiple scenes by using the NavMeshBuilder.BuildNavMeshForMultipleScenes function in an editor script.

Baking Occlusion Culling data with multiple Scenes

To bake Occlusion Culling data for multiple Scenes at once, open the Scenes that you want to bake, open the Occlusion Culling window (menu: Window > Rendering > Occlusion Culling) and click the Bake button. The Occlusion data is saved into an asset called OcclusionCullingData.asset in a folder matching the name of the current active scene.

For example Assets/ActiveSceneName/OcclusionCullingData.asset. A reference to the data is added in each open Scene. After baking the Occlusion Culling, save the Scenes affected to make the Scene-to-occlusion reference persistent.

Whenever a Scene is loaded additively, if it has the same occlusion data reference as the active Scene, the static renderers and portals culling information for that Scene are initialized from the occlusion data. Hereafter, the occlusion culling system performs as if all static renderers and portals were baked into a single Scene.

Play mode

In Play mode, with multiple scenes in the Hierarchy, an additional scene will show up called DontDestroyOnLoad.

Prior to Unity 5.3, any objects you would instantiate in Playmode marked as “DontDestroyOnLoad” would still show up in the hierarchy. These objects are not considered part of any scene but for Unity to still show the objects, and for you to inspect them, these objects are now shown as part of the special DontDestroyOnLoad scene.

You do not have access to the DontDestroyOnLoad scene and it is not available at runtime.

Scene-specific settings

A number of settings are specific to each scene. These are:

- RenderSettings and LightmapSettings (both found in the Lighting Window)
- NavMesh settings
- Scene settings in the Occlusion Culling Window.

The way it works is that each scene will manage its own settings and only settings associated with that scene will be saved to the scene file.

If you have multiple scenes open, the settings that are used for rendering and navmesh are the ones from the active scene. This means that if you want to change the settings of a scene, you must either open only one scene and change the settings, or make the scene in question the active scene and change the settings.

When you switch active scene in the editor or at runtime, all the settings from the new scene will be applied and replace all previous settings.

Scripting

Editor scripting

For editor scripting we provide a Scene struct and EditorSceneManager API and a SceneSetup utility class.

The Scene struct is available both in the editor and at runtime and contains a handful of read-only properties relating to the scene itself, such as its name and asset path.

The EditorSceneManager class is only available in the editor. It is derived from SceneManager and has a number of functions that allow you to implement all the Multi Scene Editing features described above via editor scripting.

The SceneSetup class is a small utility class for storing information about a scene currently in the hierarchy.

The Undo and PrefabUtility classes have been extended to support multiple scenes. You can now instantiate a in a given scene using [PrefabUtility.InstantiatePrefab], and you can move objects to the root of a scene in an un-doable manner using (Undo.MoveGameObjectToScene)[ScriptRef:Undo.MoveGameObjectToScene]

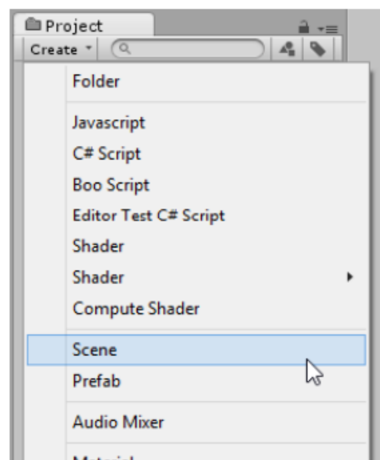
NOTE: To use Undo.MoveGameObjectToScene, you must make sure the GameObject is already at the root of the scene it is currently in.

Runtime scripting

For scripting at Runtime, the functions to work with multiple scenes such as LoadScene and UnloadScene are found on the SceneManager class.

Notes

In the File menu Save Scene As will only save the active scene. Save Scene will save all modified scenes, including prompting you to name the Untitled scene if it exists.



Creating a new Scene asset from the Project window's Create menu

Tips and tricks

It is possible to add a scene to the hierarchy while keeping it its unloaded state by holding Alt while dragging. This gives you the option to load the scene later, when desired.

New scenes can be created using the Create menu in the project window. New scenes will contain the default setup of Game Objects.

To avoid having to set up your hierarchy every time you restart unity or to make it easy to store different setups you can use EditorSceneManager.GetSceneManagerSetup to get a list of SceneSetup objects which describes the current setup. You can then serialize these into a ScriptableObject or something else along with any other information you might want to store about your scene setup.

To restore the hierarchy simply recreate the list of SceneSetups and use EditorSceneManager.RestoreSceneManagerSetup.

At runtime to get the list of loaded scenes simply get sceneCount and iterate over the scenes using GetSceneAt.

You can get the scene a GameObject belongs to through GameObject.scene and you can move a GameObject to the root of a scene using SceneManager.MoveGameObjectToScene.

It is recommended to avoid using DontDestroyOnLoad to persist manager GameObjects that you want to survive across scene loads. Instead, create a manager scene that has all your managers and use SceneManager.LoadScene(<path>, LoadSceneMode.Additive) and SceneManager.UnloadScene to manage your game progress.

Known issues

Cross-Scene references are not supported, and are prevented in Edit mode. In Play mode they are allowed, because Scenes cannot be saved.