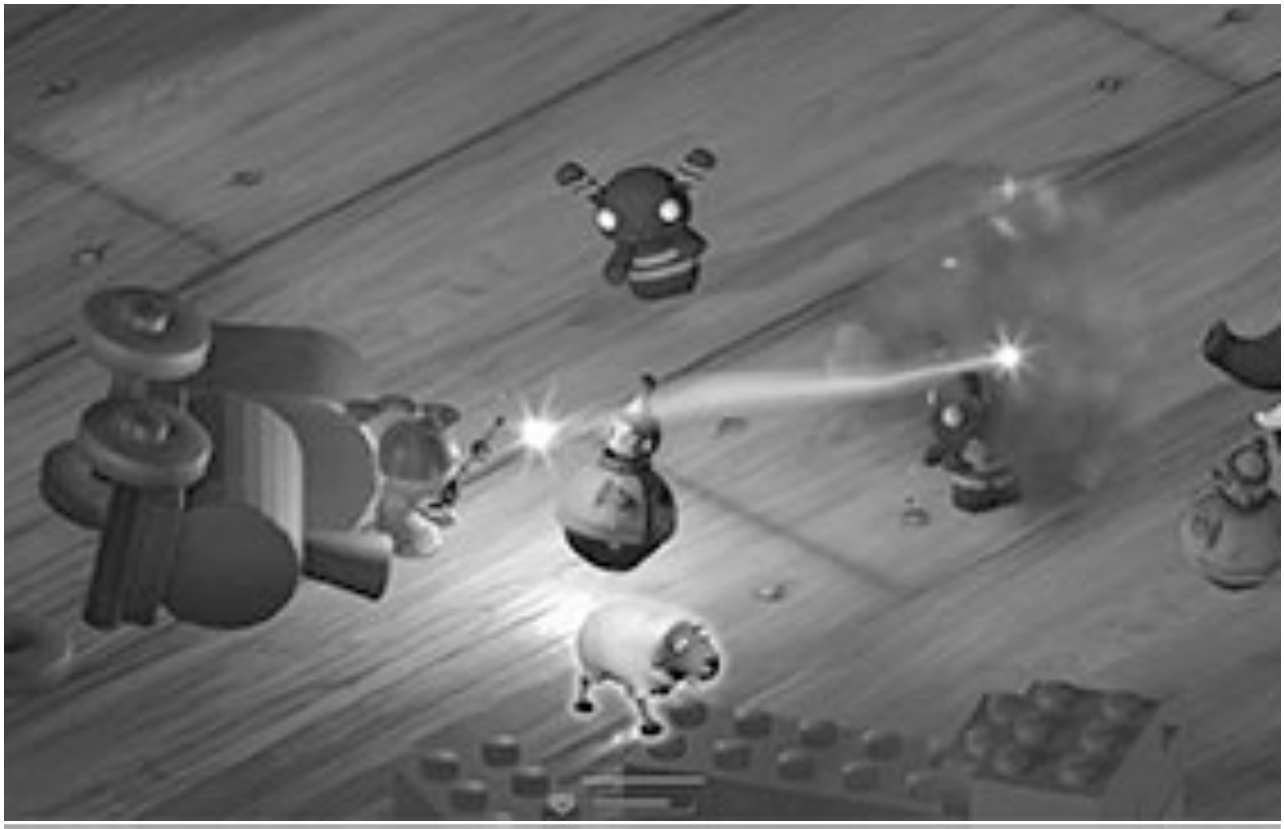
201708

Unity 課程講義



講師 連志偉

第一版

I.認識 Unity 遊戲開發引擎與軟體安裝、授權

Unity 是位居世界領先地位的遊戲創作引擎，並持續引入新功能，以協助開發團隊的人員建立共同的開發體驗，使美術人員可以直接為遊戲建立劇情動畫內容，企劃人員能夠花更多的時間在編輯遊戲內容，而不需要與技術人員排隊進行開發內容。

1.Unity 遊戲開發引擎的特點

1.1. 豐富且可擴充的編輯器

Unity 編輯器是藝術家、設計師、開發者以及其他團隊成員的創作中心。可在 Windows 和 Mac 的電腦上使用，包含了 2D 和 3D 場景的設計工具，為快速和重複編輯提供的立即播放模式，以及強大的動畫控制系統。

- **快速重複進行：**按下播放，即可立即進入遊戲遊玩並預覽在指定平台最終構建的樣子，可以隨時暫停並修改數值、資源、腳本和其它屬性並立即看到結果，也可以每一幀逐步進行以方便調適並找出問題。
- **獨特的可擴充工具：**提供專門設計的 API 讓你可以自己為 Unity 編輯器擴充，或者從 Asset Store 下載或購買其他開發者所製作的開發工具。
- **各種強大又方便的系統：**包含可以編輯過場動畫、遊戲順序、AI 尋徑、粒子特效、動畫控制、即時光照、物理渲染、聲音混合、物理碰撞、腳本整合及效能優化等等的編輯器視窗。
- 無與倫比的匯入通道：支援匯入各種圖像、聲音、影像、文件等等格式的檔案。
- 範例觀摩：https://youtu.be/csP4JQprpog。

1.2. 圖形渲染

為場景畫面創造引人注目的氣氛。從明亮的白天到晚上霓虹燈的華麗光芒，從清晨曙光到昏暗的午夜街道，建立動態的遊戲場景畫面。

- 即時渲染引擎：使用即時的全域光照與物理基礎渲染產生驚人的視覺真實度。
- 原生圖形 API：Unity 支援多平台，但仍保有接近各平台的低階圖形 API，使你能夠利用最新 GPU 和硬體改善的優勢，如 Vulan、iOS Metal、DirectX12、nVidia VRWorks 和 AMD LiquidVR。
- 範例觀摩：https://youtu.be/44M7JsKqwow。

1.3. 引擎效能

頂級效能，使用不斷改進的引擎最佳化你的創作。

- 進階分析工具：提供詳細資訊讓你了解如何改善效能。
- 原生C++效能：使用 ILCPP 使跨平台持續進化。

1.4. 跨平台

構建一次並部署到任何地方，使受眾最大化。

- 超過 25 個平台：跨越手機、桌機、遊戲機、電視、VR、AR 以及網頁。
- 引領 VR / AR：Unity 的通道輕鬆地將你的內容帶到最新的平台（支援 Vuforia、Google Tango 以及即將到來的 Microsoft Mixed Reality）。

1.5. 資源商店 Asset Store

Asset Store 協助你快速開始開發並快速完成。無論是解決問題或是開發需求，Asset Store 都是解決方案的所在。

- 有現成的內容來提升你的 Unity 專案並使開發更簡單、快速。
- 一大堆免費和付費的內容。
- 無論什麼都有：美術作品、模型、程式腳本、生產力工具等等。
- 網址：https://www.assetstore.unity3d.com/。

1.6. 多人遊戲 Multiplayer

Multiplayer 提供最簡單的方法來建立即時的網路遊戲。

- 快速設置：快速實作並高度可客製化。
- Unity 提供 Server：確保你的玩家能夠找尋到並與其他玩家一起玩。

1.7. 團隊協作 Collaboration

Collaboration 能夠使團隊更快速的運作，使創作團隊可以更有效率的共同工作，以實現協作並簡化工作流程。

- 儲存、分享以及同步妳的專案，並使用簡單的版本控管及雲端儲存空間，全部都與 Unity 無縫整合。
- 雲端構建：自動建立並與任何人分享構建。

1.8. Unity Connect

實現你的最終構想，Unity Connect 是個致力於 Unity 創作者的專業網路和人才市場。

- 使用免費的作品集展示你的工作。
- 被尋找人才的人發現。
- 與其他技術者交流，協助你克服障礙並實現你的最終構想。
- 瀏覽業界工作或尋找機會。

1.9. 即時操作分析 Live Ops Analytics

透過 Live Ops Analytics 即時洞察某些事物。使你能夠快速存取重要資訊，以協助改善遊戲內經濟以及玩家體驗。

- 完整的即時操作分析功能（遊戲和玩家分析、熱圖、效能監測）來監控玩家行為。
- 沒有 SDK！內建於 Unity 之中，所以沒有 SDK！

1.10.效能最佳化

發現應用程式錯誤。Unity Performance Reporting 能夠讓你即時解決問題。

- 查找並解決你的用戶正在經歷的最優先級問題。
- 跨越設備和平台即時收集並反應出應用程式錯誤。

1.11.貨幣化

用最簡單的方式增加收入。Unity 提供內建的解決方案，使你的成功最大化。

- 創造收入的解決方案：廣告聯播及應用內消費。
- 一套完整的即時操作分析功能來監控玩家行為。
- 持續改善：即時地為你運行中的遊戲改善效能，無需重新部署，並透過數據分析的力量增加使用者生命週期價值。

1.12.廣告

取得新用戶。在 Unity Ads 刊登廣告，直接獲得新用戶。

- 獲取優質的遊戲以及娛樂清單。
- 集中在目標受眾。
- 在正確的地點、時間接觸受眾。

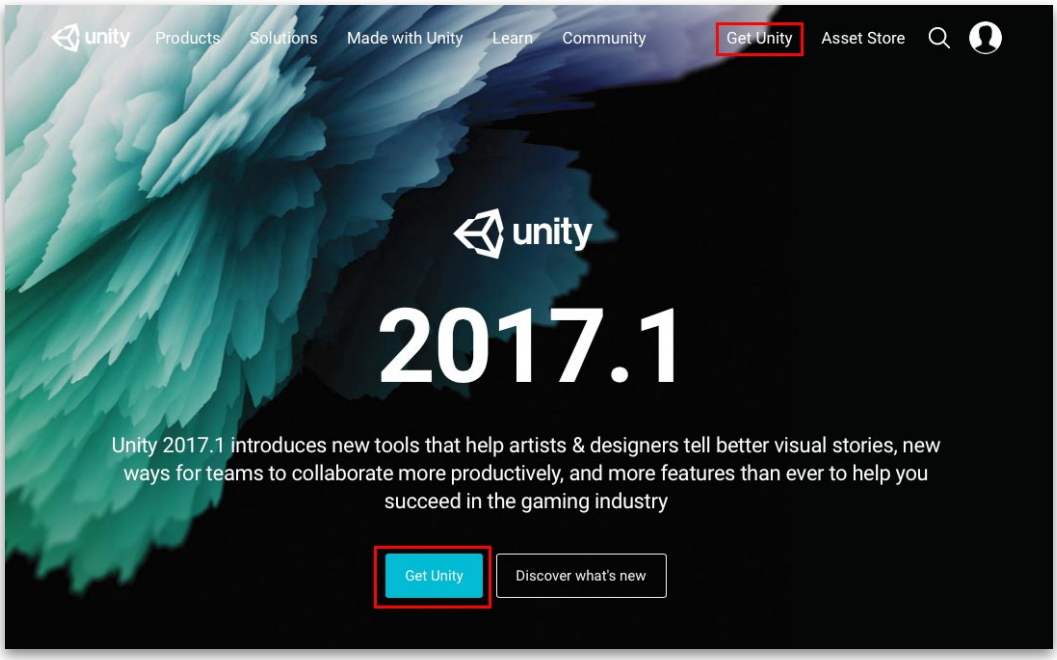
1.13.病毒式用戶增長

使用 Everyplay，你的用戶能夠分享他們表現最佳時的影片並對他們的朋友分享你的遊戲。

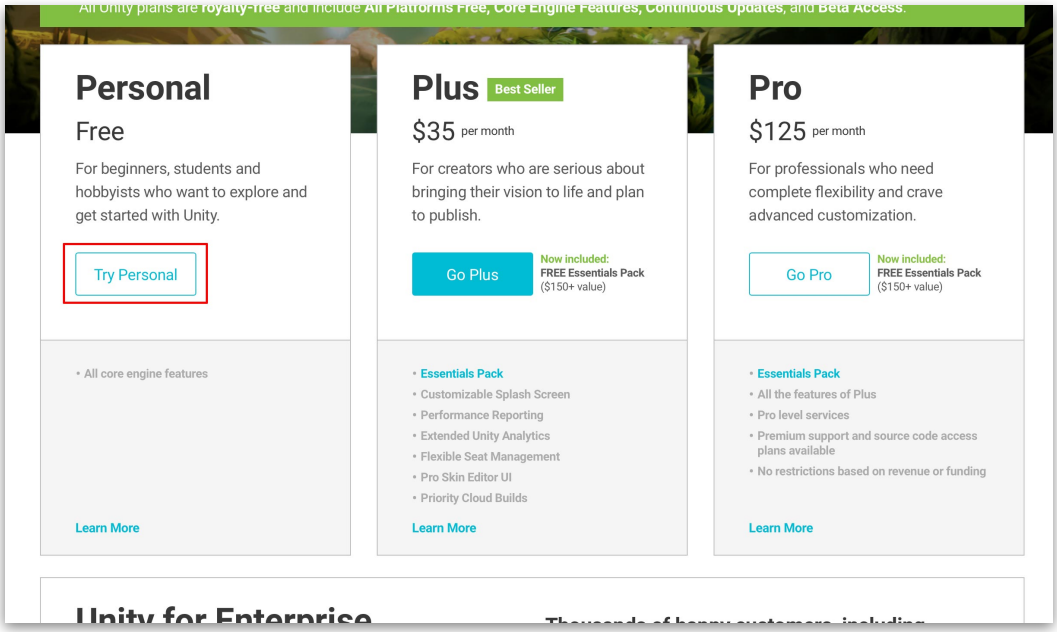
2. 下載 *Unity* 安裝程式

2.1. 下載最新版

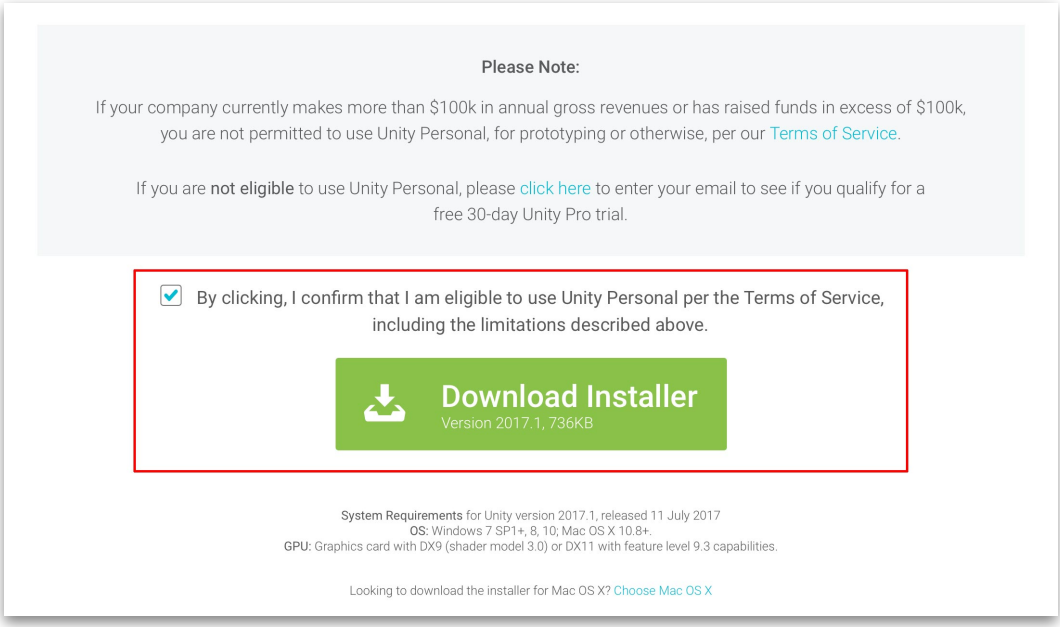
(1) 前往官方網站：https://unity3d.com，點擊 Get Unity。



(2) 點擊免費個人版（Personal）的 Try Personal 按鈕。



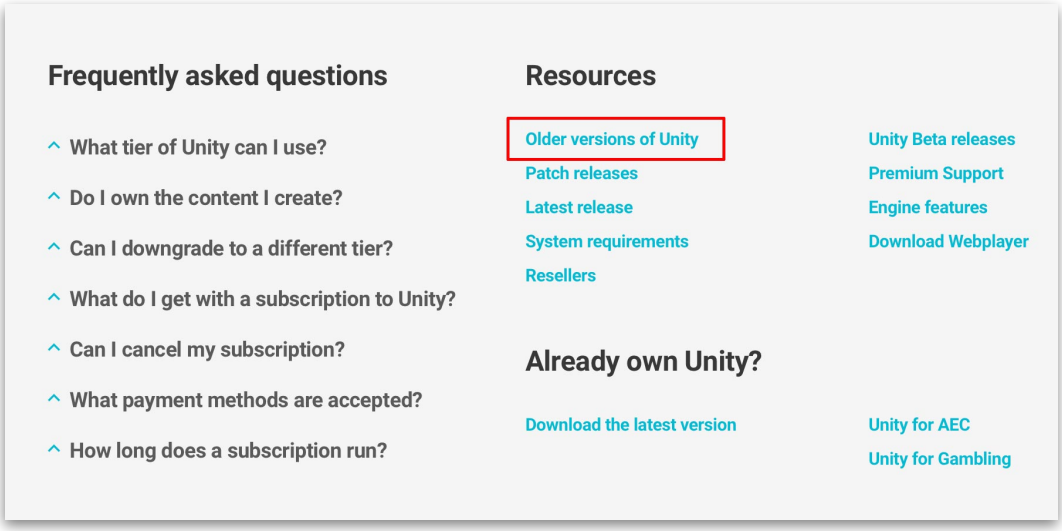
(3) 勾選「By clicking, I confirm that I am eligible to use Unity Personal per the Terms of Service, including the limitations described above」表示確認符合個人版使用資格，並點擊「Download Installer」下載安裝助理程式。



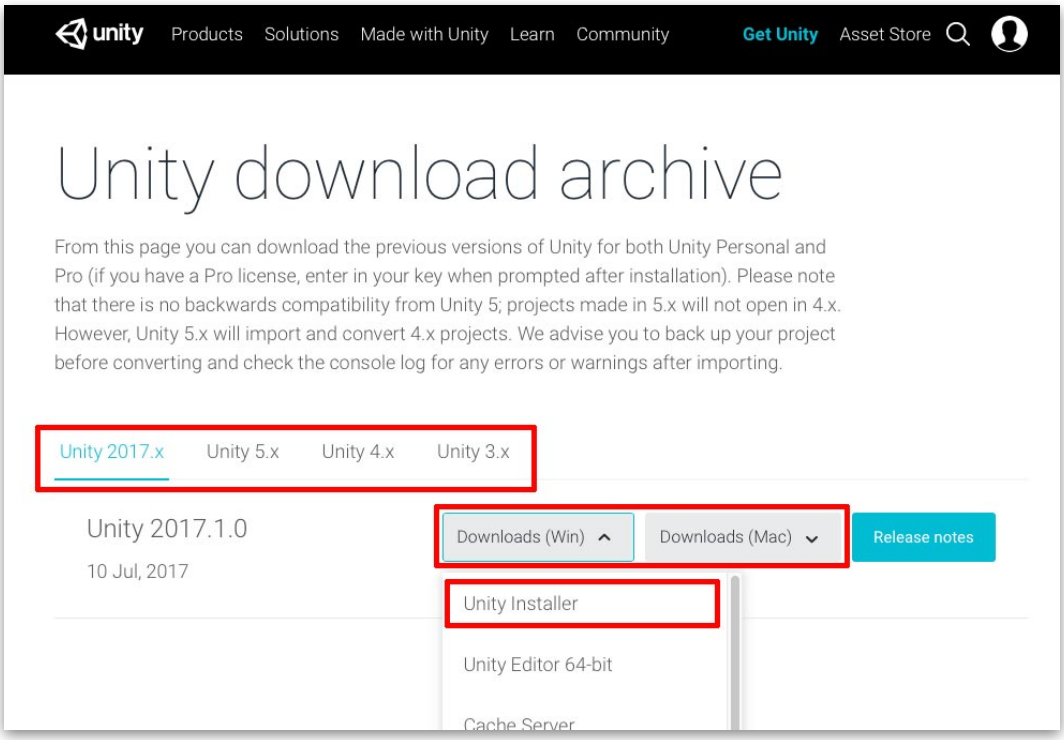
2.2. 下載指定版本

(1) 承上 (1)。

(2) 網頁最下方點擊 Older versions of Unity。



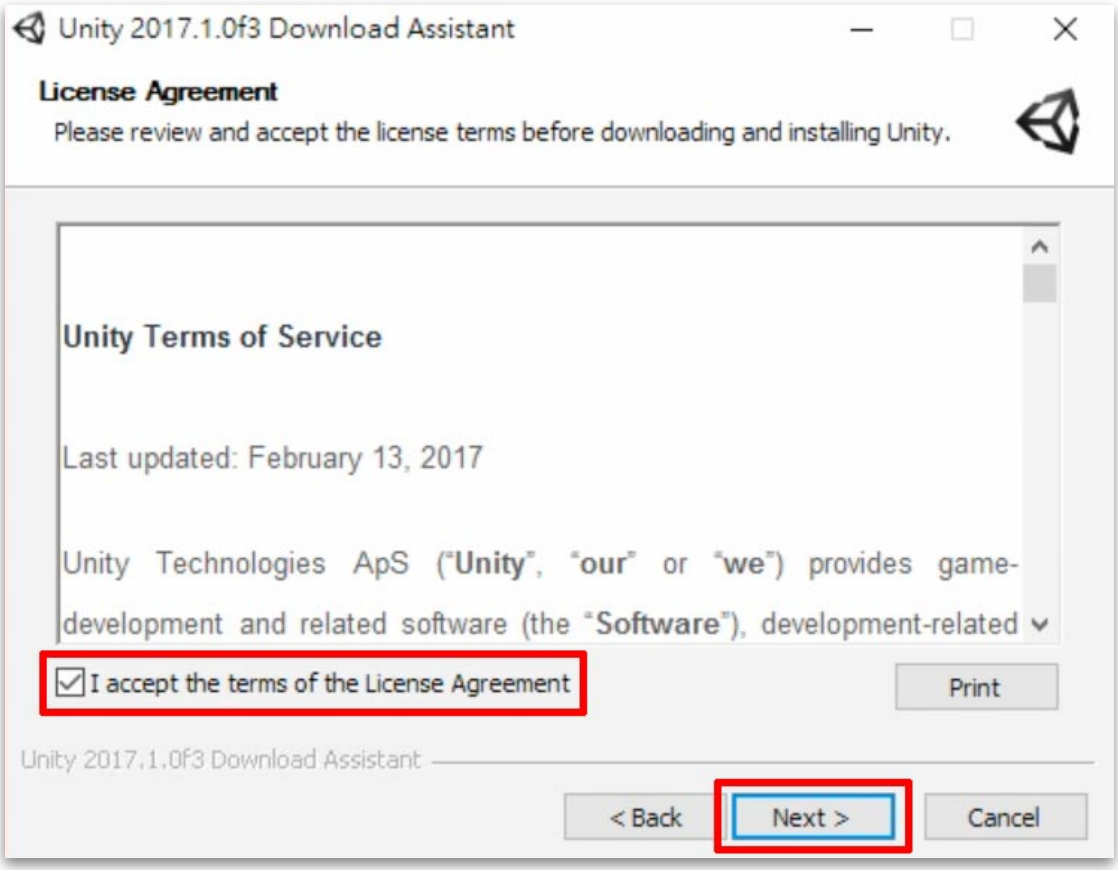
(3) 可依照需求選擇不同的 Unity 版本以及不同安裝平台的安裝程式。



3.安裝 *Unity* 開發引擎

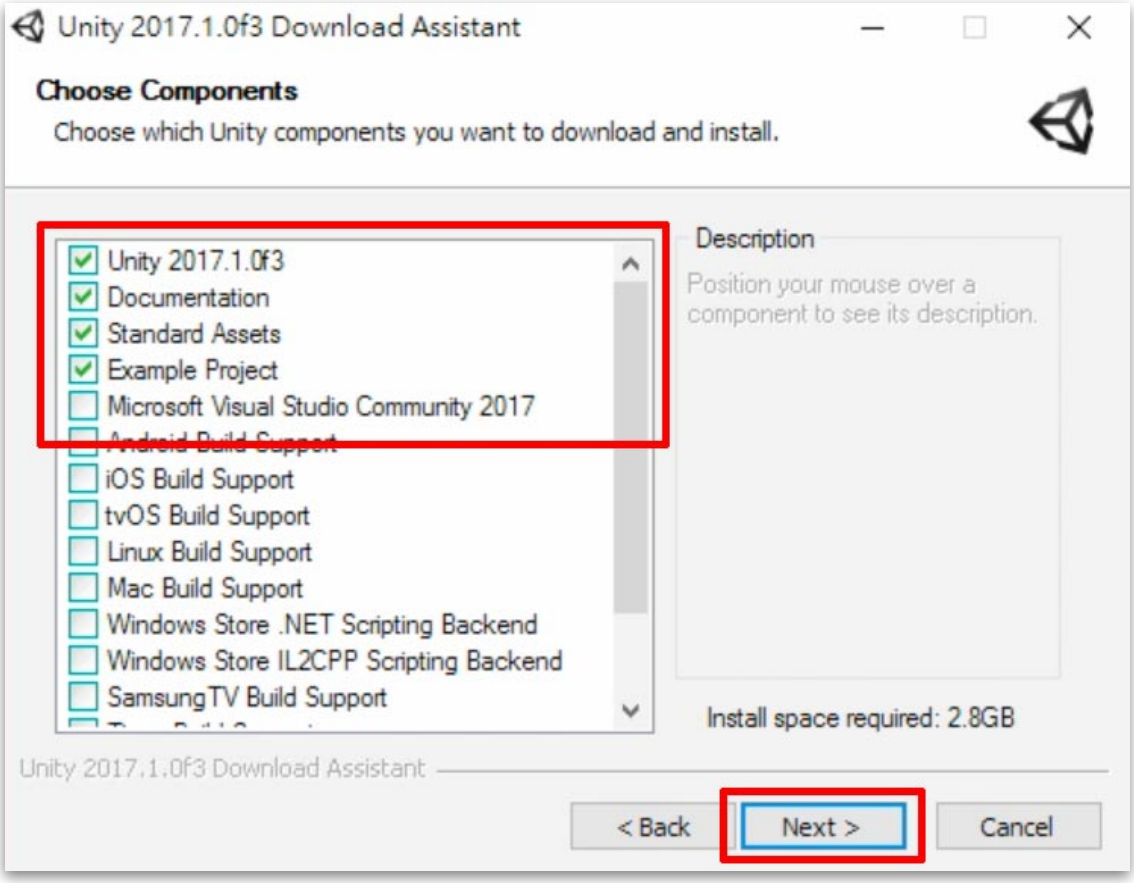
(1) 執行所下載的安裝程式（如 UnityDownloadAssistant 2017.1.0f3.exe）開啟安裝助理視窗，點擊 Next 按鈕。

(2) 勾選「I accept the terms of the License Agreement」同意許可證協議，並點擊 Next 按鈕。



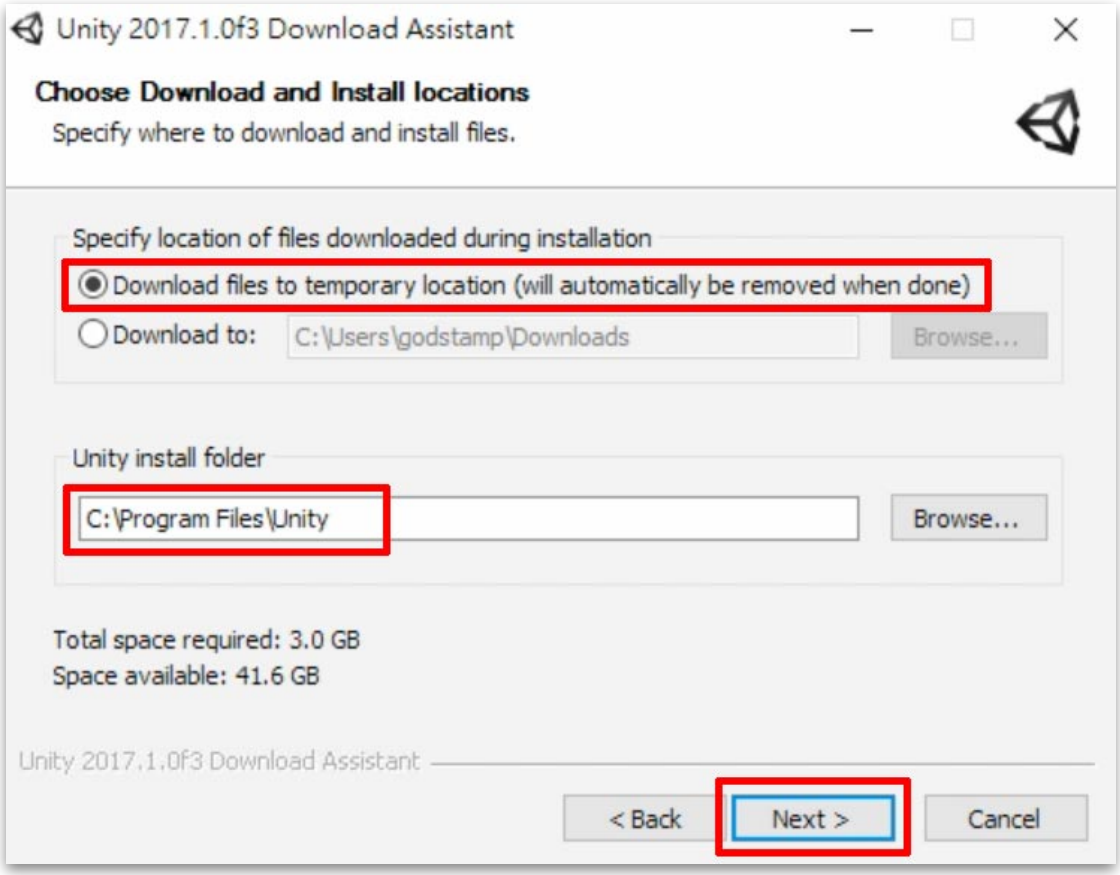
(3) 勾選需要安裝的項目，然後點擊 Next 按鈕。

- Unity 2017.1.0f3：Unity 編輯器主程式。
- Documentation：Unity 使用手冊與 API 參考文件。
- Standard Assets：Unity 內建提供的標準資源包。
- Example Project：使用 Standard Assets 所製作的範例專案。

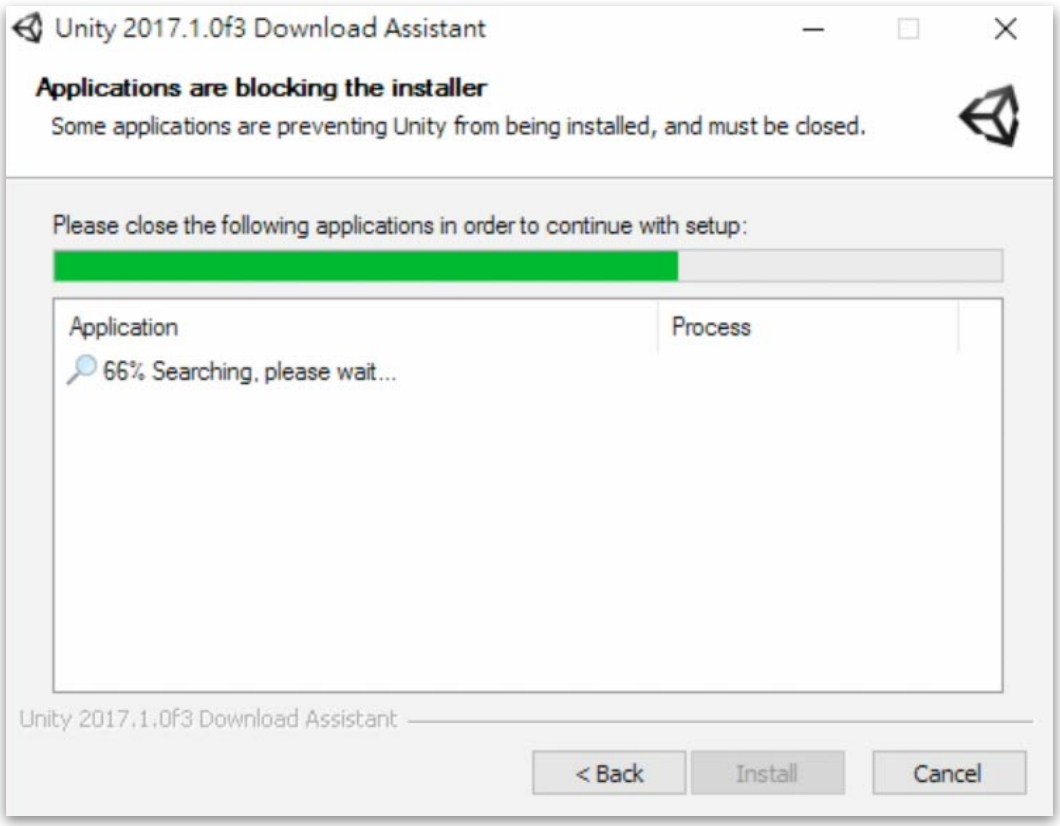


(4) 選擇下載與安裝位置，然後點擊 Next 按鈕。

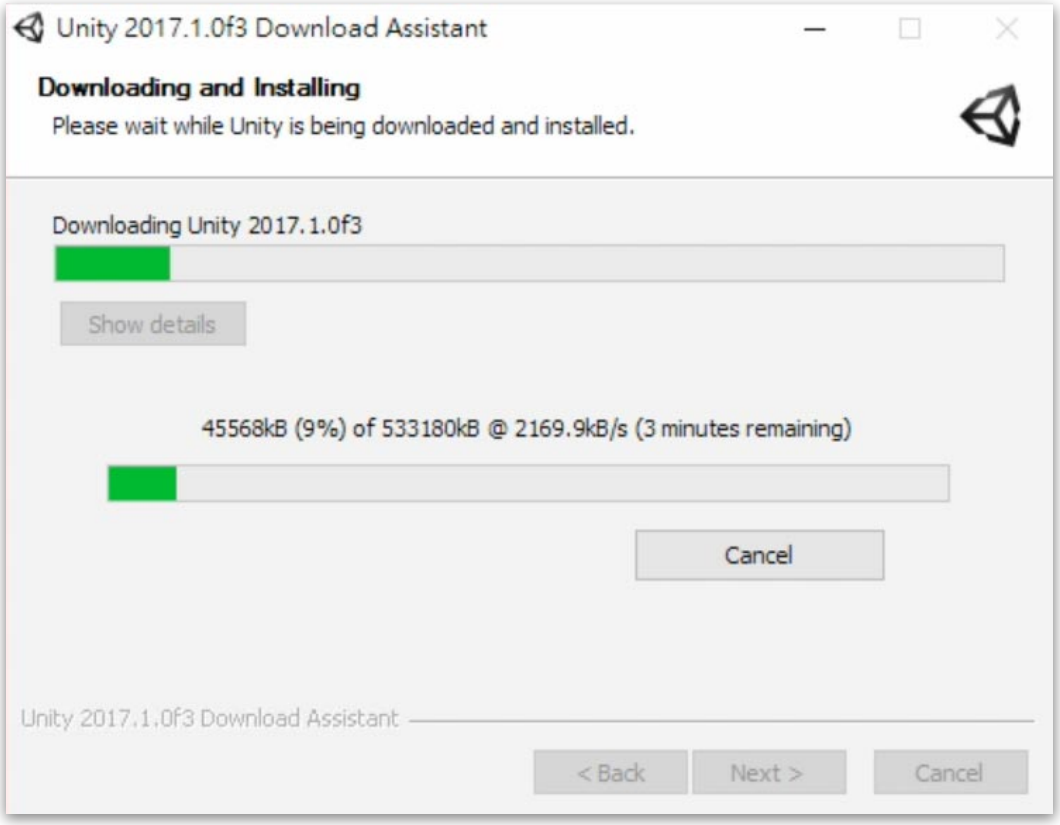
- 選擇 Download files to temporary location (will automatically be removed when done) 使安裝時下載到暫存位置並在安裝完畢後自動刪除。
- 確認 Unity 安裝的資料夾路徑，使用不同資料夾名稱可使電腦裡同時安裝多個不同版本的 Unity。



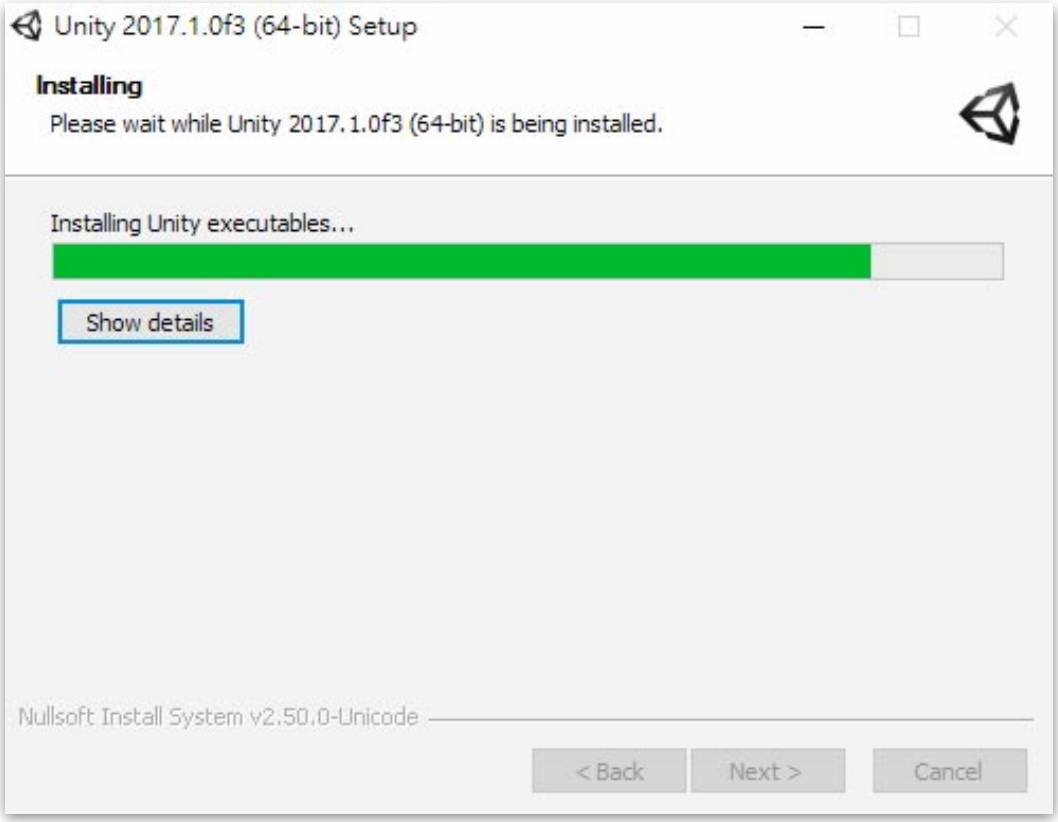
(5) 檢查是否有影響安裝程序的程式必須關閉。



(6) 下載所選擇需要安裝的相關檔案。



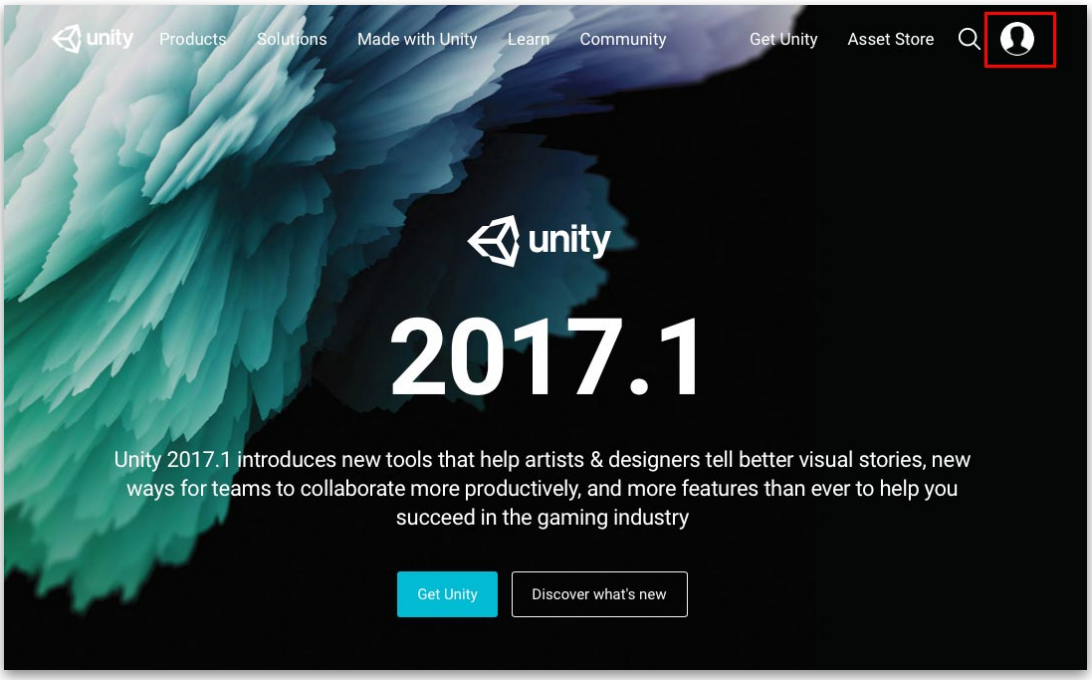
(7) 開始安裝。



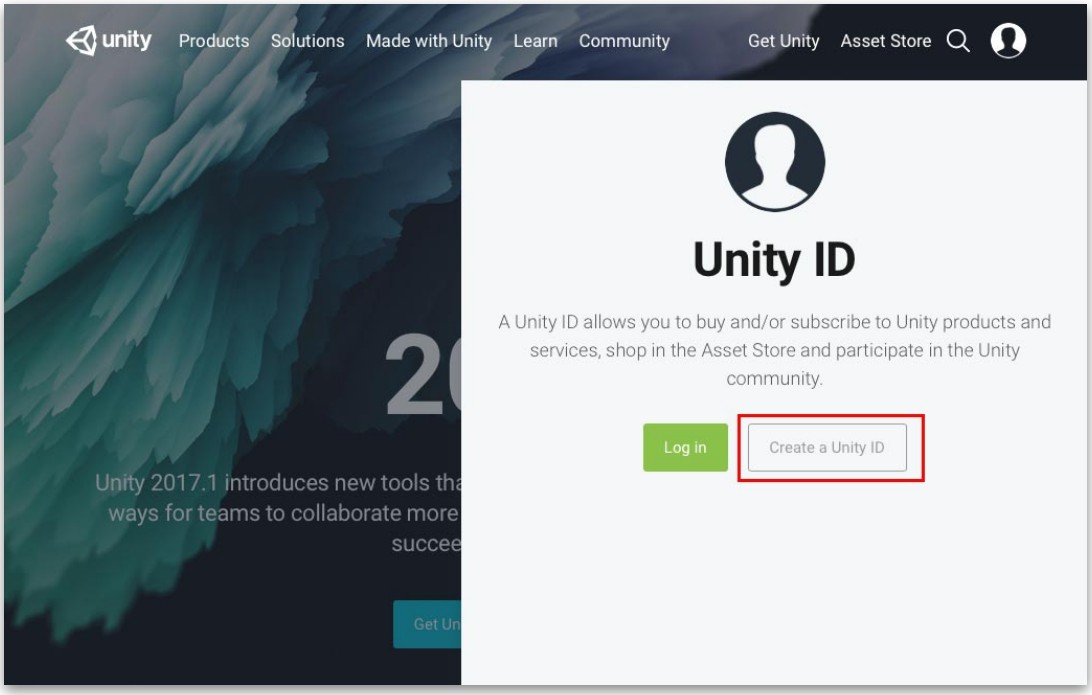
(8) 安裝完成，可勾選 Launch Unity（預設）表示直接執行 Unity，點擊 Finish 按鈕結束安裝。

4.建立 Unity ID

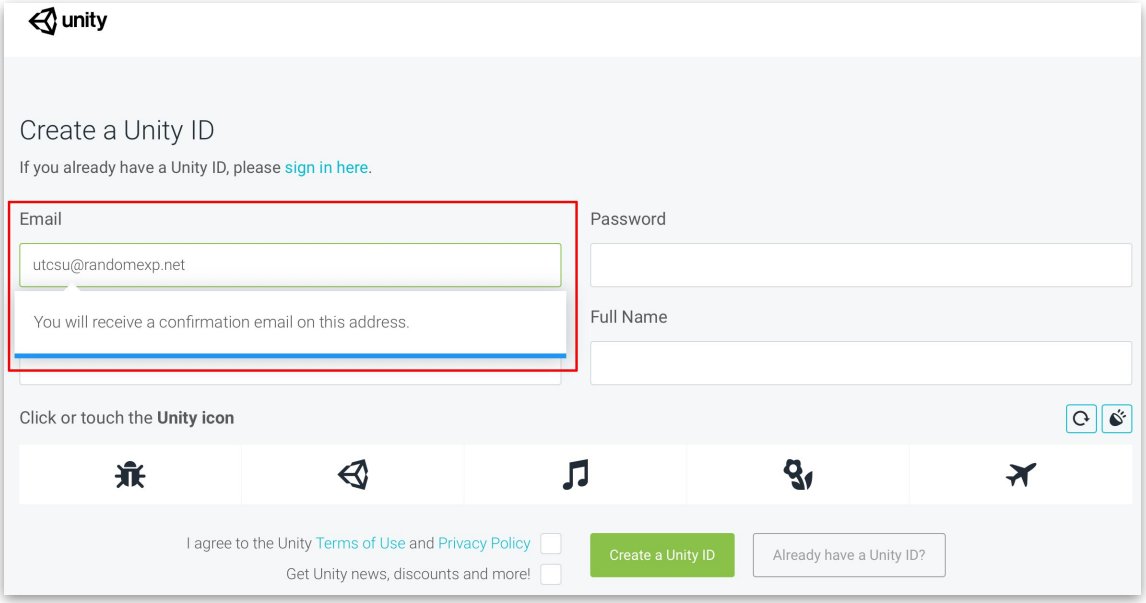
(1) 前往官方網站：https://unity3d.com，點擊右上角的頭像圖示。



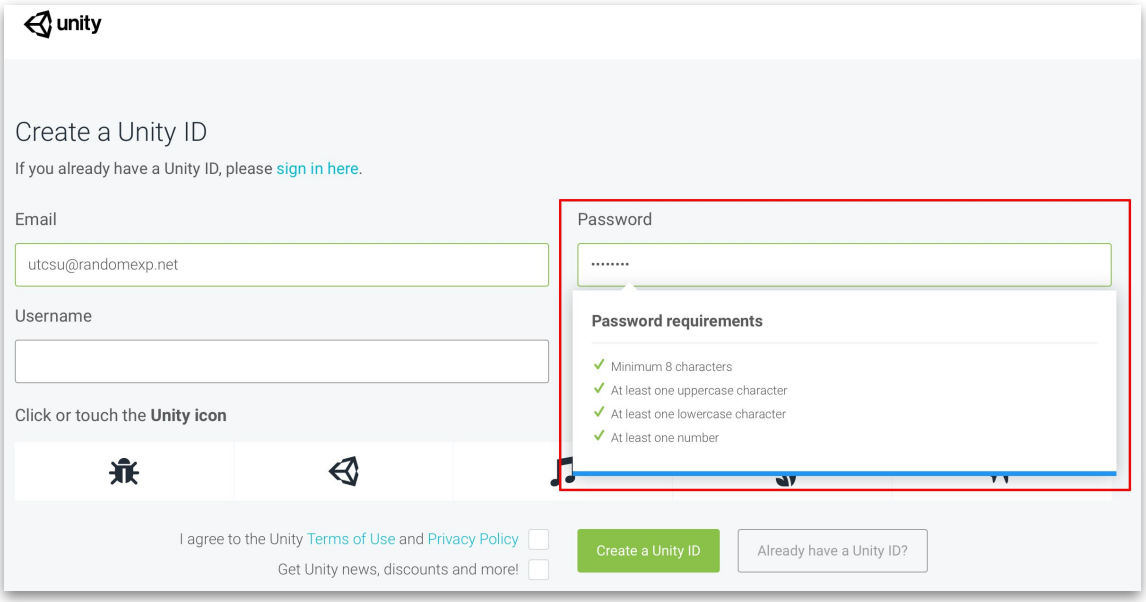
(2) 在展開的頁面區塊，點擊 Create a Unity ID 按鈕。



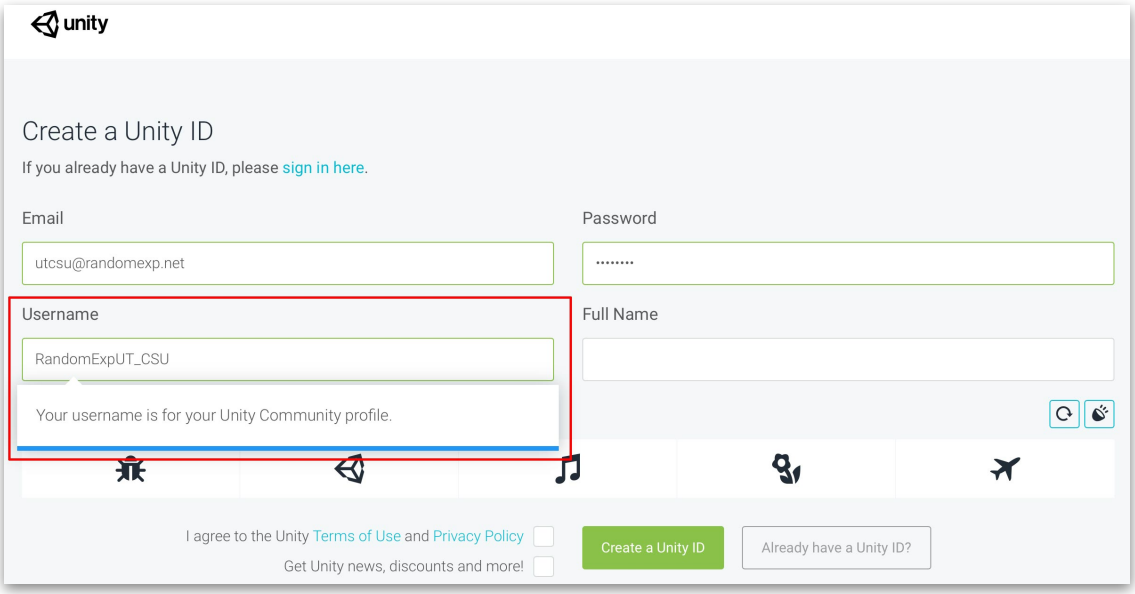
(3) 在 Email 欄位輸入電子信箱位址，此信箱將做為未來登入用的 Unity ID。



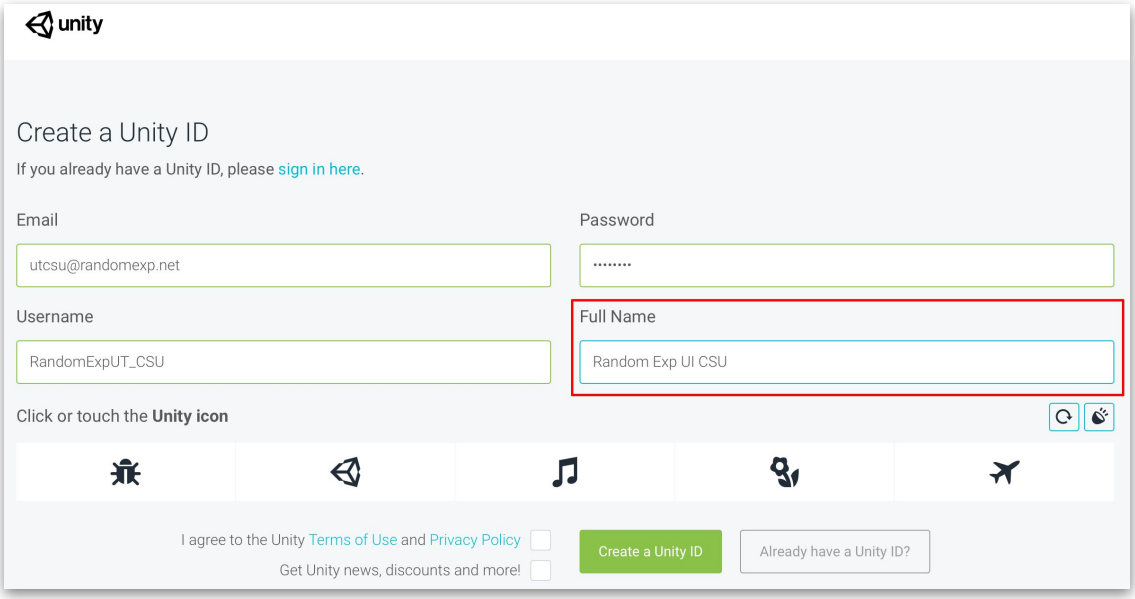
(4) 在 Password 欄位設定密碼，密碼必須至少 8 個字元並至少要有一個大寫字母、小寫字母、數字。



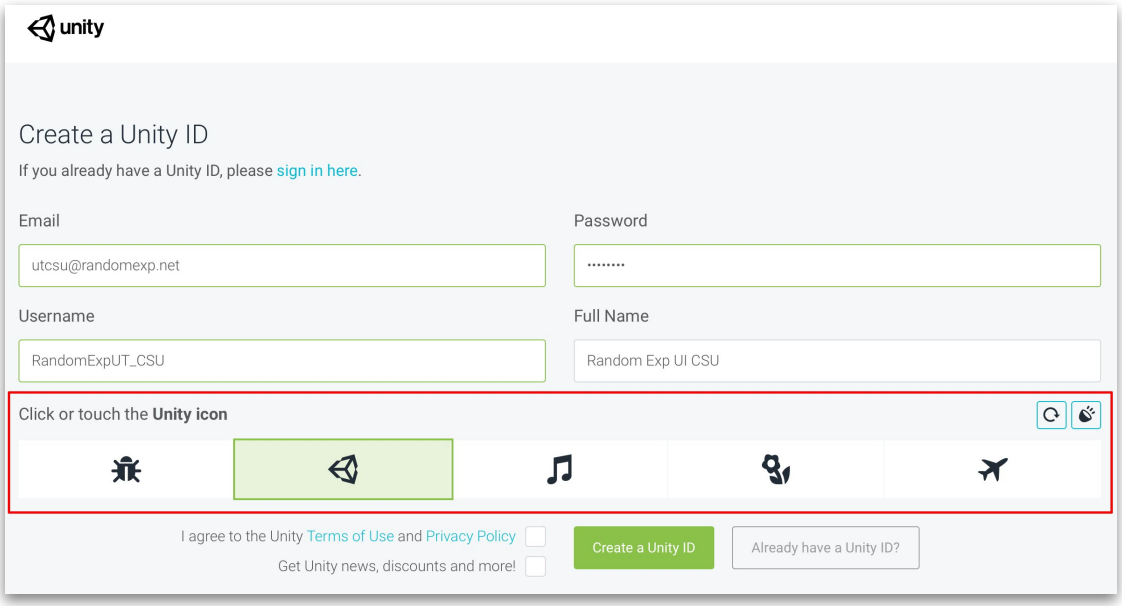
(5) 在 Username 欄位設定匿稱，此名稱將用於 Unity 社群論壇。



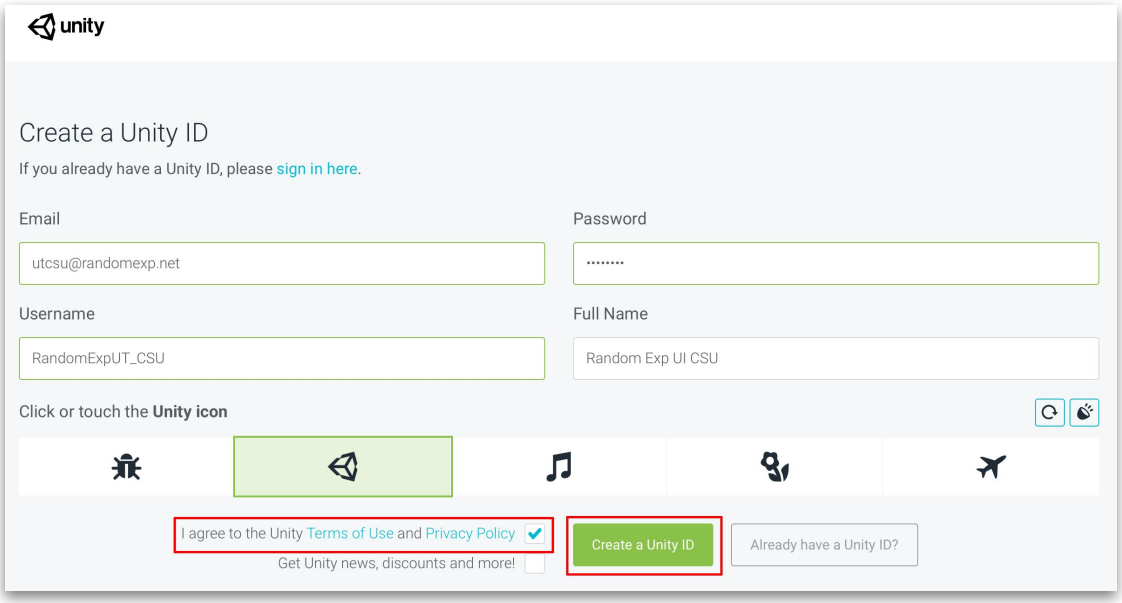
(6) 在 Full Name 欄位設定帳號資料全名。



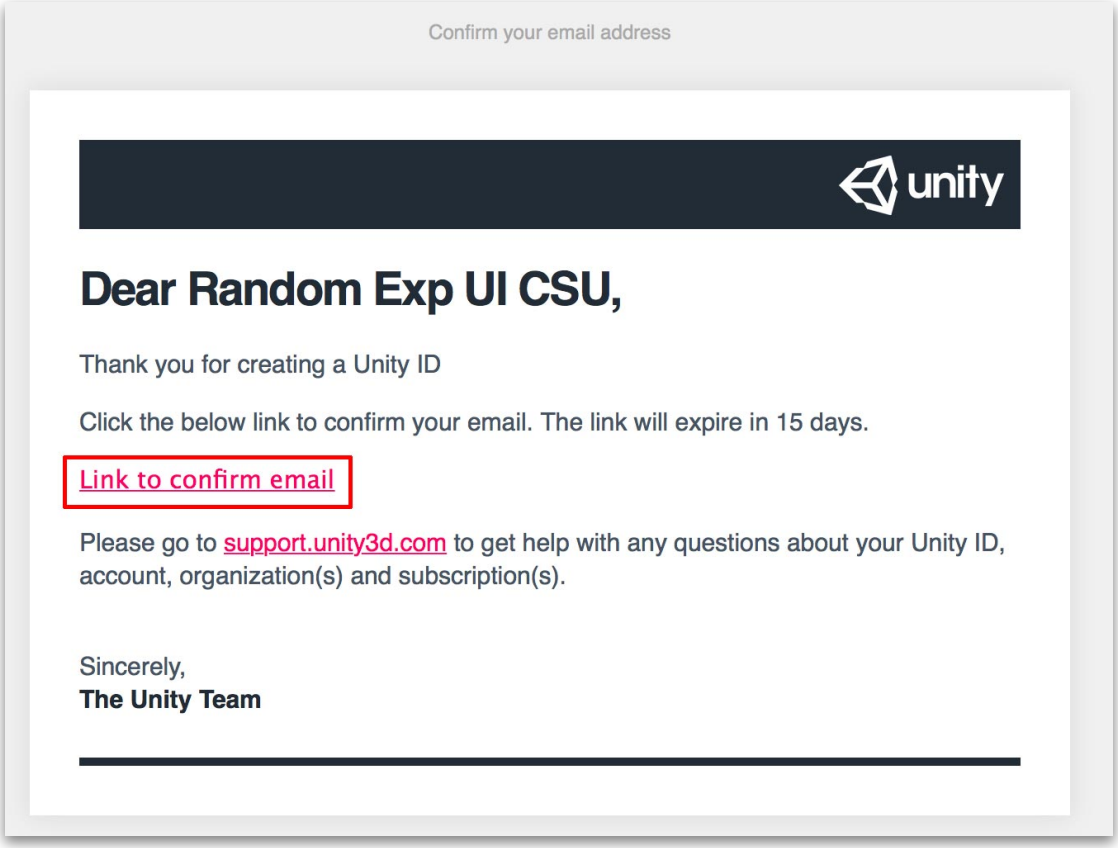
(7) 依據指示點擊驗證圖示，如下圖，指示為點擊 Unity 圖示。



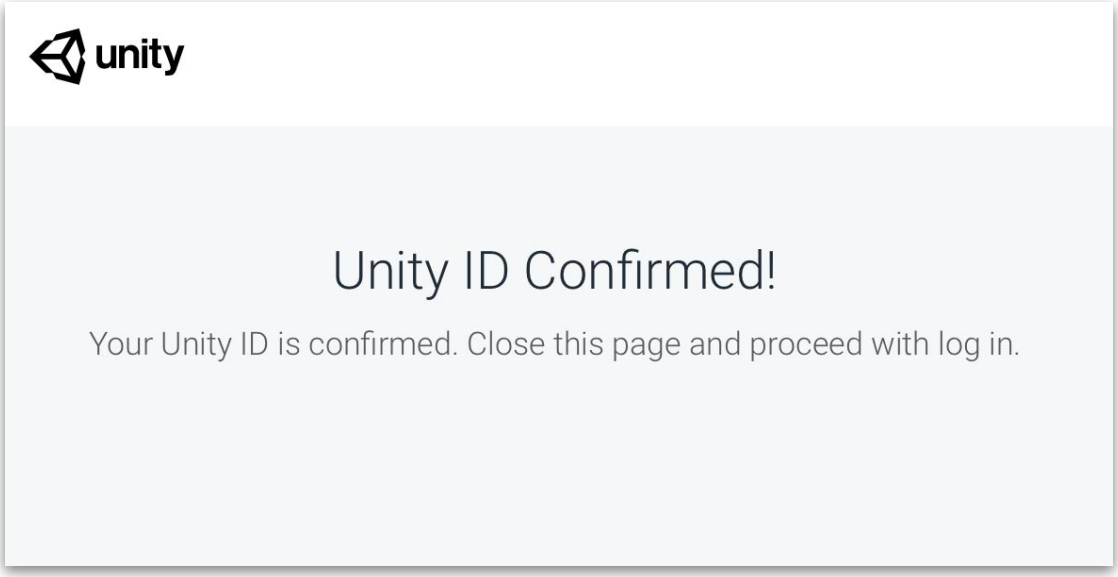
(8) 勾選「I agree to the Unity Terms of Use and Privacy Policy」表示同意使用條款和隱私政策，然後，點擊 Create a Unity ID 按鈕。



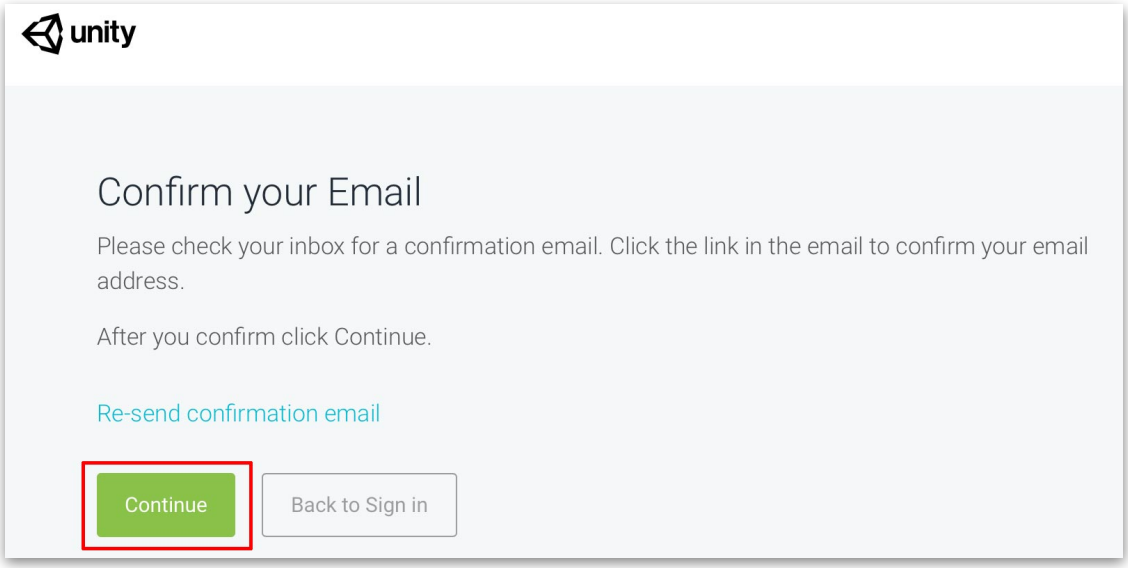
(9) 開啟信箱收取 Unity ID 確認信件，並點擊「Link to confirm email」連結確認電子信箱位址。

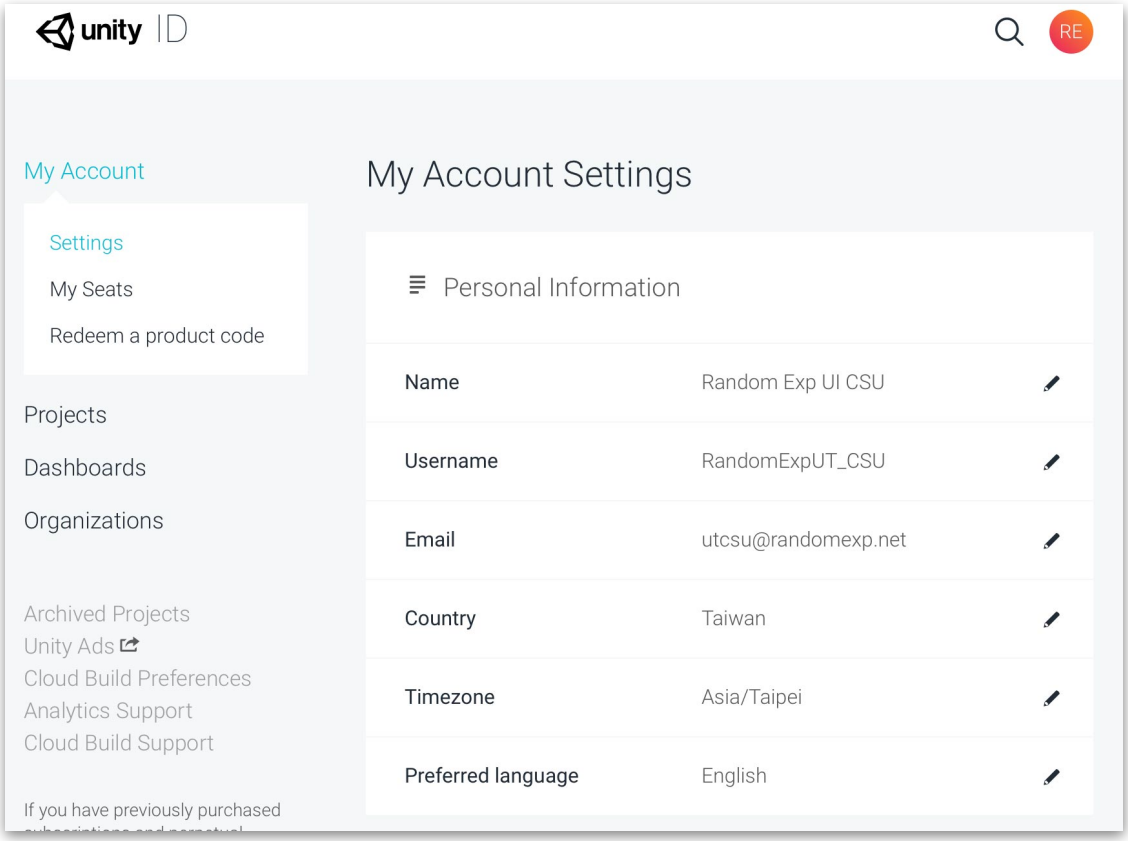


(10) 你的 Unity ID 已確認，關閉頁面並繼續登入。



(11) 回到建立 Unity ID 的頁面，點擊 Continue 按鈕即可登入。

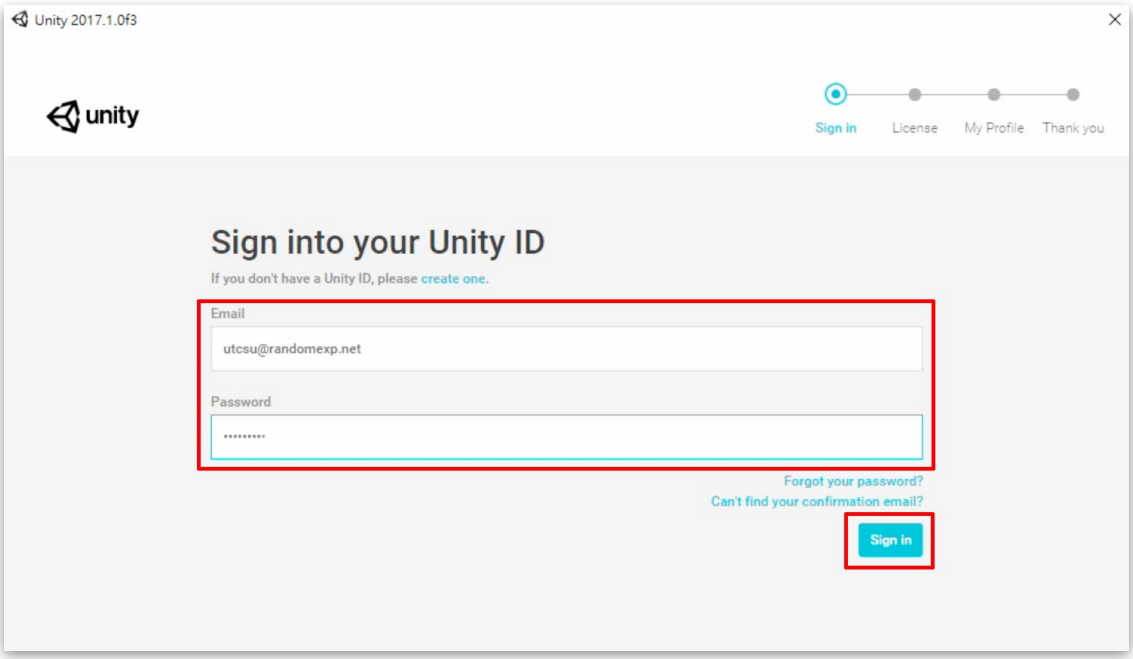




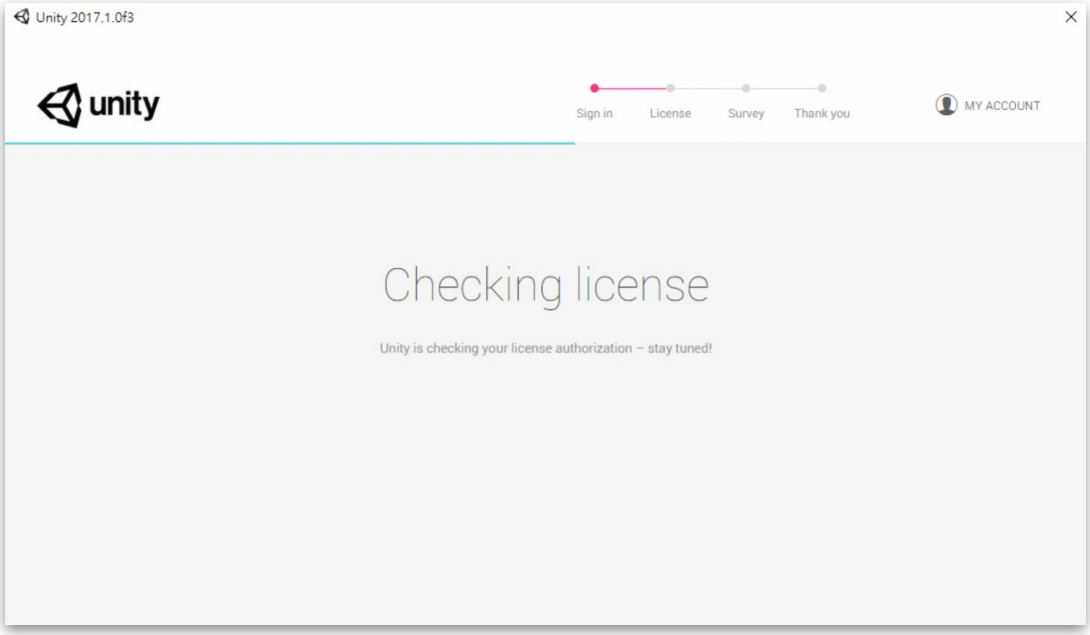
5.啟用授權

(1) 開啟 Unity 2017.1。

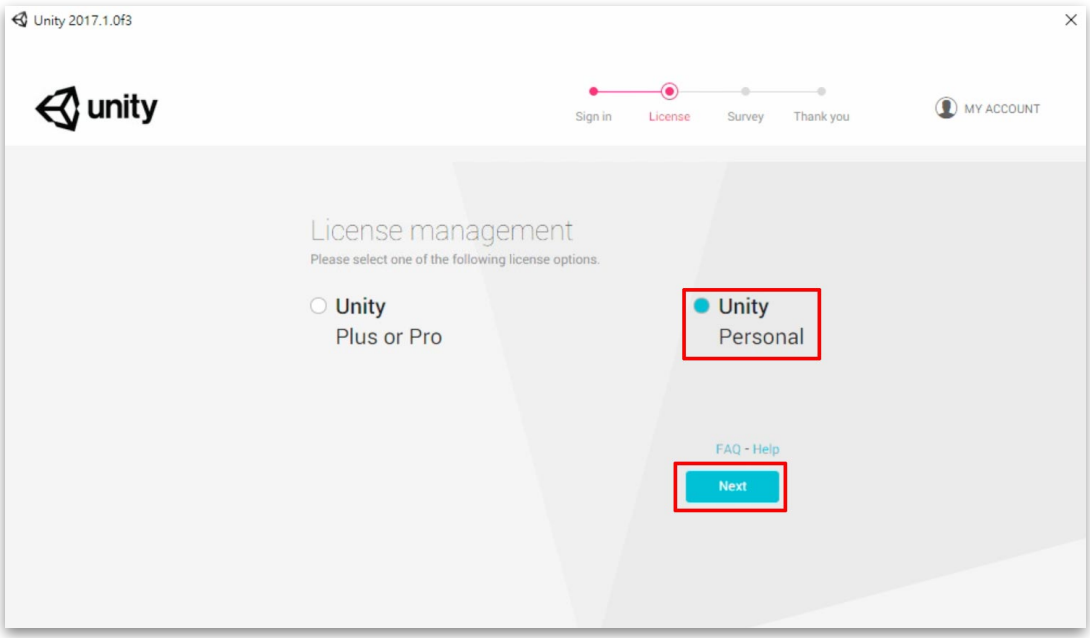
(2) 輸入建立 Unity ID 的信箱和密碼，然後，點擊 Sign in 按鈕。



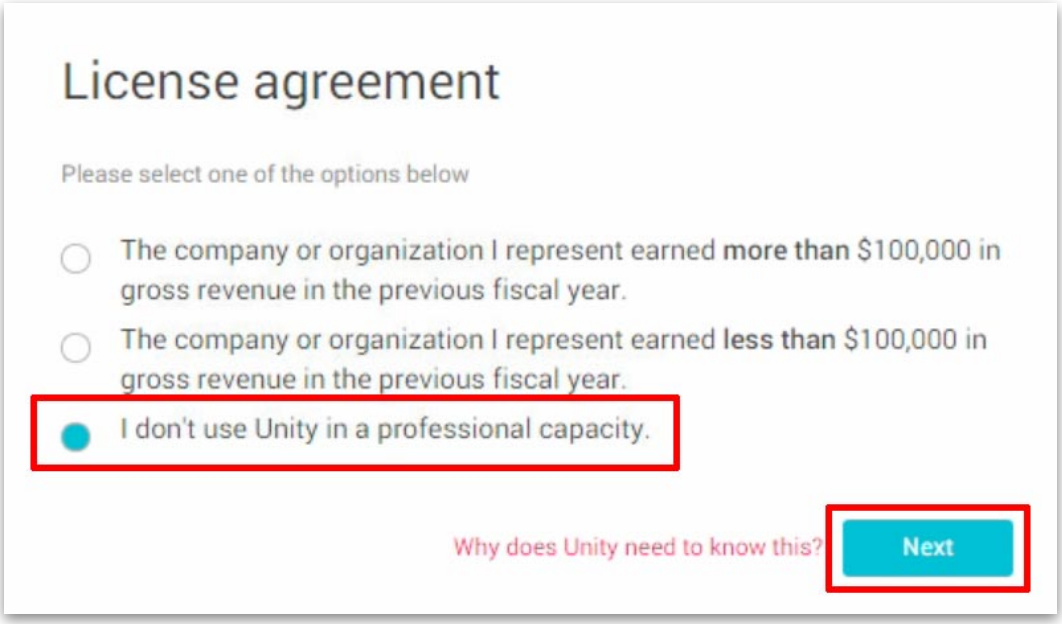
(3) 確認許可證授權。



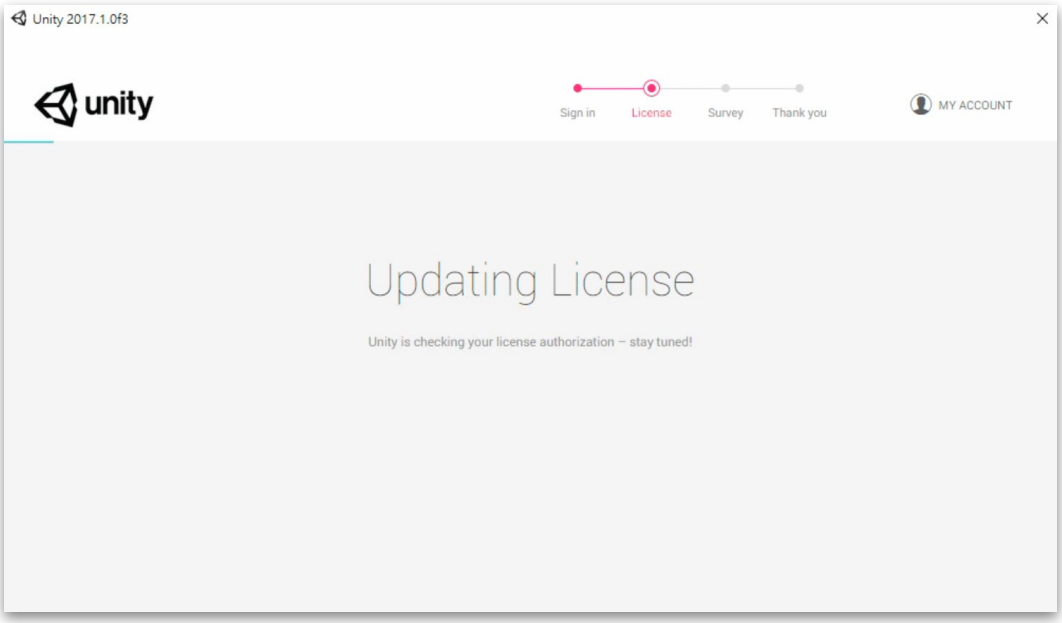
(4) 選擇個人版（Personal），並點擊 Next 按鈕。



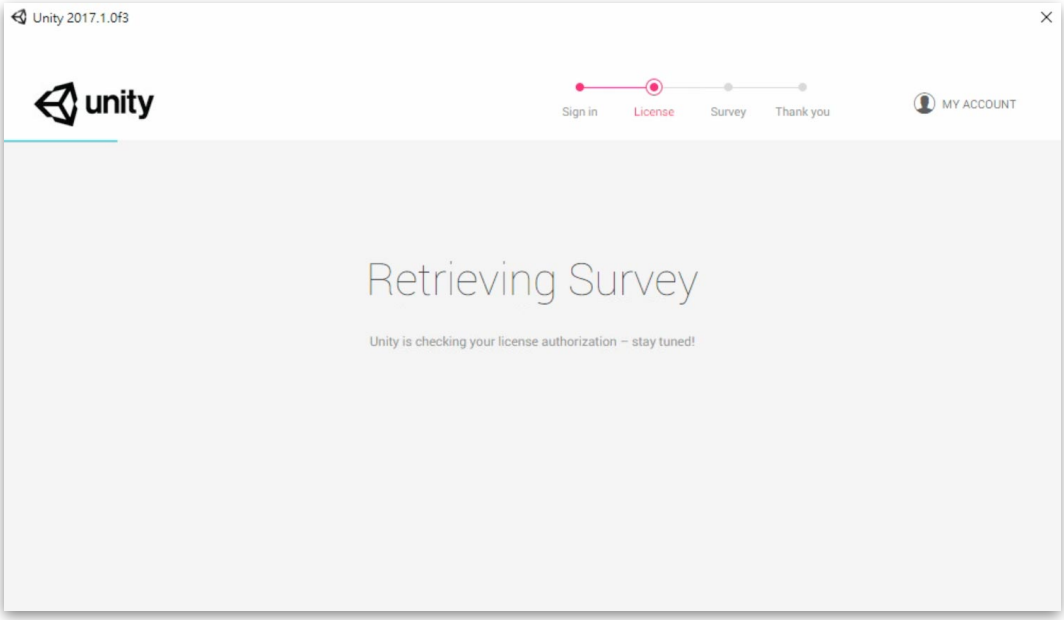
(5) 許可證協議選擇「I don't use Unity in a professional capacity」，並點擊 Next 按鈕。



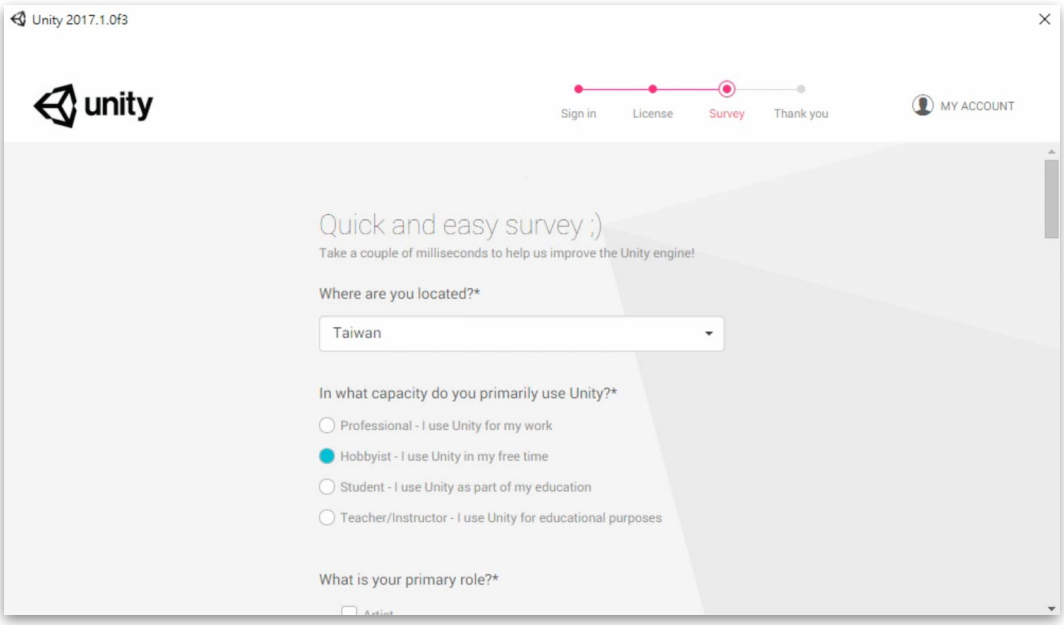
(6) 再次確認並更新許可證授權。



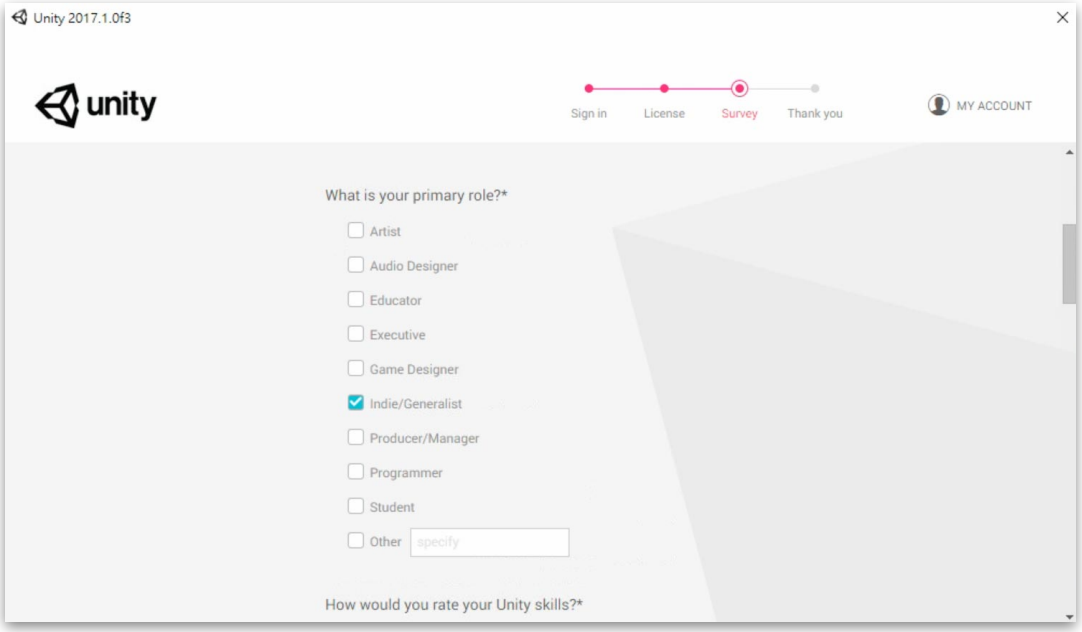
(7) 檢索調查。



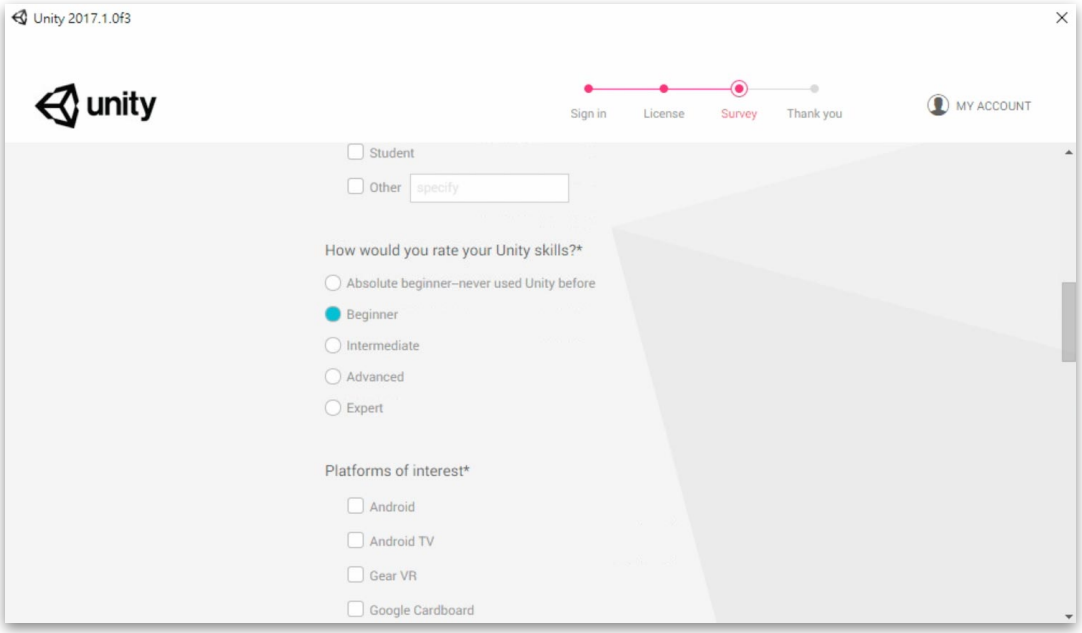
(8) Unity ID 初次使用 Unity 必須填選的問卷調查。選擇所在地以及主要使用 Unity 的能力。



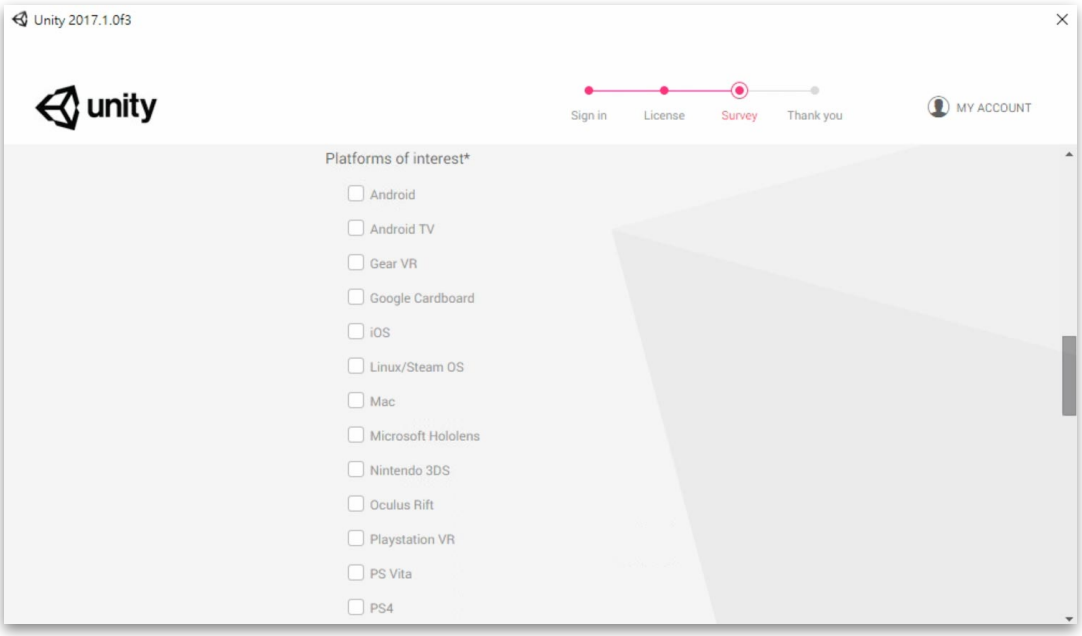
(9) 選擇主要角色是什麼。

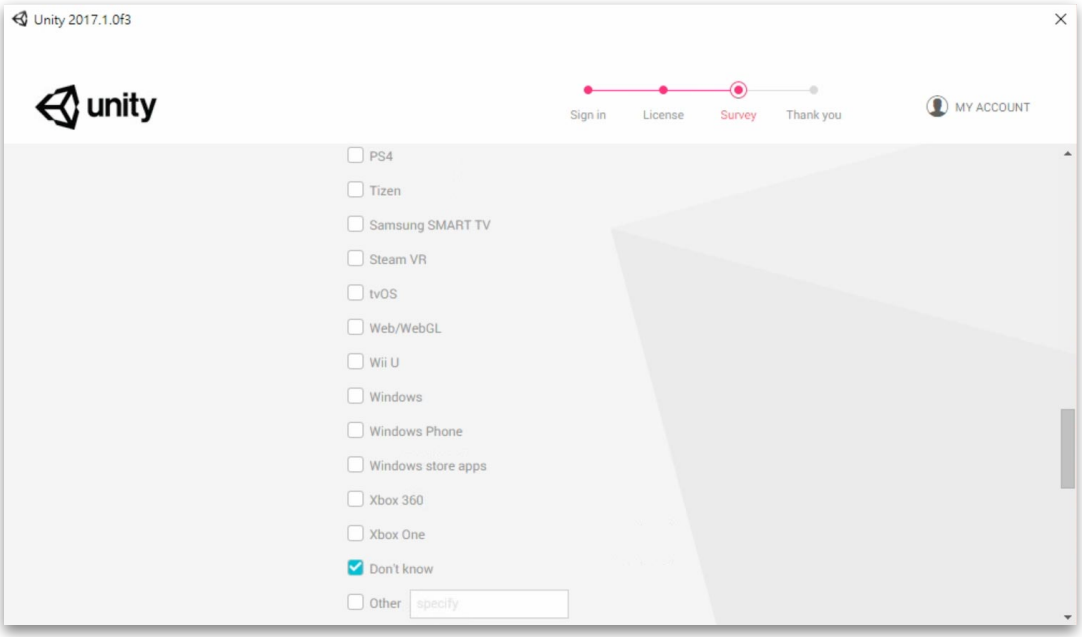


(10) 選擇自我評價 Unity 技能的程度為何。

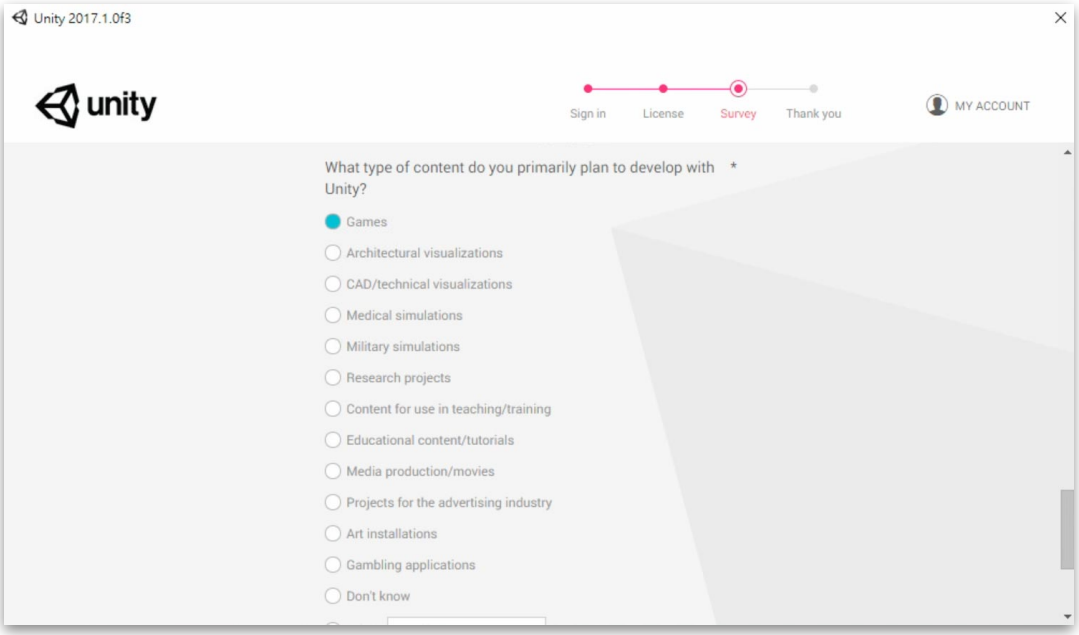


(11) 選取對哪些平台感興趣。

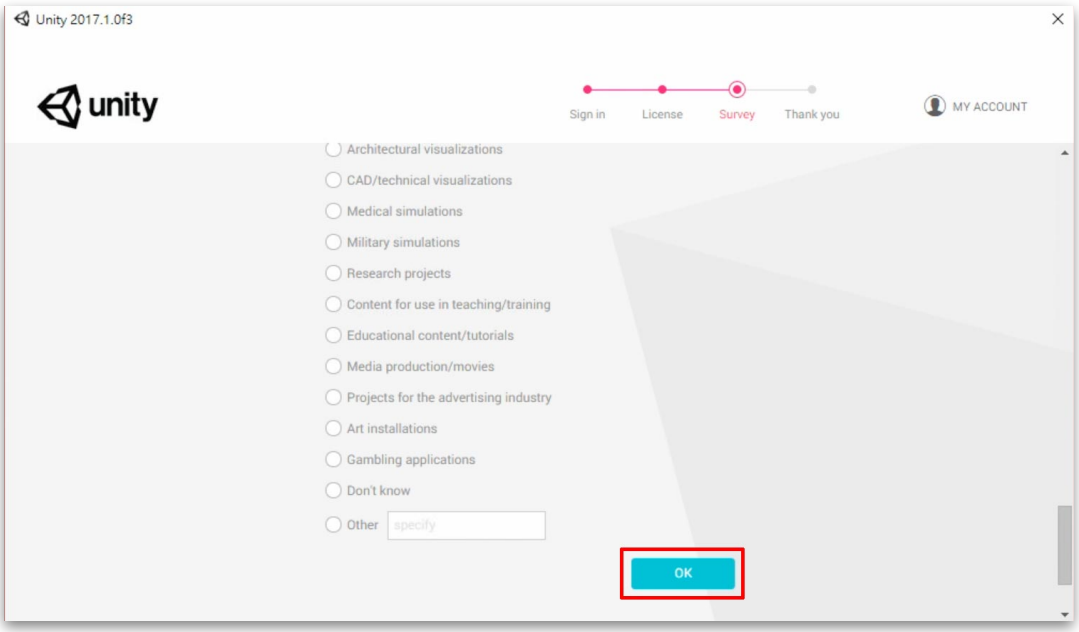




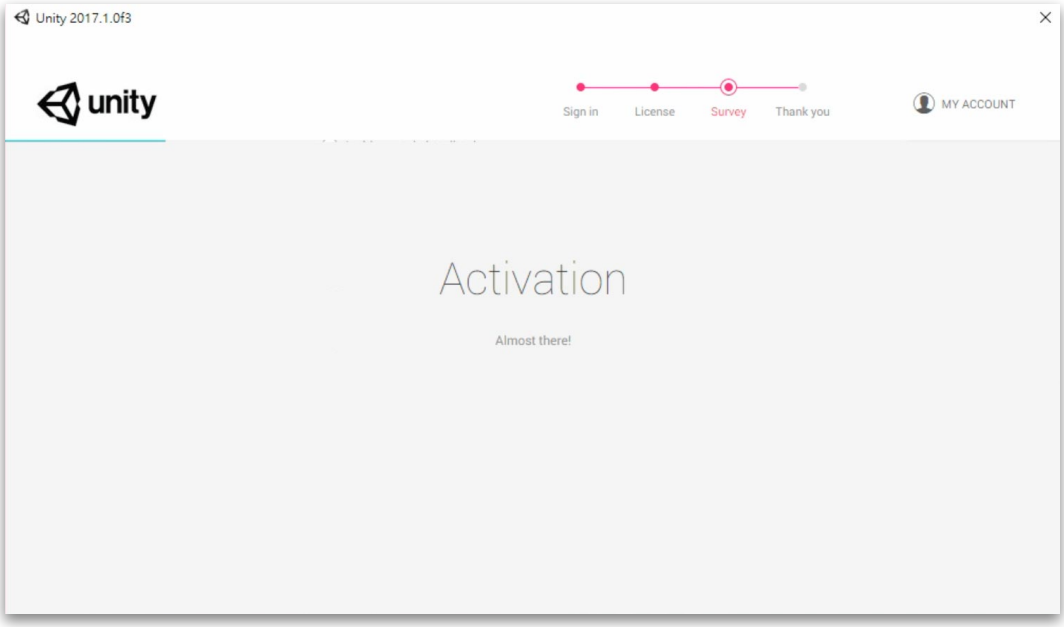
(12) 選擇主要計畫使用 Unity 來開發哪種類型的內容。



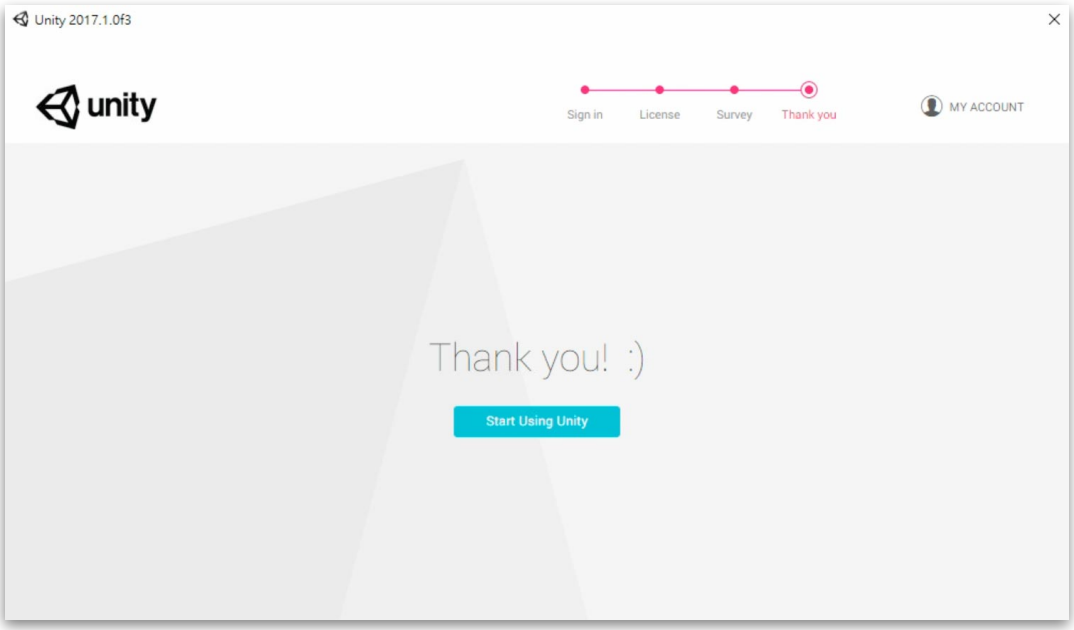
(13) 選填完畢後，點擊 OK 按鈕。



(14) 完成啟用。



(15) 開始使用 Unity。



II.開發與學習資源

有很多的方法可以幫助學習 Unity，從官方所提供的學習資源，可以找到任何能夠幫助你成為 Unity 開發者的東西。同時，不管是在開發或學習上，勢必也需要各種資源檔案來提供專案及其內容使用，

1.教程 *Tutorials*

https://unity3d.com/learn/tutorials

Unity 官方網站所提供的教程，主要分為兩個部分：

- 專案式：以遊戲實作專案為主，依照步驟來逐步完成專案內容，透過實作來有效理解引擎功能與實務使用，包含滾球、太空射擊、生存射擊、坦克對戰、2D 程序化關卡、冒險遊戲、洞穴生成等。
- 主題式：更詳細的分析及舉例解說引擎各系統或功能的系列教學，如編輯器介面基礎、2D 遊戲創作、程式腳本、圖像、物理、聲音、動畫、使用者介面、手機觸碰、導航、廣告與統計、虛擬實境、效能改善、多人連線等等。

2.文件資料 *Documentation*

Unity 官方網站或可隨 Unity 主程式一起安裝的文件資料，提供最詳細的說明和簡單範例，主要分為兩個部分：

- 使用手冊：https://docs.unity3d.com/Manual/index.html
  提供有關如何使用 Unity 編輯器以及相關服務的詳細圖文說明，每個頁面皆可切換到不同的 Unity 版本查看其相關說明，主要章節包含有完整的編輯器介紹、Unity 2D、圖像、物理、網路、程式腳本、聲音、動畫、使用者介面、導航、Unity 服務、虛擬實境、指定平台等等。
- 編程參考：https://docs.unity3d.com/ScriptReference/index.html
  包含詳細的編寫程式所需要的 API 參考資料，程式寫作所必須閱讀與查找的資料都在這裡，提供各種功能的解說和程式範例，為使用 Unity 開發遊戲、應用程式或開發工具等最核心的資料資源。

3.直播教學 *Live training*

https://unity3d.com/learn/live-training

不定期針對各種主題，由 Unity 原廠提供的專家在線上以現場直播的方式開講，使學員能夠透過網路即時互動、提問；開播之前會預告主題以及開播時間倒數，每次大約 1~2 小時，如果未能及時參與，也能於網站上查找存檔影片學習。

4.認證開發者課件 *Certified Developer Courseware*

https://certification.unity.com/courseware

更完整的系列教學課程資源，需要付費購買才可使用，另外，訂購 Unity Plus、Unity Pro、Unity Enterprise 的用戶，可享有 1~3 個月的免費使用權。

5.資源商店 *Asset Store*

https://www.assetstore.unity3d.com/

在開發或學習的過程中，透過 Asset Store 中的資源可以節省時間、提高效率。包含人物模型、動畫、粒子特效、紋理、遊戲創作工具、音效 / 音樂、可視化程式編寫方案、功能程式腳本或其他各類擴充插件等都能在這裡免費或付費的方式獲得。

III.2D 與 3D 遊戲專案概念

Unity 是同時適用於 2D 和 3D 遊戲的開發工具。當你在 Unity 中建立一個新的專案，你必須先選擇是要從 2D 或 3D 模式開始。你有可能已經知道你想要建立的是什麼，但是，有幾個細微的問題可能會影響你的選擇。

1.全 3D

3D 遊戲通常是使用具備材質和紋理的三度空間幾何描繪在遊戲物件的表面，使它們呈現立體般的環境、人物角色以及構成遊戲世界的物件。攝影機可以自由的在場景內和周圍自由的移動，而光源和陰影則以逼真的方式投射到世界的每個地方。

3D 遊戲通常使用透視的方式來描繪場景，所以物件越靠近到攝影機，則在畫面上呈現的越大。

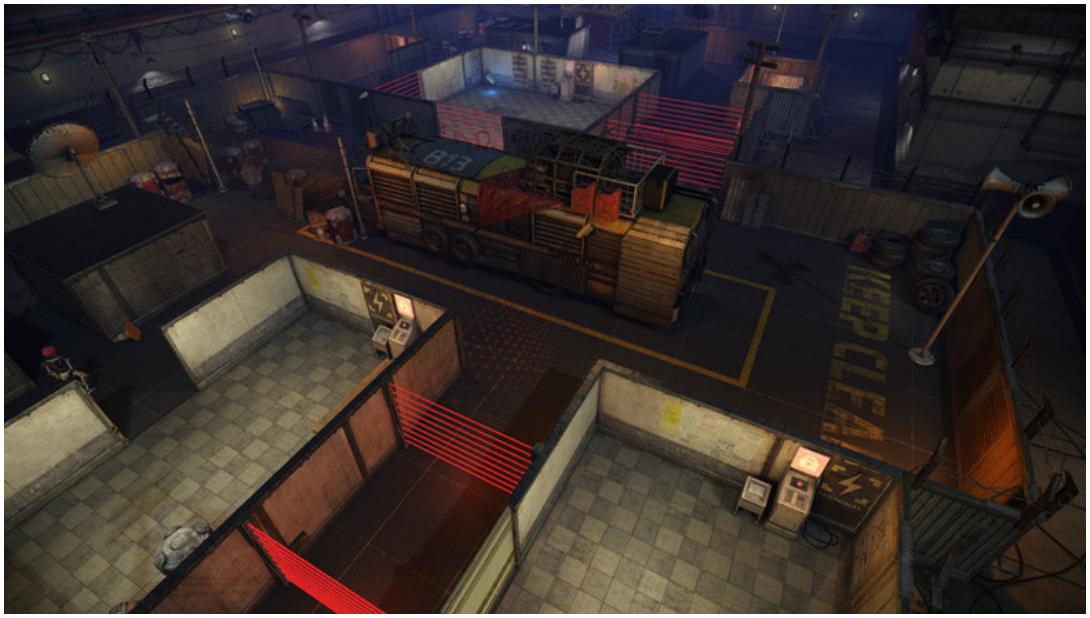
對於適合以上描述的所有遊戲，都可以 3D 模式開始。

以下為 Unity 官方放於 Asset Store 上的示範專案 3D 場景：

- Vikin Village：
  https://www.assetstore.unity3d.com/#!/content/29140

- Corridor Lighting Example：
  https://www.assetstore.unity3d.com/#!/content/33630

- Stealt（Unity 4x）：
  https://www.assetstore.unity3d.com/#!/content/7677



2.正交 3D

有時候，遊戲使用 3D 幾何，但是攝影機使用正交來取代透視。這在遊戲上是很常見的技術，提供給你一個鳥瞰的視野來運作，有時候，這也被稱為「2.5D」。

如果你正在製作一個像這樣的遊戲，你應該也是使用 3D 模式，因為雖然這樣沒有透視，但你仍然是使用 3D 模型和資源來運作。

此時，你必須將攝影機和場景視窗切換為正交（Orthographic）。

3.全 2D

許多 2D 遊戲使用平面圖像，有時候也稱為 Sprite，它們根本都沒有三度空間幾何。

它們做為平面圖像直接繪製到螢幕上，而且遊戲的攝影機沒有透視。

對於這種類型的遊戲，應該以 2D 模式啟動編輯器。

Unity 官方在 Asset Store 提供的 2D 專案：

- 2D Platformer：
  https://www.assetstore.unity3d.com/#!/content/11228

- 2D Roguelike：
  https://www.assetstore.unity3d.com/#!/content/29825

- 2D Roguelike：
  https://www.assetstore.unity3d.com/#!/content/73728

4.具備 3D 圖像的 2D 遊戲

有些 2D 遊戲針對環境和角色使用 3D 幾何，但是將遊戲限制在二度空間上。例如，攝影機可能會顯示橫軸視野，並且玩家只能在二度空間裡移動，但是遊戲本身仍然使用 3D 模型的障礙物和 3D 透視視野的攝影機。

對於這類遊戲，3D 效果可能主要提供畫面風格，而非功能上的目的。這類的遊戲，有時候也會被稱為「2.5D」。

雖然是 2D 遊戲玩法，但你主要是操作 3D 模型來建立遊戲，因此應該以 3D 模式啟動編輯器。

5.具備透視攝影機的 2D 遊戲玩法和圖像

另一種很受歡迎的 2D 遊戲風格，使用 2D 圖像，但是帶有透視攝影機以獲得視差捲動的效果。

這是種「紙板劇場」風格的場景，這裡的所有圖像都是平面的，但是佈置在與攝影機不同距離位置。這種情況下，

2D 模式將會是最適合你的開發方式。然而，你應該將攝影機的投影模式改變為透視（Perspective）並將 Scene 視窗設置為 3D。

6.其它類型

或許你計劃的專案適合以上所介紹的其中一種，或者，你可能會有一些其它完全不同或獨特的東西。無論你的計畫是怎樣的，希望以上能帶給你啟動編輯器模式的一些想法。

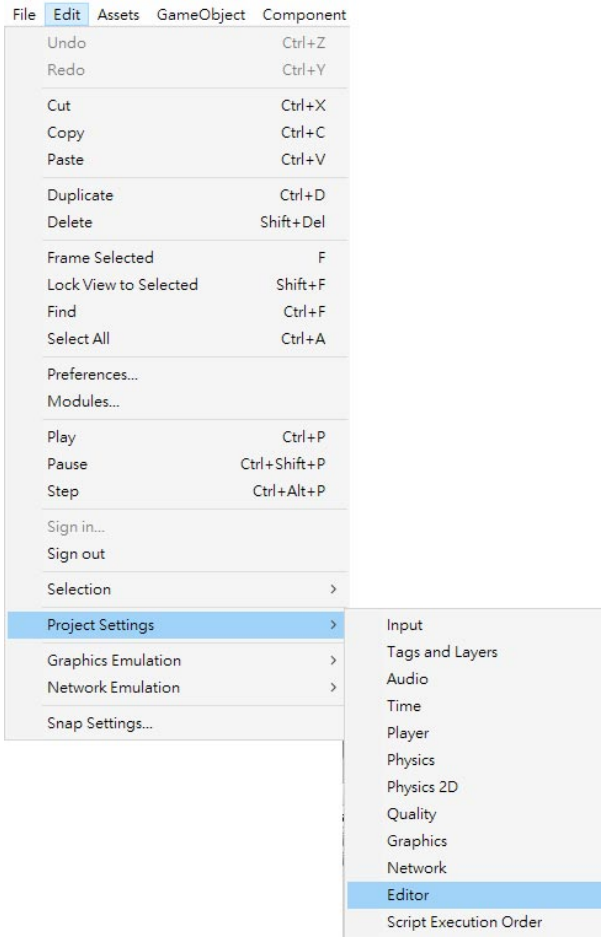
請記住，你可以隨時切換模式。

7. 2D 和 3D 模式設定

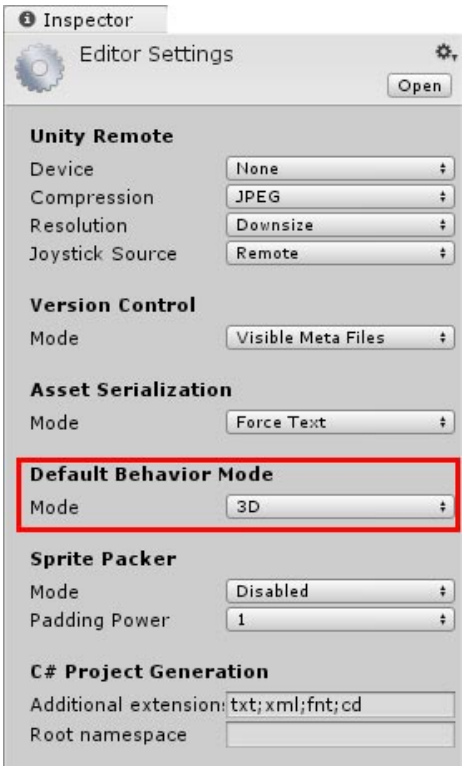
當建立一個新專案，你可以指定是要從 2D 或是 3D 模式開啟 Unity 編輯器。無論如何，你還可以隨時在 2D 和 3D 模式之間切換編輯器。

7.1. 在 2D 和 3D 模式間切換

(1) 透過選單 Edit > Project Settings > Editor 開啟編輯器設定 - Editor Settings。



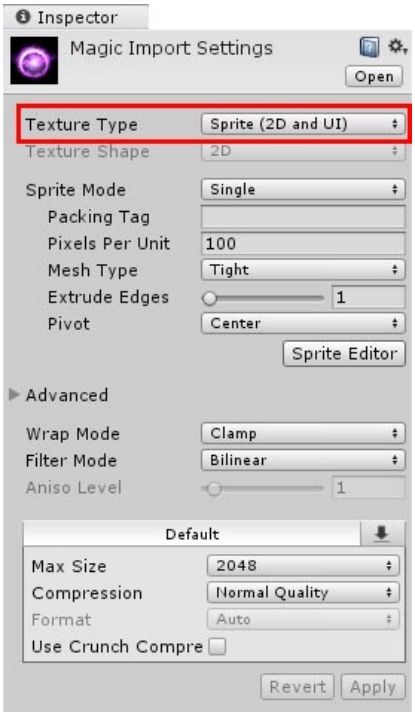
(2) 設置 Defaut Behavior Mode 為 2D 或 3D。



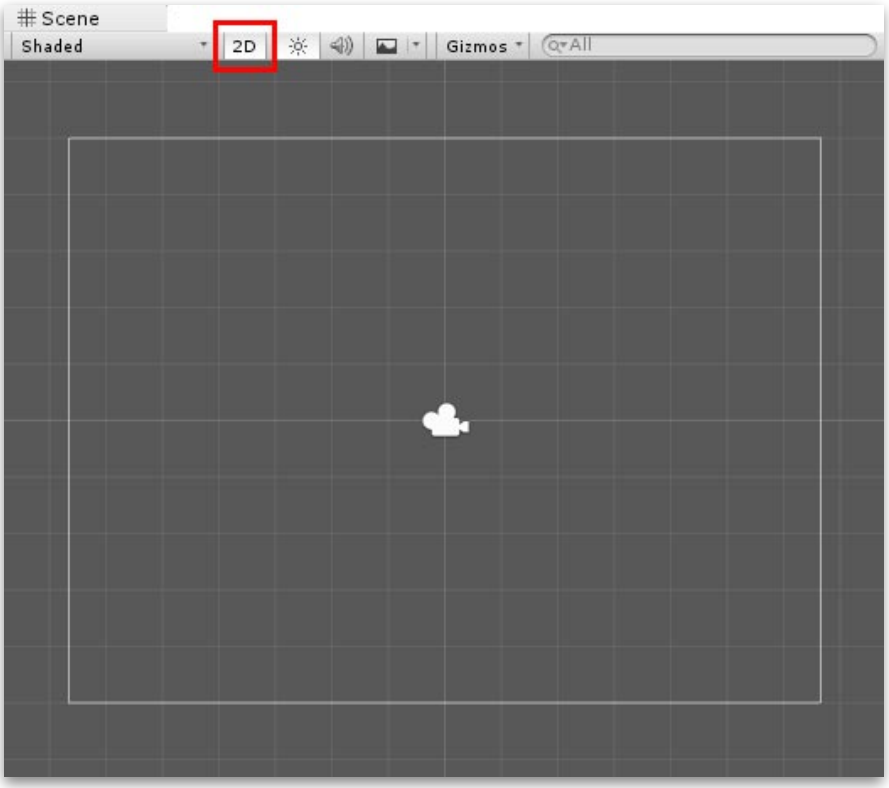
7.2. 2D 和 3D 模式的影響

(1) 2D 模式：

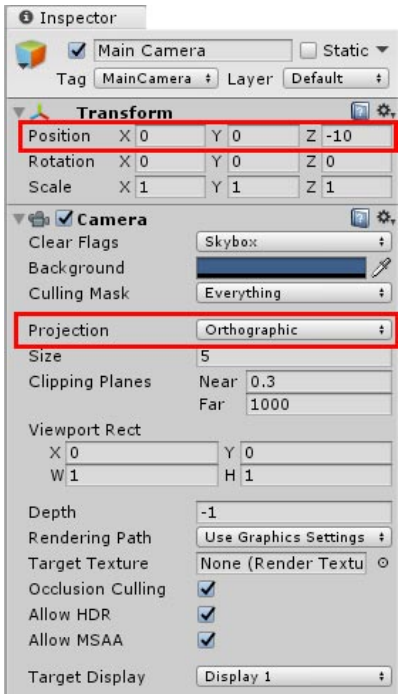
- 假定匯入的所有圖像都是要做為 2D 圖像（Sprite）使用並預設為 Sprite 模式。



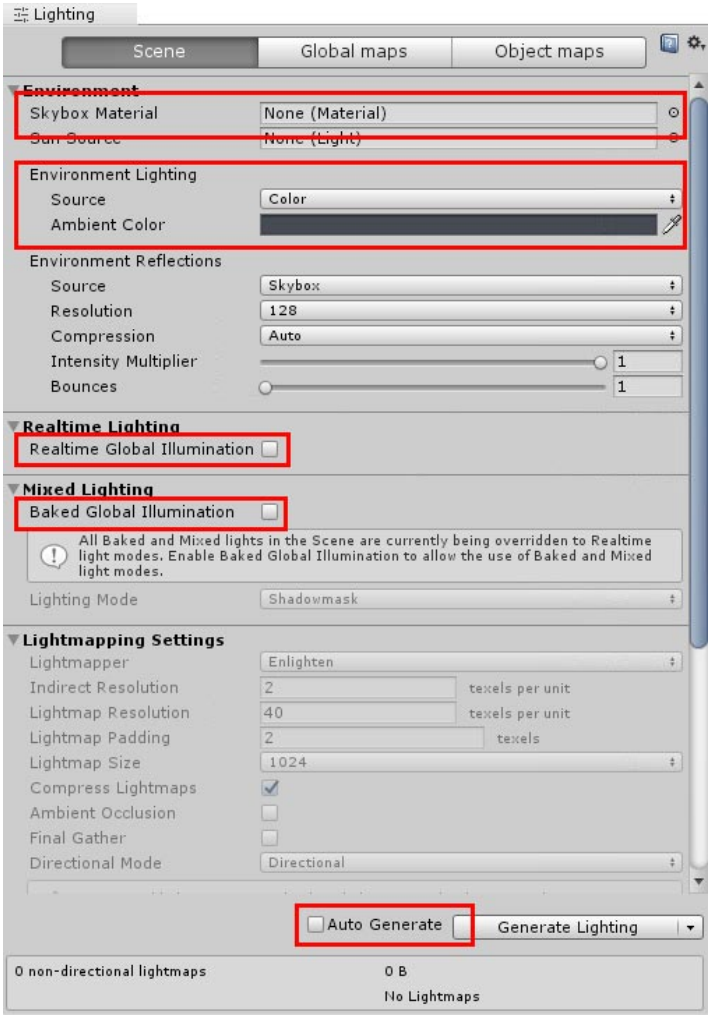
- Sprite Packer 被開啟。
- Scene 視窗設置為 2D。



- 預設的場景遊戲物件沒有定向光源（Directional Light）。
- 攝影機的預設位置為 (0,0,-10)。
- 攝影機預設為正交（Orthographic）。



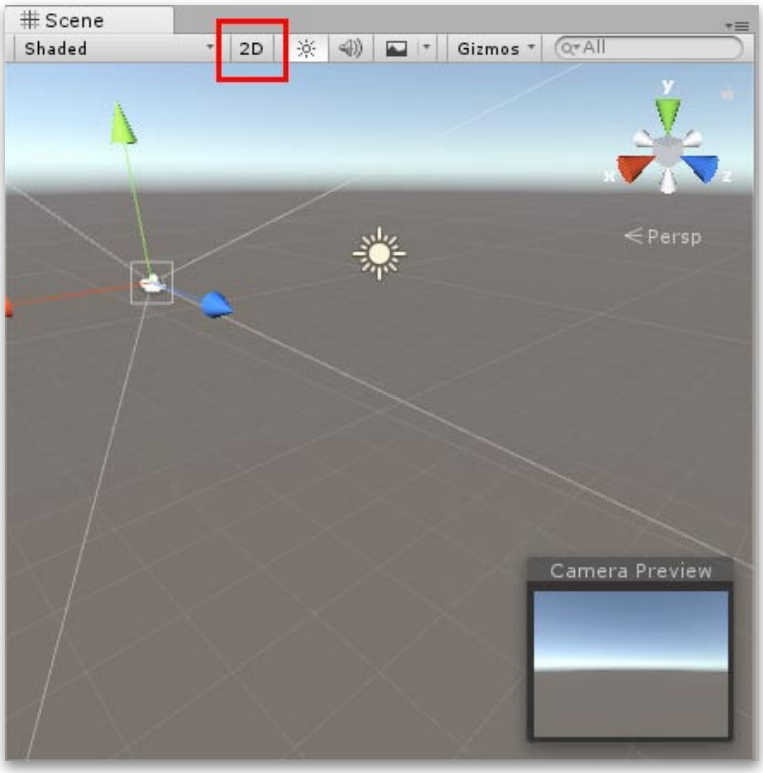
- Lighting 視窗：
  - 建立新場景時，沒有使用 Skybox。
  - 環境光來源（Ambient Source）設置為 Color（深灰色，R 54、G 58、B 66）。
  - 預計算即時全域光照（Precomputed Realtime GI）關閉。
  - 烘焙全域光照（Baked GI）關閉。
  - 自動生成（Auto Generate）關閉。



(2) 3D 模式：

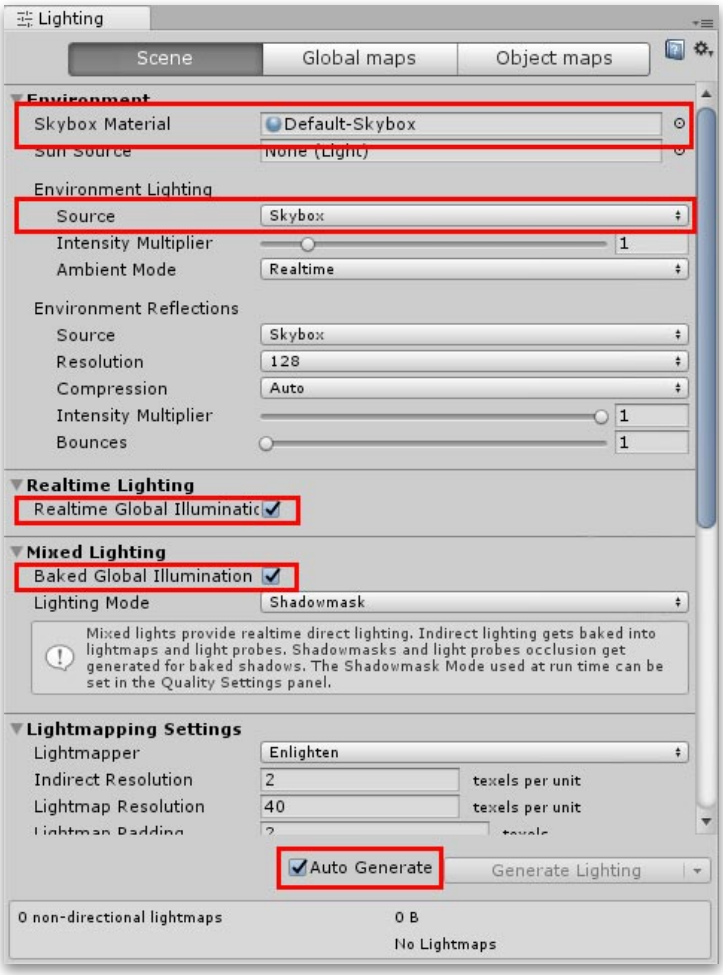
- 不會假定匯入的所有圖像都是要做為 2D 圖像（Sprite）使用。
- Sprite Packer 關閉。

- Scene 視窗設置為 3D。



- 預設的場景遊戲物件有定向光源（Directional Light）。
- 攝影機的預設位置為 (0,1,-10)。
- 攝影機預設為透視（Perspective）。

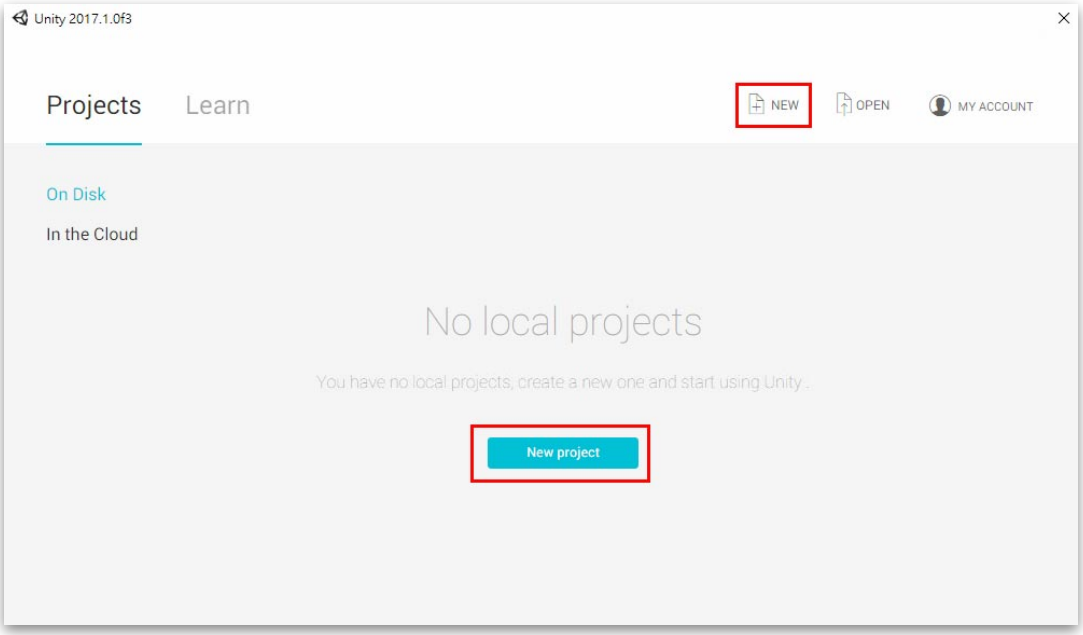
- Lighting 視窗：
  - Skybox Material 內建預設的 Skybox。
  - 環境光來源（Ambient Source）設置為 Skybox。
  - 預計算即時全域光照（Precomputed Realtime GI）開啟。
  - 烘焙全域光照（Baked GI）開啟。
  - 自動生成（Auto Generate）開啟。



IV.建立遊戲專案與專案結構介紹

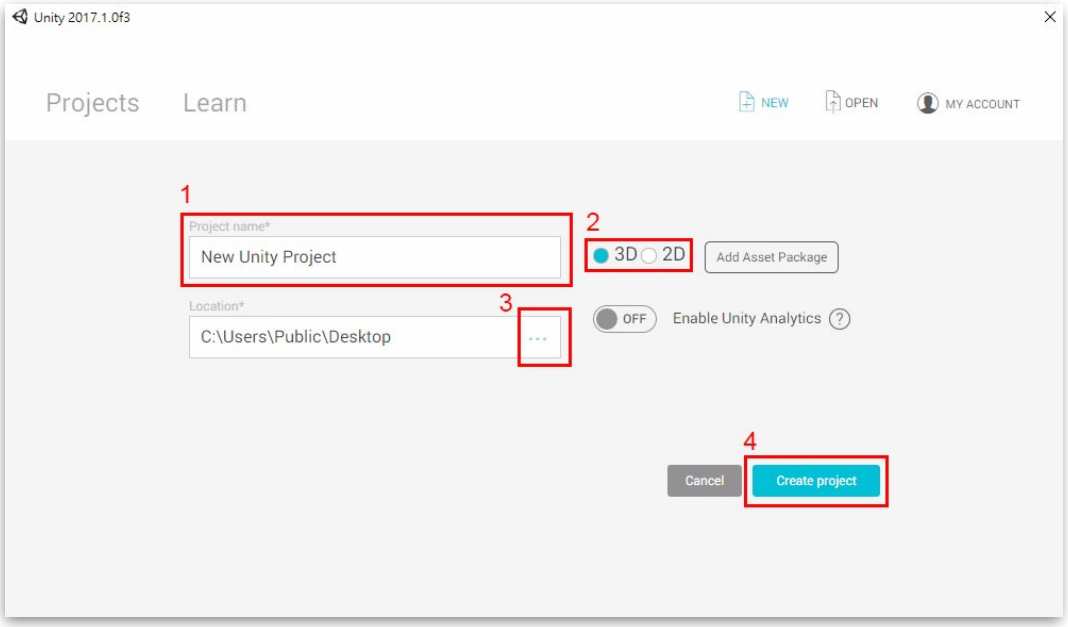
1.建立新專案

(1) 點擊 NEW 或 New Project 按鈕，準備建立新專案。



- 如果安裝於此電腦的 Unity 曾經開啟過其它專案，此處會列出那些專案名稱以及專案檔案所儲存的路徑，直接點擊專案名稱即可開啟該專案。
- 如果要開啟未列於此處的舊專案，可點擊右上方的 OPEN，透過檔案 / 資料夾選擇視窗來選取並開啟舊專案。
- 由於，目前希望建立一個全新的專案，所以，在此直接點擊右上角的 NEW，表示要建立新專案。

(2) 設置新專案。



1) 命名專案名稱：

專案名稱應該以英文字母及數字為主，避免使用中文名稱或特殊符號，此處命名之專案名稱將用來做為所建立的專案資料夾名稱。

2) 選擇 3D 或 2D 的專案類型：

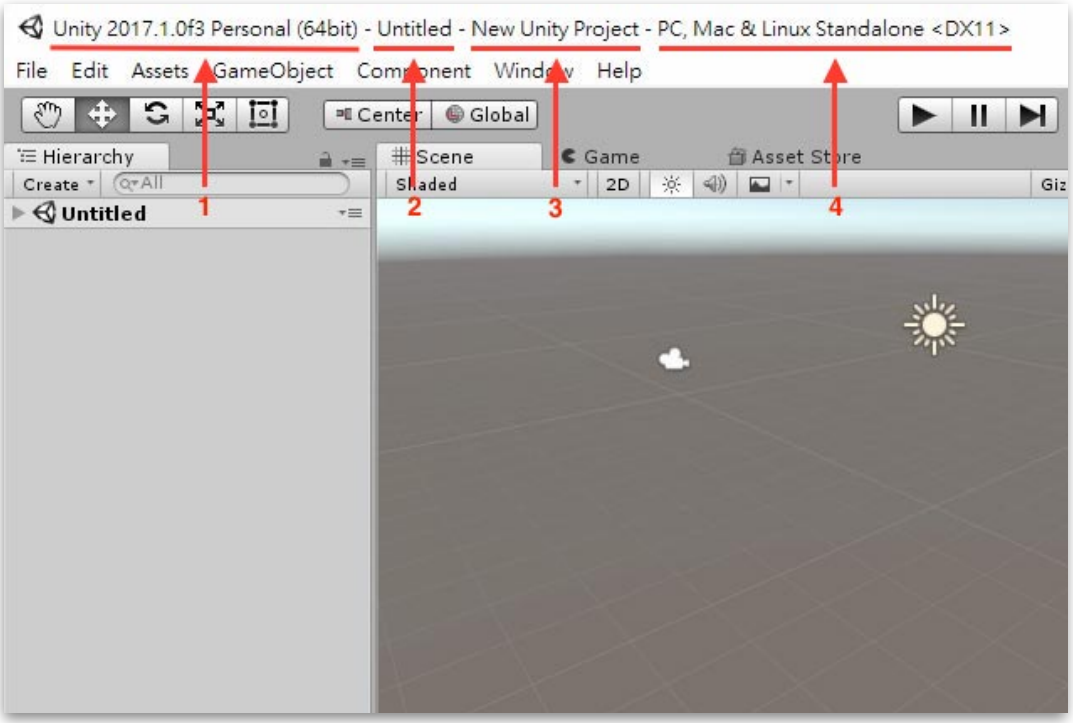
選擇預計開發的專案內容主要為 3D 或 2D，這將會影響到匯入資源以及建立新場景時的相關預設值，此項目在建立專案之後，可於編輯器中隨時更改。

3) 選擇專案儲存路徑：

指定專案資料夾所要存放的路徑位置，該路徑應該皆為英文字母及數字，避免有中文字或特殊符號。

4) 點擊 Create Project 按鈕，開始建立專案。

(3) 完成建立新專案，Unity 視窗的標題列：



1) 目前所使用的 Unity 版本。

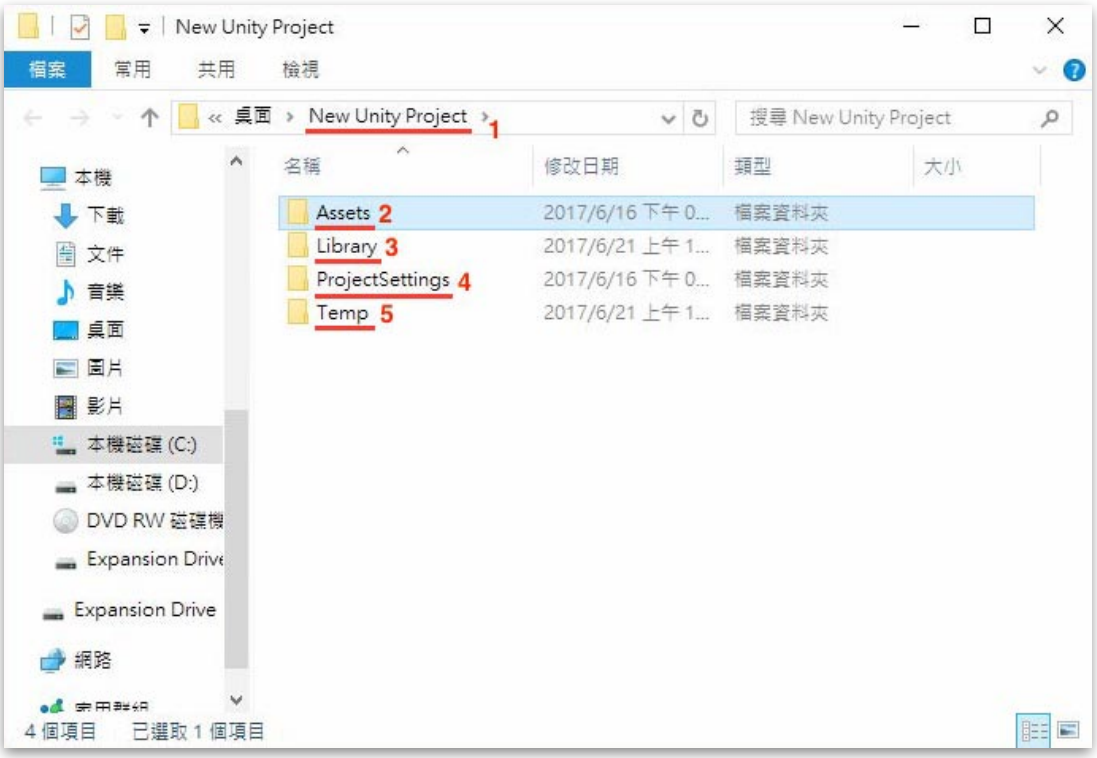
2) 目前開啟的場景名稱。

3) 專案名稱。

4) 目前編輯的目標平台。

2.專案結構

2.1. 專案資料夾：



在 Unity 編輯器上方的選單列選擇 Assets > Show in Explorer 或在 Project 視窗點擊滑鼠右鍵開啟選單選擇 Show in Explorer，可開啟專案資料夾。

1) 專案名稱同時也是專案資料夾的名稱。

2) Assets 資料夾：資料夾內的檔案，由使用者透過 Unity 編輯器建立，是實際在 Unity 編輯器內所使用的資源檔案，這個資料夾裡面的檔案更動（新增、修改名稱、刪除等）必須都在 Unity 編輯器內操作，以免發生無法預期的錯誤。

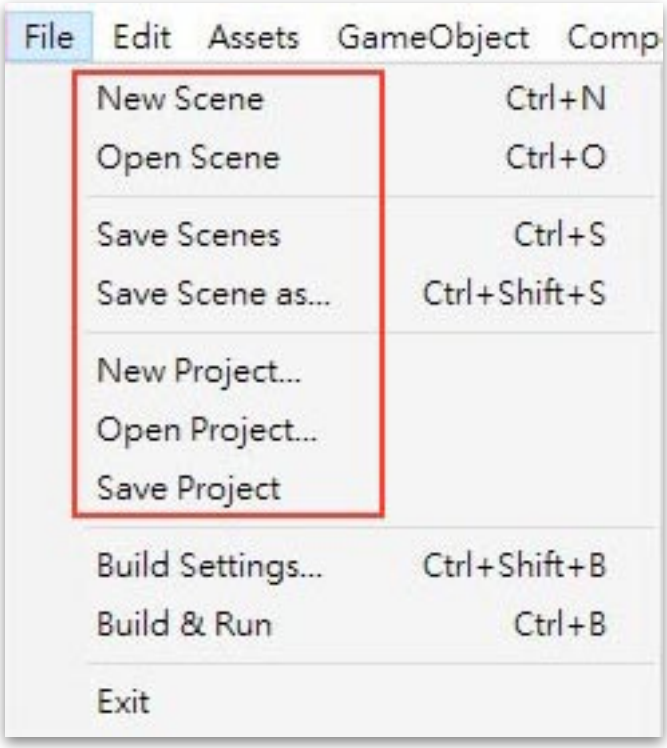
3) Library 資料夾：由 Unity 自動建立與操作的資料夾內容檔案，如無必要，使用者不應對此資料夾或其中的檔案進行任何操作。

4) ProjectSettings 資料夾：主要存放此專案相關的設定檔案，由 Unity 自動產生以及維護內容，於 Unity 編輯器中的 Edit > Project Settings 選單項目所設定的內容就是記錄在此。

5) Temp 資料夾：此為 Unity 運行時，自動產生的資料夾，其中某些檔案會被 Lock，將可能影響到專案搬移或刪除；關閉 Unity 後，此資料夾會自動刪

除；如果在 Unity 未開啟此專案時，就已有 Temp 資料夾，表示可能上次 Unity 並未正常關閉，可讓 Unity 重新開啟此專案再關閉，即可刪除。

2.2. 專案與場景 Project & Scenes：



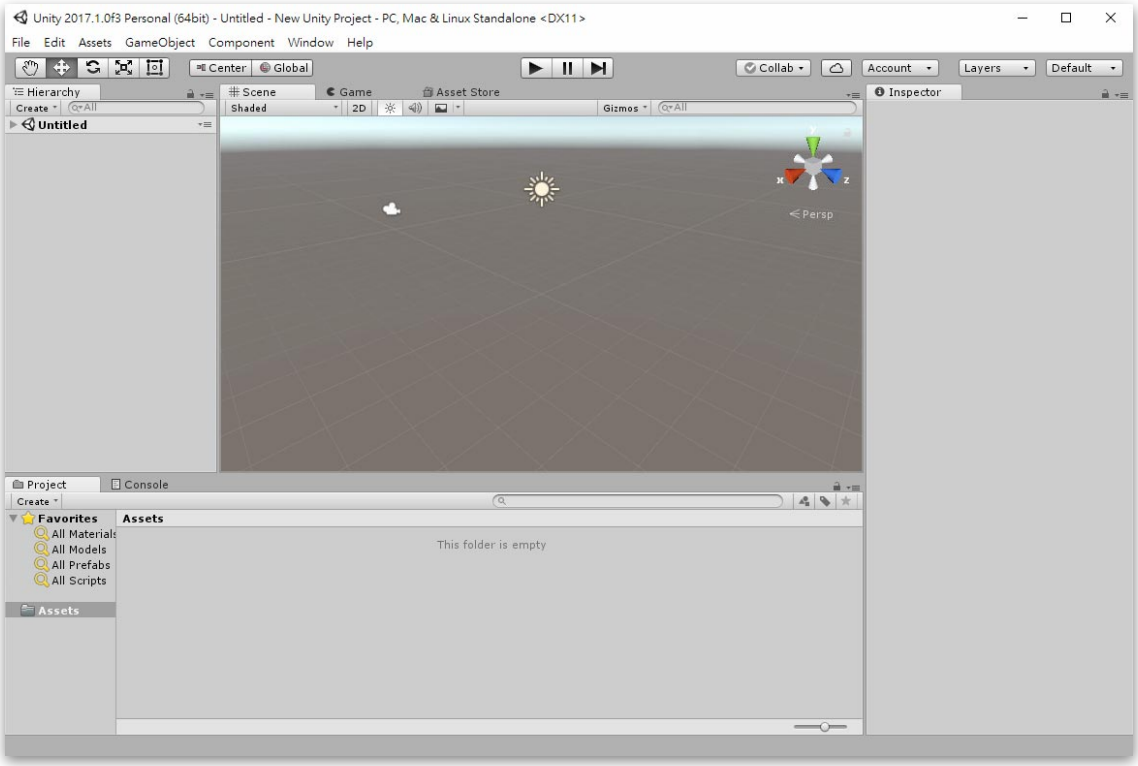
- Unity 應用或遊戲內容，是由許多個別獨立的場景所組成。
- 在 Unity 編輯器中所製作和編輯的內容，主要是用來建設場景內容。
- 每個場景在儲存之後，會成為專案內的場景資源檔案。
- 專案主要是管理所有的資源檔案以及資源的設定值，如場景、資料設定、動畫控制、動畫剪輯、字型、影片、音源、圖片、模型、材質、程式腳本、預置元件（Prefab）檔案...等等。

V.工具介面的使用

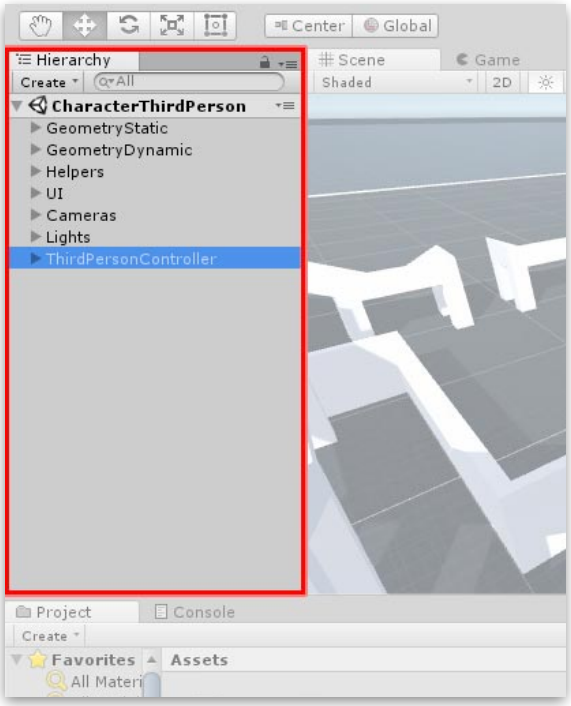
花點時間查看編輯器介面並熟悉它。主要的編輯器視窗是由標籤視窗所組成，能夠重新排列、分組、分離和停靠。

這意味著根據個人喜好以及正在做的工作，編輯器外觀將會隨著不同的專案及不同的開發者而不相同。

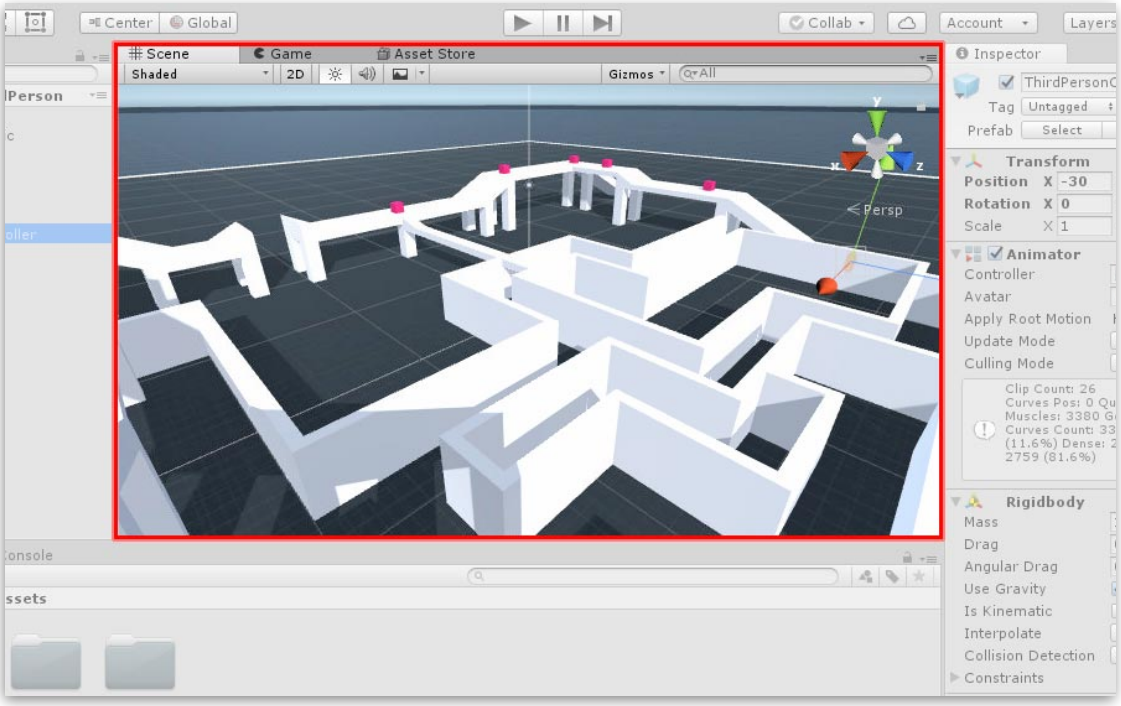
預設的視窗安排可讓你實際訪問到最常用的視窗。你可以透過標籤的名稱來辨認它們，並可依照自己的喜好重新拖拉排列或關閉視窗，也可透過選單列的 Window 選單來開啟其他視窗。



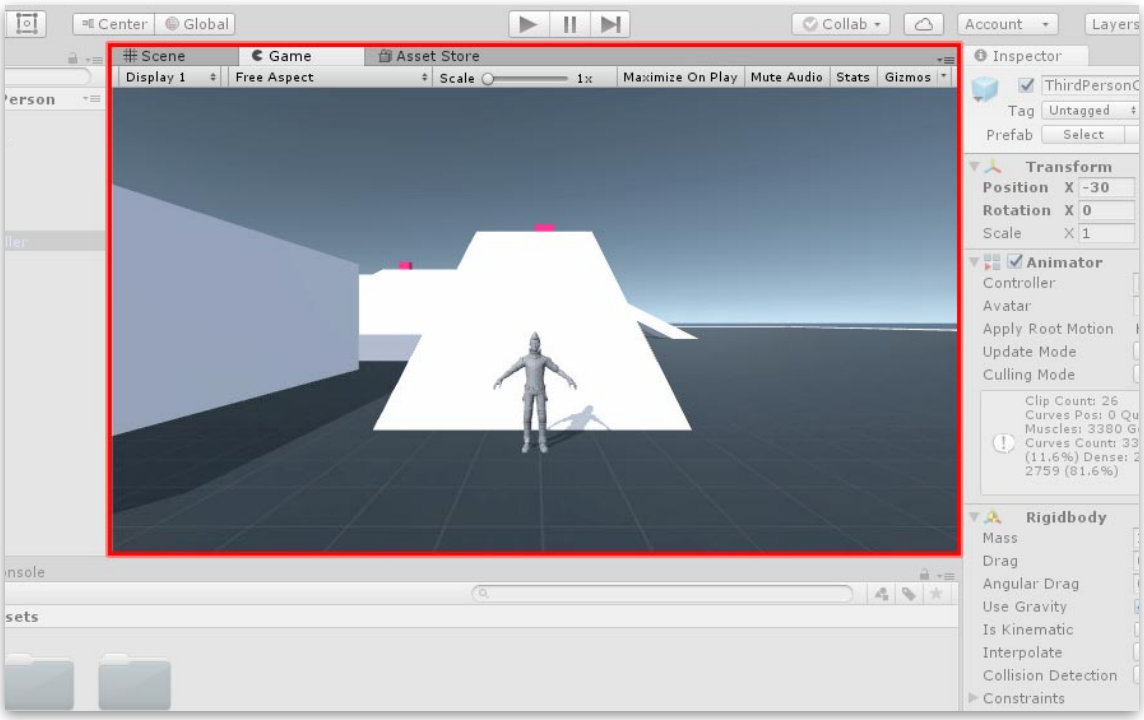
- Hierarchy 視窗：列出場景中全部的遊戲物件名稱，以及物件父子關係的層次結構，可透過點擊名稱來選取遊戲物件，或拖拉名稱來改變父子關係與排序。是場景中所有物件的層次結構文字表現。Scene 視窗中的每個東西在 Hierarchy 視窗都有一個條目，所以這兩個視窗的內部是互相連接的。



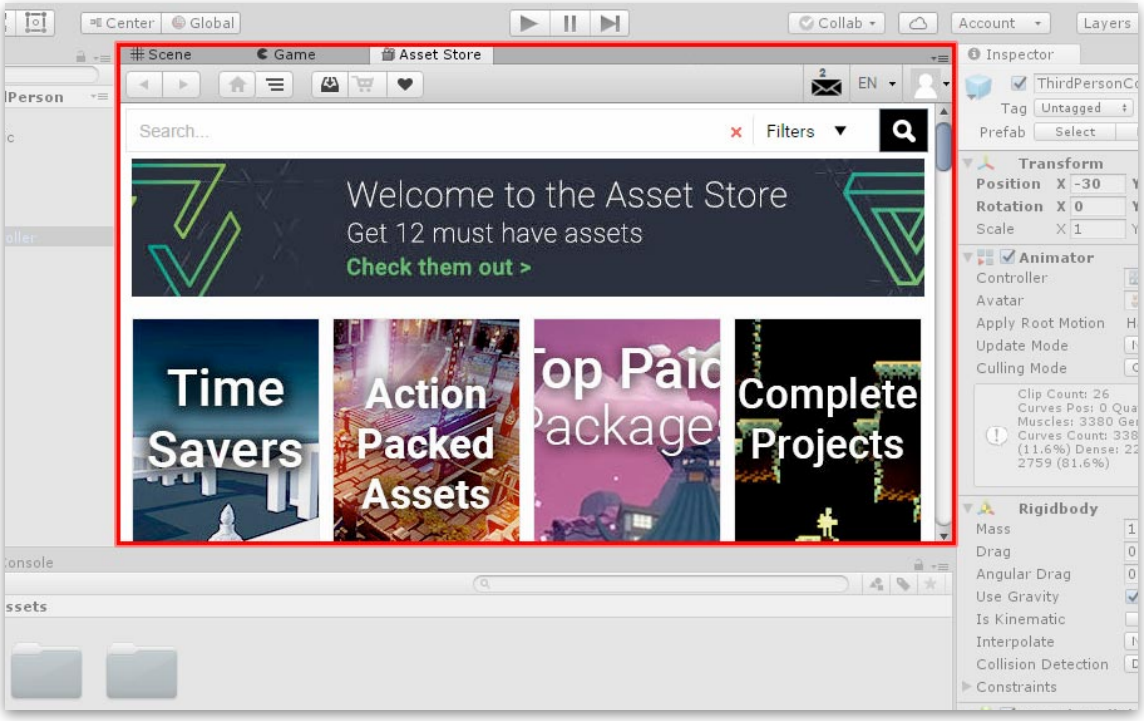
- Scene 視窗：用來編輯場景遊戲物件的位置、旋轉、縮放，讓你能夠視覺化的瀏覽和編輯你的場景。Scene 視窗能夠顯示 3D 或 2D 視角，取決於你所運作的專案類型。



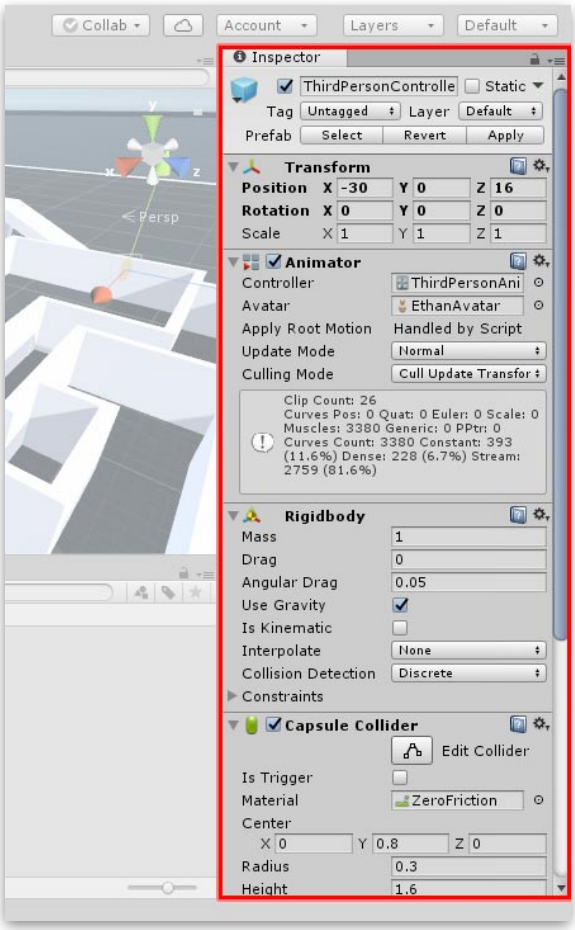
- Game 視窗：從場景中的鏡頭視野所呈現的畫面，也是玩家在遊戲中所看到的實際畫面，可直接用來觀察和測試製作結果。



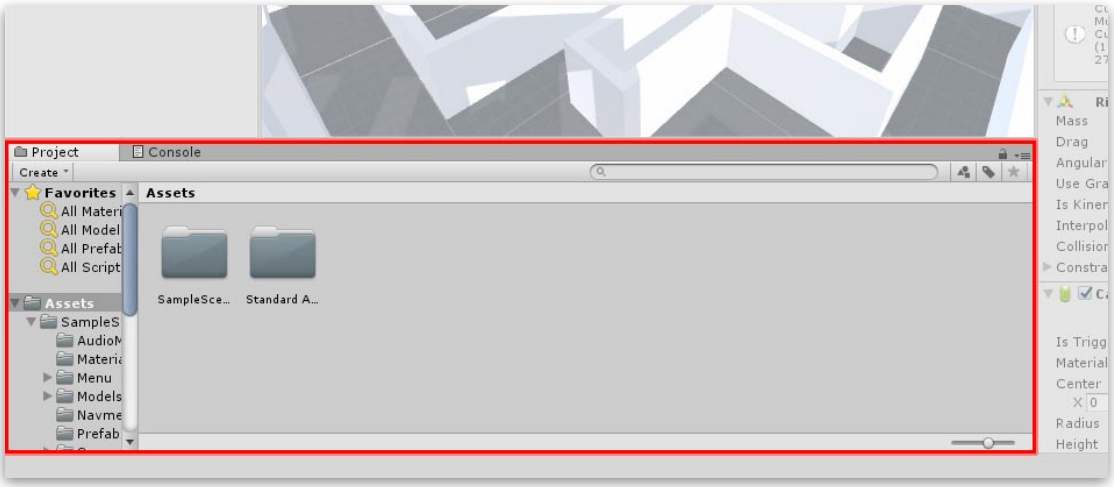
- Asset Store 視窗：Unity 專屬的資源商店平台，可免費或付費獲得各種開發相關的資源與擴充工具或學習範例。



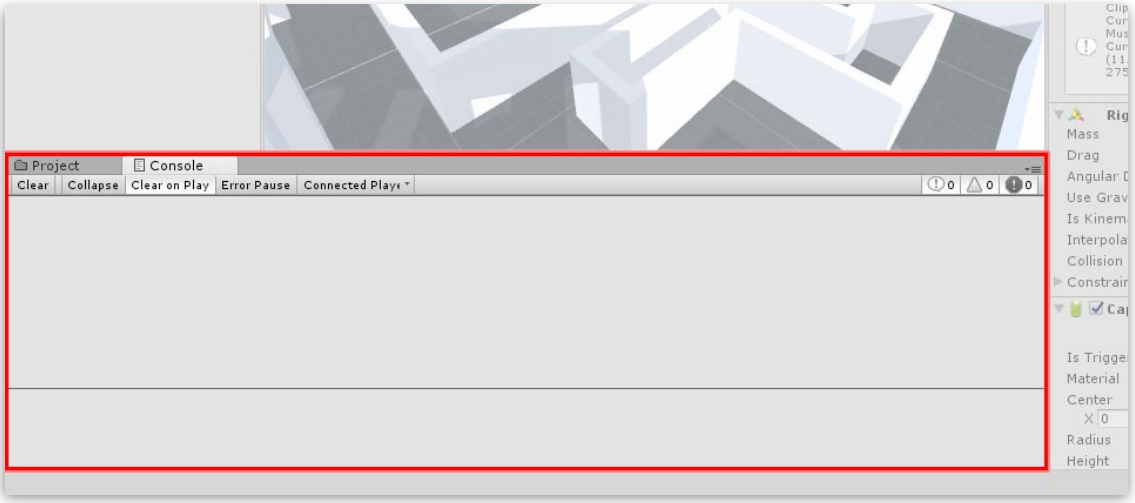
- Inspector 視窗：使你能夠查看並編輯目前所選取的遊戲物件、資源檔案等的屬性，所呈現的視窗內容，將隨著所選取的遊戲物件或資源檔案不同而有所不同。



- Project 視窗：即是專案資料夾裡的 Assets 資料夾內容，顯示專案中能夠使用的資源庫。匯入到專案中的資源，會出現在這裡。



- Console 視窗：用於顯示使用者程式裡所輸出的一般訊息以及其它警告或錯誤訊息，同時也提供所追蹤的訊息來源資訊。



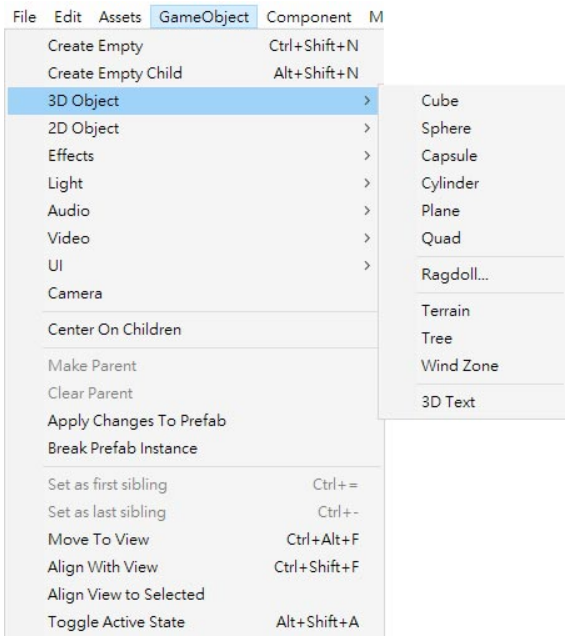
- 工具列：提供存取大部分基本工作的功能。左側包含操作 Scene 視窗中物件的基本工具。中間是執行、暫停與逐步執行的控制工具。右邊的按鈕是提供你訪問 Unity 雲端服務（Cloud Services）以及帳號、可見物件層的選單以及編輯器排列選單（提供一些替換的編輯器視窗佈局並讓你能夠儲存自己的自訂佈局）。
  工具列不是視窗，只是 Unity 介面的部分，所以不能重新排列其位置。

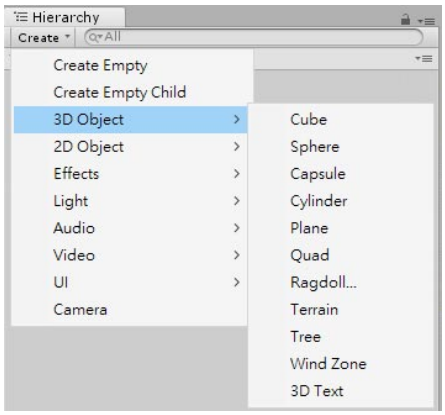
VI.遊戲資源工作流程

資源，代表可以用於你的遊戲或專案中的任何東西。資源可能來自於 Unity 外部所建立的檔案，如 3D 模型、聲音檔、圖片或任何其它各種 Unity 支援的檔案。還有一些資源類型可以在 Unity 中建立，如動畫控制器（Animator Controller）、聲音混合器（Audio Mixer）或渲染紋理（Render Texture）。

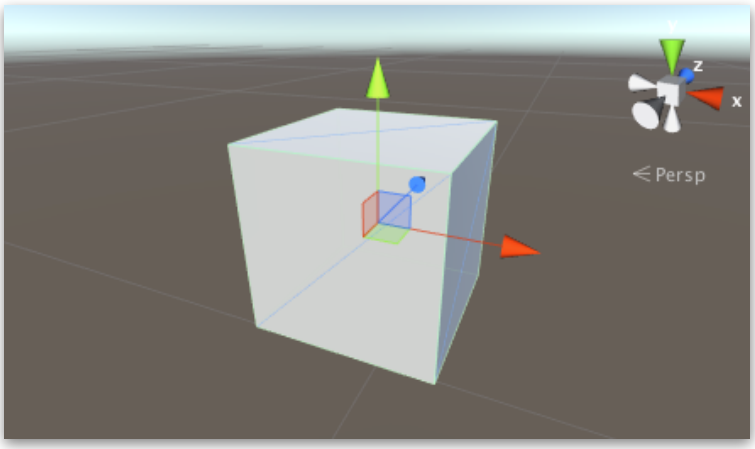
1.原型物件

Unity 可以使用任何使用建模軟體建立的各種形狀的 3D 模型。然而，也有一些原型物件可以直接在 Unity 中建立，即是立方體（Cube）、球體（Sphere）、膠囊體（Capsule）、圓柱體（Cylinder）、平面（Plane）以及四邊形（Quad）。這些物件本身的功用常常很有用（例如，平面通常用來做為平坦的地面），而它們還可以為了測試目的，提供便捷快速的方法來建立標示和原型樣品。任何原型物件都可以使用 GameObject > 3D Object 選單的合適項目來添加到場景。





- 立方體 Cube



這是使用一個單位長度的簡單立方體，其紋理為六個面重複的圖像。

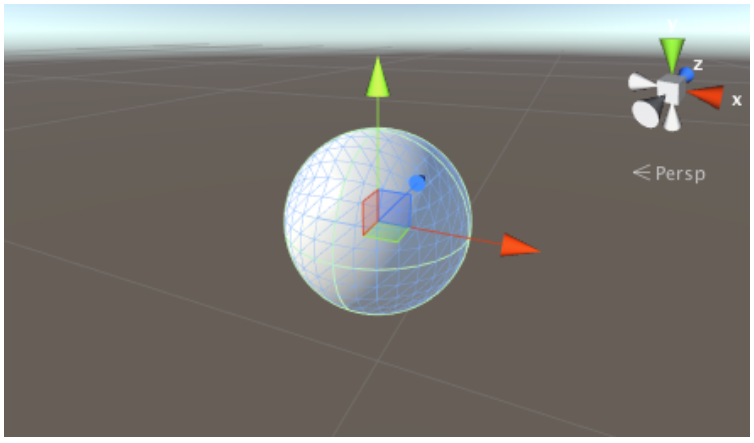
在大部分的遊戲裡，Cube 不會是很常見的物件，但是透過縮放，在做為牆壁、柱子、箱子、階梯以及其它類似的物品時會非常有用。

在開發期間，當完整的模型還不能使用時，對程式人員來說，它也是非常方便的標示物件。例如，一輛車體可以使用大致正確尺寸的細長 Cube 粗略地塑造。

雖然這不適合於完成的遊戲，但為了測試車輛控制的程式碼，做為簡單的代表物件是很好用的。

由於 Cube 的邊長為一個單位，因此，你可以透過添加一個用於比較尺寸的 Cube 來檢查匯入到場景中的網格（Mesh）比例。

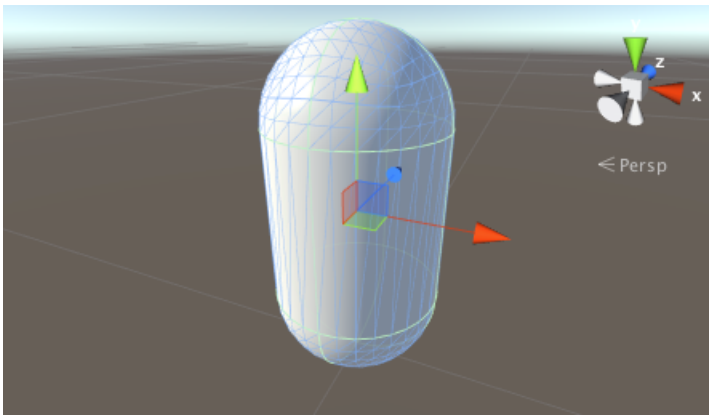
- 球體 Sphere



這是一個單位直徑的球體（即 0.5 單位的半徑），其紋理使得整個圖像在其極點的頂部和底部「擠壓」整個圍繞包裹住。

Sphere 對於代表球、行星和射彈是很有用的，而半透明球體製作 GUI 來表示一種效果的半徑。

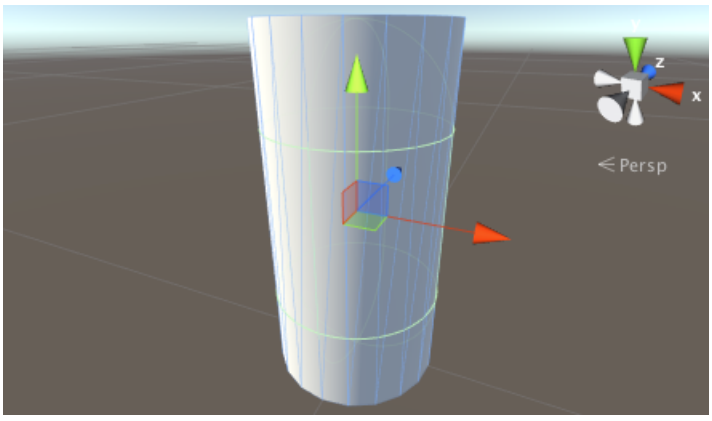
- 膠囊體 Capsule



Capsule 是在兩端具有半球形蓋子的圓筒。這種物件有一個單位直徑和兩個單位高（身體部份是一個單位以及兩個蓋子各有半個單位）。其紋理使圖像剛好圍繞包住並擠壓在兩個半球的頂點。

雖然沒有很多真實世界的物件具有這種形狀，但 Capsult 針對遊戲原型的製作是很有用的標示物。特別是有時候針對某些任務，用於圓形物件的物理會比方體還要好。

- 圓柱體 Cylinder

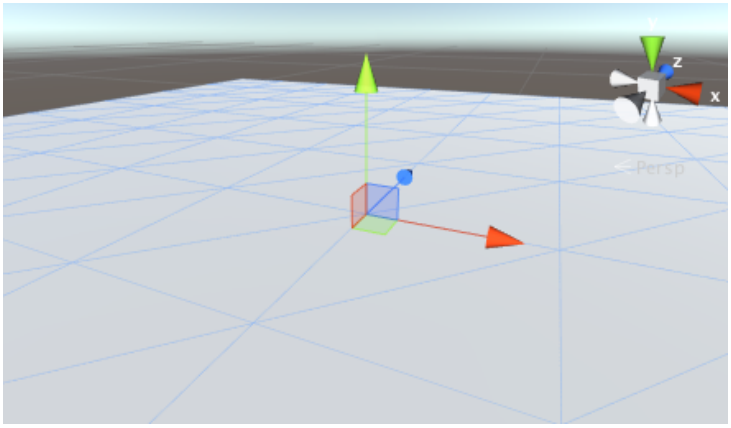


一個簡單的 Cylinder 是兩個單位高以及一個單位直徑，其紋理使得圖像環繞身體的管狀區域，但同時也會分別出現在兩個平的圓形端部。

Cylinder 非常方便於建立柱子、竿子和輪子，但要注意其碰撞器的形狀實際上是個膠囊（Unity 中沒有原生的柱體碰撞器）。如果需要準確的圓柱形碰撞器進行

物理目的，你應該要在建模軟體中建立適當形狀的網格，然後為其附加網格碰撞器（Mesh Collider）。

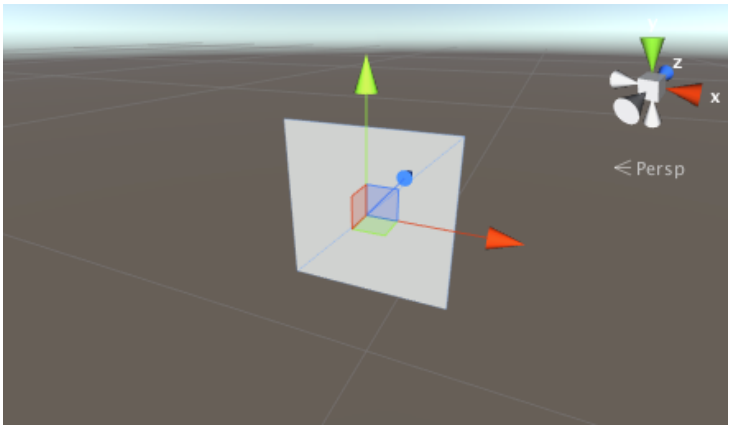
- 平面 Plane



這是個使用十單位長度邊，定位在區域座標空間的 XZ 平面上的一個平坦的正方形，其紋理讓整個圖像呈現在正方形內。

Plane 在大多數的平面表面很有用，如地板和牆壁。有時候針對顯示圖像或在 GUI 移動和特殊效果也會有需要。雖然 Plane 可以有這些用處，但更簡單的四邊形（Quad）可能會更適合。

- 四邊形 Quad



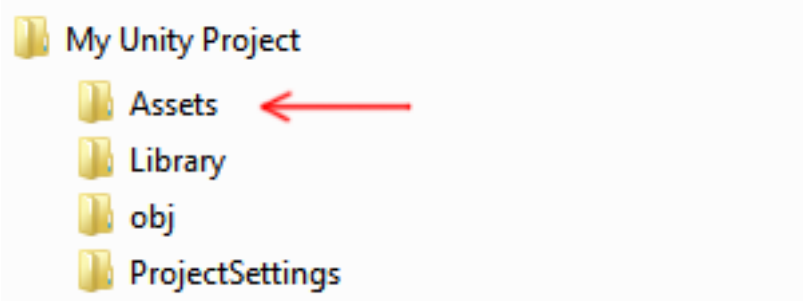
Quad 類似於 Plane，但它的邊只有一個單位長，並且其表面是定位在區域座標空間的 XY 平面上。另外，Quad 被劃分成 2 個三角形，而 Plane 則有 200 個。

在場景物件必須簡單的做為圖像或電影顯示螢幕的情況下，Quad 是很有用的。間單的 GUI 和訊息顯示，可以使用 Quad 來實現，像是粒子、Sprite 和遠距離的立體物件的替代「假」圖像。

2.匯入資源

在 Unity 的外部建立的資源，必須透過將檔案直接儲存到專案的 Assets 資料夾，或者複製到該資料夾，才能進入到 Unity 之中。對於許多常見的格式，你可以直接將你的來源檔案儲存到專案的 Asset 資料夾，Unity 將能夠讀取到。當你對該檔案儲存新的變更，Unity 會注意到並且根據需要而重新匯入。

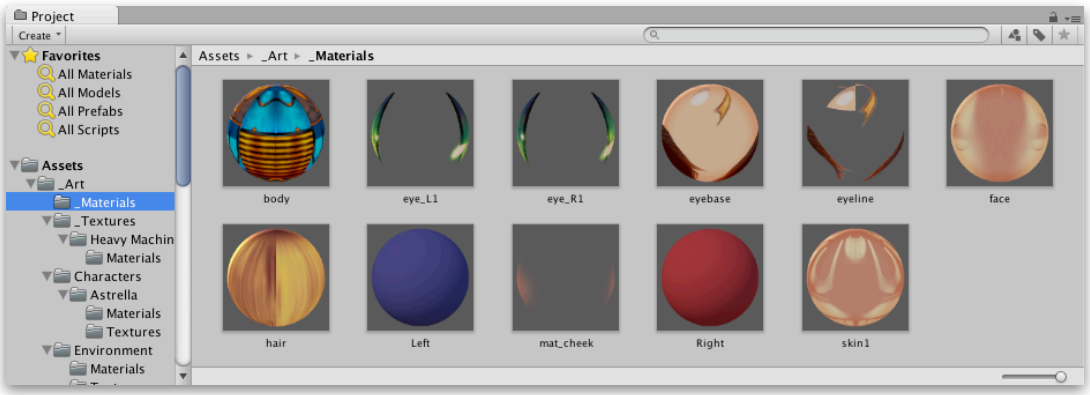
當你建立專案時，會建立一個與專案名稱相同的資料夾，其中包含一些子資料夾：



Assets 資料夾是你應該用來儲存或複製專案中要使用的檔案的位置。

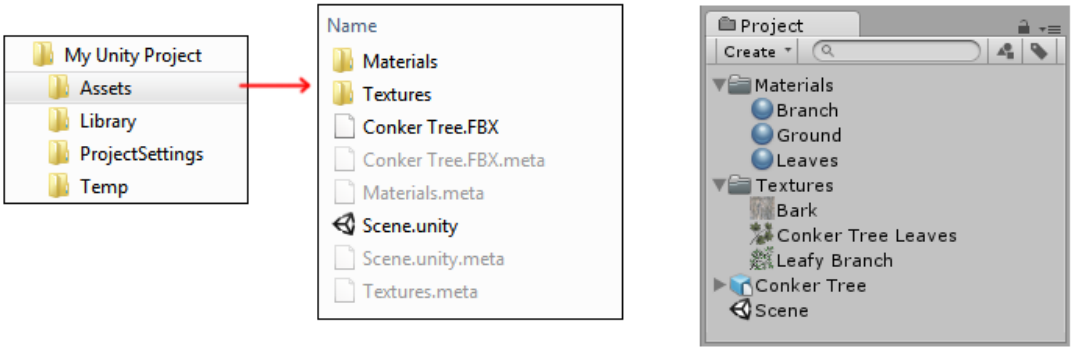
Unity 中的 Project 視窗內容顯示 Assets 資料夾內的東西。因此，如果儲存或複製檔案到 Assets 資料夾中，將會被匯入並可在 Project 視窗中看見。

當檔案被添加到 Assets 資料夾中或者檔案被修改，Unity 將會自動檢測。當你放任何資源到 Assets 資料夾，將會看到資源出現在 Project 視窗。



如果你從你的電腦拖拉檔案到 Unity 的 Project 視窗（例如，從 Mac 的 Finder 或是從 Windows 的檔案總管），檔案將被複製到 Assets 資料夾，並顯示在 Project 視窗。

大多數情況下，在 Project 視窗所看到的項目也代表是在電腦裡的真實檔案，假如你從 Unity 中刪除它們，也會同時在電腦中被刪除掉。



上圖顯示了 Unity 專案的 Asssets 資料夾中的幾個檔案和資料夾。你可以根據喜好建立許多資料夾並使用它們來組織你的資源。

上圖可以注意到，在檔案系統中列出的 .meta 檔案，但是在 Unity 的 Project 視窗中並沒有。Unity 為每個資源和資料夾建立這些 .meta 檔案，但預設是隱藏的，所以你可能在你的電腦檔案系統中並不會看到。它們包含了關於資源在專案中如何使用的重要訊息，並且必須與其相關的資源檔案共同被保留，所以，如果你在電腦的檔案系統中（檔案總管 / Finder）搬移或重新命名資源檔案，你也必須搬移或重新命名相關的 .meta 檔案。安全搬移或重新命名資源的最簡單方法就是始終從 Unity 的 Project 視窗中動作。這麼做，Unity 將自動的搬移或重新命名相對應的 .meta 檔案。

3.常見的資源類型

- 圖形檔案：
  大部分常見的圖形檔案都有支援，例如 BMP、TIF、TGA、JPG 以及 PSD。如果儲存圖層化的 Photoshop（.psd）檔案到 Assets 資料夾，它們將被做為整平的圖像來匯入。

- 3D 模型檔案：
  如果你從大多數常見的 3D 軟體使用它們的原生格式儲存 3D 檔案（如 .max、.blend、.mb、.ma）到 Assets 資料夾，它們將透過調用你的 3D 軟

體的 FBX 匯出插件來將其匯入。或者，你也可以從你的 3D 軟體匯出為 FBX 格式再匯入到 Unity 專案。

- 網格和動畫：
  無論你使用哪種 3D 軟體，Unity 將從每個檔案匯入其網格和動畫。
  網格檔案不必匯入動畫。如果你要使用動畫，可以選擇從單一個檔案匯入全部的動畫，或者從每個帶有一個動畫的個別檔案匯入。

- 聲音檔案：
  如果儲存未壓縮的聲音檔案到 Assets 資料夾，將根據所指定的壓縮設定來匯入。

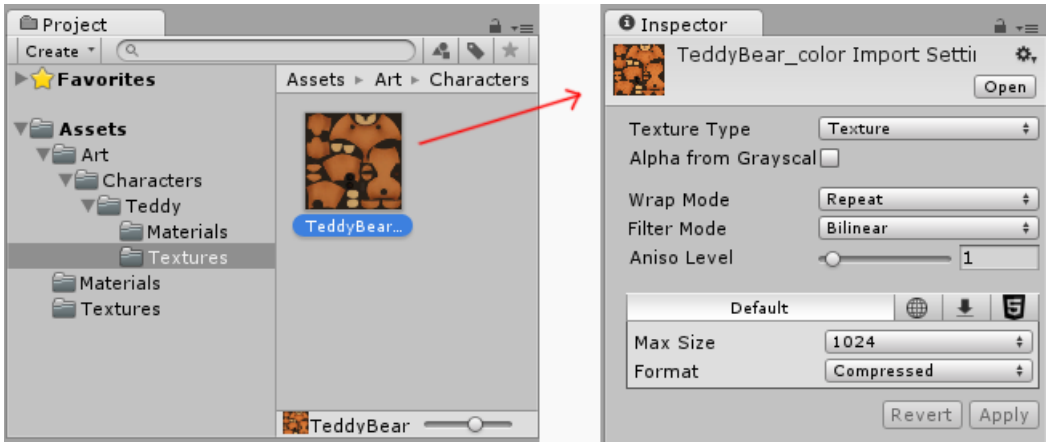
- 其它資源類型：
  在任何情況下，你的原始來源檔案不會被 Unity 所修改，即使是在 Unity 內，你會時常在各種方法間選擇壓縮、修改或以其它方式處理資源。其匯入過程是讀取來源檔案，並建立內部符合你所選擇匯入設定的資源描述，以準備給遊戲使用。如果為資源修改匯入設定，或者在 Assets 資料夾修改來源檔案，將導致 Unity 重新匯入資源以反應新的改變。

注意：匯入原生 3D 格式，需要有該 3D 軟體安裝在相同的電腦中。這是因為 Unity 使用 3D 軟體的 FBX 匯出器插件來讀取該檔案。或者，你可以直接從應用軟體匯出 FBX 檔案儲存到 Unity 的專案資料夾中。

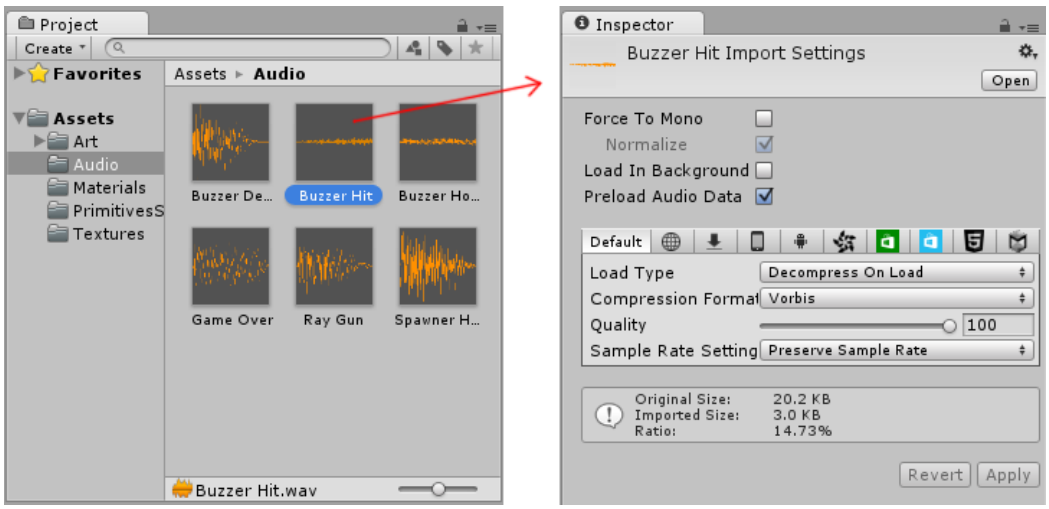
4.匯入設置

每種 Unity 支援的資源都有其匯入設置，它們會影響資源的顯示和行為。在 Project 視窗選擇資源以顯示資源的匯入設置，該資源的匯入設置將會出現在 Inspector 視窗。所顯示的選項將依據所選擇的資源種類而有所不同。

例如，圖像的匯入設置能夠讓你選擇是被匯入為紋理、2D Sprite 或法線貼圖。FBX 檔案的匯入設置讓你能夠調整縮放、生成法線或是光照貼圖座標，以及分割和修剪檔案中定義的動畫片段。



對於其它資源類型，匯入設置將看起來不同。各式各樣的設置會與所選擇的資源相關。下圖示範聲音資源在 Inspector 視窗顯示相關的匯入設置。



如果開發跨平台的專案，你可以撤銷「預設」設置並在每個平台配置不同的匯入設置。

5.從 *Asset Store* 匯入

Unity Asset Store 是由 Unity Technologies 以及社群成員所建立持續成長的免費及商用資源庫的所在地。有各式各樣的資源可用，涵蓋從紋理、模型和動畫到整個專案範例、教學和編輯器擴充的所有內容。資源從 Unity 編輯器內建的簡單界面進行存取並直接下載匯入到專案中。

Unity 使用者可以成為 Asset Store 上的發佈商，並銷售所創建的內容。

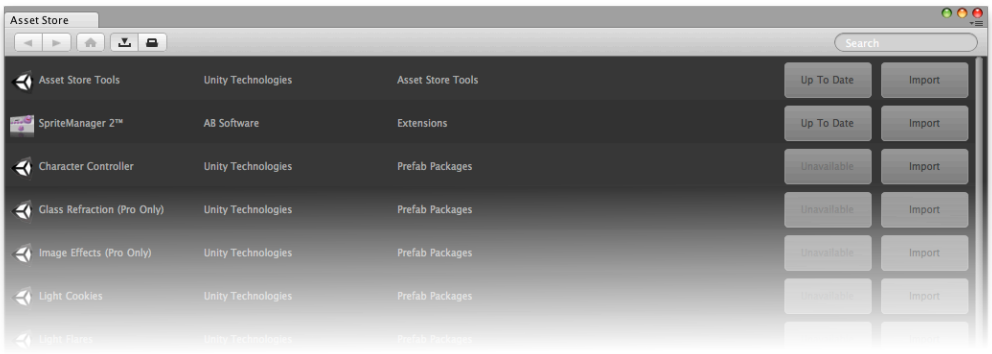
- 訪問並瀏覽 Asset Store

  可以透過選單列的 Window > Asset Store 開啟 Asset Store 視窗。第一次造訪，會被提示去建立免費的使用者帳號，隨後將用來存取 Asset Store 的內容。

Asset Store 提供類似瀏覽器的介面，使你能夠透過文字搜尋或分類來進行瀏覽。左側的工具欄是令人熟悉的瀏覽按鈕，用來瀏覽查看過的歷史紀錄項目。

稍右邊的按鈕則可查看下載管理器（Download Manager）以及查看目前的購物車內容。

下載管理器使你能夠察看已經購買過的資源包，還可以查找和安裝任何更新。此外，Unity 提供的標準資源包可以使用相同的介面查看並添加到專案中。



- 已下載的資源檔案位置：

  很少會需要直接訪問從 Asset Store 下載的檔案，但是，如果有需要，可以從這些位置找到。

  - Mac：~/Library/Unity/Asset Store
  - Windows：C:\Users\使用者名\AppData\Roaming\Unity\Asset Store

  這些資料夾包含相對應的 Asset Store 供應商名稱的子資料夾，實際的資源檔案會包含在相對應的子資料夾中。

VII.製作與匯入資源包

Unity 的資源包（Package）是分享以及重複使用 Unity 專案和所收集資源的便利方法，例如，Unity 的標準資源（Standard Assets）以及在 Unity Asset Store 上的東西都是使用資源包來提供 。

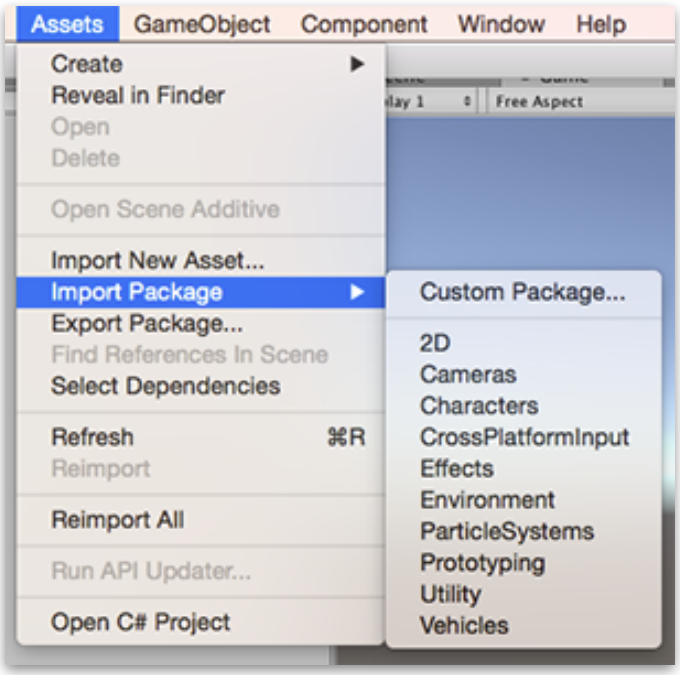
資源包是來自 Unity 專案或者專案元素的檔案和資料的集合，它們被壓縮並儲存在一個檔案中，類似於 Zip 檔案。

就像 Zip 檔案一樣，資源包解開時會維持原本的資料夾結構以及與資源相關的 meta 資料（例如匯入設置和與其它資源相關的連結）。

在 Unity 中，選單項目「Export Package」壓縮並儲存其集合，而「Import Package」解開集合內容到你目前開啟的 Unity 專案中。

1.匯入資源包

你可以匯入 Unity 提供的預製資源集合「標準資源包（Standard Asset Packages）」以及自定資源包（Custom Package），它們都是使用 Unity 製作的。



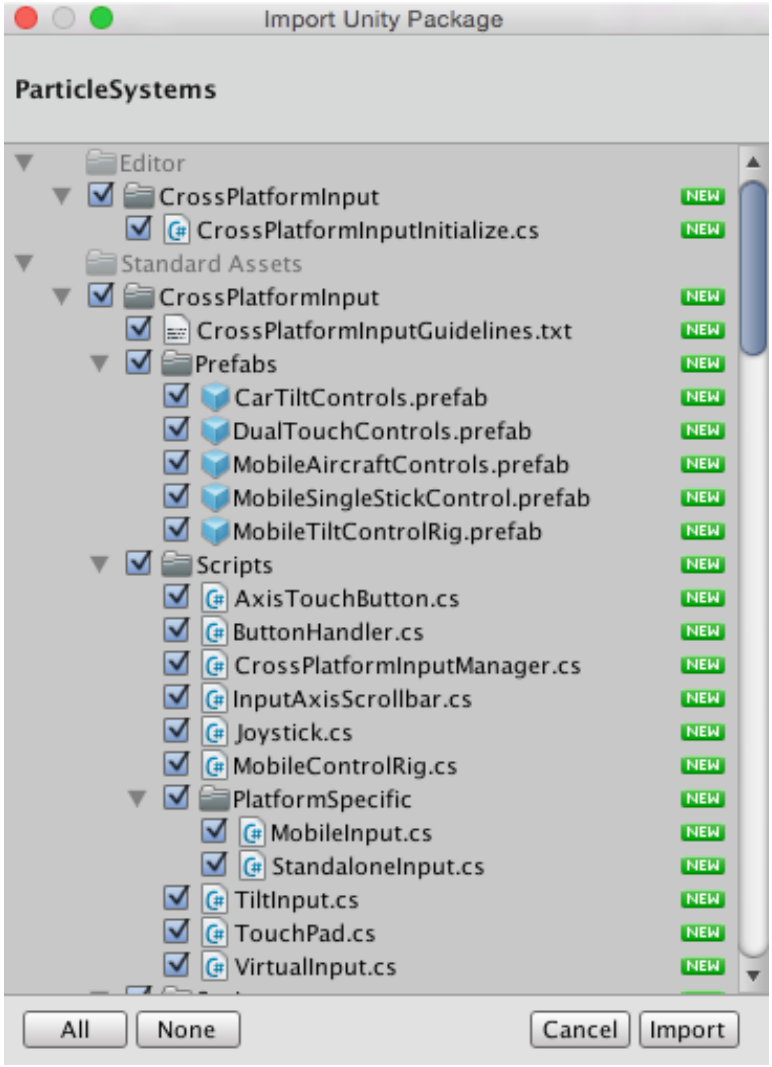
選擇 Assets > Import Package 選單來匯入它們。

2.標準資源包

Unity 的「標準資源」（Standard Assets）是由各種包裝所組成：2D、Cameras、Characters、CrossPlatformInput、Effects、Environment、ParticleSystems、Prototyping、Utility、Vehicles。

匯入新的標準資源包：

1. 開啟想要匯入資源的專案。
2. 選擇 Assets > Import Package > 資源包名稱，之後會顯示 Import Unity Package 視窗，伴隨著顯示包裝中的所有東西並預先勾選，接著就準備安裝了。
3. 點擊 Import 按鈕，Unity 會將資源包的內容放入 Project 視窗中的 Standard Asset 資料夾中。

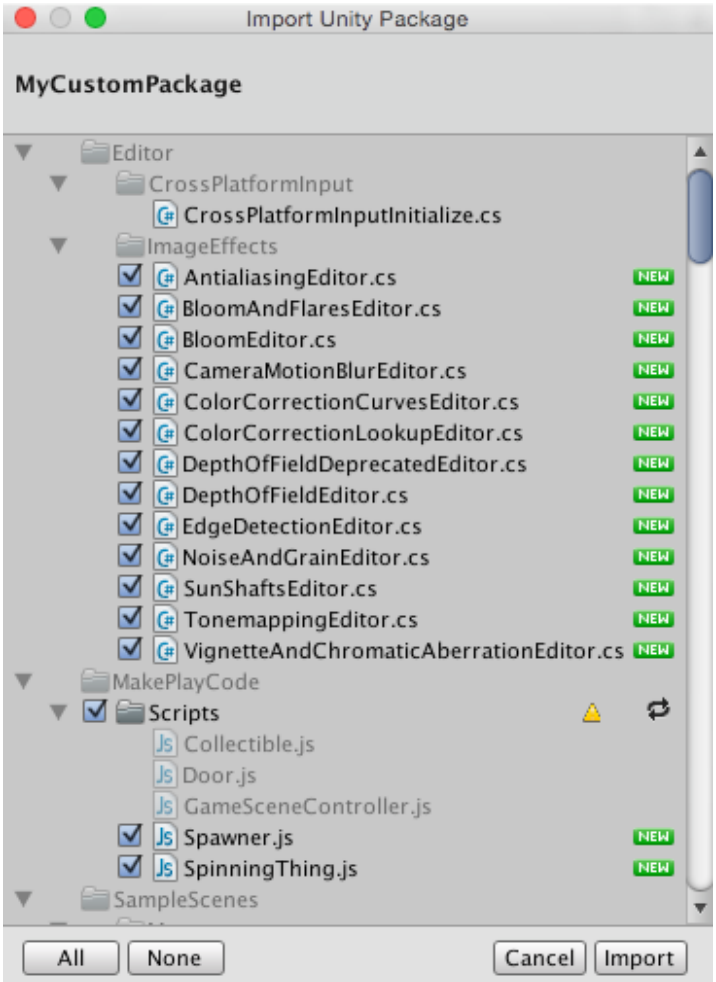


3.自定資源包

你可以匯入從你自己的專案或其它 Unity 使用者製作的專案所匯出的自定資源包。

匯入新的自定資源包：

1. 開啟想要匯入資源的專案。
2. 選擇 Assets > Import Package > Custom Package... 來開啟檔案總管（Windows）或 Finder（Mac）。
3. 選擇想要匯入的資源包檔案，並顯示 Import Unity Packgae 視窗，伴隨著顯示包裝中的所有東西並預先勾選，接著就準備安裝了。
4. 點擊 Import 按鈕，Unity 會將資源包的內容放入 Project 視窗中的 Assets 資料夾裡。

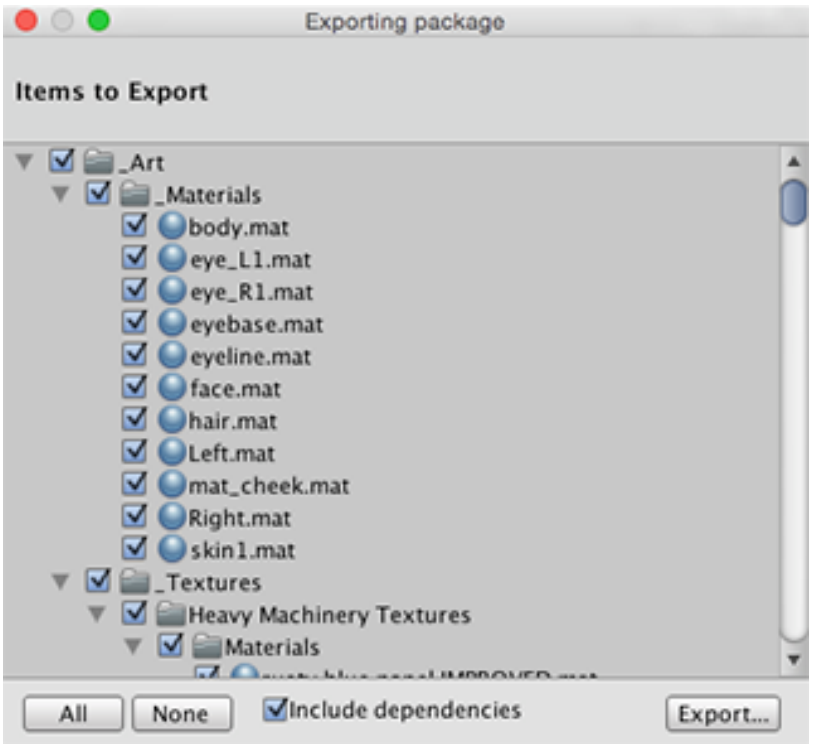


4.匯出資源包

使用 Export Package 來建立你自己的自定資源包（Custom Package）。

1. 開啟想要匯出資源的專案，並選擇想要匯出的資源檔案。
2. 選擇 Assets > Export Package...，開啟 Exporting Package 視窗。
3. 在視窗中，確認選取所想要包含在資源包中的資源。
4. 保留 include dependencies 選取，以自動選擇與所使用資源相關的檔案。
5. 點擊 Export 按鈕開啟檔案總管（Windows）或 Finder（Mac）並選擇想要儲存資源包檔案的位置，然後為其命名並儲存。

匯出資源包時，Unity 能夠匯出所有相依賴的項目。例如，如果選擇一個場景檔案，並附帶所有相依賴項目來匯出，那麼，所有的模型、紋理和其它資源也都會一起匯出。這樣可以快速地匯出一堆資源，而不用全部手動查找。



5.匯出更新的資源包

有時候可能想要改變資源包的內容並建立新的資源包更新版本。可以這麼做：

1. 選擇資源包所需的檔案（選擇未改變的和新的資源）。
2. 使用 Export Package 如上所述的匯出檔案。

注意：可以重新命名已更新的資源包，Unity 將會辨識出它做為更新，所以可以使用遞增編號來命名，例如：MyAssetPackageVer1、MyAssetPackageVer2。

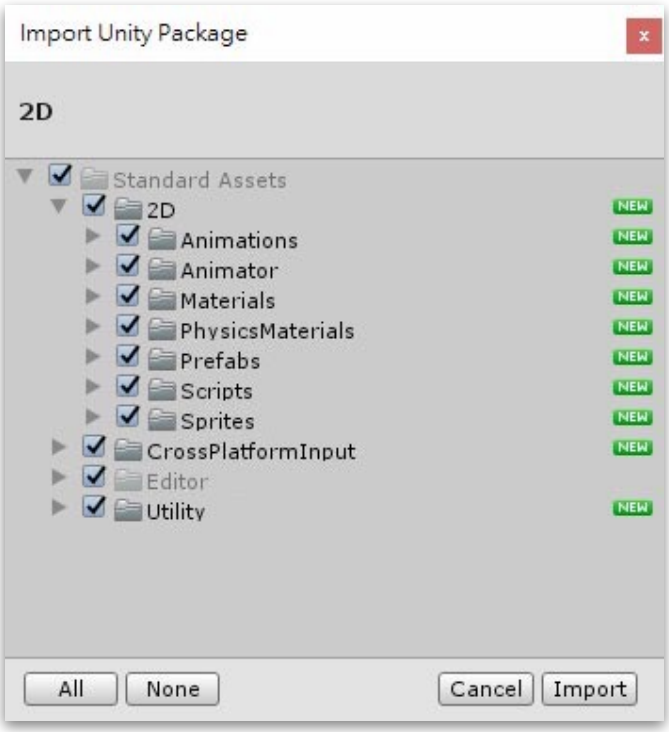
從資源包裡移除檔案，然後使用相同名稱的檔案來取代，這並不是好的做法；Unity 會辨識出它們為不同的資源，而且可能會認為是衝突的檔案，因而在它們被匯入的時候顯示警告。假如已經移除檔案並決定替換它，最好的做法，是給它一個與原來不同但又相關的名稱。

VIII.標準資源包

Unity 帶有多個標準資源，這是提供給廣大使用者的資源集。

1. 2D

匯入 2D 資源包裡除了 2D 相關的資源之外，還包含了相關的 CrossPlatformInput 以及 Utility 的資源必須同時匯入。



在 Standard Assets > 2D > Prefabs 資料夾裡，有立即可用的預置元件，直接拖拉到場景中，就能開始編輯 2D 遊戲場景：

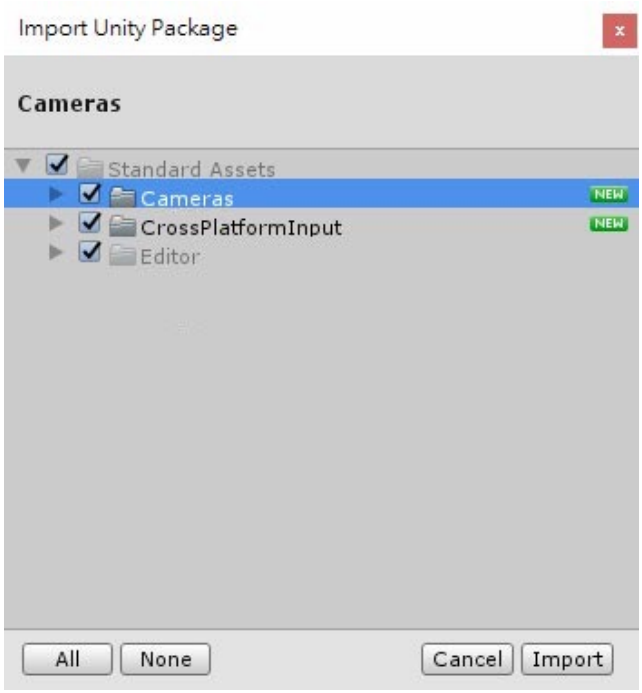
- CharacterRobotBoy：可控制前進、蹲伏、蹲伏前進和跳躍的 2D 機器人角色。角色的 Platformer Character 2D 組件提供可調整的欄位：
  - Max Speed：前進速度。
  - Jump Force：跳躍力。
  - Crouch Speed：蹲伏前進的速度為 Max Speed 的幾倍，其值範圍為 0 ~ 1，若為 1，表示速度與前進速度相同。
  - Air Control：跳躍或掉落時，角色在空中是否可控制。

  - What Is Ground：指定哪些層級的遊戲物件可做為地面，讓角色判定為「落地」。

- CollisionSlider：在遊戲畫面上看不見的碰撞區塊。
- CratePink：原色為粉紅色的可推動箱子。
- ExtentsLeft：用於限制左側行動範圍的垂直牆壁。
- ExtentsRight：用於限制右側行動範圍的垂直牆壁。
- Killzone：在遊戲畫面上看不見的死亡區域碰撞區塊，主要用來放在地板之下，當角色掉落接觸到之後，代表角色死亡，場景將立即重新載入。
- Platform04x01：4 x 1 大小的平台。
- Platform08x01：8 x 1 大小的平台。
- Platform12x01：12 x 1 大小的平台。
- Platform16x01：16 x 1 大小的平台。
- Platform36x01：36 x 1 大小的平台。
- PlatformRamp：帶有斜坡的平台。

2. Cameras

匯入 Cameras 資源包裡除了 Camera 相關的資源之外，還包含了相關的 CrossPlatformInput 的資源必須同時匯入。

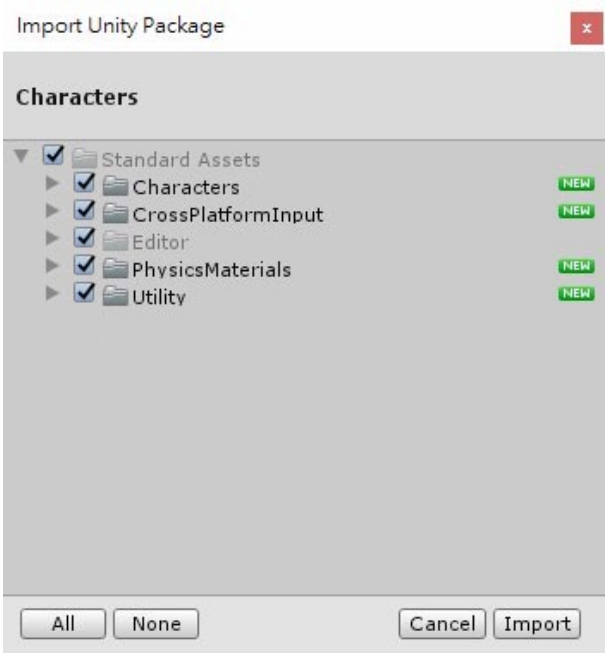


在 Standard Assets > Cameras > Prefabs 資料夾裡，有立即可用的預置元件，直接拖拉到場景中，配置鏡頭跟隨的目標物件，或將目標物件的 Tag 設置為 Player，就能提供常見的攝影鏡頭行為：

- CctvCamera：類似於監視器鏡頭，鏡頭放置在固定點，跟隨目標物件轉動。
- FreeLookCameraRig：跟隨目標物件移動的鏡頭，且可透過滑鼠或手勢（手機）轉動視角。
- HandheldCamera：以目標物件模擬手持物品般拉近畫面並自然晃動。
- MultipurposeCameraRig：跟隨目標物件移動和轉動的鏡頭。

3. Characters

匯入 Characters 資源包裡除了 Characters 相關的資源之外，還包含了相關的 CrossPlatformInput 以及 Utility 的資源必須同時匯入。

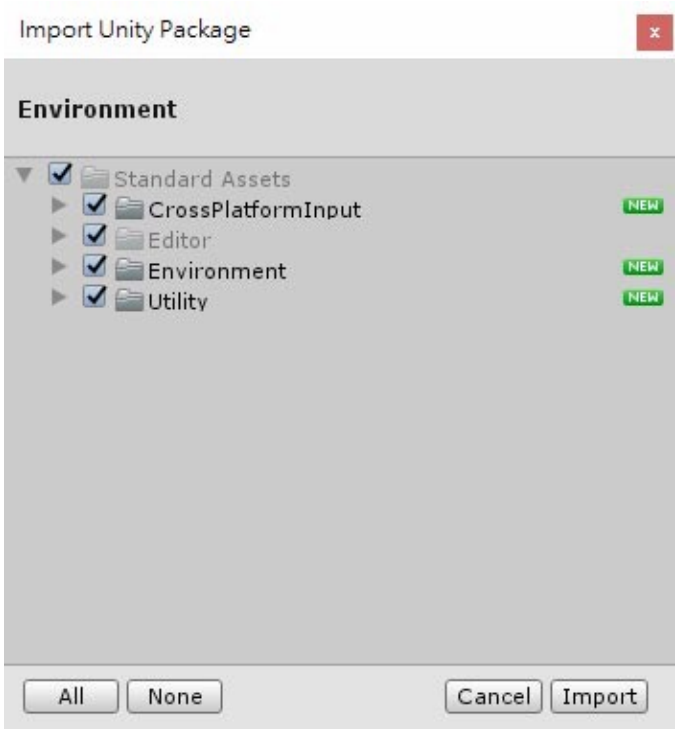


在 Standard Assets > FirstPersonCharacter > Prefabs、Standard Assets > RollerBall > Prefabs 以及 Standard Assets > ThirdPersonCharacter > Prefabs 資料夾裡，都提供有立即可用的預置元件，直接拖拉到場景中，就能進行不同種類的角色控制：

- FirstPersonCharacter：第一人稱角色控制。
  - FPSController：使用 Character Controller 組件為主的第一人稱角色控制。
  - RigidBodyFPSController：以 Rigidbody 組件的控制為主的第一人稱角色。
- RollerBall：以鏡頭視角為主要控制方向，並受力學影響的滾動球體角色控制。
- ThirdPersonCharacter：
  - ThirdPersonController：以鏡頭視角為主要控制方向的第三人稱角色控制。
  - AIThirdPersonController：配合 Unity 的 Navigation 使其具備 AI 尋徑功能的第三人稱角色控制。

4. Environment

匯入 Enviroment 資源包裡除了 Environment 相關的資源之外，還包含了相關的 CrossPlatformInput 以及 Utility 的資源必須同時匯入。



主要提供配合 Unity 的 Terrain 系統來製作地形環境所需要的資源，如樹木、草、地表貼圖以及水面效果。

5. ParticleSystems

匯入 Enviroment 資源包裡除了 Environment 相關的資源之外，還包含了相關的 CrossPlatformInput 以及 Utility 的資源必須同時匯入。

在 Standard Assets > ParticleSystems > Prefabs 資料夾裡，有立即可用的預置元件，直接拖拉到場景中，就能呈現其粒子特效，包含有爆炸、火、沙塵、水流、煙火、火光等等特效。

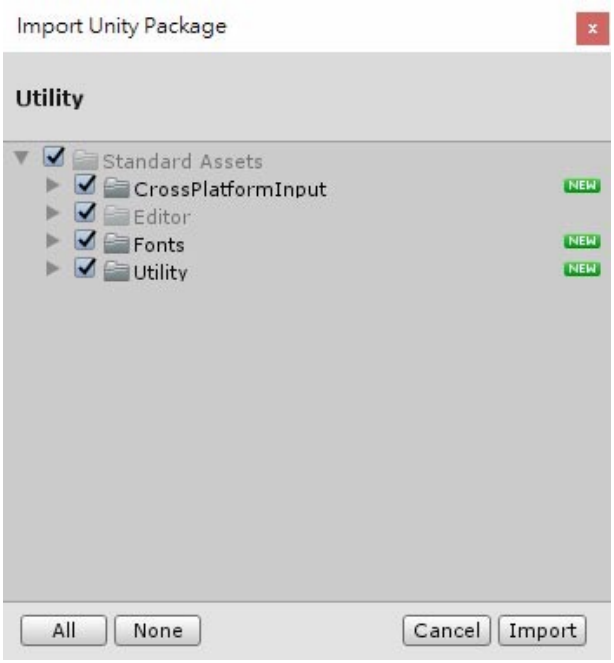
6. Prototyping

匯入 Prototyping 資源包裡除了 Prototyping 相關的資源之外，還包含了相關的 CrossPlatformInput 以及 Utility 的資源必須同時匯入。

在 Standard Assets > Prototyping > Prefabs 資料夾裡，有立即可用的預置元件，直接拖拉到場景中，就可以開始使用這些原型物件編輯場景原型，包含有方塊、立方體、地板、房屋、連接物、撿取物、柱子、平台、斜坡、階梯、牆壁等可供使用。

7. Utility

匯入 Utility 資源包裡除了 Utility 相關的資源之外，還包含了相關的 CrossPlatformInput 的資源必須同時匯入。



從選單列的 Component > Scripts 選單或者 Standard Assets > Utility 資料夾，可找到能賦予遊戲物件的實用功能：

- ActivateTrigger：使物件被觸發時，可對目標物件執行觸發、開關遊戲物件、啟動、執行動畫、替換遊戲物件等功能。
- AlphaButtonClickMask：遮蔽 UI 按鈕的透明區域，使透明區域被點擊時不會觸發按鈕功能。
- AutoMoveAndRotate：使遊戲物件依照指定值自動移動或旋轉。
- DragRigidbody：使具備 Rigidbody 的遊戲物件可被拖拉。
- EventSystemChecker：檢查 EventSystem 是否存在，不存在就會自動產生。
- FollowTarget：依照指定的偏移值，跟隨目標物件。
- FPSCounter：在 UI 的 Text 顯示 FPS 值。
- ObjectResetter：提供 DelayedReset 功能，使遊戲物件能夠被要求在指定時間後重置。
- ParticleSystemDestroyer：指定時間範圍之後清除粒子系統。

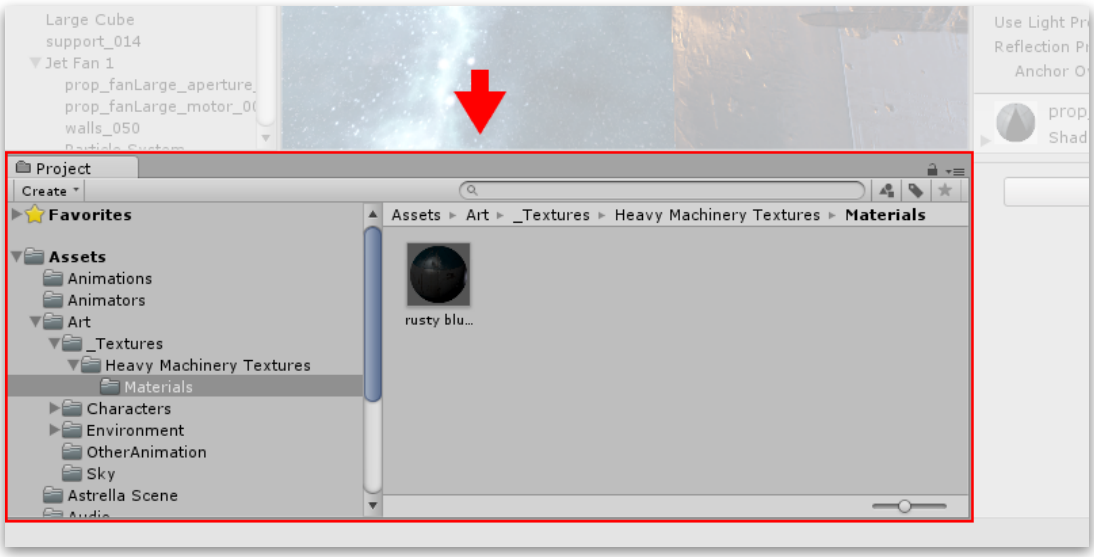
- PlatformSpecificContent：用來設置指定平台使用到的遊戲物件和組件有哪些，並依照當前平台自訂開啟或關閉。
- SimpleMouseRotator：使物件前方（Z 軸方向）跟隨滑鼠指標轉動。
- TimedObjectActivator：定時開、關、銷毀指定遊戲物件或重新載入場景。
- TimedObjectDestructor：定時銷毀遊戲物件。
- WaypointCircuit：使用遊戲物件來設定航點。
- WaypointProgressTracker：處理航點進度的追蹤器。

標準資源包不會自動更新：
在建立新專案的時候，可以選擇要包含哪些標準資源，Unity 會從安裝資料夾將所選擇的資源複製到新的專案資料夾裡。因此，在你更新 Unity 編輯器到新版本時，原本在專案中已經匯入的標準資源並不會更新，所以，你必須手動更新它們。

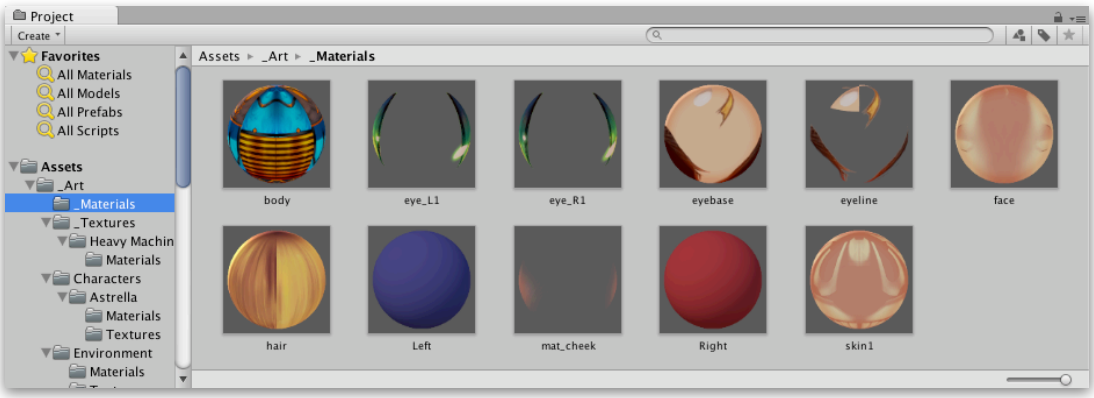
新版本的標準資源包可能會與現存安裝的表現不同（例如，效能或品質的原因）。使用新版本可能會使專案外觀或行為變得不同，可能需要重新調整參數。在決定重新安裝前，應該要先確認 Unity 發佈說明的資源包內容。

IX.主要工作視窗

1. *Project* 視窗



在這個視窗，能夠訪問和管理專案中的資源。



左側的面板以層次列表來顯示專案的資料夾結構。當點擊其中的資料夾，它的內容會顯示在右側的面板。可以透過點擊小三角形來展開或收合其內容的任何巢狀資料夾。按住 Alt 按鍵，能夠遞歸的展開或收合整個巢狀資料夾。

右側面板顯示的個別資源以圖標來指示它們的類型（如程式腳本、材質、子資料夾等）。在面板右下方可使用滑條來調整圖標大小。如果滑條移動到最左邊，將會以層次列表的顯示來取代圖標。滑條左側的空白處顯示目前所選擇的東西，包含它們的完整路徑。

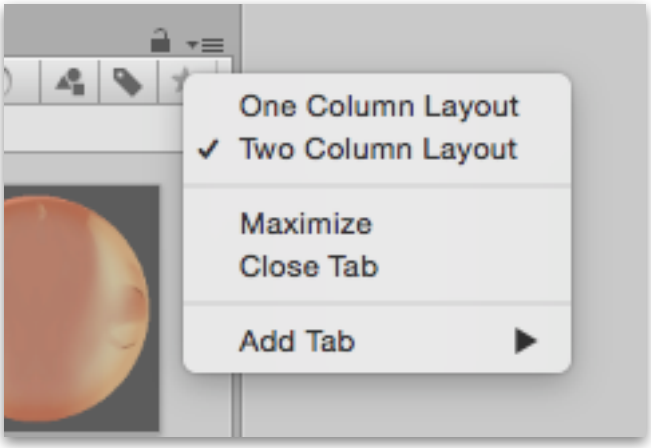
在專案結構列表之上的「Favorites」區塊，你可以保持經常使用的項目以方便訪問。你可以將專案結構列表拖拉到 Favorites，還可以儲存搜尋查詢到這裡。

面板上方是個「導航標記路徑」，顯示目前正在查看的資料夾路徑。可以點擊路徑上的個別元素來輕鬆的在資料夾層次中瀏覽。搜尋時，此欄會更改為顯示正在搜尋的區域（資源根目錄、所選資料夾或 Asset Store）以及以斜線分隔顯示在 Asset Store 中可用的免費和付費資源的數量。如果不需要，在 Unity 的 Preferences 視窗的 General 區塊有個選項能夠關閉顯示 Asset Store 符合搜尋的數量。

視窗頂部是瀏覽器的工具列。

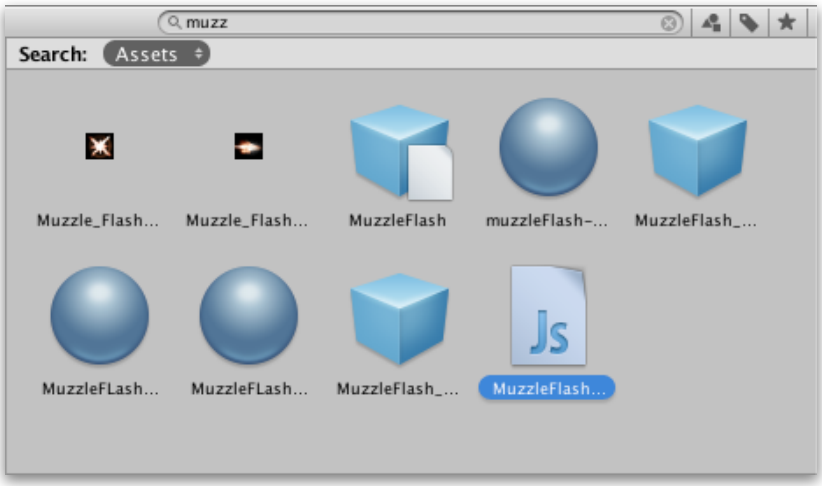
位於工具列左邊，Create 選單讓你添加新的資源和資料夾到目前的資料夾中。右邊一系列的工具則是讓你能夠在專案中搜尋資源。

視窗選單提供切換選項讓你切換 Project 視窗為一個面板的版本，只會有層次結構列表，而沒有圖標外觀。鎖頭圖標以類似 Inspector 的方式讓你凍結目前視窗的內容（避免它們被其他地方的事件所改變）。



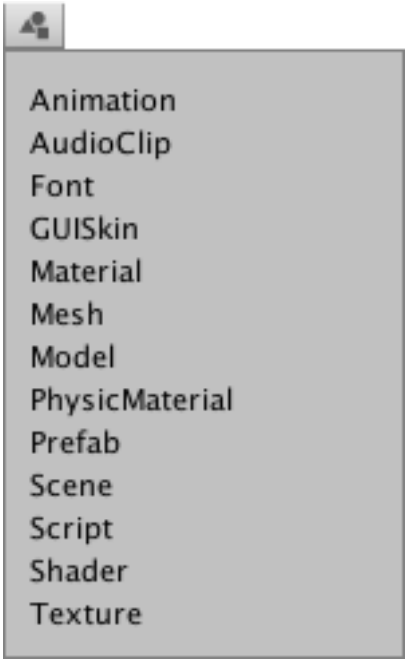
- 搜尋

  瀏覽器上有個強大的功能，特別有用於在大專案或陌生的專案中找出資源。基本上會根據在搜尋欄位輸入的文字過濾資源。

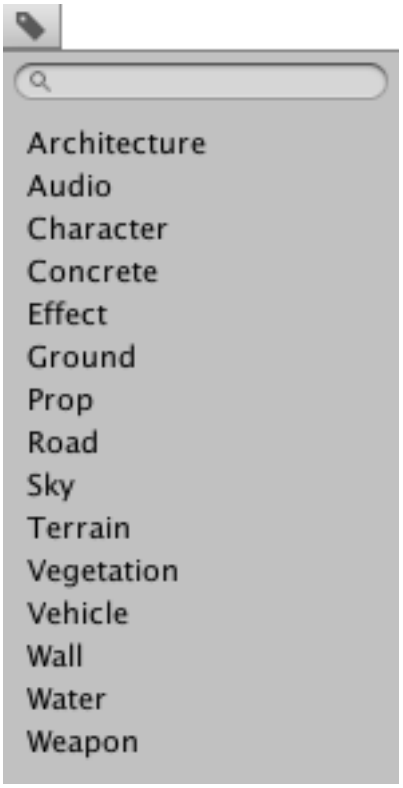


  如果提供更多個搜尋字詞，就會縮小搜尋範圍，所以，如果你輸入「coastal scene」，將只會使用「coastal」和「scene」這兩個名稱來尋找資源（字詞會 AND 在一起）。

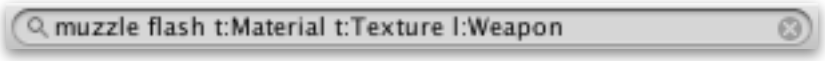
  搜尋欄位右邊三個按鈕。第一個讓你能夠根據其類型進一步過濾搜尋找到的資源。



第二個則依據它們的標籤過濾（可以為資源在 Inspector 視窗設置）。由於標籤數量可能非常多，所以標籤選單有自己的搜尋過濾欄位。



注意：過濾器的運作是使用搜尋文字透過添加額外的字詞來實現。在字詞開頭使用「t:」來過濾指定資源類型，使用「l:」來指定標籤。如果你很清楚你想要尋找什麼，可以直接在搜尋欄位輸入這些字詞，而不使用選單。還可以一次搜尋多個類型或標籤。添加幾個類型將擴展搜尋包含所有指定的類型（類型將 OR 在一起）。添加多個標籤將使搜尋縮小到具有所指定標籤的項目（標籤將被 AND）。



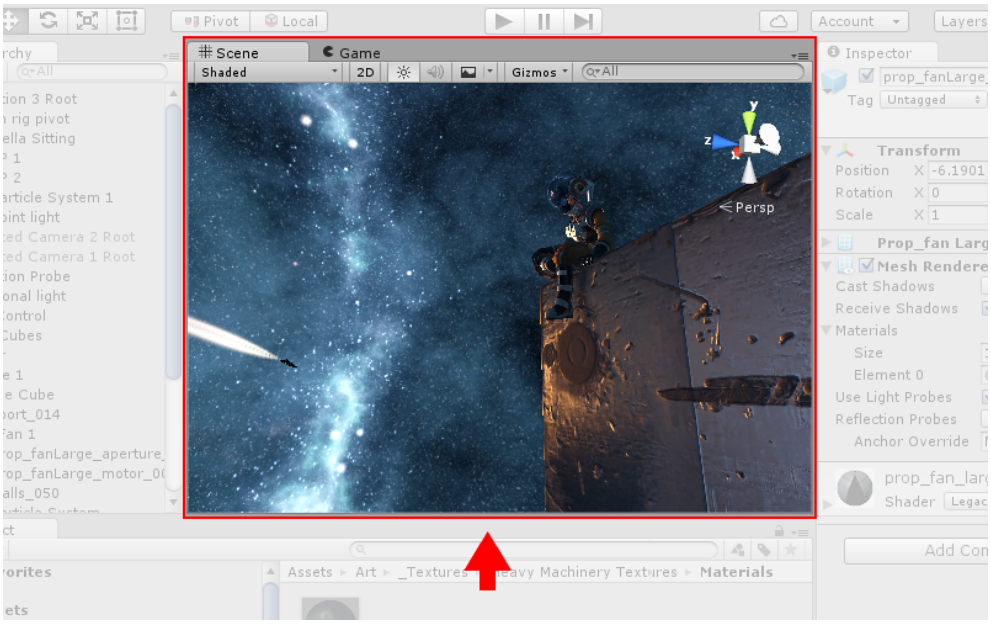
最後一個按鈕則是將所搜尋到的列表添加到 Favourites 區。

- 搜尋 Asset Store

  Project 視窗瀏覽器的搜尋也可以從 Asset Store 搜尋可用的資源。如果從「導航標記條」選擇 Asset Store，所有符合查詢的免費和付費項目都會被顯示出來。透過類型和標籤的搜尋方式與 Unity 專案中相同。首先會針對資源名稱確認來搜尋，然後依照資源包名稱、資源包標籤和資源介紹的順序來檢查（因此，名稱包含搜尋條件的排名會高於其說明中有相同字詞的物品）。

  如果從所找到的列表選擇一個項目，詳細訊息將會與購買和 / 或下載選項顯示在 Inspector 視窗。有些資源可以提供預覽，因此你可以在購買之前旋轉 3D 模型查看。Inspector 視窗也會提供 Asset Store 視窗的相關按鈕，以利查看其它詳細訊息。

2. Scene 視窗



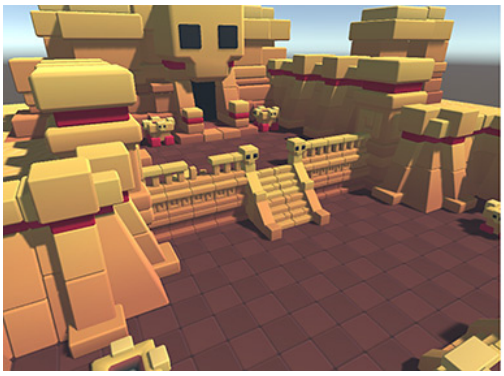
在這個視窗是與你所創建視窗互動的視窗。你將使用這個視窗選擇和放置風景、角色、攝影機、光源和所有其它類型的遊戲物件。

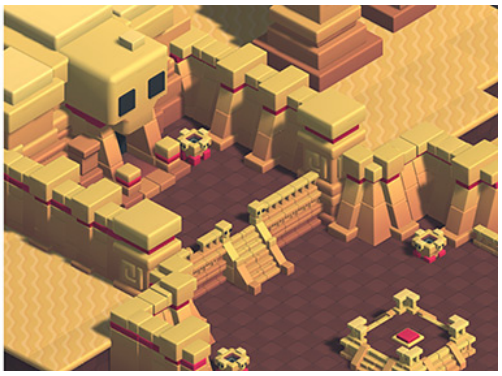
- Scene Gizom

  在 Scene 視窗右上角的 Scene Gizmo，顯示 Scene 視窗攝影機目前的方向，並讓你能夠快速的修改觀察角度和投影模式。

  Scene Gizmo 在立方體的每個邊都有個錐形臂，被標示了 X、Y 和 Z。點擊任何錐形軸臂將場景視野攝影機貼齊到其所代表的軸（例如上視圖、左視圖和前視圖）。還可以右鍵點擊立方體顯示視角選單。如果要返回預設視角，可右鍵點擊 Scene Gizmo，然後點擊 Free。

  還可以切換透視的開和關。此切換可以使場景視窗在透視（Perspective）和正交（Orthographic - 有時也稱為等距 Isometric）之間切換其投影模式。要這麼做，可以點擊在 Scene Gizmo 中間的立方體或其下的文字來切換。





如果你的場景視窗處於怪異的視角（上下顛倒或者某個混亂角度），按住 Shift 並點擊 Scene Gizmo 中間的立方體，以返回場景側面稍微上方的透視（Perspective）視角。

點擊 Scene Gizmo 右上角的鎖頭，可禁用或啟用場景的旋轉。一旦場景旋轉被關閉，拖拉滑鼠右鍵將會平移視野，而不再是旋轉視野。

注意！2D 模式下，Scene Gizmo 不會顯示，因為此時的視野只會垂直於 XY 平面。

- 方向鍵移動

  你可以使用方向鍵在場景中移動，就像「走過」一樣。上下鍵使攝影機依據所面對的方向前進和後退。左右鍵橫向平移視野。按住 Shift 鍵可以更快地移動。

- 手工具：當手工具被選擇時（快捷鍵：Q），可以使用以下的滑鼠控制。
  - 移動：拖動攝影機移動視野。

  - 繞行：按住 Alt 和滑鼠左鍵拖動，攝影機將圍繞目前的支點繞行移動。這在 2D 模式無法使用。
  - 縮放：按住 Alt 拖動和滑鼠右鍵拖動，可縮放 Scene 視窗。

  按住 Shift 鍵可提高移動和縮放的速度。

- 飛越模式

  使用飛越模式來透過使用第一人稱飛行的方式瀏覽 Scene 視窗內容，類似於在許多遊戲中瀏覽的方式。

  1. 按著滑鼠右鍵。
  2. 使用滑鼠移動視野，W、A、S、D 按鍵來向前、後、左、右移動，Q 和 E 鍵可上下移動。
  3. 按住 Shift 可移動得更快。

  飛越模式是為透視模式而設計的，在正交模式下，按住滑鼠右鍵會使攝影機繞行轉動。

  注意：在 2D 模式下，飛越模式沒有作用，此時按著滑鼠右鍵只能拖拉滑鼠平移視野。

- 視野集中在遊戲物件上

  要將 Scene 視窗中心對準遊戲物件，先在 Hierarchy 視窗選擇該遊戲物件，然後移動滑鼠到 Scene 視窗上按下 F 鍵。這個功能也能在選單列的 Edit > Frame Selected 找到。即使遊戲物件正在移動，要將視野鎖住到遊戲物件，請按 Shift + F，此功能也可在 Edit > Lock View to Selected 找到。

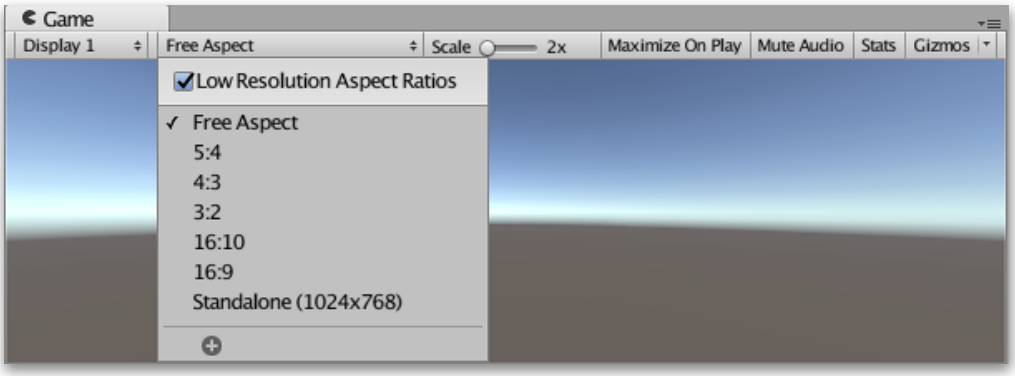
3. Game 視窗

Game 視窗的內容是從遊戲中的攝影機所呈現的。它代表你最終所要發佈的遊戲。你會需要使用一個或多個攝影機來控制玩家玩你的遊戲時所實際看到的內容。

- 播放模式

使用工具列的按鈕來控制編輯器播放模式，並查看你發佈的遊戲如何運行。在播放模式下，所做的任何改變都是暫時的，當你結束播放模式都將被重置。編輯器會稍微變暗來提示你。

- 控制列



- Display：如果場景中有多台攝影機，可點擊此處從攝影機列表選擇。預設是 Display 1。（可以在 Camera 組件的 Target Display 下拉選單指定）。
- Aspect：選擇不同的值去測試你的遊戲在使用不同長寬比的顯示器上看起來會是怎麼樣。預設是 Free Aspect。
- Scale 滑條：向右滑動以更詳細的放大並查看遊戲畫面的區域。還可讓你縮小來查看設備解析度高於 Game 視窗大小的整個畫面。你也可以在遊戲停止或暫停時使用滑鼠滾輪和中鍵來執行此操作。
- Maximize On Play：當你進入播放模式時，使 Game 視窗最大化來以全螢幕預覽（編輯器視窗的 100%）。
- Mute Audio：當進入播放模式時，使用此功能可使任何遊戲中的聲音變為靜音。
- Stats：可顯示統計訊息疊加在 Game 視窗之上，其中包含遊戲聲音和圖像的渲染統計訊息。這對於在播放模式下監看遊戲的效能非常有用。
- Gizmos：切換顯示可見的 Gizmo，要在播放模式中只看到某些類型的 Gizmo，可點擊旁邊的下拉箭頭，只勾選想要看到的 Gizmo 類型。

• 進階選項

在 Game 標籤上點擊右鍵可顯示 Game 視窗的進階選項。

- Warn if No Cameras Rendering：此項目預設是開啟的，如果沒有攝影機正在渲染到螢幕上，則會顯示警告。這對於診斷諸如意外刪除或停用攝影機的問題非常有用。保持啟用，除非你故意不使用攝影機來渲染遊戲。
- Clear Every Frame in Edit Mode：此項目預設是開啟的。當遊戲未播放時，會導致遊戲視窗每幀都被清除。這可以防止在配置遊戲時造成拖尾效應。 保持啟用，除非你在非播放模式下，正依賴於前一幀的內容。

4. Hierarchy 視窗

Hierarchy 是包含當前場景中每個遊戲物件的列表。其中一些直接是資源檔案（如 3D 模型）的實例，而其它是 Prefabs 的實例，它們是組成大部分遊戲的自定義物件。當場景中添加和刪除物件時，它們也將從 Hierarchy 視窗顯示和消失。

預設的情況下，物件按照他們被產生的順序列於 Hierarchy 視窗中。可以通過向上或向下拖動物件或使其成為「子」或「父」物件來重新排序。

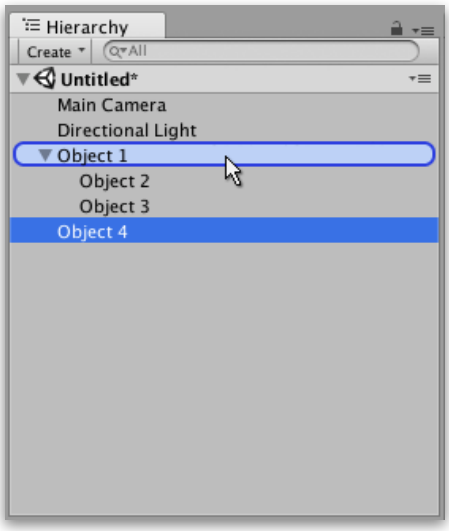
- Parenting

  Unity 使用一種叫做 Parenting 的概念。當您創建一個物件群組時，最上層的物件或場景稱為「父物件」，所有被群組於其下的物件都稱為「子物件」。還可以創建巢狀的父子物件。

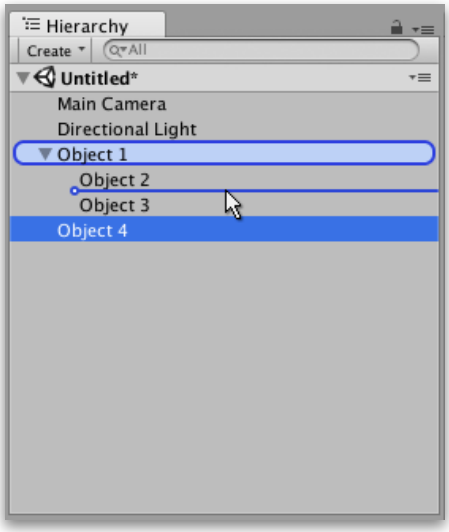
點擊父物件的下拉箭頭（在其名稱的左側）顯示或隱藏其子項。按住 Alt 鍵同時點擊下拉箭頭以切換父項的所有子物件的顯示或隱藏。

- 製作子物件

要使任何物件成為另一個物件的「子項」，在 Hierarchy 視窗將所需的子物件拖放到預期的父物件上。



還可以將物件拖放到其它物件旁，使其成為「兄弟姐妹」- 即同一父物件下的子物件。將物件拖動到現有物件的上方或下方，直到出現一條水平的藍線，並將其放在現有物件旁邊。

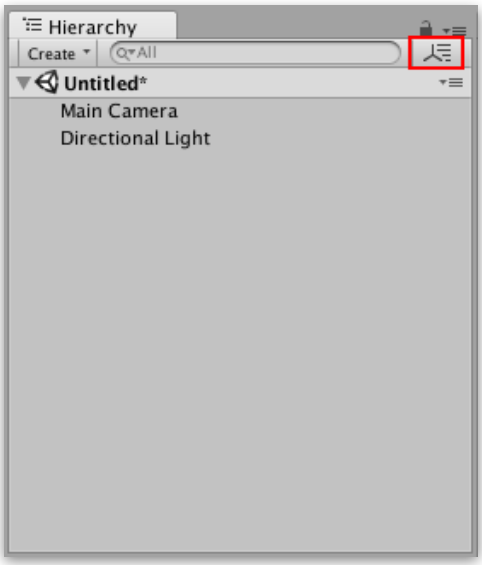


子物件會繼承父物件的移動和旋轉。

- 字母數字排序

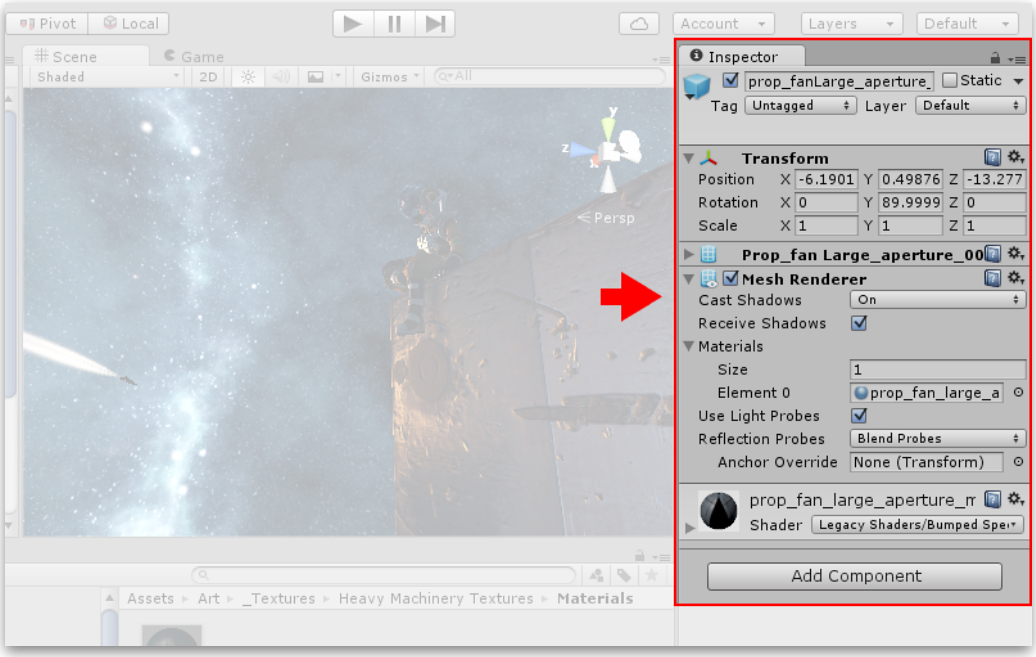
  Hierarchy 視窗中，物件的順序可以更改為字母數字順序。在選單列選擇 Edit > Preferences 開啟 Preferences 視窗。 勾選 Enable Alpha Numeric Sorting。

  當勾選後， Hierarchy 視窗的右上方會出現一個圖標，使你能夠在 Transform 排序（預設值）或字母排序之間進行切換。



5. *Inspector* 視窗

Unity 編輯器中的專案，是由多個包含程式腳本、聲音、網格和其它圖像元素的遊戲物件所組成。 Inspector 視窗顯示關於目前所選取遊戲物件的詳細訊息，包括所有連接的組件及其屬性，並允許修改場景中遊戲物件的功能。。

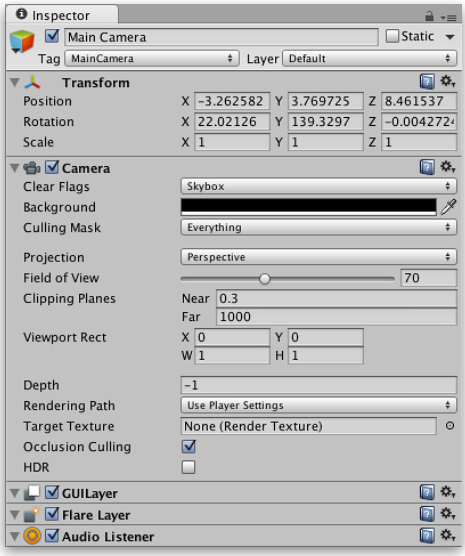


- 檢閱遊戲物件

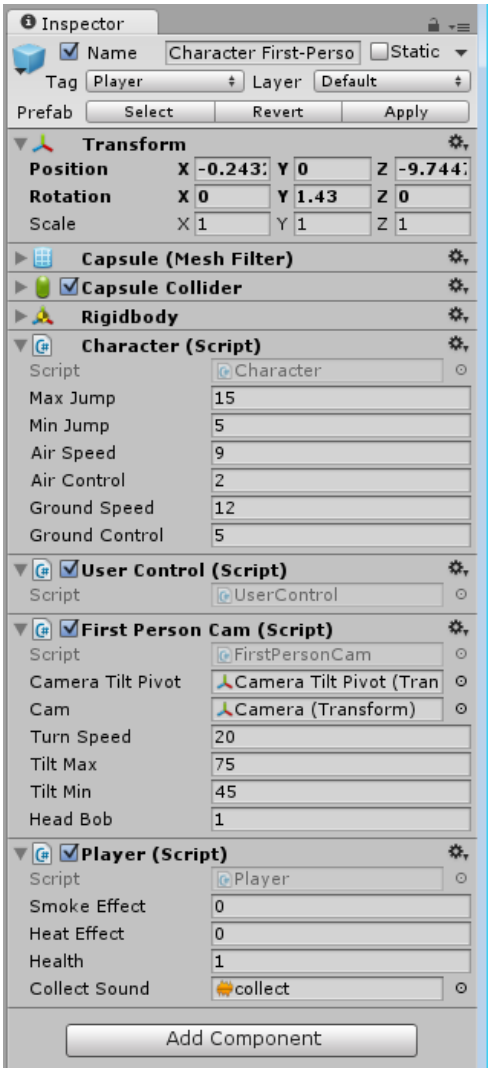
  在 Unity 編輯器中使用 Inspector 視窗去查看和編輯幾乎所有東西的屬性和設置，包括物理遊戲項目，如遊戲物件、資源和材質等，以及編輯器中的設置和喜好設定。

  當你在 Hierarchy 或 Scene 視窗中選擇一個遊戲物件時，Inspector 視窗將顯示該遊戲物件的所有組件和材質的屬性。使用 Inspector 視窗編輯這些組件和材質的設置。

  右圖顯示選擇 Main Camera 遊戲物件的 Inspector 視窗。 除了遊戲物件的位置（Position）、旋轉（Rotation）和縮放（Scale）值之外，還可以編輯 Main Camera 的所有屬性。



- 檢閱程式腳本變數

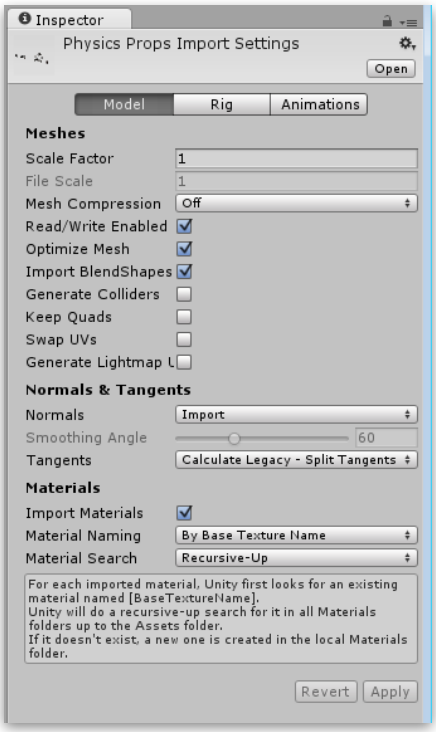


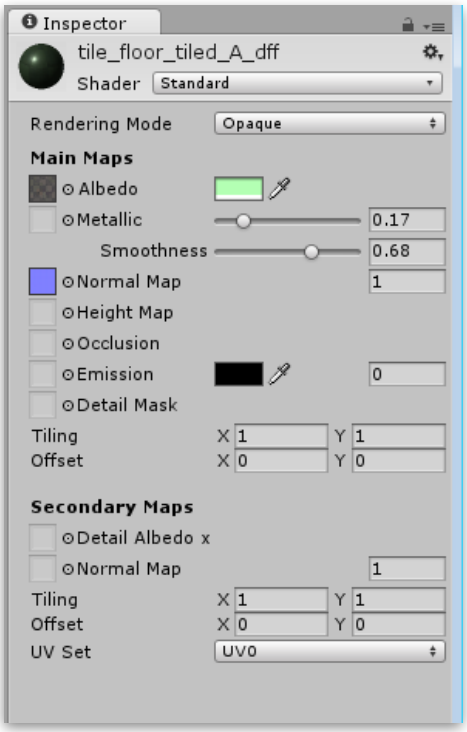
當遊戲物件附加了自定義程式腳本組件時，Inspector 視窗將顯示該程式腳本的公開變數。可以使用與設置編輯器內建組件的相同方式來編輯這些變數的設置。這意味著你可以輕鬆地設置在程式腳本中的參數和預設值，而無需修改程式碼。

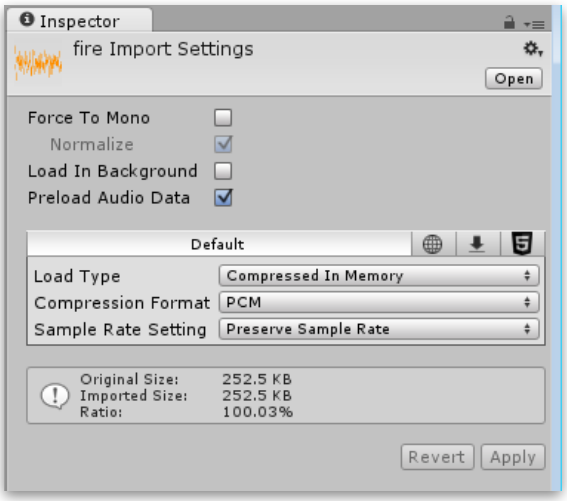
- 檢閱資源

  在 Project 視窗中選擇資源時，Inspector 視窗將會顯示與資源被匯入以及用於運行時相關的設置。

  每種類型的資源都有不同的設置選擇。 以下圖片示範 Inspector 視窗顯示其它資源類型匯入設置的一些範例：

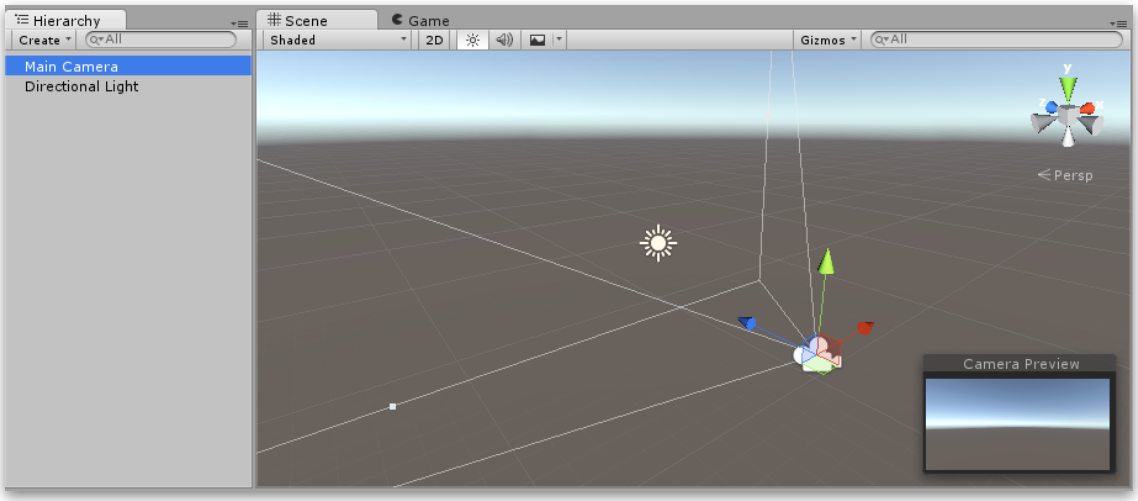






X.遊戲場景

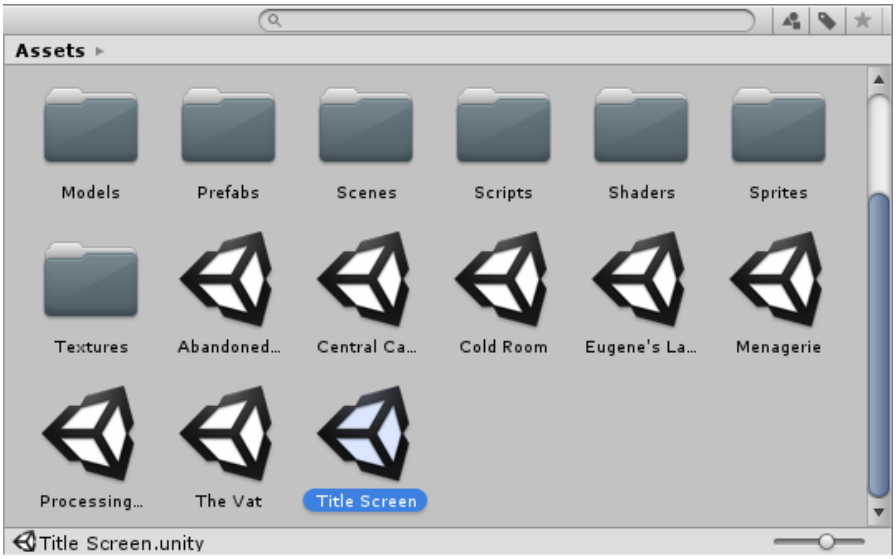
遊戲場景包含了環境以及你的遊戲選單。將每個個別的場景檔案視為一個個別的關卡。在每個場景中，放置你的環境、障礙物和裝飾品，大體上設計和構建你的遊戲。



創建新的 Unity 專案時，場景視窗將顯示一個新的場景。此場景未命名，也未儲存。除了攝影機（稱為 Main Camera）和光源（稱為 Directional Light）外，場景是空的。

1. 儲存場景

要儲存正在編輯的場景，在選單列中選擇 File > Save Scene，或者按 Ctrl + S（Windows）或 Cmd + S（masOS）。



Unity 將場景做為資源儲存在專案的 Assets 資料夾中。這意味著它們會與其餘資源一起出現在 Project 視窗中。

2. 開啟場景

要在 Unity 中打開場景，請在 Project 視窗中雙擊「場景資源」。 你必須在 Unity 中開啟來編輯它。

如果當前的場景包含未儲存的變更，Unity 會詢問是要儲存還是丟棄更改。

XI.多場景編輯

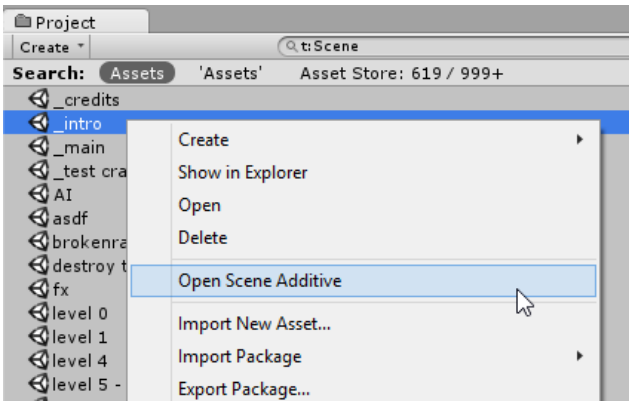
多場景編輯讓你能夠在編輯器中同時打開多個場景，並使運行時更容易管理場景。

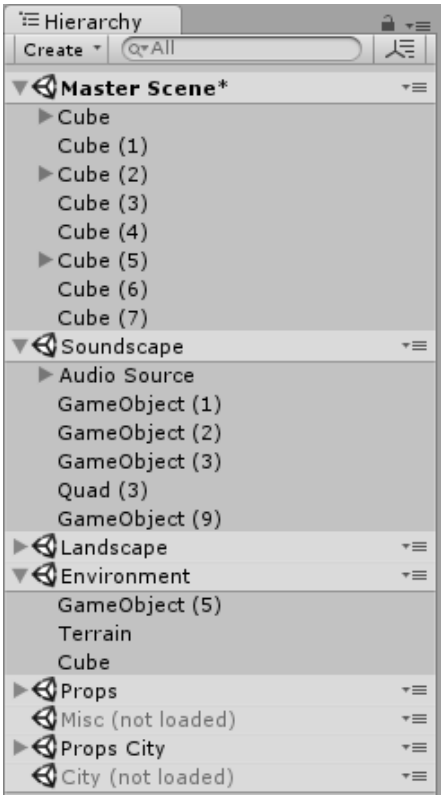
在編輯器中打開多個場景的功能使你能夠創建大型串流世界，並在場景編輯協作時改善工作流程。

1. 編輯器操作

要開啟新場景並將其添加到 Hierarchy 視窗中的目前場景列表中，可在場景資源按滑鼠右鍵選擇 Open Scene Additive，或者將一個或多個場景從 Project 視窗拖拉到 Hierarchy 視窗中。

當你在編輯器中打開多個場景時，每個場景的內容將個別的顯示在 Hierarchy 視窗中。每個場景的內容會出現在顯示場景名稱和儲存狀態的場景分隔欄下方。

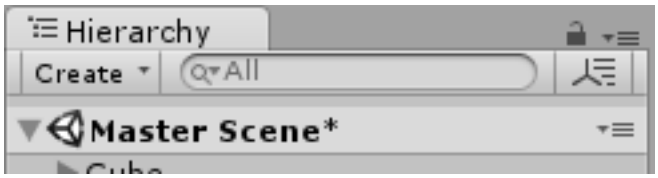




當存在於 Hierarchy 視窗中時，場景可以載入或卸載以顯示或隱藏每個場景中所包含的遊戲物件。這與從 Hierarchy 視窗中添加和刪除它們不同。

場景分隔欄可以在 Hierarchy 視窗中折疊場景的內容，如果你載入了大量的場景，這會有助於導覽。

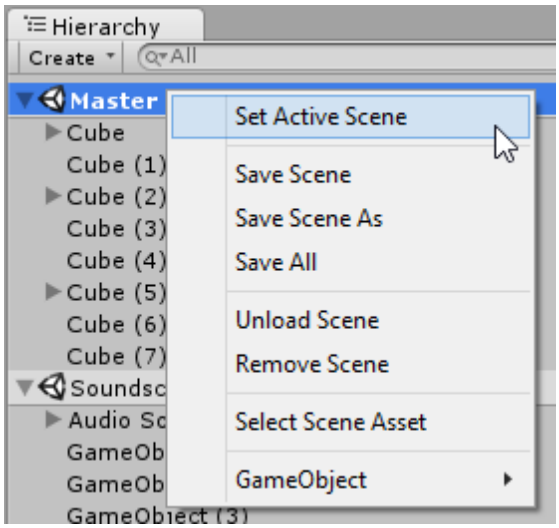
當處理多個場景時，每個被修改的場景都需要儲存其變更，因此可能有多個未儲存的場景同時開啟。未儲存的場景將在場景分隔欄中的名稱旁邊顯示一個星號。



每個場景可以通過分隔欄的選單個別儲存。從 File 選單選擇 Save Sene 或者按 Ctrl + S 可以儲存所有開啟場景的變更。

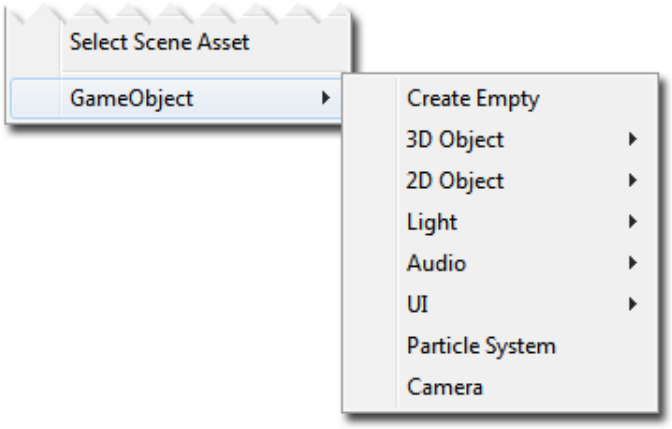
場景分隔欄的選單可以讓你對所選擇的場景執行其它的行為。

- 已載入場景的場景分隔欄選單：

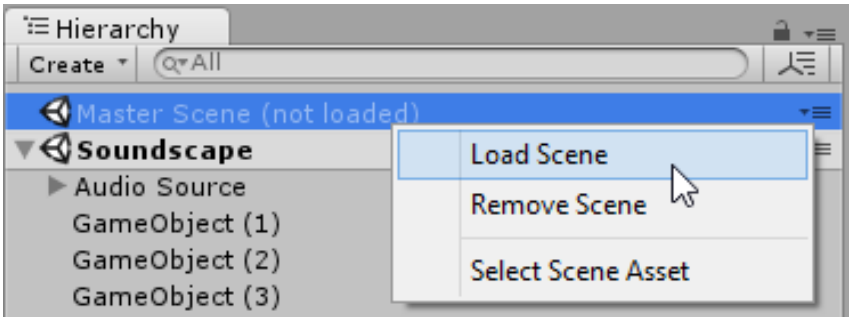


  - Set Active Scene：這讓你指定哪個場景能夠建立 / 實例化新的遊戲物件。始終要有一個場景必須被標記為 Active Scene。
  - Save Scene：只儲存變更到所選擇的場景。
  - Save Scene As：將所選場景（以及目前的任何變更）儲存為新的場景資源。
  - Save All：儲存所有場景的變更。

- Unload Scene：卸載場景，但是將場景保留在 Hierarchy 視窗。
- Remove Scene：卸載並從 Hierarchy 視窗移除場景。
- Select Scene Asset：在 Project 視窗選擇場景的資源。
- GameObject：提供子選單讓你能夠在所選擇的場景建立遊戲物件。此選單對應於 Unity 主要的 GameObject 選單。



- 未載入場景的場景分隔欄選單：



- Load Scene：載入場景內容。
- Remove Scene：從 Hierarchy 視窗移除場景。
- Select Scene Asset：在 Project 視窗選擇場景的資源。

2. 播放模式

在播放模式中，在 Hierarchy 中有多個場景，將會顯示一個名為 DontDestroyOnLoad 的附加場景。

在 Unity 5.3 之前，任何你在播放模式中標記為 DontDestroyOnLoad 的物件仍將顯示在 Hierarchy 視窗中。這些物件不被視為任何場景的一部分，但是 Unity 仍然顯示其物件，為了讓你檢查它們，這些物件現在顯示為特殊的 DontDestroyOnLoad 場景的一部分。

你無權存取 DontDestroyOnLoad 場景，並且在運行時不可用。

3. 場景的特定設置

有一些設置是特定儲存於每個場景中：RenderSettings 和 LightmapSettings（可在 Lighting 視窗找到）、NavMesh 設置、在 Occlusion Culling 視窗的場景設置。

其工作原理是，每個場景會管理它自己的設置，並且只有與該場景相關的設置才會儲存在場景檔案中。

如果打開多個場景，則用於渲染和 NavMesh 的設置來自 Active Scene。這意味著如果要更改場景的設置，則必須只打開一個場景並更改設置，或者使有問題的場景成為 Active Scene 並更改其設置。

當你在編輯器中或運行時切換 Active Scene 時，將會套用新場景中的所有設置並替換所有以前的設置。

4. 提示和技巧

透過按住 Alt 鍵拖拉場景添加到 Hierarchy 視窗，能夠保持場景的卸載狀態，這樣，就可以隨時根據需要載入場景。

新場景能夠使用 Project 視窗的 Create 建立，建立後的內容將會包含預設的遊戲物件。

要避免每次重新開啟 Unity 就必須設置 Hierarchy 視窗或想要簡單的儲存不同的設置，可以使用 EditorSceneManager.GetSceneManagerSetup 來獲得 SceneSetup 物件列表來描述目前的設定。可以將其序列化到 ScriptableObject 或伴隨一些其它你想要儲存關於場景設置的其它任何資訊。要恢復 Hierarchy，只需重新創建 SceneSetups 列表，使用 EditorSceneManager.RestoreSceneManagerSetup。

在運行時獲取載入場景的列表，只需使用取得 sceneCount 並使用 GetSceneAt 輪詢即可。

你可以透過 GameObject.scene 得知遊戲物件屬於哪個場景，並可使用 SceneManager.MoveGameObjectToScene 將遊戲物件移動到另一個場景。

建議避免使用 DontDestroyOnLoad 來保存想在場景之間生存的管理器遊戲物件。而是創建一個包含所有管理器的管理器場景，並使用 SceneManager.LoadScene（最後參數為 LoadSceneMode.Additive）和SceneManager.UnloadScene 來管理遊戲進度。

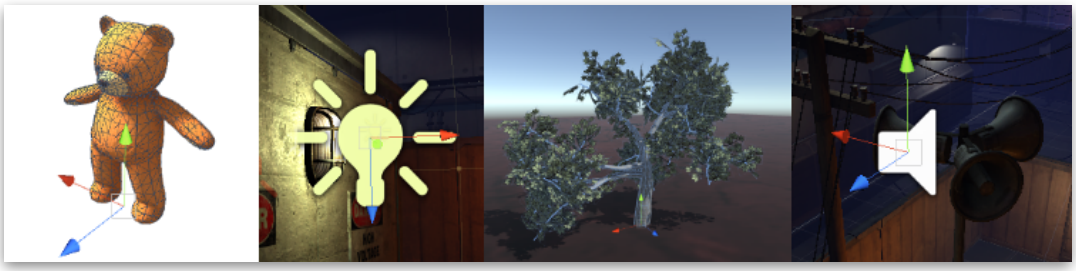
XII.遊戲物件與組件概念

1. 遊戲物件 *GameObject*

GameObject 在 Unity 編輯器中是最重要的概念。

遊戲中的每個物件都是 GameObject，從角色和收藏品到光源、攝影機和特效。然而，GameObject 不能自己做任何事情; 你需要給它屬性，才能成為一個角色、環境或特殊效果。

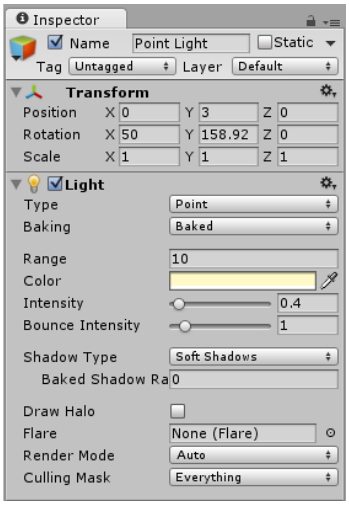
為了給予 GameObject 所需要的屬性變成一個光源、一棵樹、一個攝影機，必須為其添加組件。根據要創建的物件種類，你可以向 GameObject 添加不同的組件組合。



可以將 GameObject 視為一個空的烹飪鍋，組件可以做為組合遊戲食譜的不同原料。Unity 有許多不同的內建組件類型，你也可以使用 Unity Scripting API 去製作自己的組件。

GameObject 是 Unity 中表現角色、道具和風景的基本物件。它們本身並沒有太多的實現，而是做為實現實際功能的組件的容器。

例如，通過將 Light 組件附加到 GameObject 來創建光源物件。



實體立方體物件具有 Mesh Filter 和 Mesh Renderer 組件，用於繪製立方體的表面，以及一個 Box Collider 組件，用於在物理條件裡表示物件的實體體積。

GameObject 始終具有附加的 Transform 組件（以表示位置和方向），並且不能移除它。賦予物件功能的其它組件，可以從編輯器的 Component 選單或是從程式腳本來添加。在 GameObject > 3D Object 選單中，還有許多有用的預先構建的物件（原始形狀、攝影機等）可使用。

2. 組件 *Component* 介紹

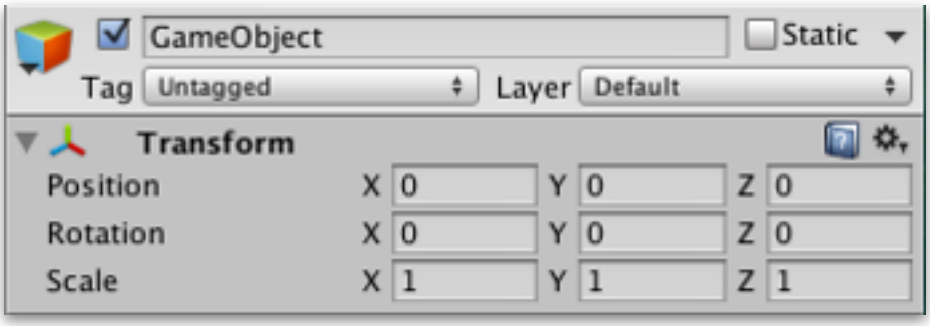
GameObject 包含多個 Component。

你可以透過在 Inspector 視窗查看新的 GameObject 看到 Transform 組件：

(1) 在 Unity 編輯器中的任何專案中打開任何場景。

(2) 創建一個新的 GameObject（選單：GameObject > Create Empty）。

(3) 新的 GameObject 被預選，Inspector 視窗顯示其 Transform 組件。（如果沒有預選，請點擊它查看其 Inspector 視窗。）



請注意，新的空的 GameObject 包含其名稱「GameObject」、Tag「Untagged」和 Layer「Default」。還包含一個 Transform 組件。

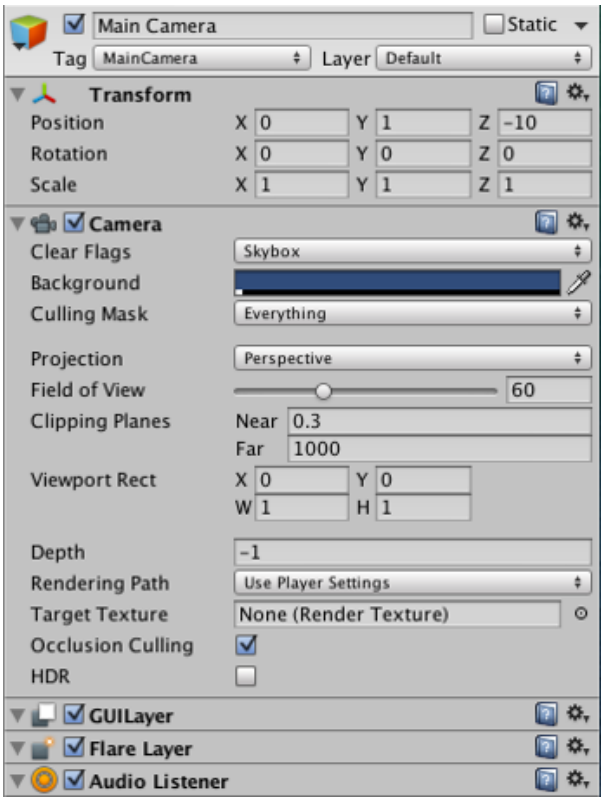
- Transform 組件

  在編輯器中創建沒有 Transform 組件的 GameObject 是不可能的。該組件定義 GameObject 在遊戲世界和 Scene 視窗中的位置、旋轉和縮放。

  Transfrom 組件還實現了一個稱為 Parenting 的概念，這是使用 GameObject 的關鍵部分。

- 其它組件
  Transform 組件對所有 GameObject 至關重要，因此每個 GameObject 都有一個，但 GameObject 也可以包含其它組件。

預設情況下，每個場景都有一個 Main Camera 的 GameObject。它具備幾個組件（可以在開啟的場景中選擇它，並在 Inspector 查看。）



在 Inspector 視窗查看 Main Camera 的 GameObject，可以看到它包含其他組件。具體來說有 Camera、GUILayer、Flare Layer 和 Audio Listener 組件。 所有這些組件都為此 GameObject 提供了功能。

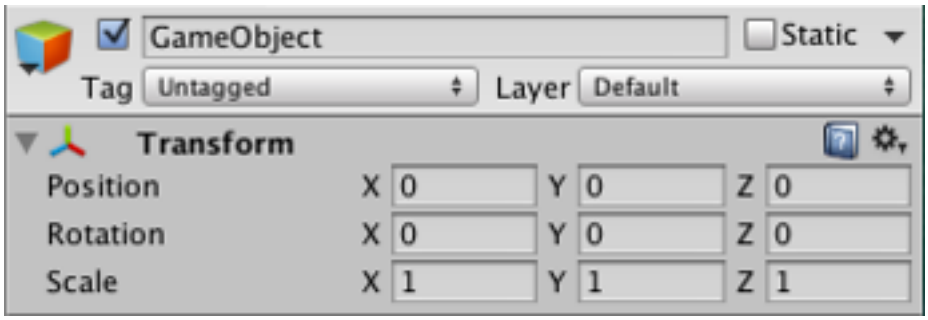
Rigidbody、Collider、Particle System、和 Audio 都是各種可以添加到 GameObject 的組件。

XIII.使用組件 Component

Component 是遊戲中物件和行為的核心。它們是每個 GameObject 的功能塊。

GameObject 是許多不同組件的容器。預設情況下，所有 GameObject 都會自動有一個 Transform Component。這是因為 Transform 主宰 GameObject 所在的位置，以及如何旋轉和縮放。 沒有 Transform Component，GameObject 在遊戲世界上不會有位置。

現在試著點擊選單列的 GameObject > Create Empty 建立一個空的 GameObject。選擇新的 GameObject，然後查看 Inspector 視窗。



請記住，你可以隨時使用 Inspector 視窗來查看哪些 Component 附加到所選的 GameObject 上。隨著添加和刪除 Component，Inspector 視窗將始終顯示當前附加的 Component。你將會使用 Inspector 視窗去更改任何 Component（包括 Script）的所有屬性。

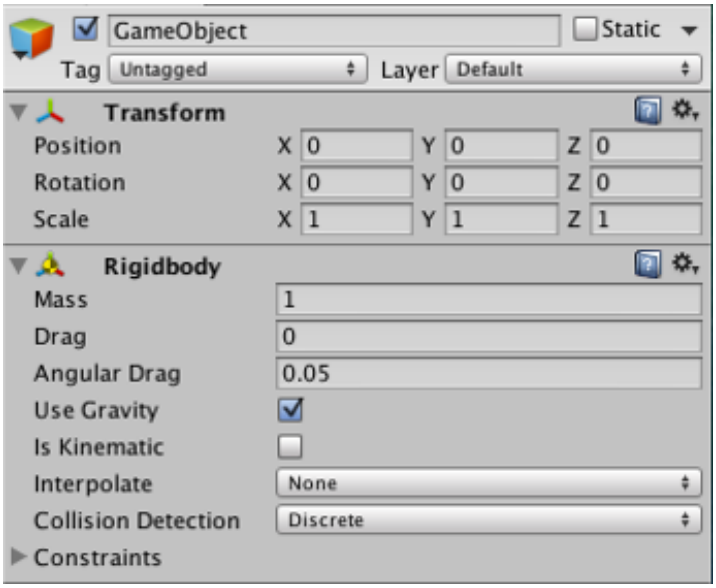
1. 添加 *Component*

你可以通過 Component 選單將 Component 添加到所選的 GameObject。

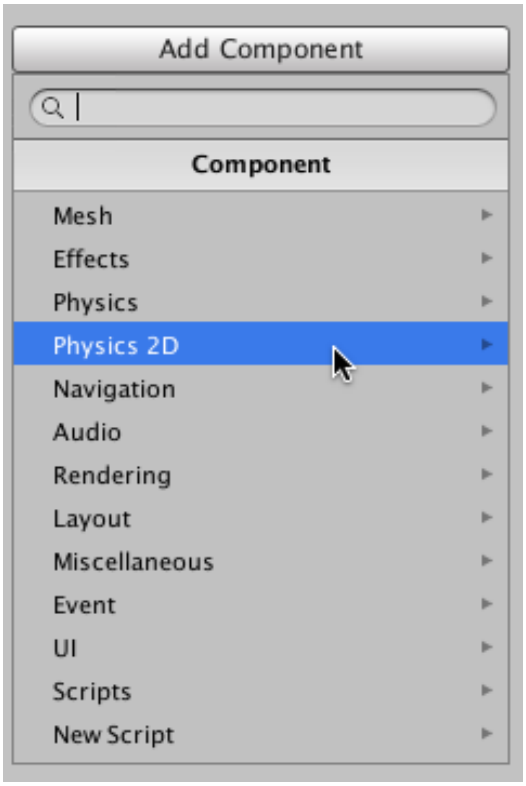
我們現在將嘗試通過在剛剛創建的空的 GameObject 中添加一個 Rigidbody。選擇它，然後從選單中選擇 Component > Physics > Rigidbody。

當你這樣做時，你會看到 Rigidbody 的屬性出現在 Inspector 視窗中。 如果你仍然選擇空的 GameObject，則按 Play 鍵查看一下。

你會注意到，添加了 Rigidbody 給空的 GameObject，GameObject 的 Transform 的 Y 軸位置開始往下降，這是因為 Unity 的物理引擎導致 GameObject 受重力影響而掉落。



另一個選項是使用 Component 瀏覽器，可以在物件的 Inspector 視窗使用 Add Component 按鈕來啟用它。



這個瀏覽器讓你能夠按照類別方便地導覽 Component，並且還具有可按名稱查找 Component 的搜尋欄。

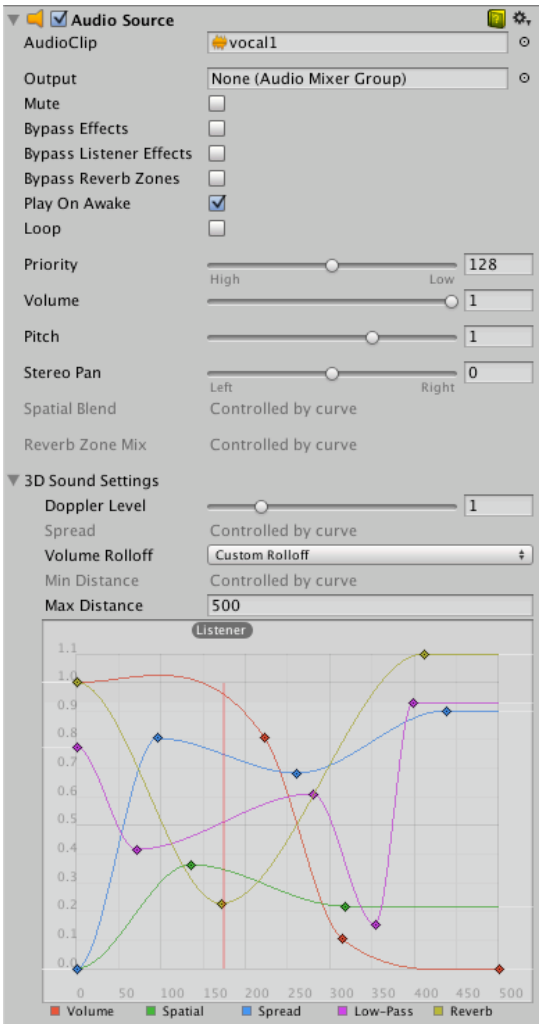
你可以將任何數量或 Component 的組合附加到單個 GameObject。有些 Component 能夠與其它 Component 結合使用。例如，Rigidbody 與任何 Collider 共同運作。Rigidbody 通過 NVIDIA PhysX 物理引擎控制 Transform，Collider 允許 Rigidbody 與其它 Collider 進行碰撞和互動。

如果想了解有關使用特定 Component 的更多訊息，可以在 Inspector 視窗點擊 Component 名稱上的小書圖標來查看 Unity 手冊中的 Component 說明頁面。

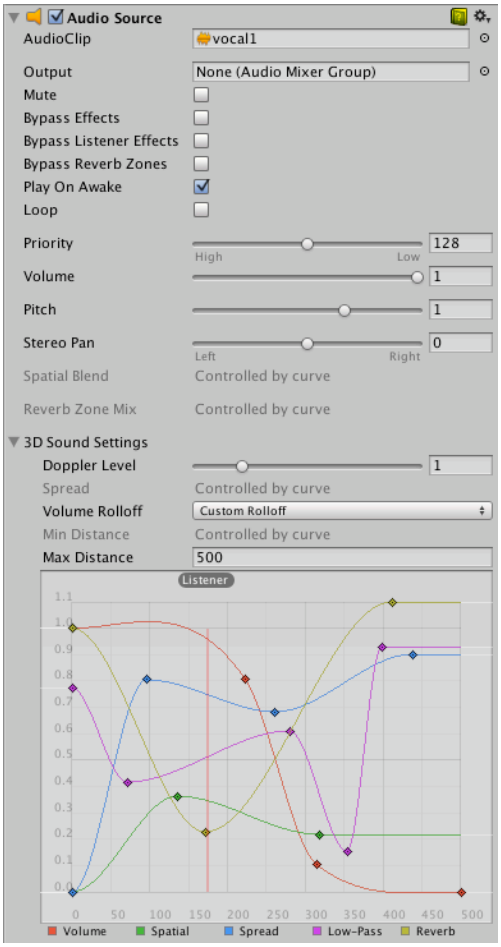
2. 編輯 *Component*

Component 的一個重要方面是靈活性。當你將 Component 附加到 GameObject 時，Compoennt 中有不同的值或屬性，可以在構建遊戲時於編輯器中進行調整，或者在運行遊戲時透過程式進行調整。屬性有兩種主要種類：值（Value) 和參考（Reference）。

如下圖所示。它是一個空的 GameObject 帶有 Audio Source Component。在 Inspector 視窗中的 Audio Source 的所有的值，都是預設值。



這個 Component 包含 Reference 屬性和 Value 屬性。Audio Clip 是 Reference 屬性。當此 Audio Source 開始播放時，將嘗試播放 Audio Clip 屬性中所參考到的聲音檔案。 如果沒有參考，則會發生錯誤，因為沒有聲音可以播放。你必須在 Inspector 視窗中參考該檔案。這個步驟很簡單，將聲音檔案從 Project 視窗拖到該 Reference 屬性，或是使用物件選擇器（Object Selector）。

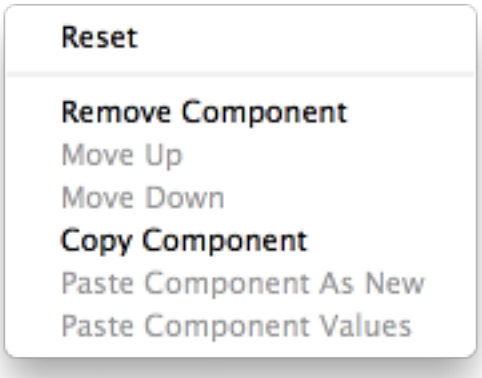


Component 可以包含對任何其它類型的 Component、GameObject 或資源的參考。

Audio Clip 以外的其餘屬性都是 Value 屬性。這些可以在 Inspector 視窗中直接調整。這些 Value 屬性有切換（Toggle）、數值、下拉選單，Value 屬性也可以是文字字串、顏色、曲線和其它類型。

3. *Component* 環境選單指令

Component 的環境選單（Context Menu）有許多有用的指令。



Inspector 視窗的 Component 面板右上角的「齒輪」圖標也可以獲得相同的指令。

- Reset：可恢復 Component 屬性到最近的編輯期之前所具有的值。
- Remove：可用於移除不再需要連接到 GameObject 的 Component。請注意，有一些相互依賴的 Component（例如，Hing Joint 僅在 Rigidbody 被賦予時才起作用）；如果嘗試移除其它相依賴的 Component，將彈出警告消息。
- Move Up / Down：使用上移（Move Up）和下移（Move Down）指令來重新排列 Inspector 視窗中 GameObject 的 Component 順序。
- Copy / Paste：Copy Component 指令儲存 Component 的類型和當前屬性設置。然後可以使用 Paste Component Values 將這些貼到相同類型的另一個 Component。還可以通過使用 Paste Component As New 將物件複製的值創建為新的 Component。

4. 測試屬性

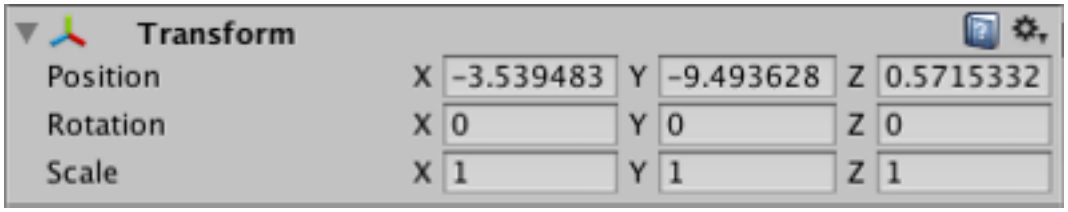
當遊戲處於播放模式時，可以隨意更改任何 GameObject 在 Inspector 視窗中的屬性。例如，你可能想要嘗試不同的跳躍高度。

如果你在程式腳本中創建 Jump Height 屬性，可以在進入播放模式時，更改其值，然後按跳躍按鈕查看會發生什麼。然後，不要退出播放模式，可以再次更改它，並在幾秒鐘內查看結果。當退出播放模式時，屬性將恢復為其播放模式前的值，因此你不會丟失任何東西。

這個工作流程提供了測試、調整和改善遊戲的強大能力，而無需重複循環的投入大量時間。

XIV.轉換組件 Transform 簡介

Transform 組件決定場景中每個物件的位置，旋轉和縮放。每個 GameObject 都有一個 Transform。



- 屬性：
  - Position：X、Y 和 Z 座標變換的位置。
  - Rotation：圍繞 X、Y 和 Z 軸變換的旋轉，以角度為單位。
  - Scale：沿著 X、Y 和 Z 軸變換的比例大小。值 1 是原始的大小（物件被匯入時的大小）。

Transform 的位置、旋轉和縮放值相對於父項 Transform 來測量。 如果 Transform 沒有父項，則其屬性是使用世界空間來測量的。

XV.編寫腳本建立組件

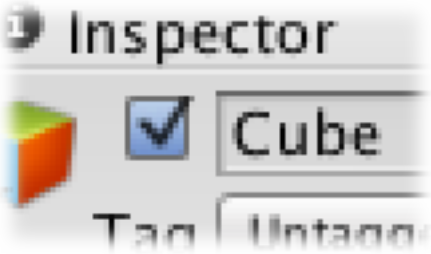
編寫腳本（或創建 Script）是使用 Unity Scripting API 在程式碼中編寫自己附加到 Unity 編輯器的功能。

當你創建一個 Script 並將其附加到 GameObject 時，該 Script 就像內建 Component 一樣顯示在 GameObject 的 Inspector 視窗中。這是因為當你將它們儲存在你的專案中時，Script 會成為 Component。

在技術術語中，將使編譯的任何 Script 做為 Component 的一種類型，因此 Unity 編輯器會將 Script 視為內建的 Component。你定義要在 Inspector 視窗中顯示的 Script 成員，而編輯器將執行你所編寫的任何功能。

XVI.解除遊戲物件

GameObject 可以透過標記為非活動狀態來暫時從場景中移除。這可以在程式中使用 activeSelf 屬性或者在 Inspector 視窗中使用啟用勾選框來達成。



- 停用 GameObject 父項的影響

  當父物件被停用時，也將覆蓋所以子物件上的 activeSelf 設置而將其全部停用，因此從該父項向下的整個層次結構都將被停用。要注意的是，這不會更改子物件上的 activeSelf 屬性的值，因此，一旦父物件重新啟用，它們將返回原本的狀態。這意味著，你無法透過讀取其 activeSelf 屬性來確定子物件目前所處的活動狀態。取而代之的，應該使用 activeInHierarchy 屬性，該屬性會以父項為首要。

  Unity 4.0 中引入了這種壓倒性的行為。在早期版本中，有一個名為 SetActiveRecursively 的功能可用於啟用或停用給定父物件的子項。然而，這個功能的作用是不一樣的，因為每個子物件的啟用設置都被改變了-整個層次結構都會被關閉，但是子物件沒有辦法「記住」原來的狀態。為了避免破壞舊程式碼，SetActiveRecursively 為了 4.0 而保留在 API 中，但是不推薦使用，而將來也可能會移除。在你真正希望改變子項的 activeSelf 設置的特殊情況下，可以使用類似以下程式碼：

```
void DeactivateChildren(GameObject g, bool a) {
    g.SetActive (a);
    foreach (Transform child in g.transform) {
        DeactivateChildren(child.gameObject, a);
    }
}
```

XVII.遊戲物件標籤

Tag 是可以分配給一個或多個 GameObject 的參考字。例如，你可以為玩家控制的角色定義「Player」標籤，並為非玩家控制的角色定義「Enemy」標籤。 可以使用「Collectable」標籤來定義玩家可以在場景中收集的道具。

Tag 可以為你程式編寫的目的來幫忙識別 GameObject。它們保障你不需要使用拖放的方式手動將 GameObject 添加到 Script 的公開屬性，從而在多個 GameObject 中使用相同的 Script 時節省時間。

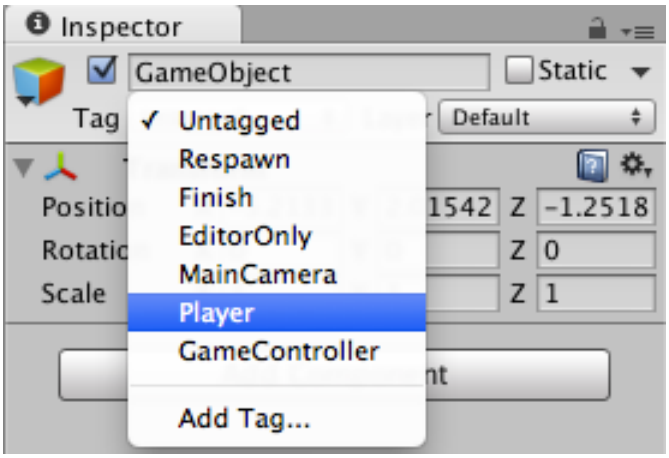
Tag 有用於 Collider 控制程式中的觸發器；它們需要確定玩家是否與敵人、道具或可收藏品進行互動。

可以使用 GameObject.FindWithTag() 功能透過設置 Tag 來查找包含你所想要的 Tag 的任何物件。以下示範使用 GameObject.FindWithTag()。它將 respawnPrefab 實例化在具有標籤「Respawn」的 GameObject 的位置：

```
using UnityEngine;

public class Example : MonoBehaviour {

    public GameObject respawnPrefab;
    public GameObject respawn;

    void Start() {

        if (respawn == null)
            respawn = GameObject.FindWithTag("Respawn");

        Instantiate(respawnPrefab,
            respawn.transform.position, respawn.transform.rotation);
    }
}
```

1. 建立新 *Tag*

Inspector 視窗在 GameObject 名稱的下方顯示 Tag 和 Layer 下拉選單。



要創建新的 Tag，請選擇 Add Tag...。 這將在 Inspector 視窗中打開 Tag 和 Layer 的管理器。 請注意，Tag 一旦被命名，它將不能再重新命名。

Layer 與 Tag 類似，但用於定義 Unity 如何在場景中呈現 GameObject。

2. 套用 *Tag*

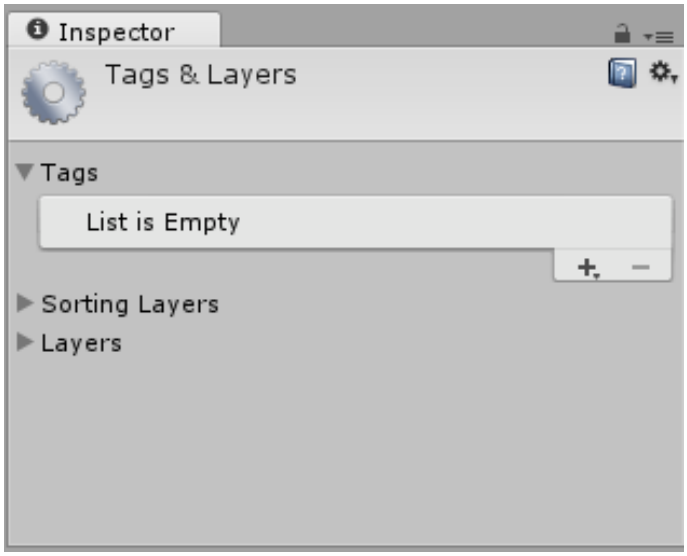
Inspector 視窗在任何 GameObject 名稱的下方顯示 Tag 和 Layer 下拉選單，要將現有 Tag 套用於 GameObject，請打開 Tag 下拉菜單，然後選擇要套用的標籤。 GameObject 就會與此 Tag 相關聯。

- 每個 GameObject 只能配置一個 Tag。
- Unity 包含一些內建的 Tag，不會出現在 Tag 管理器中。
  - Untagged
  - Respawn
  - Finish
  - EditorOnly
  - MainCamera
  - Player
  - GameController

- 可以使用任何你喜歡的字詞做為 Tag。甚至可以使用短語，但這樣的話，你可能需要拉寬 Inspector 視窗才能查看到 Tag 的全名。

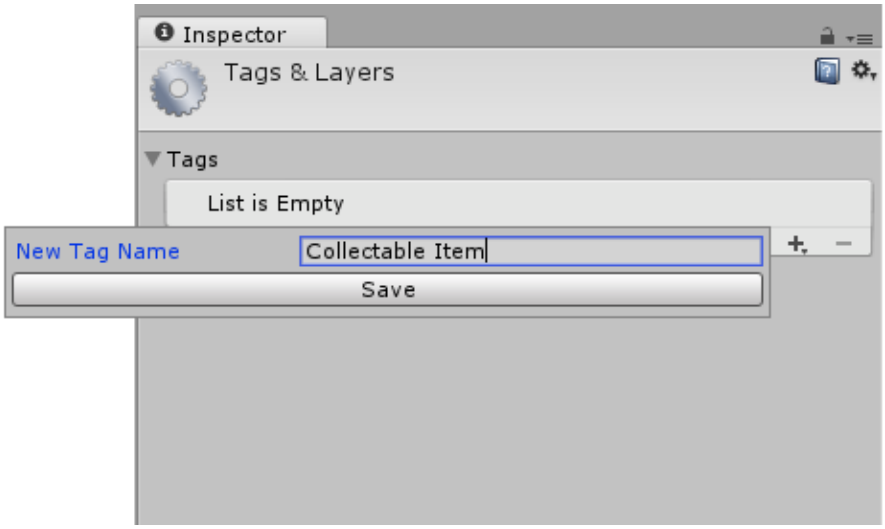
3. *Tags and Layers*

Tags and Layers 管理器允許你設置 Tags、Sorting Layers 和 Layers。要查看 Tages and Layers 管理器的話，使用編輯器選單的 Edit > Project Settings > Tags and Layers。

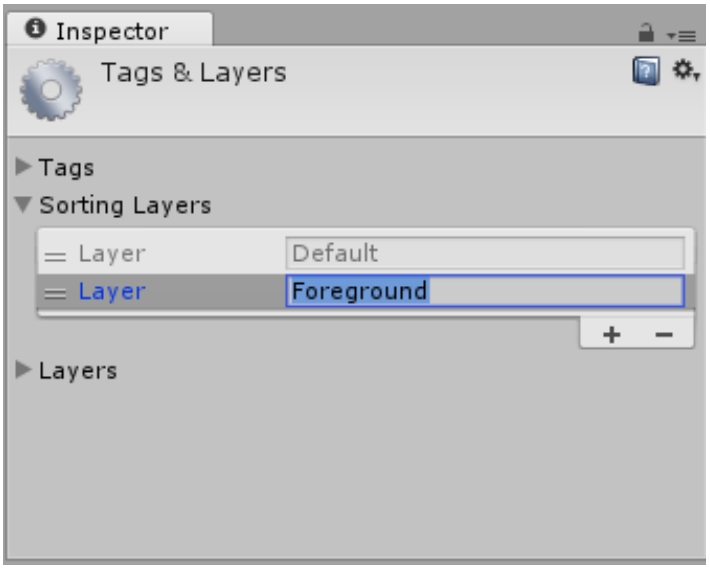


- Tags

  可用於標識專案中物件的標記值。要添加新的 Tag，請點擊列表右下角的加號按鈕（+），並為新 Tag 命名。

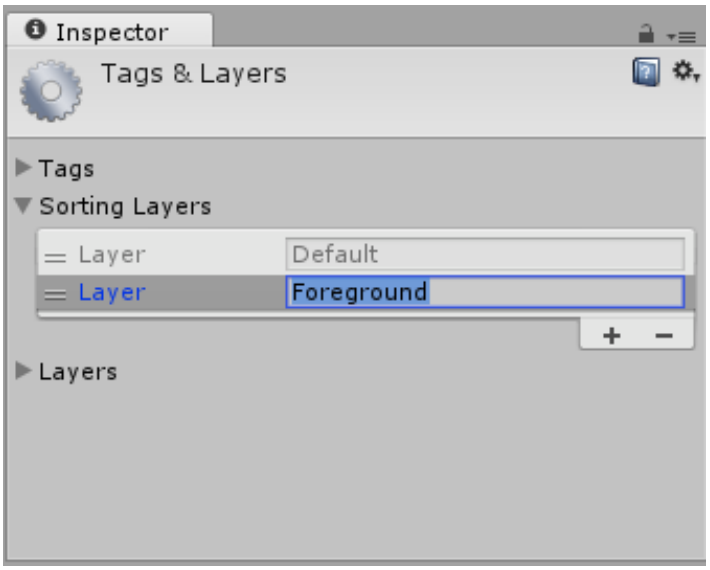


請注意，一旦命名了 Tag，就無法重新命名。要刪除 Tag，請點擊選取它，然後點擊列表右下角的減號（ - ）按鈕。

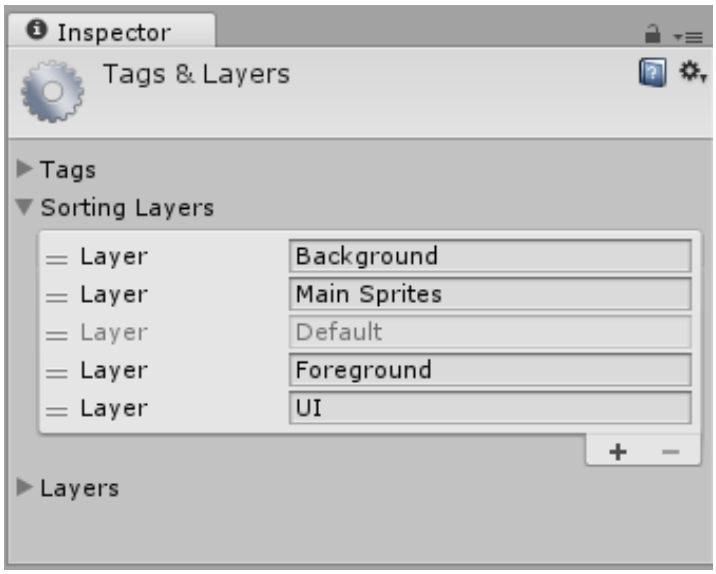


- Sorting Layers

  與 2D 系統中的 Sprite 圖像結合使用時，「Sorting」是指不同 Sprite 的重疊順序。

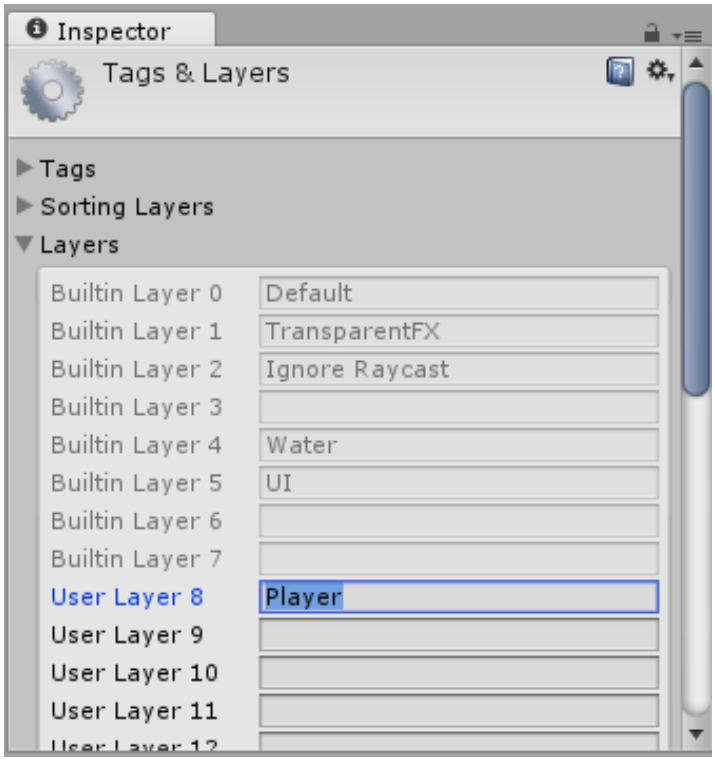


  要添加和刪除 Sorting Layers，請使用列表右下方的加號和減號（+/-）按鈕。要更改其順序，請拖動每個 Layer 項目左側的拉柄。



- Layers

  在整個 Unity 編輯器中使用此方法來創建共享特定特徵的物件群組。User Layer 主要限制諸如射線處理或渲染處理之類的操作，以使它們僅適用於相關物件群組。 在 Tags and Layers 管理器中，前 8 個內建 Layer 是 Unity 預設使用的，因此無法編輯它們。但是，您可以從 8 到 31 個自定義 User Layer。



要從 8 到 31 個自定義 User Layer；可在每個你要使用的文字欄位中輸入自定義名稱。請注意，你無法添加 Layers 數量，但與 Tags 不同，可以重新命名 Layer。

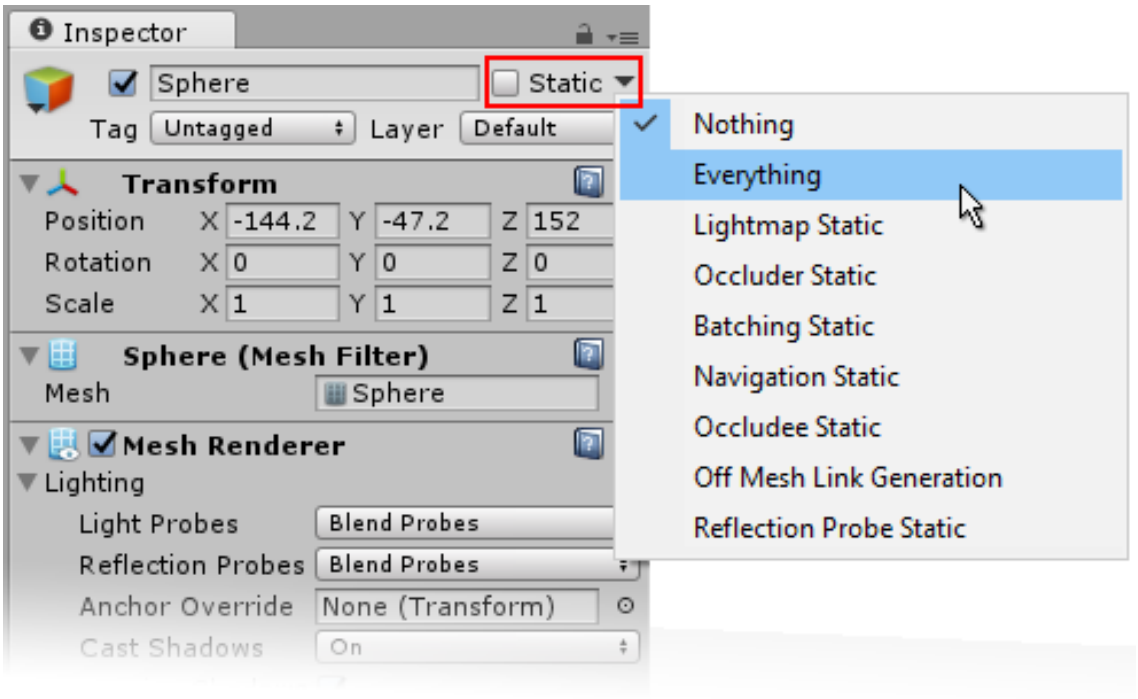
XVIII.靜態遊戲物件

許多最佳化需要知道物件是否能夠在遊戲進行過程中移動。

關於靜態（即不移動）物件的訊息，在知道它不會被物件位置變化而作廢，通常可以在編輯器中預先計算。

例如，可以通過將幾個靜態對象合併成稱為批次處理的單一個大物件來最佳化渲染。

GameObject 的 Inspector 視窗右上角有一個 Static 勾選框和選單，用於通知 Unity 中各種不同的系統，讓其知道該物件不會移動。可以將物件分別標記為每個這些系統的靜態物件，因此當沒有獲得好處時，可以選擇不計算物件的靜態最佳化。



Everything 和 Nothing 會同時啟用或禁用所有使用它的系統的靜態狀態。

使用到靜態物件的系統有：

- Lightmapping：提供場景進階的照明。

- Occluder 和 Occludee：基於來自特定攝影機位置的物件可視性進行渲染最佳化。
- Batching：將多個物件組合成一個較大物件的渲染最佳化。
- Navigation：啟動角色去判斷場景障礙物的系統。
- Off-mesh Links：透過 Navigation 系統在場景的不連續區域之間建立連接。
- Reflection Probe：在各個方向捕捉其周圍環境的球形視野。

XIX.預製元件 Prefab

通過添加組件並將其屬性設置為適當的值，可以方便地在場景中構建 GameObject。然而，如果你有像 NPC、道具或一片景色這樣在場景中多次重複使用的物件，這可能會產生問題。

簡單地複製物件肯定會產生副本，但它們都會是獨立可編輯的。通常，你想要的是特定物件的所有實例具有相同的屬性，因此，當你編輯場景中的一個物件時，你不必對所有副本重複進行相同的編輯。

幸運的是，Unity 有一個 Prefab 資源類型，可以讓你儲存具備完整組件和屬性的 GameObject 物件。

Prefab 可做為讓你能夠在場景中建立新的物件實例的模板。對 Prefab 資源進行任何編輯都會立即反映到從其生成的所有實例中，但也可以單獨覆蓋每個實例的組件和設置。

注意：當您將資源檔案（例如 Mesh）拖動到場景中時，它將創建一個新的物件實例，並且所有這些實例都會在原始資源被更改時，也跟隨更改。 然而，雖然它的行為表面相似，但資源不是 Prefab，所以將無法添加組件或使用以下描述的其它 Prefab 的功能。

1. 使用 *Prefab*

你可以透過選擇 Asset > Create Prefab 來創建 Prefab，然後將物件從場景拖動到剛出現的「空」Prefab 資源上。

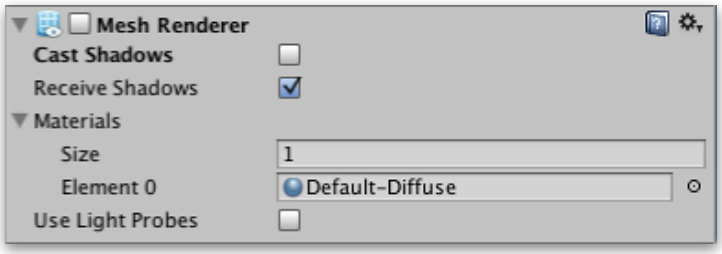
如果之後將不同的 GameObject 拖到 Prefab 上，將會被詢問是否要使用新的 GameObject 替換掉目前的內容。

簡單的將 Prefab 資源從 Project 視窗拖動到 Scene 視窗，然後就會建立 Prefab 的實例。

做為 Prefab 實例所建立的物件將以藍色文字顯示在 Hierarchy 視窗中（一般物件使用黑色文字顯示）。

如上所述，Prefab 資源本身的變更將反映在所有實例中，但你也可以個別的修改單獨的實例。這是很有用的，比方說，當你想要創建幾個類似的 NPC，但想要引進一些變化以使它們更現實。

當一個屬性值被覆蓋時，它會將其名稱標籤在 Inspector 視窗中顯示為粗體（當一個完全新的組件被添加到 Prefab 實例中時，它的所有屬性將以粗體顯示）。



你還可以從程式運行時創建 Prefab 實例。

2. 從實例編輯 *Prefab*

Prefab 實例的 Inspector 視窗具有不存在於一般物件的三個按鈕：Select、Revert 和 Apply。

Select 按鈕會在 Project 視窗選擇出該實例來源的 Prefab 資源。這讓你能夠編輯主要的 Prefab，從而更改其所有的實例。

然而，你也可以使用 Apply 按鈕將實例中的覆蓋值儲存回原始的 Prefab（由於顯而易見的原因，被修改的 Transform 位置和旋轉值會被排除）。 這有效地讓你能夠通過任何單獨的實例來編輯所有實例（覆蓋值已被改變的除外），並且是進行全體變更非常快捷方便的方法。

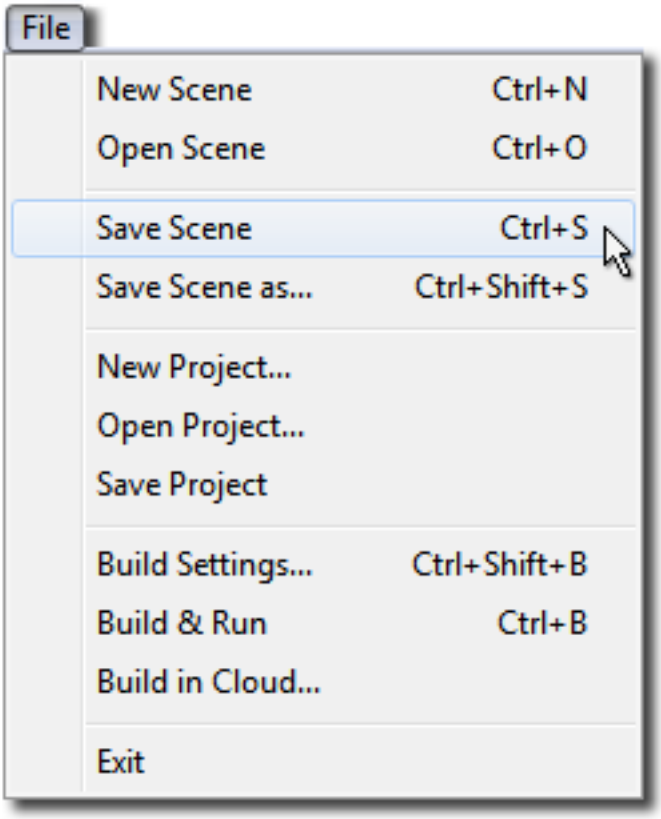
如果你嘗試覆蓋屬性，但是之後決定使用預設值，則可以使用 Revert 按鈕將該實例還原 Prefab 原本的屬性值。

XX.各種儲存工作

Unity 儲存關於你專案的許多各種類型的訊息，其中有些以不同的方式儲存。這意味著你的工作被儲存時，取決於你正在做什麼樣的更改。

當然，這裡要特別建議你定期儲存，並使用版本控制系統（VCS）來保留對工作的增量更改，並讓你能夠去試探並回滾更改，而不會對你的工作造成任何損失。

1. 儲存目前場景的更改

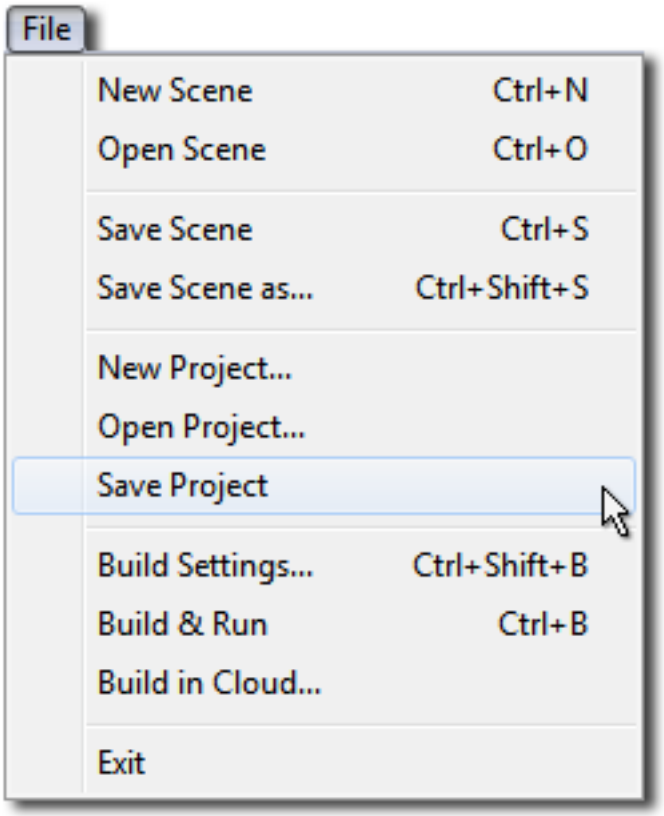


場景更改包括對 Hierarchy 視窗中的任何物件的修改（例如，添加、移動或刪除遊戲物件），以及更改 Inspector 視窗中層次化遊戲對象的參數。

要儲存對場景的更改，請從 File 選單中選擇 Save Scene，或者按 Ctrl / Cmd + S。這樣可以將目前的變更儲存到場景並執行 Save Project。

這意味著，當你執行 Save Scene，所有的東西都被儲存了。

2. 儲存專案範圍的變化



你可以在 Unity 中進行的一些在專案範圍內非特定場景的變更。這些設置可以通過 File 選單中的 Save Project 有別於場景變更來儲存。

使用 Save Project 不會儲存對場景的變更，只會保存專案範圍的變更。 例如，如果你使用臨時場景對 Prefab 進行某些變更，則可能希望儲存專案，而不希望儲存被變更的場景。

使用 Save Project 時，儲存的專案範圍變更包含：

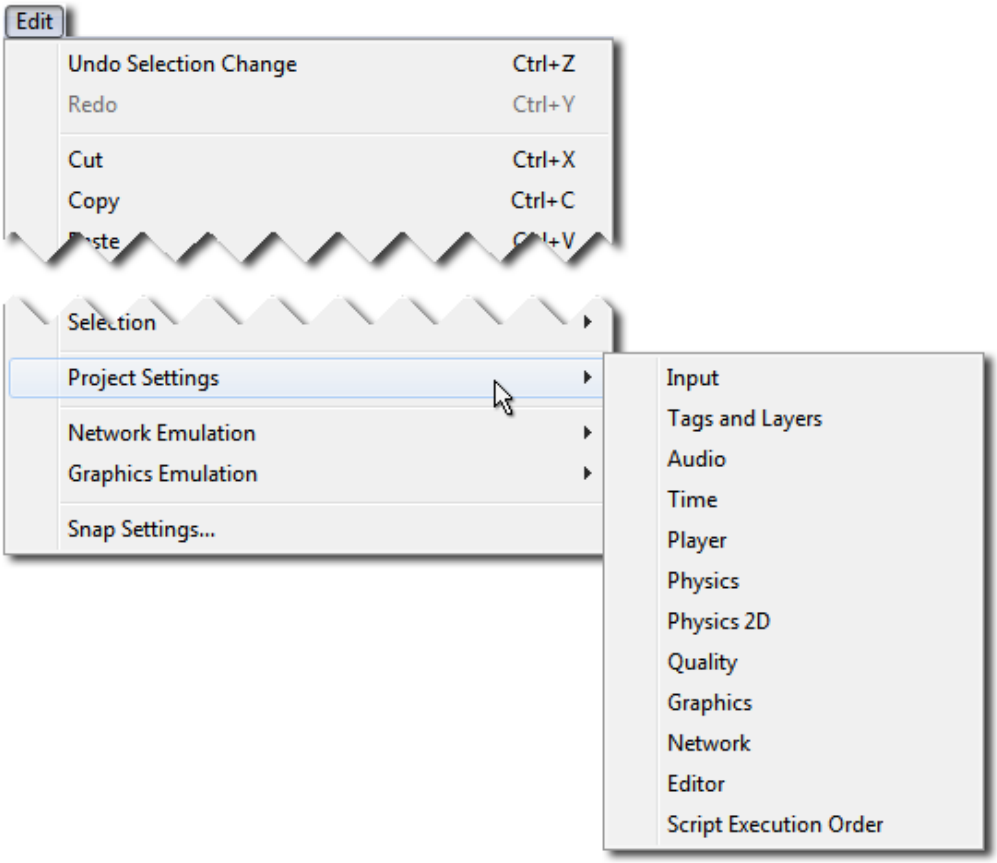
- 所有的專案設置（Project Settings）：

  執行 Save Project 時，將儲存選單裡每個 Project Settings 的所有設置，如自定義輸入軸、使用者定義的 Tag 或 Layer 以及物理重力強度。

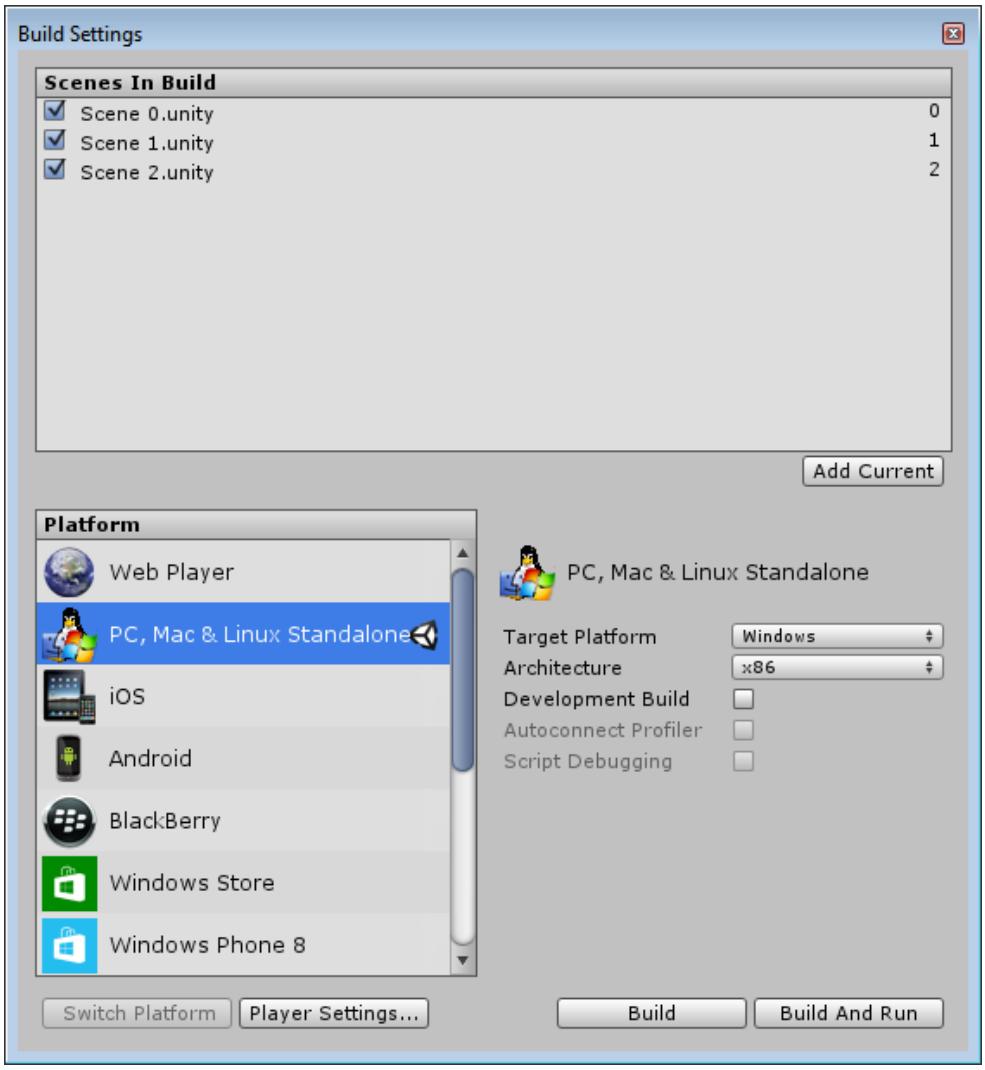
  當儲存專案時，這些設置的變更將儲存到 ProjectSettings 資料夾中：

  - Input：儲存為 InputManager.asset。

- Tags And Layers：儲存為 TagManager.asset。
- Audio：儲存為 AudioManager.asset。
- Time：儲存為 TimeManager.asset。
- Player：儲存為 ProjectSettings.asset。
- Physics：儲存為 DynamicsManager.asset。
- Physics 2D：儲存為 Physics2DSettings.asset。
- Quality：儲存為 QualitySettings.asset。
- Graphics：儲存為 GraphicsSettings.asset。
- Network：儲存為 NetworkManager.asset。
- Editor：儲存為 EditorUserSettings.asset。



- Build Settings 的內容：
  Build Settings 也在 ProjectSettings 資料夾中儲存為 EditorBuildSettings.asset。



- 在 Project 視窗中所變更的資源：
  伴隨專案範圍設置一起儲存的還有不具有 Apply 按鈕的資源的變更，例如變更任何以下的內容：
  - 材質（Material）的參數。
  - Prefab。
  - Animator Controller（狀態機）。
  - Avatar Mask。
  - 任何其它沒有 Apply 按鈕的資源。

3. 立即寫入磁碟的變更（不需要儲存）

有一些類型的變更會立即寫入磁碟，而不需要執行「儲存」的操作。這些包括以下內容：

- 變更任何匯入設置會要求使用者按 Apply 按鈕：

  大多數資源類型的匯入設置會要求按 Apply 按鈕使變更生效。這將影響到資源重新匯入時可以符合新的設置。所以當按下 Apply 按鈕時，這些變更將立即儲存。例如：

  - 變更圖像資源的紋理類型。
  - 改變 3D 模型資源的比例因子（Scale Factor）。
  - 變更聲音資源的壓縮設置。
  - 任何其它具有 Apply 按鈕的匯入設置。

- 其它立即儲存的變更：

  一些其它類型的資料會立即或自動儲存到磁碟，而無需執行「儲存」操作：

  - 創建新資源，例如：新材質或 Prefab（但不是後續更改資源）。
  - 烘焙照明資料（烘焙完成時儲存）。
  - 烘焙導航資料（烘烤完成時儲存）。
  - 烘焙遮擋剔除資料（烘烤完成時儲存）。
  - Script 執行順序變更（按下 Apply 後，此資料儲存在每個 Script 的 .meta 檔案中）。

XXI.執行期使用 Prefab（一）

到目前為止，你應該已從根本上了解 Prefab 的概念了。它們是你在遊戲中可重複使用的預先定義 GameObject 與 Component 的集合。

當你要在執行期實例化複雜的 GameObject 時，使用 Prefab 會非常方便。實例化 Prefab 的替代方法是使用程式碼從頭開始創建 GameObject。實例化 Prefab 與其替代方法相比具有許多優點：

- 你可以用一行程式碼就實例化一個 Prefab 並具有完整的功能。用程式碼創建相同的 GameObject 平均需要五行程式碼，但可能會更多。
- 您可以在 Scene 和 Inspector 視窗中快速方便地設置、測試和修改 Prefab。
- 您可以更改要被實例化的 Prefab，而無需更改實例化的程式碼。一個簡易的火箭可能會變成超級火箭，而不需要更改程式碼。

1. 常見情況

在以下這些例子的情況，Prefab 能夠派上用場：

- 透過在不同的位置創建多次來產生整體，例如使用單一個「磚塊」Prefab 建立一面牆壁。
- 火箭發射器在發射時實例化飛出的火箭 Prefab。Prefab 包含一個網格（Mesh）、Rigidbody、Collider 和一個子項 GameObject 與它自己的軌跡拖尾線（Trail）粒子系統。
- 機器人爆炸成許多碎件。完整的、可操作的機器人被破壞，使用毀壞的機器人 Prefab 替換。這個 Prefab 將由分成許多部分的機器人組成，全部都設置了自己的 Rigidbody 和粒子系統。這種技術使你能夠將機器人炸成許多塊，只需一行程式碼，使用 Prefab 替換掉原物件。

2. 建立牆壁

在此將說明使用 Prefab 與從程式碼建立物件的好處。

首先，讓我們從程式碼建立磚牆：

```
public class Instantiation : MonoBehaviour {

    void Start() {

        for (int y = 0; y < 5; y++) {

            for (int x = 0; x < 5; x++) {

                GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
                cube.AddComponent<Rigidbody>();

                cube.transform.position = new Vector3(x, y, 0);
            }
        }
    }
}
```

(1) 使用 GameObject > Create Empty 建立空的 GameObject。

(2) 使用上面的程式碼儲存為一個 Script 檔案，並將此 Script 檔案拖拉到剛剛建立的空的 GameObject。

如果執行該程式碼，將會看到當你進入播放模式時，會看到建立了整個磚牆。CreatePrimitive 和 AddComponent 那兩行是與每個單獨的磚相關的功能。現在還不錯，但是我們的每一塊磚都沒有紋理。 要在磚上執行的每一個額外的動作，如改變紋理、摩擦力或剛體質量，都要額外再寫幾行。

如果你創建了一個 Prefab 並且手動執行所有的設置，則只要使用一行程式碼來執行每個磚塊的創建和設置。當你決定要進行更改時，這可以減輕維護和更改大量程式碼。使用 Prefab，只需對其進行更改即可執行。無需更改程式碼。

如果你為每個磚塊使用 Prefab，這是創建牆壁所需的程式碼。

```
public class Instantiation : MonoBehaviour {

    public Transform brick;

    void Start() {

        for (int y = 0; y < 5; y++) {

            for (int x = 0; x < 5; x++) {

                Instantiate(brick, new Vector3(x, y, 0), Quaternion.identity);
            }
        }
    }
}
```

這不僅非常整潔，而且非常能夠重複使用。沒有地方提到我們正在實例化一個立方體或者必須包含一個剛體。所有的這些都在 Prefab 中定義並且能夠在編輯器中快速創建。

現在我們只需要在編輯器中創建 Prefab。就是這樣：

- 在選單列選擇 GameObject > 3D Object > Cube。
- 在選單列選擇 Component > Physics > Rigidbody。
- 在選單列選擇 Assets > Create > Prefab。
- 在 Project 視窗中，將新的 Prefab 的名稱更改為 Brick。
- 將 Hierarchy 視窗中創建的 Cube 拖放到 Project 視窗中的 Brick Prefab 上。
- 使用 Prefab 創建，你可以從 Hierarchy 視窗中安然地刪除 Cube（在 Windows 上 Delete 鍵，Mac 上的 Command+delete）。

我們已經創建了 Brick Prefab，所以現在必須將它附加到 Scrpit 中的 brick 變數中。當選擇包含 Script 的空的 GameObject 時，將可在 Inspector 視窗中看見 Brick 欄位。

現在將 Brick Prefab 從 Project 視窗拖動到 Inspector 視窗中的 Brick 欄位。按下播放，將可看到使用 Prefab 構建的牆壁。

這是一個可以在 Unity 中重複使用的工作流程模式。一開始你可能會驚異於為什麼這樣好多了，因為從程式碼創建立方體的 Script 只有2行多。

由於現在使用的是 Prefab，你可以在幾秒鐘內調整 Prefab。想改變所有這些實例的質量？只要在 Prefab 中的 Rigidbody 調整一次。想為所有實例使用不同的材質？只要將材質拖到 Prefab 上一次。想改變摩擦力？在 Prefab 的 Collider 中使用不同的物理材質一次。要將粒子系統添加到所有的這些盒子上？只要在 Prefab 裡添加一次子項。

XXII.在執行期使用 Prefab（二）

1. 實例化火箭和爆炸

這裡是 Prefab 如何適合這種情況：

(1) 當使用者射擊時，火箭發射器實例化一個火箭 Prefab。Prefab 包含網格、Rigidbody、Collider 以及包含拖尾軌跡粒子特效的子 GameObject。

(2) 火箭撞擊並實例化爆炸 Prefab。爆炸 Prefab 包含一個粒子系統、隨著時間推移而淡出的光線，以及對周圍 GameObject 造成傷害的 Script。

雖然可以完全從程式碼中構建一個火箭 GameObject，但是手動添加組件和設置屬性，實例化 Prefab 是更容易的。你可以用一行程式碼實例化火箭，無論火箭的 Prefab 有多複雜。 在實例化 Prefab 之後，還可以修改實例化物件的任何屬性（例如，可以設置火箭剛體的速度）。

除了更容易使用之外，可以稍後更新 Prefab。所以如果你正在建一個火箭，你不需要立即添加一個粒子軌跡，可以稍後再做。 一旦將 Trail 做為子 GameObject 添加到 Prefab，所有實例化的火箭將具有粒子軌跡。最後，可以快速調整火箭 Prefab 的屬性，使遊戲更容易微調。

```
public class LaunchRocket : MonoBehaviour {

    public Rigidbody rocket;
    public float speed = 10f;

    void FireRocket () {

        Rigidbody rocketClone = (Rigidbody) Instantiate(rocket,
            transform.position, transform.rotation);

        rocketClone.velocity = transform.forward * speed;
    }

    void Update () {

        if (Input.GetButtonDown("Fire1")) {

            FireRocket();
        }
    }
}
```

2. 使用破碎物件替代原始物件

假設你有一個完整裝束控制的敵人角色死亡。可以簡單地在角色上播放死亡動畫，並停用處理敵人邏輯的所有腳本。可能需要刪除多個腳本，添加一些自定義邏輯，以確保沒有人會繼續攻擊死掉的敵人，以及做其他的清理任務。

另一個更好的方法是立即刪除整個角色，並將其替換為被破壞的 Prefab 的實例。這就給了你很大的靈活性。你可以為死亡角色使用不同的材質，附加完全不同的腳本，產生一個 Prefab 檔案，包含破碎成許多部位的物件，以模擬一個粉碎的敵人。

這些都可以通過一次呼叫 Instantiate() 來實現，只需將其連接到正確的 Prefab，即可設置！

要記住的重要部分是，Instantiate() 的殘骸可以由與原始物體完全不同的物件製成。例如，如果你有一架飛機，你可以建立兩個版本。一個飛機由一個具有網格渲染器的 GameObject 和飛機物理腳本組成。將模型保持在一個 GameObject 中，遊戲將運行得更快，因為可以使用較少的三角形來形成模型，並且由於它比使用許多小部件的渲染速度更快，所以渲染的物件數量將會減少。而且，當你的飛機高興地飛來飛去時，沒有理由將它分開。

要建造一架失事的飛機 Prefab，典型的步驟是：

(1) 使用你熟悉的建模軟體建造飛機的許多部位。

(2) 創建一個空的場景。

(3) 將模型拖動到空的場景中。

(4) 選擇所有部件並選擇 Component-> Physics-> Rigidbody，將剛體添加到所有部件。

(5) 選擇所有部件並選擇 Component-> Physics-> Box Collider 將 Box Colliders 添加到所有部件。

(6) 要獲得額外的特殊效果，可以將每個零件添加一個煙霧粒子系統作為子 GameObject。

(7) 現在你有一架有多個爆炸部件的飛機，它們通過物理掉落到地面，並由於附著的粒子系統而造成一個粒子軌跡。點擊播放以預覽模型如何反應並進行任何必要的調整。

(8) 選擇 Assets->Create Prefab。

(9) 將包含所有飛機零件的 GameObject 拖到 Prefab 中。

以下示範顯示了這些步驟使用的程式碼。

```
public class Wreck : MonoBehaviour {

    public GameObject wreck;

    IEnumerator Start() {

        yield return new WaitForSeconds(3);

        KillSelf();
    }

    void KillSelf () {

        Instantiate(wreck, transform.position, transform.rotation);

        Destroy(gameObject);
    }

}
```

3. 使用特定模式放置一堆物件

如果你想把一堆物件放在一個網格或圓形圖案中。傳統上，可以通過以下兩種方式完成：

(1) 從程式碼完全構建一個物件。這很麻煩！從腳本輸入值是緩慢、不直覺、沒價值的麻煩。

(2) 製作完全裝備的物件，複製並放置在場景中。這是很乏味的，將物件精確地放置在網格中是很困難的。

所以使用 Instantiate() 與 Prefab 取代！你會了解為什麼 Prefab 在這些情況下非常有用。 以下是這些場景所需的程式碼：

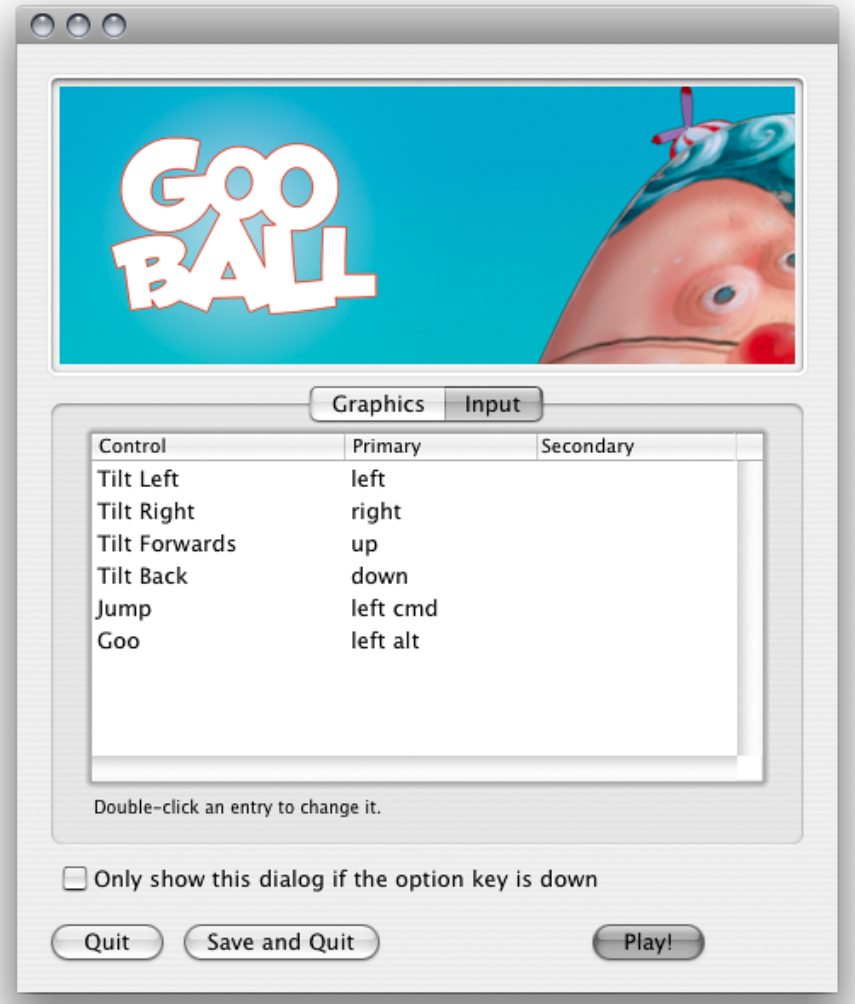
```
public class PlacingCircle : MonoBehaviour {

    public GameObject prefab;
    public int numberOfObjects = 20;
    public float radius = 5f;

    void Start() {

        for (int i = 0; i < numberOfObjects; i++) {

            float angle = i * Mathf.PI * 2 / numberOfObjects;
            Vector3 pos = new Vector3(Mathf.Cos(angle),
                0, Mathf.Sin(angle)) * radius;

            Instantiate(prefab, pos, Quaternion.identity);
        }
    }
}
```

```
public class PlacingGrid : MonoBehaviour {

    public GameObject prefab;
    public float gridX = 5f;
    public float gridY = 5f;
    public float spacing = 2f;

    void Start() {

        for (int y = 0; y < gridY; y++) {

            for (int x = 0; x < gridX; x++) {

                Vector3 pos = new Vector3(x, 0, y) * spacing;

                Instantiate(prefab, pos, Quaternion.identity);
            }
        }
    }
}
```

XXIII.傳統遊戲輸入控制

Unity 支援鍵盤、操縱桿和遊戲手把輸入。

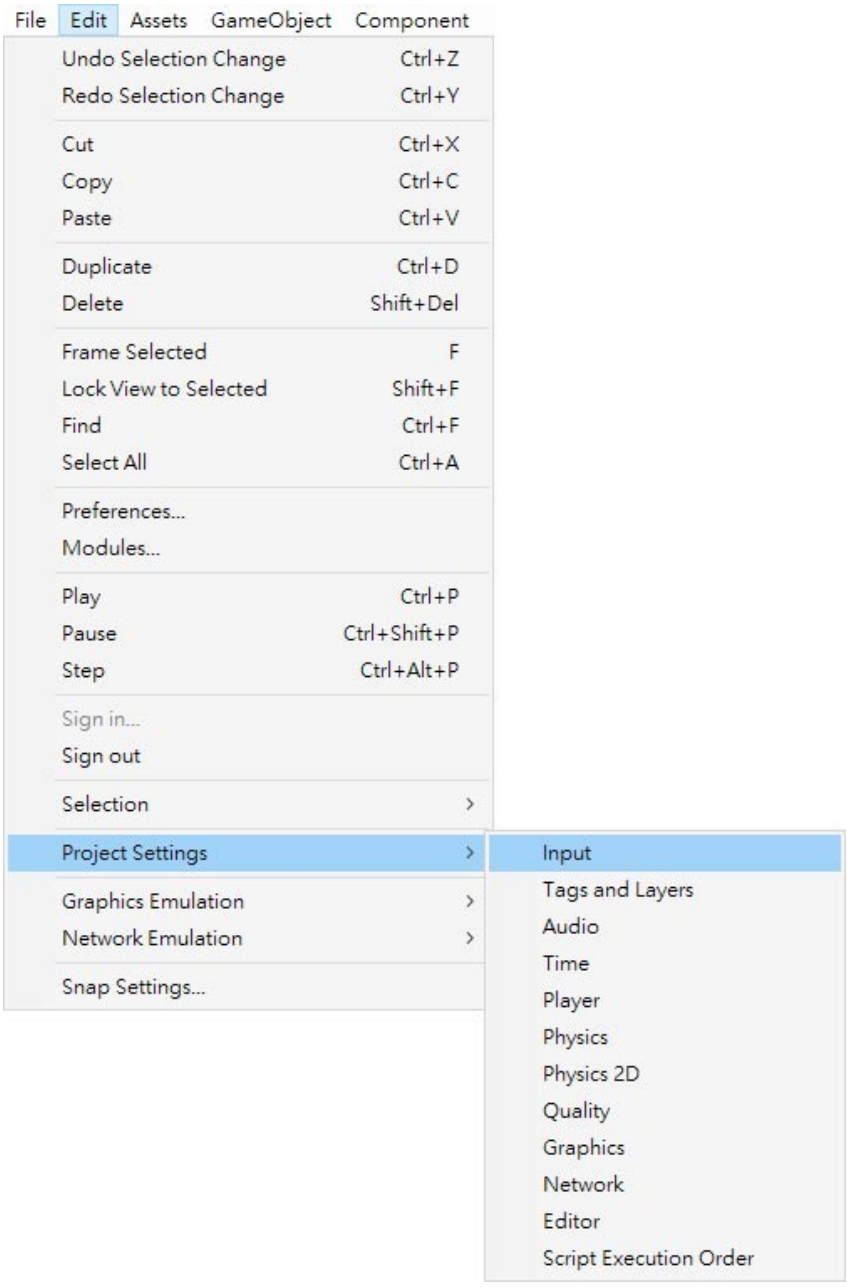
可以在輸入管理器（Input Manager）中創建虛擬軸（Virtual Axes）和按鈕，玩家可以在配置視窗中配置鍵盤輸入。



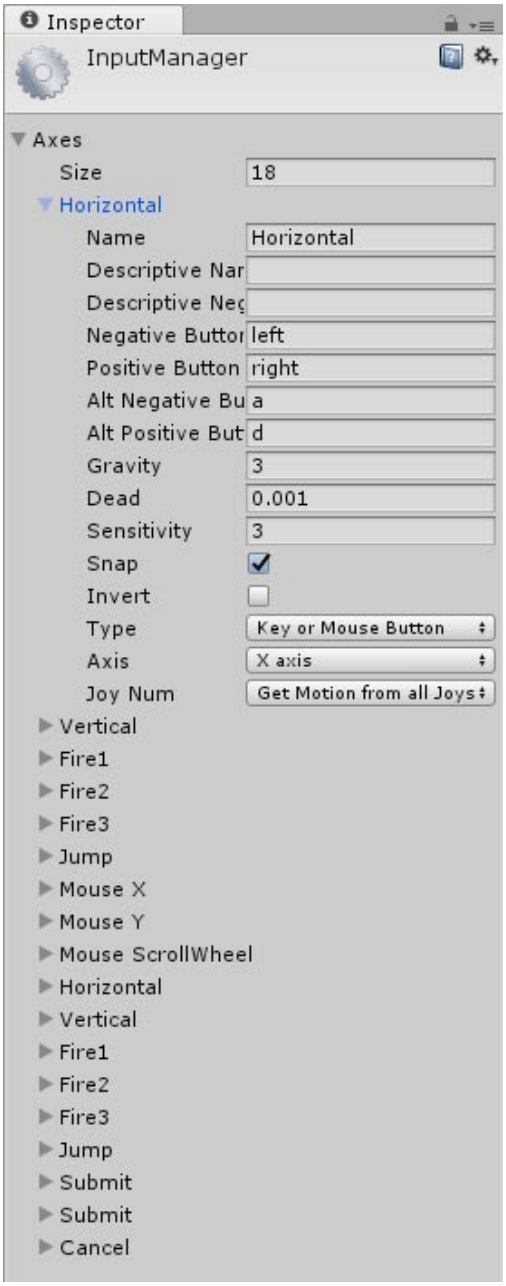
你可以設置操縱桿、遊戲手把、鍵盤和滑鼠，然後通過一個簡單的腳本界面訪問它們。通常使用軸和按鈕來偽裝遊戲機控制器。或者，你也可以訪問鍵盤上的按鍵。

1. *Virtual Axes*

- 在程式中，可透過 Virtual Axis（虛擬軸）的名稱來獲得使用者的輸入值，這些名稱可從選單列的 Edit > Project Settings > Input 開啟的 InputManager 獲得，並且可依據需求自行新增或修改。



- 在每個專案被建立時，都會伴隨著一些預設的 Virtual Axes：
  - Horizonal 和 Vertical 對應到 W、A、S、D 以及方向鍵。
  - Fire1、Fire2、Fire3 個別對應到 Control、Alt 和 Windows 按鍵。
  - Mouse X 和 Mouse Y 對應到滑鼠移動的增量。
- 可使用相同的名稱定義多個 Virtual Axis，運行時將會回傳其中最大的絕對值，因此，可以為鍵盤、搖桿等輸入設備使用多個相同名稱來設置，使得程式碼裡不用考慮到使用者的輸入來源。



- 每個 Axis 可使用名稱透過程式碼獲取 -1 到 1 之間的輸入值，中間值為 0。

```
void Update(){
    float h = Input.GetAxis ("Horizontal");
    this.slider.value = h;
}
```

- 每個 Axis 可使用名稱透過程式碼得知是否該按鍵發生作用。

```
void Update(){
    if(Input.GetButtonDown ("Fire1")){
        Debug.Log ("Fire");
    }
}
```

- Axis 設置的按鍵名稱可依據以下規則，這些名稱也可用於程式碼中：
  - 一般鍵：a、b、c ...。
  - 數字鍵：1、2、3 ...。
  - 方向鍵：up、down、left、right。
  - 小鍵盤：[1]、[2]、[3]、[+]、[equals]。
  - 修飾鍵：right shift、left shift、right ctrl、left ctrl、right alt、left alt、right cmd、left cmd。
  - 滑鼠按鍵：mouse 0 、mouse 1、mouse 2 ...。
  - 搖桿鍵（任何來源）：joystick button 0、joystick button 1、joystick button 2 ...。
  - 搖桿鍵（指定來源）：joystick 1 button 0、joystick 1 button 1、joystick 2 button 0、Joystick 2 button 1 ...。
  - 特殊鍵：backspace、tab、return、escape、space、enter、insert、home、end、page up、page down。
  - 功能鍵：f1、f2、f3 ... 。

2. *Key Code*

- 在 Virtual Axes 可設置的按鍵，在程式碼中都有對應的 KeyCode 列舉。
- 使用 Input.GetKey、Input.GetKeyUp 和 Input.GetKeyDown，可透過 KeyCode 或按鍵名稱來獲得指定按鍵的輸入狀態。

```
void Update(){
    if(Input.GetKey (KeyCode.Space)){
        Debug.Log ("Hold down the space key.");
    }
    if(Input.GetKeyDown (KeyCode.Space)){
        Debug.Log ("Pressing down the space key.");
    }
    if(Input.GetKeyUp (KeyCode.Space)){
        Debug.Log ("Release the space key.");
    }
}
```

XXIV.行動裝置觸控輸入

在行動設備上，Input Class 提供對觸摸螢幕、加速度器等的訪問。

1. 多點觸碰螢幕

iPhone 和 iPod Touch 設備能夠追蹤多達五個手指同時觸摸屏幕。你可以通過訪問 Input.touches 屬性陣列來檢索在最後一幀期間觸摸螢幕的每個手指的狀態。

Android 設備對追踪的手指數量沒有統一的限制。反而，因設備之間的不同，會是在較舊設備上的兩點觸摸到一些較新設備上的五個手指。

每個手指觸碰經由 Input.Touch 資料結構表示：

- finger：觸碰的唯一索引。
- position：觸碰的螢幕位置。
- deltaPosition：自上一幀起，所改變的螢幕位置。
- deltaTime：自上一次狀態改變以來，所經過的時間。
- tapCount：iPhone / iPad 螢幕能夠區分使用者的快速手指敲擊。該計數器讓你知道使用者在不移動手指兩側的情況下輕拍螢幕的次數。Android 設備不計算敲擊數，此欄位始終為 1。
- phase：描述所謂的「階段」或觸碰的狀態。它可以幫助你確定觸碰是剛剛開始、使用者是否移動手指或者只是抬起手指。

  phase 有以下狀態：

  - Began：手指剛觸碰螢幕。
  - Moved：手指在螢幕移動。
  - Stationary：手指觸摸螢幕，但自從上一幀起就沒有移動。
  - Ended：手指從螢幕上抬起。這是觸碰的最後階段。
  - Canceled：系統取消追蹤觸碰，例如用戶將設備放置在臉部或同時發生超過五次觸碰時。這是觸碰的最後階段。

以下是個示範腳本，當使用者點擊螢幕時，它將射出一個射線：

```
public class TouchToHit : MonoBehaviour {

    public GameObject particle;

    private RaycastHit hit;

    void Update () {

        foreach(Touch touch in Input.touches){

            if(touch.phase == TouchPhase.Began){

                Ray ray = Camera.main.ScreenPointToRay (touch.position);

                if(Physics.Raycast (ray , out hit)){

                    Instantiate (particle , hit.point , Quaternion.identity);
                }
            }
        }

    }
}
```

2. 滑鼠模擬

在原生觸碰支援上，Unity iOS / Android 提供鼠標模擬。你可以使用標準 Input Class 的滑鼠功能。不過請注意，iOS / Android 設備旨在支援多種手指觸摸。使用滑鼠功能只支援單一手指觸碰。而且，行動設備上的手指觸碰可以從一個區域移動到另一個區域，而不是在它們之間游動。行動設備上的滑鼠模擬將提供移動，因此與觸碰輸入相比是非常不同的。建議是在早期開發過程中使用滑鼠模擬，在之後盡可能使用觸碰輸入。

3. 加速度傳感器

隨著行動設備的移動，內建的加速度器會在三維空間中彙報沿三個主軸的線性加速度變化。每個軸的加速度直接由硬體直接彙報為 G 力值。1.0 的值表示沿著特定軸約 +1g 的負荷，而 -1.0 的值表示 -1g。如果你保持設備直立（Home 鍵在底部），X軸的正值在右邊，Y軸的正值在上面，Z軸正值指向你。

以下是使用加速度器移動物件的示範腳本：

```
public class AccelerationExample : MonoBehaviour {

    public float speed = 10;

    void Update () {

        Vector3 dir = Vector3.zero;

        dir.x = -Input.acceleration.y;
        dir.z = Input.acceleration.x;

        if (dir.sqrMagnitude > 1)
            dir.Normalize ();

        dir *= Time.deltaTime;

        transform.Translate (dir * speed);
    }
}
```

4. 低通濾波器

加速度器的讀數可能會變得很嘈雜。對信號進行低通濾波可以使其平滑並擺脫高頻噪聲。

以下腳本顯示如何將低通濾波應用於加速度器讀數：

```
public class AccelerometerLowPassFilter : MonoBehaviour {

    public float accelerometerUpdateInterval = 1.0f / 60.0f;
    public float lowPassKernelWidthInSeconds = 1.0f;

    private float lowPassFilterFacctor;
    private Vector3 lowPassValue = Vector3.zero;

    void Start () {

        lowPassFilterFacctor = accelerometerUpdateInterval /
            lowPassKernelWidthInSeconds;

        lowPassValue = Input.acceleration;
    }

    public Vector3 LowPassFilterAccelerometer () {

        lowPassValue = Vector3.Lerp (lowPassValue,
            Input.acceleration, lowPassFilterFacctor);

        return lowPassValue;
    }
}
```

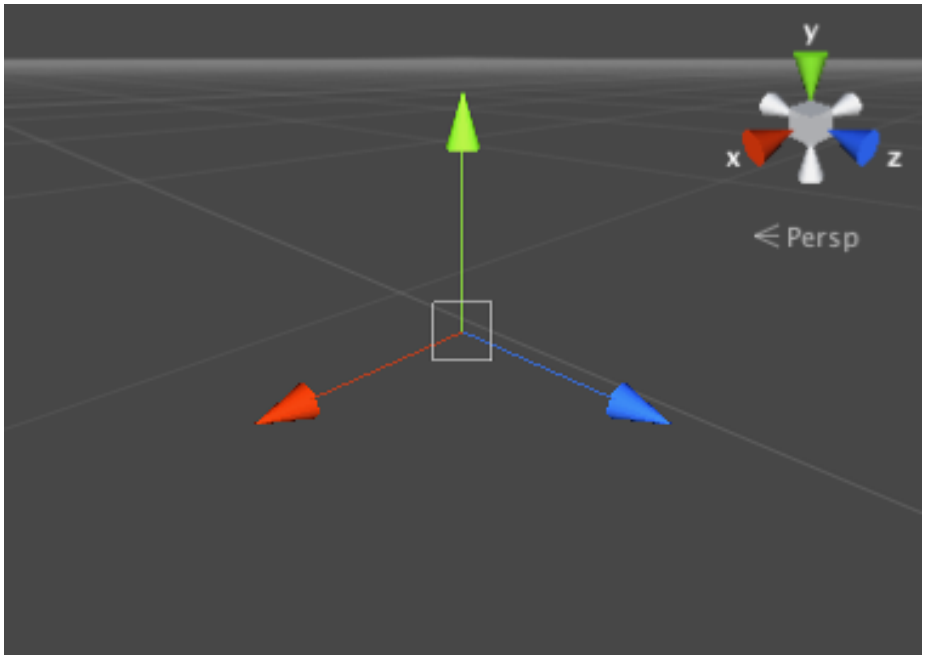
lowPassKernelWidthInSeconds 的值越大，濾波值越接近於當前輸入樣本（反之亦然）。

XXV.轉換組件 Transform（一）

Transform 用於儲存 GameObject 的位置、旋轉、縮放和父子狀態，因此非常重要。GameObject 將始終具有附加的 Transform 組件 - 無法刪除 Transform 或創建沒有 Transform 的 GameObject。

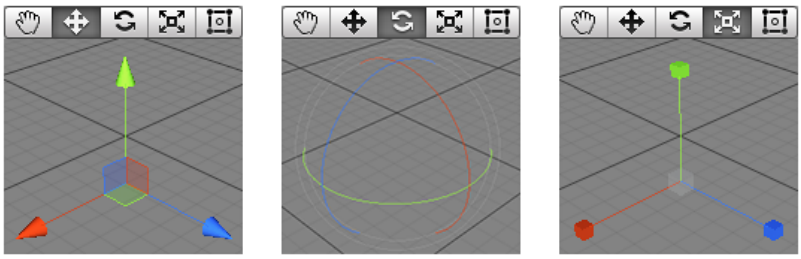
1. 編輯 *Transform*

Transform 在 3D 空間中使用 X、Y 和 Z 軸操作，或在 2D 空間中僅使用 X 和 Y 軸進行操作。在 Unity 中，這些軸分別由紅色、綠色和藍色表示。

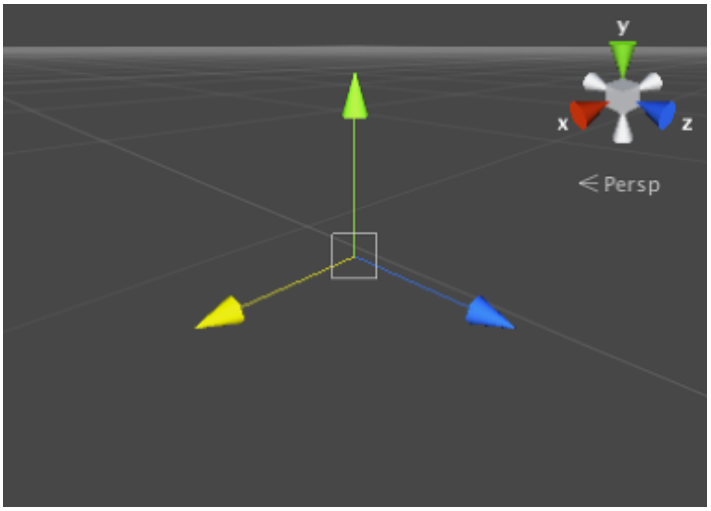


Transform 可以在 Scene 視窗中編輯，或在 Inspector 視窗中更改其屬性。在場景中，可以使用移動、旋轉和縮放工具修改 Transform。這些工具位於 Unity 編輯器的左上角。

這些工具可以使用在場景中的任何物件。當你點擊物件時，將看到工具出現在物件上。工具的外觀取決於選擇了哪個工具。



當你點擊並拖動三個工具軸之一時，會注意到其顏色變為黃色。當拖動滑鼠時，將看到物件沿所選軸的平移、旋轉或縮放。放開滑鼠按鈕時，軸保持選中狀態。

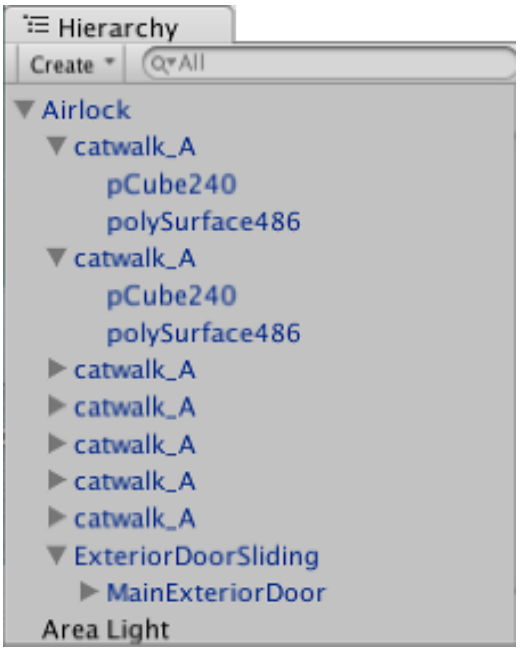


在 Translate 模式中還有一個附加選項可將移動鎖定到特定的平面上（即允許拖動兩個軸，同時保持第三個不變）。Translate 工具中心附近的三個小彩色方塊啟動每個平面鎖；顏色對應於當點擊方形時將被鎖定的軸（例如，藍色鎖定Z軸）。

2. 父子化

父子化是使用 Unity 時所要了解的最重要的概念之一。當 GameObject 是另一個 GameObject 的父級時，子 GameObject 會隨其父級一樣移動、旋轉和縮放。你可以將父子化視為像你的手臂和身體之間的關係；每當你的身體移動，你的手臂也隨之移動。子物件也可以擁有自己的子物件等等。所以你的雙手可以被視為手臂的「孩子」，然後每隻手都有幾根手指等等。任何物件都可以有多個子物件，但只有一個父級。這些多級別的父子關係形成一個 Transform 層次結構。層次結構頂端的物件（即層次結構中唯一沒有父項的物件）稱為根（Root）。

可以透過將 Hierarchy 視窗中的任何 GameObject 拖動到另一個來創建一個父級。這將在兩個 GameObject 之間創建一個父子關係。



請注意，對於任何子 GameObject 的 Inspector 視窗中的 Transform 值將顯示相對於父物件的 Transform 值。這些值被稱為局部坐標。回到身體和手臂的類比，你的身體的位置可能會隨著你的行走而移動，但你的手臂仍然會在相同的位置被附著。對於場景構造，通常使用本地坐標來進行子物件，但是在遊戲中，通常可以在世界空間或全域坐標中找到它們的確切位置。Transform 組件的腳本 API 具有單獨的屬性，用於區域和全域位置、旋轉和縮放，還允許轉換區域和全域坐標之間的任意點。

XXVI.轉換組件 Transform（二）

1. 非均勻縮放的局限性

不均勻縮放是當 Transform 中的縮放對於 X、Y 和 Z 具有不同的值時，例如（2,4,2）。相反，均勻縮放對於 X、Y 和 Z 具有相同的值，例如（3,3,3）。在一些特定情況下，非均勻縮放可能是有用的，但是它有一些在均勻縮放不會出現的怪異現象：

- 某些組件不完全支援非均勻縮放。例如，一些組件具有由半徑屬性定義的圓形或球形元素，其中包括 Sphere Collider、Capsule Collider、Light 和 Audio Source。在這種情況下，圓形的形狀將不會像所期望的那樣在非均勻縮放下變為橢圓形，並且將保持圓形。
- 當子物件具有不均勻縮放的父項並相對於該父物件旋轉時，它可能會出現傾斜或「剪切」。有些組件支援簡單的非均勻縮放，但是在這樣的傾斜時不能正常工作。例如，傾斜的 Box Collider 將不會准確地匹配渲染網格的形狀。
- 出於效能原因，非均勻縮放父項的子物件在旋轉時不會自動更新其縮放比例。結果，當縮放比例最終被更新時，子物件的形狀可能會突然改變，比如說子物件可能與父物件分離。

2. 比例的重要性

Transform 的比例決定了建模應用軟體中網格的大小與 Unity 中該網格的大小之間的差異。Unity 中的網格大小（因此是 Transform 的 Scale）非常重要，特別是在物理模擬過程中。預設情況下，物理引擎假設世界空間中的一個單位對應於一米。如果物體非常大，可能會出現「慢動作」降落；模擬實際上是正確又有效的，因為你正在看著很遠的一個非常大的物體下降。

有三個因素會影響物件的比例：

- 你的3D 建模應用程式中的網格尺寸。
- 物件匯入設置中的網格比例因子設置。
- Transform 組件的縮放值。

理想情況下，你不應該調整 Transform 組件中物件的縮放。最好的選擇是依據現實生活中的比例創建你的模型，所以你不必改變 Transform 的比例。下一個最佳選擇是在單一個網格的匯入設置中調整匯入網格的比例。基於匯入大小發生某些最佳化，實例化具有調整縮放比例的物件會降低性能。

3. 使用 *Transform* 的幾個提示

- Transform 父子化時，在添加子物件之前將父節點的位置設置為（0,0,0）是非常有用的。這意味著子物件的區域坐標將與全局坐標相同，從而更容易確保你的子物件處於正確的位置。
- 你可以從 Preferences（選單：Unity > Preferences，然後選擇 Colors & keys 面板）更改 Transform 軸（和其他 UI 元素）的顏色。
- 改變比例會影響子物件 Transform 的位置。例如，將父物件縮放為（0,0,0）將會相對於父物件將所有子物件置於（0,0,0）。

XXVII.添加隨機性遊戲元素（一）

隨機選擇的項目或值在許多遊戲中都很重要。接下來要介紹如何使用 Unity 的內建隨機函數來實現一些常見的遊戲機制。

1. 從陣列中選擇隨機項目

隨機抽取陣列元素可以選擇零和陣列最大索引值之間的隨機整數（等於陣列的長度減去 1）。這可以使用內建的 Random.Range 函數輕鬆完成：

```
public class RandomArrayItem : MonoBehaviour {

    public int[] intArray;

    void Update(){

        if(Input.GetKeyDown (KeyCode.Space)){

            int ele = intArray [Random.Range (0, intArray.Length)];
            Debug.Log (ele);
        }
    }
}
```

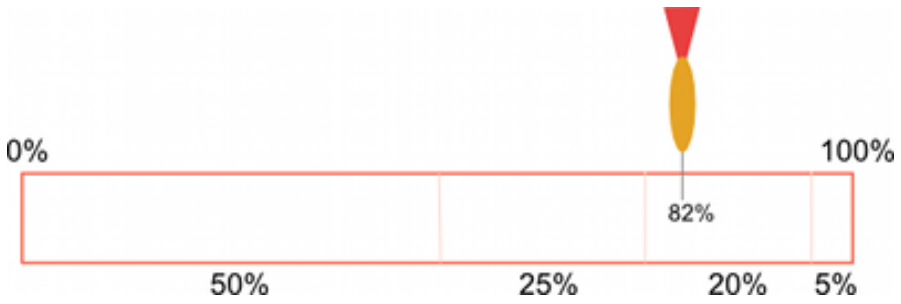
請注意，Random.Range 是從包含第一個參數的範圍返回值，但不包括第二個參數值的範圍，因此在此處使用 intArray.Length 提供正確的結果。

2. 選擇不同機率的項目

有時候，需要隨意選擇項目，但有些項目比其他項目更有可能被選擇。例如，NPC 在遇到玩家時可能會以幾種不同的方式作出反應：

- 50％ 的機會友善問候。
- 25％ 的機會逃跑。
- 20％ 的機會立即攻擊。
- 5％ 的機會提供錢做為禮物。

你可以將這些不同的結果視覺化為紙條，分為各自佔據條帶總長度的一部分。所佔據的比例等於選擇結果的機率。做出選擇等同於沿著條帶的長度（就像投擲飛鏢）選擇一個隨機點，然後查看它在哪一個部分。



在腳本中，紙條實際上是一系列浮點數，其中包含項目的不同機率。隨機點是通過將 Random.value 乘以陣列中所有浮點數的總和獲得的（它們不需要加起來為 1;重要的是不同值的相對大小）。要查找的點為「in」陣列元素，首先檢查它是否小於第一個元素中的值。如果是這樣，那麼第一個元素是所選的元素。否則，從點值中減去第一個元素的值，並將其與第二個元素進行比較，依此類推，直到找到正確的元素。在程式碼中，這看起來像下面這樣：

```
public class RandomArrayProb : MonoBehaviour {

    public float[] probs;

    void Update () {

        if(Input.GetButtonDown ("Fire1")){

            float res = Choose (probs);

            Debug.Log (res);
        }
    }

    private float Choose (float[] probs) {

        float total = 0;

        foreach (float elem in probs) {

            total += elem;
        }

        float randomPoint = Random.value * total;

        for (int i= 0; i < probs.Length; i++) {

            if (randomPoint < probs[i]) {

                return i;

            } else {

                randomPoint -= probs[i];
            }
        }

        return probs.Length - 1;
    }
}
```

請注意，最終返回語句是必需的，因為 Random.value 可以返回 1 的結果。在這種情況下，搜索將不會在任何地方找到隨機點。

3. 加權連續隨機值

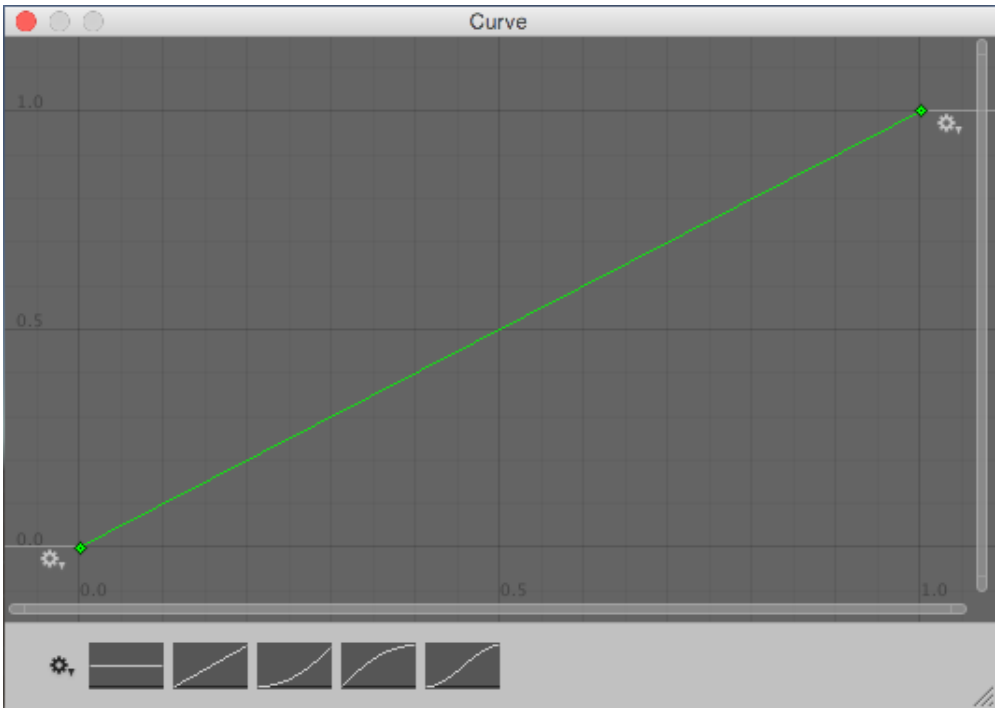
如果你要有不連續的結果，那麼浮點數陣列的方法會運行地很好，但是你也可能會想要產生更連續的結果 - 比如說，要隨機分配寶箱中發現的金幣數量，並且希望可能會在 1 到 100 之間的任何數字中得到最終結果。使用浮點數陣列的方法來執行將需要設置一個上百個浮動數的陣列（即紙條上的部分），這是很笨重的；如果您不限於整數，而是希望在範圍內使用任何數字，則不可能使用該方法。

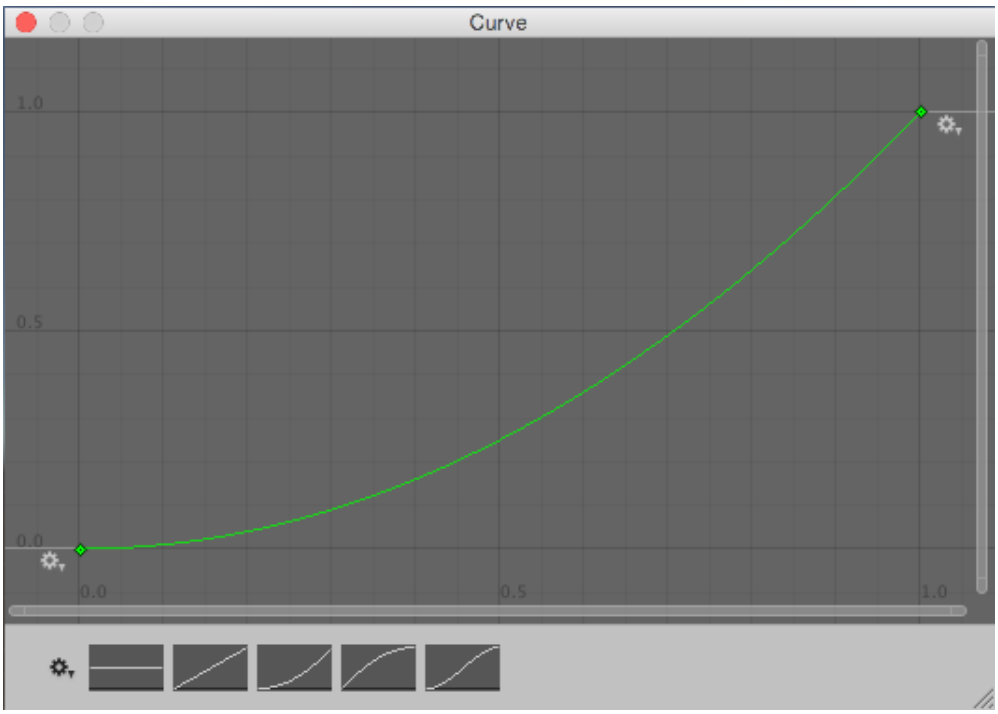
一個更好的連續結果的方法是使用 AnimationCurve 將「原始」隨機值轉換為「加權」值；通過繪製不同的曲線形狀，可以產生不同的權重。程式碼也更簡單：

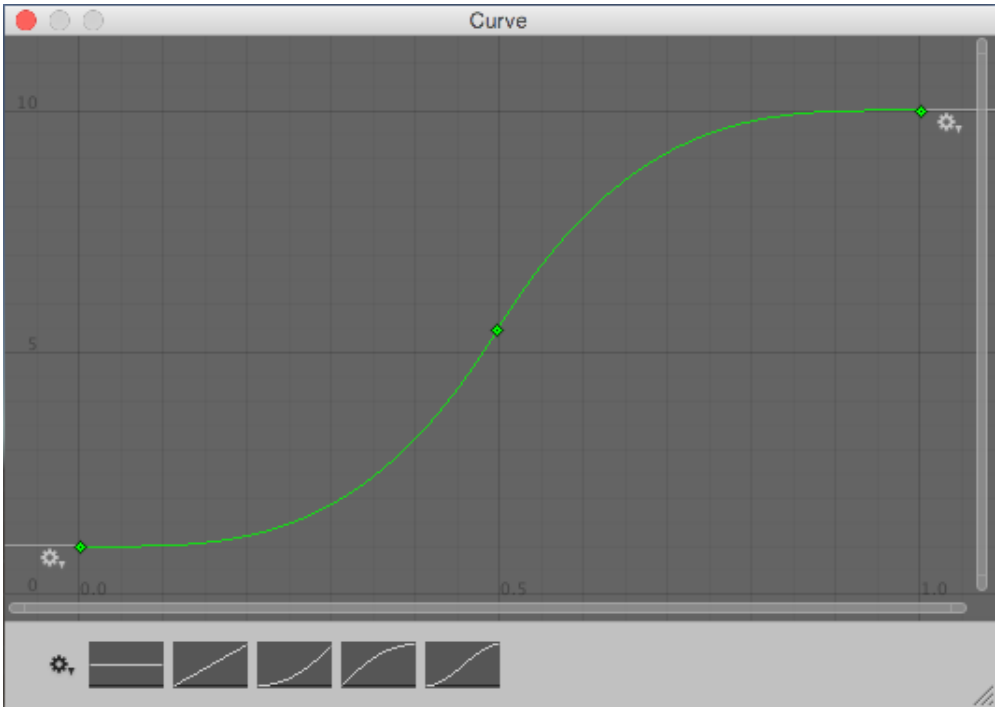
```
public class RandomCurveWeighted : MonoBehaviour {

    public AnimationCurve curve;

    void Update () {

        if(Input.GetButtonDown ("Fire1")){

            float res = curve.Evaluate (Random.value);

            Debug.Log (res);
        }
    }
}
```

通過從 Random.value 取得 0 到 1 之間的「原始」隨機值。然後將其傳遞給 curve.Evaluate()，將其視為水平坐標，並返回該水平位置處的曲線的相應垂直坐標。

曲線的平坦部分具有更大的機會被選擇到，而較陡部分被選到的機會較低。







請注意，這些曲線不是機率分佈曲線，而更像是反向累積機率曲線。

透過在腳本上定義一個公開 AnimationCurve 變數，將可以通過 Inspector 視窗查看和編輯曲線，而不需要計算值。

這種產生浮點數的技術。如果要計算整數結果 - 例如，需要 82 個金幣，而不是 82.1214 個金幣，可以將計算的值傳遞給像 Mathf.RoundToInt() 這樣的函數。

4. 洗牌

一種常見的遊戲機制是從一組已知的項目中進行選擇，但是按照隨機順序進行。例如，一副撲克牌通常會被洗牌，因此它們不能以可預測的順序抽出。你可以通過訪問每個元素並將其與陣列中的隨機索引中的另一個元素進行交換來隨機排列陣列中的項目：

```
public class RandomShuffle : MonoBehaviour {

    public int[] deck;

    void Start () {

        for (int i = 0; i < deck.Length; i++) {

            int temp = deck[i];
            int randomIndex = Random.Range(0, deck.Length);

            deck[i] = deck[randomIndex];
            deck[randomIndex] = temp;
        }
    }
}
```

XXVIII.添加隨機性遊戲元素（二）

1. 選出不重複的項目

一種常見的任務是從一個集合中隨機選擇一些項目，而不是多次挑選相同的項目。例如，你可能希望在隨機生成點生成一些 NPC，但要確保在每個點只生成一個 NPC。這可以通過重複排序項目來完成，為每個項目隨機決定是否將其添加到所選擇的集合中。隨著每個項目被造訪，其被選擇的機率仍然等於需要的數量除以可供選擇的數量。

例如，假設有十個產生點可用，但只能選擇五個。選擇第一個項目的概率為 5/10 或 0.5。如果選擇一個，那麼第二個項目的概率將是 4/9 或 0.44（即仍然需要四個項目，九個可供選擇）。然而，如果沒有選擇第一個，那麼第二個的概率將為 5/9 或 0.56（即，仍然需要五個，剩下九個可供選擇）。這一直持續到集合包含所需的五個項目。可以通過以下程式碼完成此操作：

```
public class RandomRepetition : MonoBehaviour {

    public int required;
    public int[] source;
    public int[] result;

    void Start () {

        result = new int[required];

        int choose = required;

        for(int total = source.Length ; total > 0 ; total--){

            float prob = (float)choose / (float)total;

            if(Random.value <= prob){

                choose--;

                result [choose] = source [total - 1];

                if(choose == 0){

                    break;
                }
            }
        }
    }
}
```

請注意，雖然選擇是隨機的，但所選集中的項目將與原始陣列中的順序相同。如果要按順序一次使用這些項目，則排序可以使其部分可預測，因此可能需要在使用前洗牌。

2. 空間裡的隨機點

可以通過將 Vector3 的每個組件設置為 Random.value 返回的值來選擇立方體體積中的隨機點：

```
public class RandomVector : MonoBehaviour {

    public float side = 5;

    void Start () {

        transform.position = new Vector3 (Random.value,
            Random.value, Random.value) * side;

    }
}
```

這給予一個立方體內一個單位長的一個點。可以通過將向量的 X、Y 和 Z 分量乘以所需的邊長度來簡單地縮放立方體。如果其中一個軸設置為零，則該點將始終位於單個平面內。 例如，選擇地面上的隨機點通常是隨機設置 X 和 Z 分量，並將 Y 分量設置為零。

當體積是一個球體（即，當你想要從原點起一個給定半徑內的隨機點）時，可以使用 Random.insideUnitSphere 乘以所需的半徑：

```
public class RandomInsideSphere : MonoBehaviour {

    public float radius = 5;

    void Start () {

        transform.position = Random.insideUnitSphere * radius;
    }
}
```

請注意，如果將其中一個向量組件的結果設置為零，則不會在圓內獲得正確的隨機點。雖然這一點確實是隨機的，並且位於正確的半徑範圍之內，但是這個概率對於圓的邊緣有很大的偏向，所以所獲得的點將非常不均勻地擴散。應該要為此目的使用 Random.insideUnitCircle：

```
public class RandomInsideCircle : MonoBehaviour {

    public float radius = 5;

    void Start () {

        transform.position = Random.insideUnitCircle * radius;
    }
}
```

XXIX.Unity 的旋轉與方向（一）

3D 應用程式中的旋轉通常用兩種方式之一表示，即四元數或歐拉角。每個都有自己的用途和缺點。Unity 在內部使用四元數 ，但是在 Inspector 視窗中顯示等效的歐拉角度的值，以便讓你編輯。

1. 歐拉角 *Euler Angle*

歐拉角具有更簡單的表示，它們是順序施加的三個角度值 X，Y 和 Z。為了將歐拉旋轉應用於特定物體，每個旋轉值依次被施加為圍繞其對應軸線的旋轉。

- 優點：歐拉角具有直覺的「可讀」格式，由三個角度組成。
- 優點：歐拉角可以表示從一個方向到另一個方向的轉動超過 180 度。
- 限制：歐拉角受到萬向鎖（Gimbal Lock）的影響。依次施加三次旋轉時，第一或第二旋轉可能導致第三軸指向與先前軸之一相同的方向。這意味著「自由度數」已經喪失，因為第三個旋轉值不能應用於唯一的軸。

2. 四元數 *Quaternion*

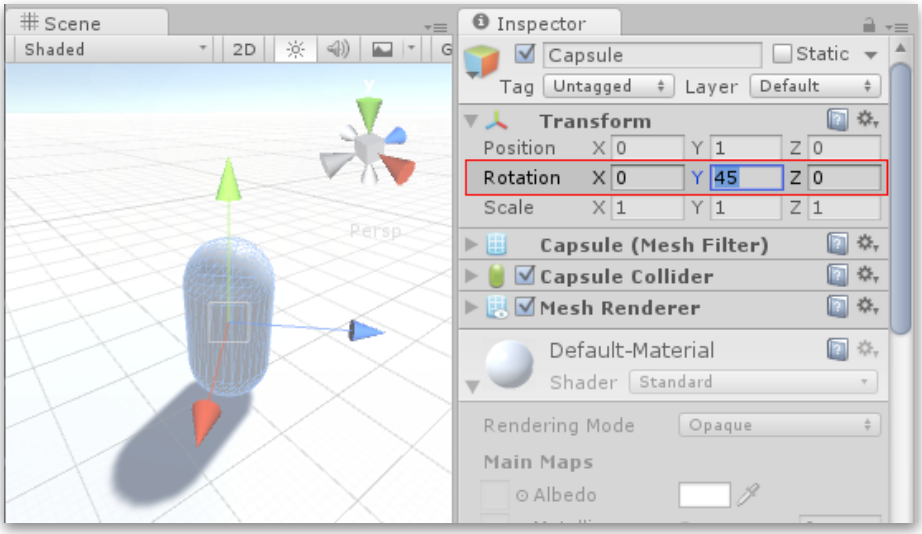
四元數可以使用來表示物件的方向或旋轉。此表示內部由四個數字組成（在 Unity 中參考為 X、Y、Z 和 W），但是這些數字並不代表角度或軸，通常不需要直接訪問它們。除非你特別有興趣於研究四元數學，否則你只需要知道四元數代表 3D 空間中的旋轉，你通常不需要知道或修改 X，Y 和 Z 屬性。

以向量可以表示位置或方向（其方向從原點測量）相同的方式，四元數可以表示方向或旋轉 - 其中旋轉是從旋轉「原點（Origin）」或「標識（Identity：這個四元數對應於“不旋轉” - 物件與世界或父軸完全對齊。）」測定的。因為旋轉以這種方式測量 - 從一個方向到另一個方向 - 四元數不能表示超過 180 度的旋轉。

- 優點：四元數旋轉並不受到萬向鎖的影響。
- 限制：單一個四元數不能表示任何方向超過180度的旋轉。
- 限制：四元數的數字表示方式是不能直覺的理解的。

在 Unity 中，所有的遊戲物件旋轉都是內部儲存為四元數，因為它的優點超過限制。

然而，在 Inspector 視窗中的 Transform，使用歐拉角顯示旋轉，因為這更容易理解和編輯。輸入到 Inspector 中的遊戲物件旋轉的新值將被轉換為「防護」下的物件的新的四元數旋轉值。



做為副作用，Inspector 視窗可以為遊戲物件的旋轉旋轉輸入 X：0、Y：365、Z：0 的值。這是一個不可能表示為四元數的值，所以當你點擊 Play 時，會看到物件的旋轉值更改為 X：0、Y：5、Z：0（或其周圍）。這是因為旋轉被轉換為不具有「完整 360 度旋轉加 5 度」概念的四元數，而是簡單地將其設置為與旋轉結果相同的方向。

3. 對編寫腳本的影響

使用你的腳本應對處理旋轉時，應該使用 Quaternion Class 及其函數來創建和修改旋轉值。在某些情況下，使用歐拉角是有效的，但你應該牢記：你應該使用處理歐拉角的 Quaternion Class 函數 - 從旋轉中檢索、修改和重新套用歐拉值可能導致偶然的副作用。

然而有時候，腳本中可能需要使用歐拉角。在這種情況下，重要的是要注意，必須將角度保持在變數中，並且只能使用它們將它們應用於您的旋轉的歐拉角。雖然可以從四元數中檢索歐拉角，但如果檢索，修改和重新套用，則會出現問題。

以下是使用假設的範例，嘗試以每秒 10 度旋轉物件的錯誤例子。 這是你應該避免的：

```
public class MistakeRotation1 : MonoBehaviour {

    void Update () {

        Quaternion rot = transform.rotation;
        rot.x += Time.deltaTime * 10;
        transform.rotation = rot;
    }
}
```

這裡的錯誤是我們正在持續修改四元數的 X 值，該值不表示角度，並不會產生期望的結果。

```
public class MistakeRotation2 : MonoBehaviour {

    void Update () {

        Vector3 angles = transform.rotation.eulerAngles;
        angles.x += Time.deltaTime * 10;
        transform.rotation = Quaternion.Euler(angles);
    }
}
```

這裡的錯誤是我們正在讀取、修改然後從四元數寫入歐拉值。因為這些值是從四元數計算出來的，所以每個新的旋轉可能會返回非常不同的歐拉角，這可能會遭受萬向鎖定。

這裡是正確使用歐拉角度的例子：

```
public class CorrectlyRotation : MonoBehaviour {

    private float x;

    void Update () {

        x += Time.deltaTime * 10;
        transform.rotation = Quaternion.Euler(x , 0 , 0);
    }
}
```

在這裡，我們將歐拉角儲存在一個變數中，只用它來應用為歐拉角，但是從不依賴於讀取歐拉。

XXX.Unity 的旋轉與方向（二）

Unity 的 Quaternion Class 具有許多功能，允許你創建和操作旋轉，而無需使用歐拉角。

1. 使用 *Quaternion* 建立旋轉

- Quaternion.LookRotation：使用指定的向前和向上方向創建旋轉。回傳計算出的四元數。

```
public class QuaternionLookRotation : MonoBehaviour {

    public Transform target;

    void Update() {

        Vector3 relativePos = target.position - transform.position;

        Quaternion rotation = Quaternion.LookRotation(relativePos);

        transform.rotation = rotation;
    }
}
```

- Quaternion.AngleAxis：創建圍繞軸旋轉角度的旋轉。

```
public class QuaternionAngleAxis : MonoBehaviour {

    public float speed = 50;

    private Vector3 up;
    private Quaternion rotate;

    private float angle;

    void Start(){

        up = transform.up;
        rotate = transform.rotation;
    }

    void Update () {

        angle += Time.deltaTime * speed;

        transform.rotation = Quaternion.AngleAxis (angle, up) * rotate;
    }
}
```

- Quaternion.FromToRotation：從一個方向朝另一個方向建立旋轉。

```
public class QuaternionFromToRotation : MonoBehaviour {

    private Vector3 up;
    private Quaternion rotate;

    void Start(){

        up = transform.up;
        rotate = transform.rotation;
    }

    void Update () {

        if (Input.GetButtonDown ("Fire1")) {

            transform.rotation = Quaternion.
                FromToRotation (up, transform.forward) * rotate;
        }
    }
}
```

2. 使用 *Quaternion* 操作旋轉

- Quaternion.Slerp：透過一個 t 值在兩個旋轉四元數之間球體插值，t 值介於 0~1 的範圍之間。

```
public class QuaternionSlerp : MonoBehaviour {

    public Transform from;
    public Transform to;
    public float t;

    void Update() {

        transform.rotation = Quaternion.
            Slerp(from.rotation, to.rotation, t);
    }
}
```

- Quaternion.Inverse：回傳旋轉的反向。

```
public class QuaternionInverse : MonoBehaviour {

    public Transform target;

    void Update() {

        transform.rotation = Quaternion.
            Inverse(target.rotation);
    }
}
```

- Quaternion.RotateTowards：從來源向目標旋轉一個角度。透過最大度數變化（maxDegreesDelta）將來源的四元數轉向目標（但旋轉不會超過目標），最大度數變化為負值的話，將會旋轉到完全相反的方向。

```
public class QuaternionRatateTowards : MonoBehaviour {

    public Transform target;
    public float speed;

    void Update() {

        float step = speed * Time.deltaTime;

        transform.rotation = Quaternion.
            RotateTowards(transform.rotation,
                target.rotation, step);
    }
}
```

3. 對動畫的影響

許多 3D 創作包，以及 Unity 自己的內部 Animation 視窗，允許在動畫過程中使用歐拉角來指定旋轉。

這些旋轉值往往可以超過四元數能夠表達的範圍。例如，如果物件應該在原位旋轉720度，則可以用歐拉角 X：0、Y：720、Z：0 表示。但這根本不是四元數所能表達代表的值。

- Unity 的 Animation 視窗

  在 Unity 自己的 Animation 視窗中，有一些選項允許指定如何插值旋轉 - 使用四元數或歐拉插值。通過指定歐拉插值，表示告訴 Unity，你希望透過角度指定的全部運動範圍。然而，使用四元數旋轉，只是想要旋轉到特定方向結束，Unity將使用四元數插值，並在最短距離內旋轉到達。

- 外部動畫來源

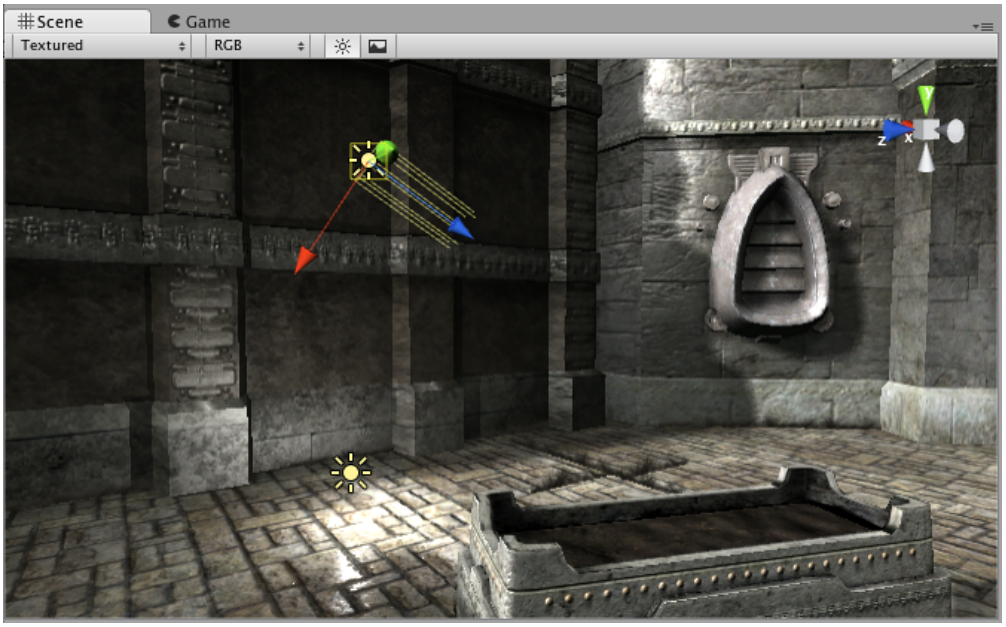
  從外部來源匯入動畫時，這些檔案通常包含歐拉格式的旋轉關鍵幀動畫。Unity 預設行為是重新採樣這些動畫，並為動畫中的每個幀生成一個新的四元數關鍵幀，以避免關鍵幀之間的旋轉可能超過四元數有效範圍的任何情況。

  例如，假設兩個關鍵幀相隔 6 幀，第一個關鍵幀上的 X 為 0，第二個關鍵幀為 270。在沒有重新採樣的情況下，這兩個關鍵幀之間的四元數插值將沿相反方向旋轉 90 度，因為這是從第一方向到第二方向的最短方式。然而，通過在每個幀上重新採樣和添加關鍵幀，關鍵幀之間現在只有 45 度，所以旋轉將正常工作。

  仍然有一些情況 - 即使重新採樣 - 匯入的動畫四元數表示可能與原始的不完全相符，因此，在Unity 5.3 及以上版本中，可以選擇關閉動畫重新採樣，以便可以在運行時使用原始的歐拉動畫關鍵幀。

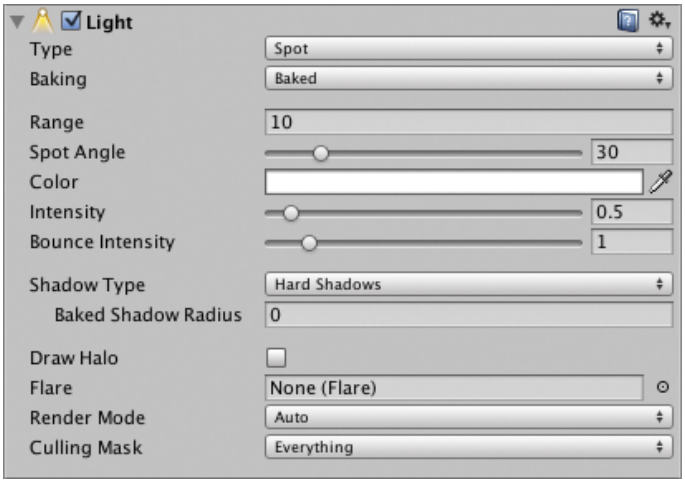
XXXI.遊戲中的照明

照明是每個場景的必要組成部分。網格和紋理定義場景的形狀和外觀，照明定義 3D 環境的顏色和氣氛。可能會在每個場景中使用多個光源。使它們共同運作需要一點練習，但結果會是相當驚人的。

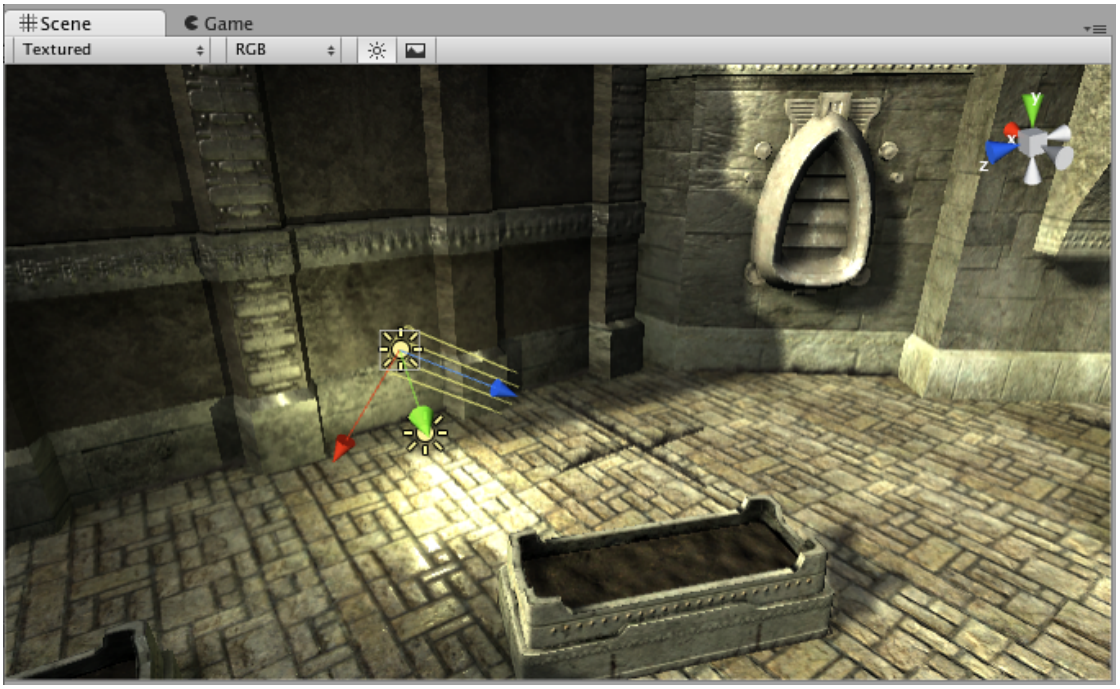


可以從 GameObject > Light 選單將光源添加到場景中。從出現的子選單中選擇所需的光源格式。一旦添加了光，你可以像任何其它 GameObject 一樣操縱它們。此外，可以通過使用 Component > Rendering > Light 向所選的 GameObject 添加 Light Component。

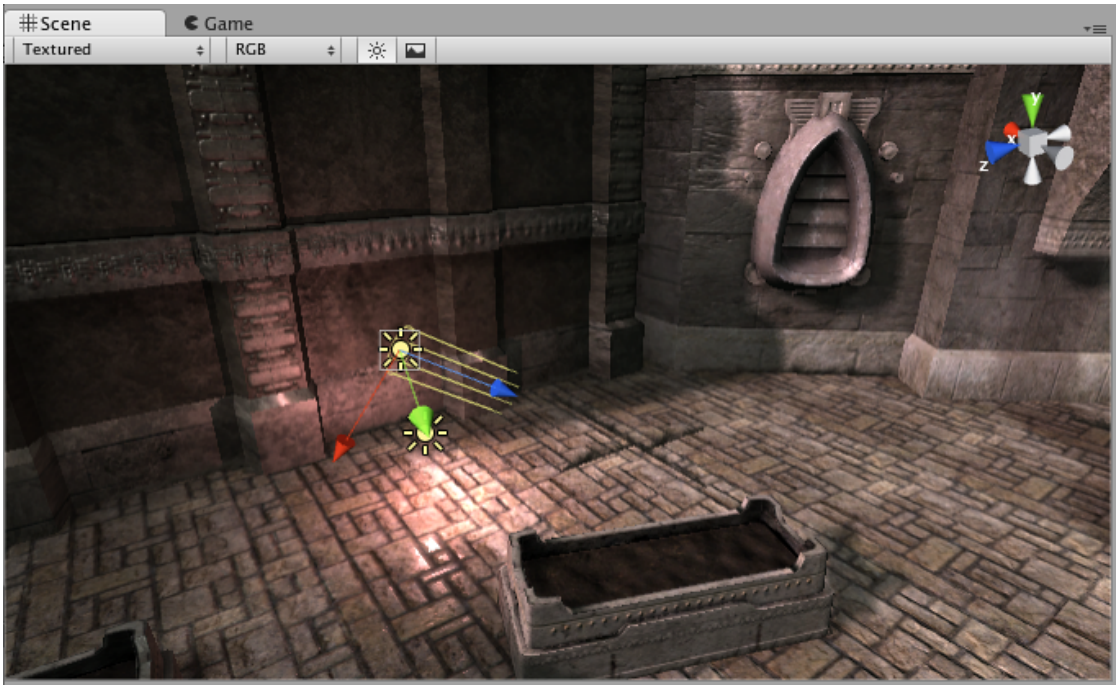
Inspector 視窗中，Light Component 中有許多不同的選項。



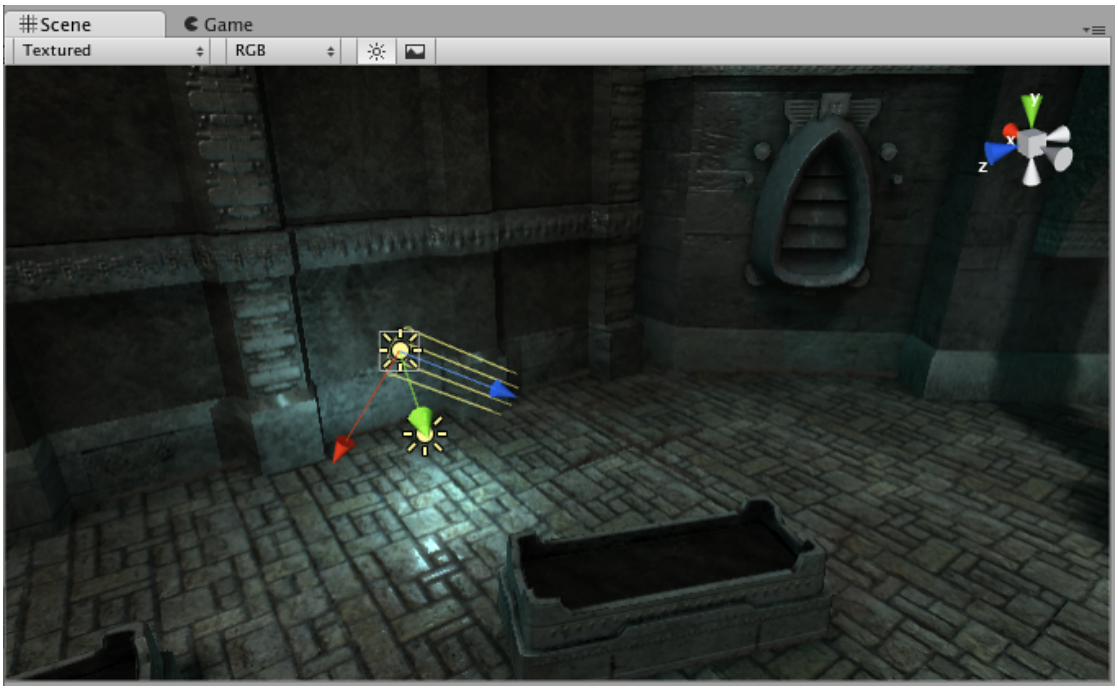
通過簡單的改變光的顏色，能夠獲得完全不同氣氛的場景。



陽光明媚的照明



黑暗、中世紀的照明



陰森的夜間照明

1. 渲染路徑

Unity 支持各種渲染路徑。這些路徑主要影響燈光和陰影，因此根據遊戲需求選擇正確的渲染路徑可以提高專案的效能。

你應該根據你的遊戲內容和目標平台/硬體選擇使用哪一個。不同的渲染路徑具有不同的效能特徵，主要影響光和陰影。

你的專案使用的渲染路徑可在 Graphics Settings 中選擇。此外，可以針對每個攝影機覆蓋這個設置。

如果顯示卡無法處理所選的渲染路徑，Unity 將自動使用較低保真度的。例如，在不能處理延遲著色（Deferred Shading）的 GPU 上，將使用前向渲染（Forward Rendering）。

- 延遲著色 Deferred Shading：延遲著色是具有最多照明和陰影保真度的渲染路徑，如果有許多即時照明，則最適合。它需要一定程度的硬體支援。
- 前向渲染 Forward Rendering：這是傳統的渲染路徑。它支持所有典型的 Unity 圖形功能（法線貼圖、每像素照明、陰影等）。然而，在預設設置下，只有少數最亮的燈在每像素照明模式下渲染。剩下的燈是在物件頂點或每個物件上計算的。

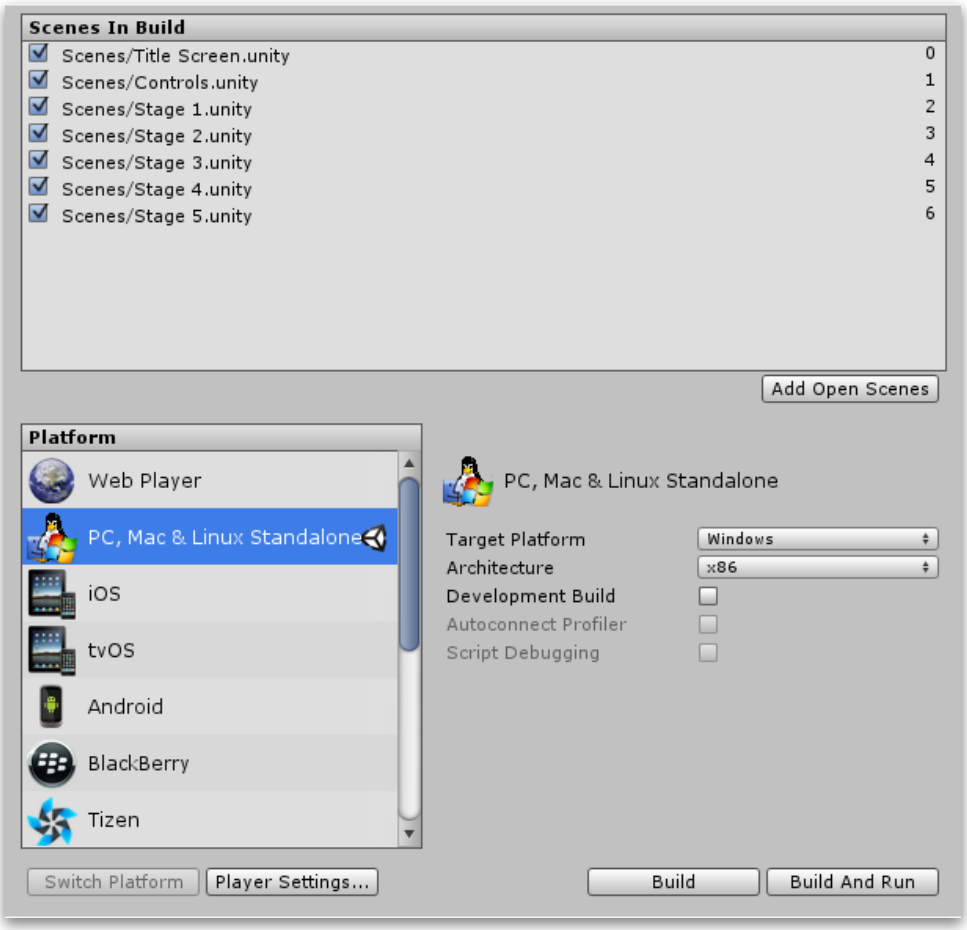
- 傳統延遲 Legacy Deferred：傳統延遲與延遲著色類似，只是使用不同的技術，不同的權衡。它不支持基於 Unity 5 的標準著色器。
- 傳統頂點光 Legacy Vertex Lit：Legacy Vertex Lit 是具有最低照明保真度的渲染路徑，不支援即時陰影。它是前向渲染路徑的一個子集。

注意：使用正交（Orthographic）投影時不支持延遲渲染。如果攝影機的投影模式設置為「正交（Orthographic）」，則這些值將被覆蓋，攝影機將始終使用前向渲染。

XXXII.跨平台遊戲構建與發佈要項

在創建遊戲的任何時候，你可能希望看到以單機方式構建並在編輯器之外運行時的外觀。

選單 File > Build Settings... 開啟 Build Settings 視窗。會彈出一個可編輯的場景列表，當構建遊戲時將包含這些場景。



第一次在專案中查看此視窗時，它顯示為空白。如果在此列表空白時構建遊戲，則只有當前打開的場景才會包含在構建中。如果要快速構建一個只有一個場景檔案的測試播放器，只需構建一個空白場景列表的播放器。

要構建多個場景，只要簡單的添加場景檔案到列表。有兩種方法可以添加它們。第一種方法是點擊 Add Open Scenes 按鈕。將看到當前打開的場景出現在列表中。添加場景檔案的第二種方法是將它們從 Project 視窗拖動到列表中。

此時，請注意，每個場景都有不同的索引值。場景 0 是要載入的第一個場景。當要載入新場景時，請在腳本中使用 SceneManager.LoadScene()。

如果添加了多個場景檔案並想重新排列，只需點擊並拖動列表上方或下方的場景，直到按照所需順序排列。

如果要從列表中刪除場景，請點擊選取場景，然後按 Delete。場景將從列表中消失，不會包含在構建中。

當您準備發佈構建時，選擇一個平台，並確保 Unity 標誌位於該平台旁邊；如果不是，點擊 Switch Platform 按鈕，讓 Unity 知道要構建哪個平台。最後按 Build 按鈕。可以使用標準的「儲存」視窗來決定遊戲的名稱和位置。點擊儲存時，Unity 會建立你的遊戲內容。這很簡單 如果不確定要在哪裡儲存你構建的遊戲，請考慮將其儲存到專案的根目錄中。不要將構建儲存到 Assets 文件夾中。

勾選 Development Build 將啟用 Profiler 功能，並且會使 Autoconnect Profiler 和 Script Debugging 選項可用。

1. 建立單機播放器

使用 Unity 可以為 Windows、Mac 和 Linux 構建獨立的應用程式。這只要在 Build Settings 中選擇構建目標，並點擊 Build 按鈕。構建單機播放器時，生成的檔案將根據構建目標而有所不同。對於 Windows，將構建可執行檔案（.exe），並包含應用程式所有資源的 Data 資料夾。對於 Mac，將構建一個包含運行應用程式所需的檔案以及資源的應用程式包。

在 Mac 上發佈，只是提供應用程式包（所有東西都包裝在裡面）。在 Windows 上，你需要提供 .exe 檔案和 Data 資料夾給其他人運行。其他人必須在他們的電腦上具有相同的檔案，以你為 Unity 生成的檔案來運行你的遊戲。

2. 構建過程

構建過程將會在你指定的地方放置構建的遊戲應用程式的空白副本。然後，它將透過 Build Settings 中的場景列表工作，一次在編輯器中打開、最佳化它們，並

將它們整合到應用程式包中。還將會計算所包含場景所需的所有資源，並將該資料儲存在應用程式包中的個別檔案中。

- 已發佈的構建中，不包含場景中標有「EditorOnly」的任何 GameObject。這對於最終不需要包含在遊戲中的除錯腳本很有用。
- 當新的關卡載入時，上一個關卡的所有物件都將被銷毀。為了防止這種情況，請對任何不想銷毀的物件使用 DontDestroyOnLoad()。這最常用於在載入關卡時保持音樂播放或者用於保持遊戲狀態和進度的遊戲控制器腳本。
- 載入新的關卡完成後，OnLevelWasLoaded() 將被發送到所有現行的遊戲物件。

3. 預載

已發佈的構建會在場景載入時自動預載場景中的所有資源。場景 0 是這規則的例外，這是因為第一個場景通常是一個啟動畫面，你會希望盡可能快地顯示。

為了確保所有內容都已預載，可以創建一個呼叫 SceneManager.LoadScene(1) 的空場景。在構建設置中，使這個空場景的索引為 0，隨後的所有關卡都將被預載。

XXXIII.Unity 程式設計介紹

腳本是所有遊戲的重要組成部分。即使是最簡單的遊戲也需要腳本來回應來自玩家的輸入並安排遊戲中應該發生的事件。除此之外，腳本可用於創建圖像效果、控制物件的物理行為，甚至可以為遊戲中的角色實現自定義 AI 系統。

腳本是一種需要一些時間和精力去學習的技能。這裡不是教你如何從頭開始編寫腳本程式碼，而是解釋適用於 Unity 中腳本的主要概念。

雖然 Unity 使用標準 Mono 執行期的腳本來實現，但它仍然有自己從腳本訪問引擎的實踐和技術。

GameObject 的行為由附加到它們的組件控制。雖然 Unity 的內建組件可以有非常多的用途，但你很快就會發現，你需要在它可以提供的內容之外實現自己的遊戲功能。Unity 讓你能夠使用腳本創建自己的組件。這些，讓你能夠觸發遊戲事件、隨時間修改組件屬性，並以任何你喜歡的方式回應使用者輸入。

Unity 本身支援兩種程式語言：

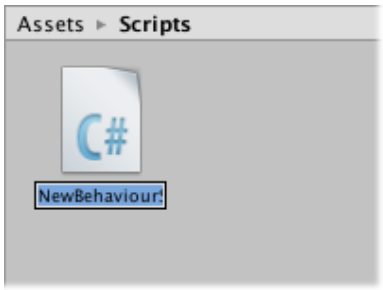
- C#（發音為 C-sharp），類似於 Java 或 C++ 的業界標準語言。
- UnityScript，專為 Unity 使用而設計並用 JavaScript 塑造的語言（即將廢除）。

除了這些，如果可以編譯相容的DLL，可以使用許多其他 .NET 語言。

1. 創建腳本

與大多數其他資源不同，腳本通常直接在 Unity 內創建。你可以從 Project 視窗左上角的 Create 選單創建新腳本，或者從主選單中選擇 Assets > Create > C# Script (或 JavaScript) 。

將會在 Project 視窗中被選擇的資料夾中創建新腳本。新腳本檔案的名稱將被選取，並提示你輸入新名稱。



最好是現在輸入新腳本的名稱，而不是稍後再編輯。輸入的名稱將用於檔案內容的 Class 名稱。

2. 腳本檔案剖析

當您雙擊 Unity 中的腳本資源時，它將在文字編輯器中打開。預設情況下，Unity 將使用 MonoDevelop，但你可以從 Unity 的 Preferences 中的 External Tools 面板中選擇所需的任何編輯器。

該檔案的的初始內容將如下所示：

```
public class NewBehaviourScript : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

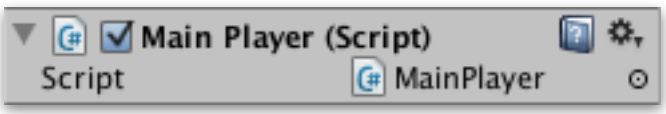
一個腳本通過實現一個繼承自 MonoBehaviour 內建 Class 來與 Unity 的內部運作聯繫起來。你可以將該 Class 視為一種創建可以附加到 GameObject 的新組件類型的藍圖。每次將腳本組件附加到 GameObject 時，它將創建由藍圖定義的物件新實例。該 Class 的名稱取自創建檔案時提供的名稱。Class 名稱和檔名必須相同才能使腳本組件附加到 GameObject。

然而，主要需要注意的是在 Class 中定義的兩個函數。Update 功能是放置將處理 GameObject 於每次畫面更新的程式碼的地方。這可能包括移動、觸發動作和回應使用者輸入，基本上任何需要在遊戲過程中隨時隨地處理的任何事情。通常在啟用 Update 功能來執行工作、在任何遊戲操作發生之前，能夠設置變數、讀取喜好設定並與其他 GameObject 進行連接會很有用。在遊戲開始之前（即，在第一次執行 Update 功能之前），Unity 將執行 Start 函數，而且這是進行任何初始化的理想場所。

有經驗的程式人員注意：可能會感到驚訝的是，使用建構函數不會對物件進行初始化。這是因為物件的建構由編輯器處理，並且不會像所期望的那樣在遊戲開始時進行。如果嘗試為腳本組件定義建構函數，它將干擾 Unity 的正常運行，並可能會引發專案的重大問題。

3. 控制 *GameObject*

如上所述，腳本僅定義了組件的藍圖，因此在腳本的實例附加到 GameObject 之前，它們的程式碼都不會被啟動。你可以通過將腳本資源拖動到 Hierarchy 視窗中的 GameObject 或當前選擇的 GameObject 的 Inspector 視窗來附加腳本。在 Component 選單上還有一個 Scripts 子選單，其中包含專案中可用的所有腳本，包括你自己創建的腳本。腳本實例看起來與 Inspector 視窗中的任何其他組件類似：



一旦附加上去，當你按播放來運行遊戲，腳本將開始運作。你可以通過在 Start 功能中添加以下程式碼來檢查：

```
// Use this for initialization
void Start () {

    Debug.Log("I am alive!");
}
```

Debug.Log 是一個簡單的指令，它只向 Unity 的 Console 視窗輸出訊息。如果你立即按播放，應該會在 Unity 編輯器底部和 Console 視窗（選單：Window > Console）中看到該訊息。

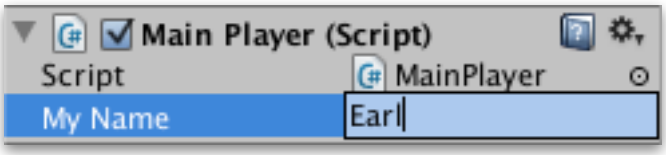
XXXIV.Unity 程式構造與編輯器的關係

創建腳本時，本質上是創建自己的新類型的組件，可以像任何其他組件一樣附加到 Game Object。

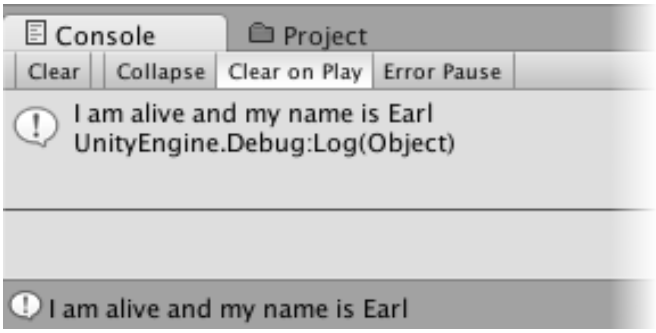
就像其他組件一樣，通常在 Inspector 視窗中會有可編輯的屬性，可以從 Inspector 視窗編輯腳本中的值。

```
public class NewBehaviourScript : MonoBehaviour {

    public string myName;

    void Start () {

        Debug.Log("I am alive and my name is " + myName);
    }
}
```

此程式碼在 Inspector 視窗建立可編輯的欄位，並將欄位名稱標示為「My Name」。



Unity 通過在變數名稱中出現大寫字母的位置引入空格來創建 Inspector 欄位標籤。但是，這純粹僅用於顯示，你應該始終在程式碼中使用原本的變數名稱。如果編輯該欄位，然後按播放，將看到該訊息包含所輸入的文字。



在 C# 中，須將變數聲明為 public，才能在 Inspector 視窗中出現欄位。

Unity 將實際上讓你在遊戲運行時更改腳本變數的值。這對直接查看更改的影響非常有用，無需停止並重新啟動。當遊戲結束時，變數的值將重置為按下 Play 之前的值。這可以確保你能夠自由地調整物件的設置，而不用擔心造成永久性的傷害。

XXXV.程式組件對遊戲物件的控制

在 Unity 編輯器中，可以使用 Inspector 視窗更改組件屬性。因此，例如對 Transform 組件的位置值進行更改將導致對 GameObject 的位置變更。相同地，可以更改 Renderer 材質顏色或 Rigidbody 的質量，將對 GameObject 的外觀或行為具有相對應的影響。在大多數情況下，腳本也是針對修改組件屬性來操縱 GameObject。不同之處在於，腳本可以隨時間的變化或回應使用者的輸入來改變屬性的值。通過在正確的時間更改、創建和銷毀物件，可以實現任何類型的遊戲。

1. 訪問組件

最簡單和最常見的情況是腳本需要訪問附加到同一 GameObject 的其他組件。如上所述，組件實際上是一個 Class 的實例，所以第一步是獲取要使用的組件實例的參考。這是通過 GetComponent 函數完成的。通常，你要將組件物件分配給變數，該變數使用以下語法在 C# 中完成：

```
void Start () {
    Rigidbody rb = GetComponent<Rigidbody>();
}
```

一旦對組件實例進行了參考，就可以像 Inspector 視窗那樣設置其屬性的值：

```
void Start () {
    Rigidbody rb = GetComponent<Rigidbody>();
    rb.mass = 2;
}
```

在 Inspector 視窗中不可用的額外功能可以在組件實例上呼叫函數來執行：

```
void Start () {
    Rigidbody rb = GetComponent<Rigidbody>();
    rb.AddForce(Vector3.up * 500);
}
```

```
public class NewBehaviourScript : MonoBehaviour {

    void Start () {

        Rigidbody rb = GetComponent<Rigidbody>();

        rb.mass = 2;

        rb.AddForce(Vector3.up * 500);
    }
}
```

還要注意的是，沒有理由不能將多個自定義腳本連接到同一個物件。如果需要從一個腳本訪問另一個腳本，則可以照常使用 GetComponent，只需使用腳本 Class 的名稱來指定所需的組件類型。

如果嘗試取得尚未實際添加到 GameObject 的組件，則 GetComponent 將返回 null；如果嘗試更改 null 物件上的任何值，將在運行時得到一個空參考錯誤。

2. 訪問其它物件

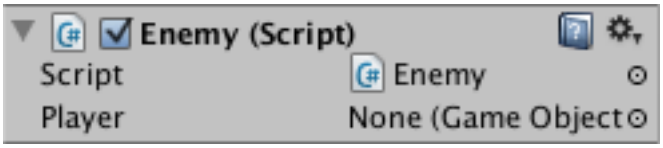
雖然它們有時是隔離的操作，但腳本通常保持追蹤其他物件。例如，敵人可能需要知道玩家的位置。Unity 提供了許多不同的方式來取得其他物件。

- 使用變數連結物件

  找到相關 GameObject 最簡單的方法是將一個公開的 GameObject 變數添加到腳本中：

  ```
  public GameObject player;
  ```

  該變數將像其他欄位一樣在 Inspector 視窗顯示：



現在，你可以將場景或 Hierarchy 視窗中的物件拖動到此變數上進行配置。就像任何其他物件一樣，可使用 GetComponent 函數和訪問組件變數，因此可以使用如下程式碼：

```
public class Example : MonoBehaviour {

    public GameObject player;

    void Start() {

        transform.position =
            player.transform.position - Vector3.forward * 10f;
    }
}
```

另外，如果在腳本中聲明組件類型的公開變數，則可以將具有該組件的任何 GameObject 拖動到其上。這將直接訪問組件，而不是 GameObject 本身。

```
public Transform playerTransform;
```

當處理具有永久連接的各個物件時，將物件與變數鏈接起來最為有用。你可以使用陣列變數鏈接相同類型的多個物件，但連接仍然必須在 Unity 編輯器中進行，而不是在運行時。在運行時定位物件通常很方便，如下，Unity 提供了兩種基本的方法來實現。

- 尋找子物件

  有時候，遊戲場景會使用相同類型的物件，如敵人、航點和障礙物。這些可能需要由管理或對其進行反應的特定腳本進行跟踪（例如，所有航點可能需要可用於航點腳本）。使用變數鏈接這些物件是一種可能性，但是如果每個新的航點都必須被拖動到腳本上的變數，那麼這將使設計過程變得乏味。同樣地，如果一個航點被刪除，那麼必須刪除對該丟失物件的變數參考也是麻煩的。在這種情況下，通過使它們成為一個父物件的所有子物件來管理整組物件通常會更好。可以使用物件的 Transform 組件（因為所有的 GameObject 具有 Transform）來取得子物件：

```
public class WaypointManager : MonoBehaviour {

    public Transform[] waypoints;

    void Start() {

        waypoints = new Transform[transform.childCount];

        int i = 0;

        foreach (Transform t in transform) {

            waypoints[i++] = t;
        }
    }
}
```

還可以使用 Transform.Find 函數透過名稱查找特定的子物件：

```
transform.Find("Gun");
```

當物件具有可以在遊戲過程中添加和刪除的子項時，這可以很有用。能夠拾起和放下的武器就是一個很好的例子。

- 透過名稱或 Tag 尋找物件

只要有一些訊息來識別，就可以在場景層次結構中的任何位置找到 GameObject。可以使用 GameObject.Find 函數通過名稱找出單一個物件：

```
public class GameObjectFind : MonoBehaviour {

    public GameObject player;

    void Start() {

        player = GameObject.Find("Main Camera");
    }
}
```

物件或物件的集合也可以使用 GameObject.FindWithTag 和 GameObject.FindGameObjectsWithTag 函數通過其 Tag 來找出：

```
public class GameObjectFindWithTag : MonoBehaviour {

    public GameObject player;
    public GameObject[] enemies;

    void Start() {

        player = GameObject.FindWithTag("Player");

        enemies = GameObject.FindGameObjectsWithTag("Enemy");
    }
}
```

XXXVI.Unity 事件功能與執行順序

Unity 中的腳本不像程式的傳統想法，程式碼在循環中持續運行，直到完成其任務。相反的，Unity 通過呼叫在其中聲明的某些函數間接地將控制傳遞給腳本。一旦功能完成執行，控制就會傳回 Unity。這些功能被稱為事件功能，因為它們被 Unity 啟動以回應遊戲過程中發生的事件。Unity使用命名方案來識別要呼叫特定事件的函數。例如，已經看過的 Update 函數（在幀更新之前呼叫）和 Start 函數（在物件的第一幀更新之前呼叫）。Unity 中還有更多的事件功能可用；完整列表可以在 MonoBehaviour Class 的腳本參考頁面中找到，以及它們的使用細節。以下是一些最常見和重要的事件。

1. 定期更新事件

遊戲就像一個動畫，動畫幀是在製作中生成的。在遊戲程式中的一個關鍵概念是在渲染每幀之前對遊戲中的物件位置、狀態和行為進行更改。Update 功能是 Unity 中此類程式碼的主要地方。Update 在渲染幀之前以及在計算動畫之前被呼叫。

```
void Update() {

    float distance =
        speed * Time.deltaTime * Input.GetAxis("Horizontal");

    transform.Translate(Vector3.right * distance);
}
```

物理引擎以類似的方式在分離的時間階段進行更新來進行幀渲染。在每次物理更新之前呼叫一個名為 FixedUpdate 的事件函數。由於物理更新和幀更新不會以相同的頻率發生，因此如果將其放置在 FixedUpdate 函數而不是 Update 中，則可以從物理程式碼中獲得更準確的結果。

```
void FixedUpdate() {

    Vector3 force =
        transform.forward * driveForce * Input.GetAxis("Vertical");

    rigidbody.AddForce(force);
}
```

有時也可以在對場景中的所有物件呼叫 Update 和 FixedUpdate 函數之後以及在計算所有動畫之後的某個時間點進行其他更改。例如攝影機應該保持對目標物件的瞄準；必須在目標物件移動後對攝影機的方向進行調整。另一個例子是腳本程式碼應該覆蓋動畫的效果（比如說，讓角色的頭部朝著場景中的目標物件看）。LateUpdate 功能可用於這些情況。

```
void LateUpdate() {

    Camera.main.transform.LookAt(target.transform);
}
```

2. 初始化事件

能夠在遊戲過程中發生的任何更新之前呼叫初始化程式碼通常是有用的。Start 函數在物件的第一幀或物理更新之前被呼叫。當場景載入時，對場景中的每個物件呼叫 Awake 功能。請注意，儘管各種物件以任意順序呼叫 Start 和 Awake 功能，但所有的 Awake 都將在第一個 Start 被呼叫之前完成。這意味著 Start 函數中的程式碼可以預先在 Awake 階段執行其他初始化。

3. *GUI* 事件

Unity 具有用於在場景中的主要操作上呈現 GUI 控制以及回應這些控制點擊的系統。該程式碼的處理方式與正常的幀更新有所不同，因此它應該放在 OnGUI 函數中，將會被定期呼叫。

```
void OnGUI() {

    GUI.Label(labelRect, "Game Over");
}
```

還可以檢測出現在 GameObject 中的滑鼠事件。這可以用於定位武器或顯示當前在滑鼠指針下的訊息。一組 OnMouseXXX 事件函數（例如，OnMouseOver、OnMouseDown）可用於允許腳本使用滑鼠對使用者操作做出反應。例如，如果在特定物件上方按下滑鼠按鈕，則該物件腳本中的 OnMouseDown 函數將被調用（如果存在）。

4. 物理事件

物理引擎將通過呼叫該物件腳本的事件函數來反應物件的碰撞。OnCollisionEnter、OnCollisionStay 和 OnCollisionExit 函數將被呼叫做為建立、維持和斷開的聯繫。當物件的碰撞器被配置為觸發器（即，只是簡單地檢測到什麼東西進入而不做出反應的碰撞器）時，將呼叫相對應的 OnTriggerEnter、OnTriggerStay 和 OnTriggerExit 函數。如果在物理更新期間檢測到多個聯繫，則可以連續呼叫這些功能，因此參數將傳遞給函數提供碰撞的細節（如進入物件的位置等）。

```
void OnCollisionEnter(Collision otherObj) {

    if (otherObj.gameObject.tag == "Arrow") {

        ApplyDamage(10);
    }
}
```

XXXVII.時間與幀率管理

Update 功能讓你能夠監測輸入和定期來自腳本的事件，並採取適當的行為。例如，當按下「前進」鍵時，可以移動角色。在處理基於時間的操作時，要記住的一件重要的事情是遊戲的幀率以及呼叫 Update 功能之間的時間長度不是恆定的。

舉例，思考一下將物體逐漸向前移動的任務，一次一幀。首先看起來可能只是將對象每幀移動一個固定的距離：

```
public class ExampleDeltaTime : MonoBehaviour {

    public float distancePerFrame;

    void Update() {

        transform.Translate(0, 0, distancePerFrame);
    }
}
```

然而，由於幀時間不是恆定的，所以物體將以不規則的速度移動。如果幀時間是 10 毫秒，那麼物件將以每秒 100 次的 distancePerFrame 的速度前進。但是，如果幀時間增加到 25 毫秒（由於 CPU 負載），所以說，它將只會向前邁進 40 次，因此涵蓋較少的距離。解決方案是透過你能夠從 Time.deltaTime 屬性讀取的幀時間縮放移動的大小：

```
public class ExampleDeltaTime : MonoBehaviour {

    public float distancePerSecond;

    void Update() {

        transform.Translate(0, 0, distancePerSecond * Time.deltaTime);
    }
}
```

請注意，現在是以 distancePerSecond 移動，而不是 distancePerFrame。隨著幀速率的變化，移動長度將相應地改變，因此物件的速度將會是恆定的。

1. 固定時間步驟 *Fixed Timestep*

與主幀更新不同，Unity 的物理系統確實能夠處理固定的時間步驟，這對於模擬的準確性和一致性很重要。在物理更新開始時，Unity 通過將固定時間步驟的值添加到最後一個物理更新結束的時間來設置「警報」。然後，物理系統將執行計算，直到警報熄滅。

你可以從 Time Manager 更改固定時間步驟大小，可使用 Time.fixedDeltaTime 屬性從腳本中讀取它的大小。請注意，對於時間步驟，較低的值將導致更頻繁的物理更新和更精確的模擬，但代價是增加 CPU 負載。你可能不需要更改預設的固定時間步驟，除非你對物理引擎需要很高要求。

2. 最大允許時間步驟 *Maximum Allowed Timestep*

固定的時間步驟保持物理模擬在真實時間上的準確性，但是會在遊戲大量使用物理的情況下帶來問題，遊戲幀速率也會變低（由於運行的大量物件）。主幀更新進程在頻繁的物理更新之間被「壓縮」，並且如果有很多進程，那麼會在單個幀期間進行多次物理更新。由於幀時間、物件位置和其他屬性在幀的開始處被凍結，所以圖形可能與更頻繁更新的物理不同步。

理所當然的，只有這麼多的 CPU 功率可用，但 Unity 有一個選項，讓你有效地減慢物理時間，讓幀處理趕上。Maximum Allowed Timestep 設置（在 Time Manager 中）限制了 Unity 在給定幀更新期間處理物理和 FixedUpdate 呼叫的時間量。如果幀更新花費的時間超過 Maximum Allowed Timestep，則物理引擎將「停止時間」，並使幀處理趕上。一旦幀更新完成，物理將恢復，因為它已經停止，沒有時間過時。這樣做的結果是，剛體不會像通常那樣即時地完全移動，但會稍微減慢。然而，物理學「時鐘」仍將跟踪它們，就像它們正常地移動一樣。物理時間的減慢通常是不明顯的，是遊戲效能可接受的折中做法。

3. 時間比例 *Time Scale*

對於特殊效果，如「子彈時間」，有時可以減慢遊戲時間的流逝，從而使動畫和腳本應對以較低的速度發生。此外，你有時可能會完全凍結遊戲時間，就像遊戲暫停時一樣。Unity 具有 Time Scale 屬性，可以控制多快的遊戲時間來相對應真實時間。如果 Time Scale 設置為 1.0，則遊戲時間與真實時間相同。設為 2.0 會使得 Unity 的時間速度快兩倍（即，動作將被加速），而 0.5 的值將使遊戲速度

降低到一半。零值將使時間完全停止。請注意，Time Scale 實際上並不會減慢執行速度，而是通過 Time.deltaTime 和 Time.fixedDeltaTime 更改報告給 Update 和 FixedUpdate 函數的時間步驟。當遊戲時間減慢時，Update 功能可能比平常更頻繁地呼叫，但是報告每幀的 deltaTime 步驟將減少。其他腳本功能不受 Time Scale 影響，例如，當遊戲暫停時，可以顯示具有正常互動功能的 GUI。

Time Manager 有一個屬性讓你可以全域設置 Time Scale，但從腳本使用 Time.timeScale 屬性設置該值，通常會更為有用。

```
public class ExampleTimeScale : MonoBehaviour {

    void Update(){

        if(Input.GetMouseButtonDown (0)){

            Pause ();

        }else if(Input.GetMouseButtonDown (1)){

            Resume ();
        }
    }

    void Pause() {

        Time.timeScale = 0;
    }

    void Resume() {

        Time.timeScale = 1;
    }
}
```

4. 奪取幀率 *Capture Framerate*

一種非常特殊的時間管理情況是你想要將遊戲記錄為影像的地方。由於儲存螢幕圖像的任務需要相當長的時間，因此如果在正常的遊戲過程中嘗試執行此操作，則遊戲幀率會大大降低。這將導致影像不能反映出遊戲的真實表現。

所幸，Unity 提供了一個 Capture Framerate 屬性，可以讓你解決這個問題。當該屬性的值設置為零以外的任何值時，遊戲時間將會減慢，幀更新將以精確的定期間隔發佈。幀之間的間隔等於 1 / Time.captureFramerate，因此如果該值設置

為 5.0，則每五分之一秒更新一次。隨著對幀速率的要求有效降低，就可以在 Update 功能中儲存螢幕截圖或執行其他操作：

```
public class ExampleCaptureFramerate : MonoBehaviour {

    public string folder = "ScreenshotFolder";
    public int frameRate = 25;

    void Start () {

        Time.captureFramerate = frameRate;

        System.IO.Directory.CreateDirectory(folder);
    }

    void Update () {

        if (Input.GetMouseButton (0)) {

            string name =
                string.Format ("{0}/{1:D04} shot.png",
                    folder, Time.frameCount);

            ScreenCapture.CaptureScreenshot (name);
        }
    }
}
```

雖然使用這種技術錄製的影像通常看起來非常好，但是當劇烈的放慢時，遊戲很難玩。可能需要使用 Time.captureFramerate 的值試驗，來給予充足的錄製時間，而不會使測試播放器的任務過度複雜化。

XXXVIII.協程 Coroutine（一）

當你呼叫一個函數時，它返回之前會運行完畢。這實際上意味著在功能中發生的任何動作都必須在單幀更新中進行；函數呼叫不能用於包含程序動畫或隨時間推移的一系列事件。例如，逐漸減少物件的 alpha 值（不透明度）直到完全看不見的任務。

```
public class ExampleCoroutine : MonoBehaviour {

    public SpriteRenderer spriteRenderer;

    void Update () {

        if(Input.GetButtonDown ("Fire1")){

            Fade ();
        }
    }

    private void Fade() {

        for (float f = 1f; f >= 0; f -= 0.1f) {

            Color color = spriteRenderer.color;
            color.a = f;

            spriteRenderer.color = color;
        }
    }
}
```

像這樣，淡入淡出功能將不會有你所期望的效果。為了使褪色可見，必須通過一系列幀來減少 alpha，以顯示被渲染的中間值。但是，該功能在單幀更新中全部執行了。中間值將永遠不會被看到，物件將立即消失。

通過將程式碼添加到逐幀執行漸變的 Update 功能，可以處理這種情況。然而，對於這種任務使用 Coroutine 通常更為方便。

Coroutine 就像一個功能，可以暫停執行並將控制權返回給 Unity，然後停留到下一個幀再繼續。Coroutine 的聲明如下：

```
private IEnumerator Fade() {

    for (float f = 1f; f >= 0; f -= 0.1f) {

        Color color = spriteRenderer.color;
        color.a = f;

        spriteRenderer.color = color;

        yield return null;
    }
}
```

它實質上是一個聲明具備回傳 IEnumerator 類型的函數，並且將 yield return 語句包含在函數本體的某個位置。yield return 這一行是執行暫停並在下一幀恢復的位置。要設置 Coroutine 運行，需要使用 StartCoroutine 函數：

```
using UnityEngine;
using System.Collections;

public class ExampleCoroutine : MonoBehaviour {

    public SpriteRenderer spriteRenderer;

    void Update () {

        if(Input.GetButtonDown ("Fire1")){

            StartCoroutine (Fade ());
        }
    }

    private IEnumerator Fade() {

        for (float f = 1f; f >= 0; f -= 0.1f) {

            Color color = spriteRenderer.color;
            color.a = f;

            spriteRenderer.color = color;

            yield return null;
        }
    }
}
```

你將注意到，淡化功能中的迴圈在 Coroutine 生命週期內保持正確的值。實際上，任何變數或參數都將在 yield 之間被正確保留。

預設情況下，Coroutine 在它的 yield 之後的下一幀恢復，但也可以使用 WaitForSeconds 採用時間延遲：

```
using UnityEngine;
using System.Collections;

public class ExampleCoroutine : MonoBehaviour {

    public SpriteRenderer spriteRenderer;

    void Update () {

        if(Input.GetButtonDown ("Fire1")){

            StartCoroutine (Fade ());
        }
    }

    private IEnumerator Fade() {

        for (float f = 1f; f >= 0; f -= 0.1f) {

            Color color = spriteRenderer.color;
            color.a = f;

            spriteRenderer.color = color;

            yield return new WaitForSeconds(.1f);
        }
    }
}
```

XXXIX.協程 Coroutine（二）

Coroutine 可以用於在一段時間內傳播效果的一種方式，而它也是種有用的最佳化。遊戲中的許多任務需要定期執行，最常見的方法是將它們包含在 Update 功能中。然而，這個功能通常會每秒呼叫多次。當任務不需要如此頻繁地重複時，可以將其放在 Coroutine 中以定期更新，而不是每一幀。這樣的例子可能是一個警告，如果敵人在附近，則警告玩家。程式碼可能如下所示：

```
private bool ProximityCheck() {

    for (int i = 0; i < enemies.Length; i++) {

        if (Vector3.Distance(transform.position,
            enemies[i].transform.position)
            < dangerDistance) {

            return true;
        }
    }

    return false;
}
```

如果有很多敵人，那麼每一幀都要呼叫這個功能可能會有很大的效能開銷。但是，你可以使用 Coroutine 每十分之一秒才呼叫一次。

```
private IEnumerator DoCheck() {

    while(true) {

        ProximityCheck();

        yield return new WaitForSeconds(.1f);
    }
}
```

這將大大減少執行的檢查次數，而對遊戲沒有任何明顯的影響。

以下是完整的範例程式碼：

```
using UnityEngine;
using System.Collections;

public class CheckEnemies : MonoBehaviour {

    public bool isProximity;
    public float dangerDistance;
    public GameObject[] enemies;

    void Start(){

        StartCoroutine (DoCheck ());
    }

    private bool ProximityCheck() {

        for (int i = 0; i < enemies.Length; i++) {

            if (Vector3.Distance(transform.position,
                enemies[i].transform.position)
                < dangerDistance) {

                return true;
            }
        }

        return false;
    }


    private IEnumerator DoCheck() {

        while(true) {

            isProximity = ProximityCheck();

            yield return new WaitForSeconds(.1f);
        }
    }
}
```

注意：當 MonoBehaviour 被停用時，Coroutine 不會被停止，而只能明確地將它銷毀。你可以使用 MonoBehaviour.StopCoroutine 和 MonoBehaviour.StopAllCoroutines 來停止 Coroutine。當 MonoBehaviour 被銷毀時，Coroutine 也會被停止。

XL.命名空間的使用

隨著專案越來越大，腳本數量的增加，腳本 Class 名稱之間發生衝突的可能性也越來越大。尤其是當幾個程式人員分別在遊戲的不同方面進行工作時，最終會將他們的工作合併到一個專案中。例如，一個程式人員可能正在編寫程式碼來控制主要玩家角色，而另一個程式人員也在編寫敵人的同等效果的程式碼。兩個程式人員都可以選擇稱他們的主要腳本 Class 名稱為 Controller，但是當它們在專案合併時，將會發生衝突。

在某種程度上，通過採用命名約定或者發現衝突時重新命名 Class 可以避免這個問題（例如，上面的 Class 可以給出像 PlayerController 和 EnemyController 這樣的名字）。然而，當有好幾個 Class 有衝突的名稱或使用這些名稱聲明變數時，這很麻煩，每個提到的舊 Class 名稱都必須要替換程式碼來編譯。

C# 語言提供了一個稱為命名空間的功能，以一種可靠的方式解決這個問題。命名空間只是個被參考的 Class 集合，以在 Class 名稱之上使用一個被選定的前綴。在下面的例子中，名為 Player 和 Enemy 的命名空間分別都有名為 Controller 的成員：

```
namespace Player {
    public class Controller : MonoBehaviour {
        ...
    }
}
namespace Enemy {
    public class Controller : MonoBehaviour {
        ...
    }
}
```

在程式碼中，這些 Class 分別被稱為 Player.Controller 和 Enemy.Controller。這比重新命名 Class 更好，因為命名空間可以圍繞現有的 Class 進行括號（即，不必單獨更改所有 Class 的名稱）。此外，即使相同命名空間的 Class 在不同的檔案中，你也可以在 Class 的任何位置使用括號中的命名空間部分。

可以在檔案頂部添加 using 指令來避免重複輸入命名空間前綴。

如果腳本還需要從不同的命名空間引用具有相同名稱的 Class，那麼仍然要使用前綴。如果 using 指令同時導入包含衝突 Class 名稱的兩個命名空間，則編譯器仍將報錯。

XLI.屬性的使用

屬性（Attribute）是可以放置在腳本中的 Class、特性（Property）或函數上方的標記，以指示特殊行為。例如，HideInInspector 屬性可以添加到欄位變數聲明之上，以防止其在 Inspector 視窗中顯示，即使它是公開的。在 C＃中，它包含在中括號內：

```
public class AttributeEx : MonoBehaviour {

    [HideInInspector]
    public float strength;
}
```

Unity 提供的腳本參考中列出了許多屬性可使用。還有在 .NET 庫中定義的屬性，有時在 Unity 程式碼中也會有用。

XLII. 了解自動記憶體管理

當創建物件，字串或陣列時，記憶體會被要求從稱為堆（Heap）的中央池分配來儲存它們。當該項目不再使用時，它曾經佔用的記憶體可以被回收並用於其他的東西。在過去，通常由程式人員通過適當的函數調用明確分配和釋放這些堆記憶體塊。 如今，像 Unity 的 Mono 引擎這樣的執行期系統會自動為你管理記憶體。自動記憶體管理需要比明確的分配/釋放更少的程式編寫工作，並大大減少記憶體洩漏的可能性（記憶體被分配但從未隨後釋放的情況）。

1. 實值與參考類型 *Value and Reference Types*

當呼叫函數時，會將其參數的值複製到為該特定呼叫保留的記憶體區域。只佔用幾個 Byte 的資料類型可以非常快速容易地複製。然而，通常物件，字串和陣列會大得多，如果這些類型的資料被頻繁的複製，效率會非常低。所幸，這不是必需的；從堆中分配大項目的實際儲存空間，並使用小的「指針（pointer）」值來記住其位置。從此，在參數傳遞過程中只需要複製指針。只要執行期系統可以定位由指針標識的項目，資料的單一副本就可以如往常所需要的來使用。

在參數傳遞期間直接儲存和複製的類型稱為實值類型。這些包括整數、浮點數、布林值和 Unity 的結構物件（例如 Color 和 Vector3）。分配在堆上然後通過指針訪問的類型稱為參考類型，因為儲存在變數中的值僅僅是「參考」到實際資料。參考類型包括類物件、字串和陣列。

2. 分配與垃圾回收 *Allocation and Garbage Collection*

記憶體管理器保持追踪它所知道未被使用的堆中的區域。當請求新的記憶體塊時（例如，當物件被實例化時），管理器從所選擇的未使用區域來分配記憶體塊，然後從已知的未使用空間中移除被分配的記憶體。連串的請求以相同方式處理，直到沒有足夠大的自由空閒區域來分配所需的塊大小。在這一點上，從堆中分配的所有記憶體都仍然在使用中是非常不可能的。一個在堆上的參考項目只要仍然有參考變數可以定位它就可以被訪問到。如果對記憶體塊的所有參考都消失了

（即，參考變數已被重新分配，或者它們現在是超出範圍的區域變數），那麼它佔用的記憶體就可以安全地被重新分配。

要確定哪些堆塊不再使用，記憶管理器將搜索所有當前活動的參考變數，並將其標識為「活的」。在搜索結束時，被記憶體管理器認定活的塊之間的任何空間是空的，就會被用於後續的分配。顯而易見的，找出並釋放未使用的記憶體的過程稱為垃圾回收（或簡稱GC）。

3. 最佳化 *Optimization*

垃圾回收對於程式人員來說是自動且看不見的，但回收過程實際上需要在背後花費大量的 CPU 時間。正確使用時，自動記憶體管理普遍會均等分配整體效能。但是，對於程式人員來說，重要的是要避免過多觸發回收器而導致執行暫停的錯誤。

有一些臭名昭著的算法會是 GC 惡夢，即使它們看起來沒什麼。重複的字串連接是個典型的例子：

```
private void ConcatExample(int[] intArray) {
    string line = intArray[0].ToString();
    for (int i = 1; i < intArray.Length; i++) {
        line += ", " + intArray[i].ToString();
    }
    return line;
}
```

這裡的關鍵細節是，新的片段不會被逐個添加到字串中。實際發生的是，每次循環時，line 變數的先前內容都將變為無效的 - 一個全新的字串被分配以包含原始的部分加上最後的部分。由於字串隨著 i 的增加值而變長，所以佔用的堆空間量也會增加，因此每次呼叫該函數時，都容易的就使用上百個 Byte 的堆空間。如果你需要連接多個字串，那麼一個更好的選擇是 Mono 庫的 System.Text.StringBuilder 類。

然而，如果不被頻繁的呼叫，即使重複連接也不會引起太多麻煩，而在 Unity 中通常隱含著幀更新。就像是：

```
using UnityEngine;
using UnityEngine.UI;

public class Example : MonoBehaviour {

    public Text scoreBoard;
    public int score;

    void Update() {

        string scoreText = "Score: " + score.ToString();

        scoreBoard.text = scoreText;
    }
}
```

每次呼叫 Update 時都會分配新的字串，並不斷生成新垃圾。這大多數可以透過只有當分數改變時的更新文字來儲存：

```
using UnityEngine;
using UnityEngine.UI;

public class Example : MonoBehaviour {

    public Text scoreBoard;
    public string scoreText;
    public int score;
    public int oldScore;

    void Update() {

        if (score != oldScore) {

            scoreText = "Score: " + score.ToString();
            scoreBoard.text = scoreText;
            oldScore = score;
        }
    }
}
```

當函數回傳陣列值時，會發生另一個潛在的問題：

```
private float[] RandomList(int numElements) {

    float[] result = new float[numElements];

    for (int i = 0; i < numElements; i++) {

        result[i] = Random.value;
    }

    return result;
}
```

當創建一個充滿值的新陣列時，這種類型的函數非常優雅和方便。但是，如果重複呼叫，則每次都會分配新的記憶體。由於陣列可能非常大，可用堆空間可能會迅速消耗，導致垃圾回收頻繁。避免這個問題的一個方法是利用陣列是參考類型的事實。做為參數傳遞給函數的陣列可以在該函數內修改，結果將在函數回傳後保留。像上面這樣的功能通常可以被替換為：

```
private void RandomList(float[] arrayToFill) {

    for (int i = 0; i < arrayToFill.Length; i++) {

        arrayToFill[i] = Random.value;
    }
}
```

這只是用新的值替換陣列的現有內容。雖然這需要在函數程式碼中完成陣列的初始分配，但是在呼叫該函數時不會產生任何新的垃圾。

4. 請求回收

如上所述，最好盡量避免分配。但是，鑑於不能完全消除這些問題，你可以使用兩種主要策略來最大限度地減少它們對遊戲遊玩的侵擾：

- 快速頻繁的小堆垃圾回收：

  這個策略通常最適合長時間遊玩的遊戲，其中平穩的幀速率是主要的關注點。這樣的遊戲通常會頻繁地分配小塊，但這些塊將僅在短時間內使用。在iOS 上使用此策略時，典型的堆大小約為200KB，iPhone 3G 上的垃圾回收大約需要 5ms。 如果堆增加到1MB，則收集大約需要7ms。因此有時可以以規則的

幀間隔請求垃圾回收。這通常會使回收發生的次數比絕對必要的更多，但它們將被快速處理，對遊戲的影響最小：

```
if (Time.frameCount % 30 == 0){

    System.GC.Collect();
}
```

但是，你應該謹慎使用此技術，並檢查 Profiler 統計訊息，以確保它真正減少了遊戲的回收時間。

- 緩慢但不常見的大堆垃圾回收：

  這種策略對於遊戲分配相對不頻繁並且可以在遊戲暫停期間處理的遊戲最適合。它對於盡可能大的堆是有用的，對於由於系統記憶體不足而使你的應用程式被作業系統關掉而言並不大。然而，如果可能，Mono 運行時會自動避免擴張堆。你可以通過在啟動期間預先分配一些佔位符空間來手動擴展堆（即，實例化一個純粹為了對記憶體管理器影響分配的「無用」物件）：

```
void Start() {

    System.Object tmp = new System.Object[1024];

    //在較小的塊中進行分配，以避免用特殊方式對待它們，這是專為大塊而設計的
    for (int i = 0; i < 1024; i++){

        tmp[i] = new byte[1024];
    }

    // 釋放參考
    tmp = null;
}
```

  一個足夠大的堆不應該在可容納一個回收的遊戲進行中的暫停期間被完全填滿。當暫停發生時，你可以明確請求回收：

```
System.GC.Collect();
```

  同樣的，你應該在使用此策略時注意並關注 Profiler 統計訊息，而不僅僅是假設它具有預期的效果。

5. 可重複使用的物件池

在許多情況下，你可以通過減少創建和銷毀的物件數量來避免生成垃圾。遊戲中存在某些類型的物件，例如投射物，儘管只會有一小部分會立刻執行，但可能會一再遇到。在這種情況下，通常可以重複使用物件而不是破壞舊的物件，並將其替換為新物件。

XLIII.平台相依編譯與定義

Unity 包含一個名為 Platform Dependent Compilation 的功能。 這包括一些預處理器指令，可以讓你分割腳本來編譯和執行專用於一個受支援平台的程式碼。

可以在 Unity 編輯器中運行此程式碼，因此可以專門為你的目標平台編譯程式碼，並在編輯器中進行測試！

1. 平台定義指令

Unity 支援你腳本的平台定義指令如下：

- UNITY_EDITOR：用於從你的遊戲程式碼呼叫 Unity Editor 腳本。
- UNITY_EDITOR_WIN：用於 Windows 的 Editor 程式碼。
- UNITY_EDITOR_OSX：用於 Mac OS X 的 Editor 程式碼。
- UNITY_STANDALONE_OSX：用於特別提供 Mac OS X（包含 Universal、PPC 和 Intel 架構）編譯 / 執行程式碼。
- UNITY_STANDALONE_WIN：用於特別提供 Windows 單機應用程式編譯 / 執行程式碼。
- UNITY_STANDALONE_LINUX：用於特別提供 Linux 單機應用程式編譯 / 執行程式碼。
- UNITY_STANDALONE：用於特別提供單機平台（Mac OSX、Windows 或 Linux）編譯 / 執行程式碼。
- UNITY_WII：用於 Wii 遊戲機編譯 / 執行程式碼。
- UNITY_IOS：用於 iOS 平台編譯 / 執行程式碼。
- UNITY_ANDROID：用於 Android 平台。
- UNITY_PS4：用於運行 PlayStation 4 程式碼。
- UNITY_SAMSUNGTV：用於執行 Samsung TV 程式碼。
- UNITY_XBOXONE：用於執行 Xbox One 程式碼。

- UNITY_TIZEN：用於 Tizen 平台。
- UNITY_TVOS：用於 Apple TV 平台。
- UNITY_WSA：用於通用 Windows 平台。另外，NETFX_CORE 是根據 .NET Core 編譯 C# 檔案並使用 .NET 腳本後端定義的。
- UNITY_WSA_10_0：用於通用 Windows 平台。 另外 WINDOWS_UWP 是針對 .NET Core 編譯 C# 檔案時定義的。
- UNITY_WINRT：等同於 UNITY_WSA。
- UNITY_WINRT_10_0：等同於 UNITY_WSA_10_0。
- UNITY_WEBGL：用於 WebGL。
- UNITY_ADS：用於從你的遊戲程式碼呼叫 Unity Ads 方法。版本 5.2 及以上。
- UNITY_ANALYTICS：用於從你的遊戲程式碼呼叫 Unity Analytics 方法。版本 5.2 及以上。
- UNITY_ASSERTIONS：用於中斷控制處理。

從 Unity 2.6.0 起，可以有選擇地編譯程式碼。可用的選項取決於你正在編輯的編輯器版本。給定版本號 X.Y.Z（例如2.6.0），Unity 用以下格式公開了三個全域定義指令：UNITY_X，UNITY_X_Y 和 UNITY_X_Y_Z。

以下是 Unity 5.0.1 中定義指令的範例：

- UNITY_5：用於發佈版本 Unity 5，包含每個 5.X.Y 版本。
- UNITY_5_0：用於主要版本 Unity 5.0，包含每個 5.0.Z 版本。
- UNITY_5_0_1：用於 Unity 5.0.1 的次要版本。

從 Unity 5.3.4 開始，你可以根據更早版本的 Unity 所需要編譯或執行所給予的部分程式碼，選擇性的編譯程式碼。給定與上述相同的版本格式（X.Y.Z），Unity 以格式 UNITY_X_Y_OR_NEWER 顯示一個全域定義，可以用於此目的。

支援的定義指令是：

- ENABLE_MONO：提供 Mono 的腳本後端。

- ENABLE_IL2CPP：提供 IL2CPP 的腳本後端。
- ENABLE_DOTNET：提供 .NET 的腳本後端。
- NETFX_CORE：在 .NET 上針對 .NET Core 類庫構建腳本時定義。
- NET_2_0：在 Mono 和 IL2CPP 針對 .NET 2.0 API 相容級構建腳本時定義。
- NET_2_0_SUBSET：在 Mono 和 IL2CPP 上針對 .NET 2.0 子集 API 相容級別構建腳本時定義。
- NET_4_6：在 Mono 和 IL2CPP 針對 .NET 4.6 API 相容級構建腳本時定義。
- ENABLE_WINMD_SUPPORT：當 Windows Runtime 在 IL2CPP 和 .NET 上被啟動時定義。

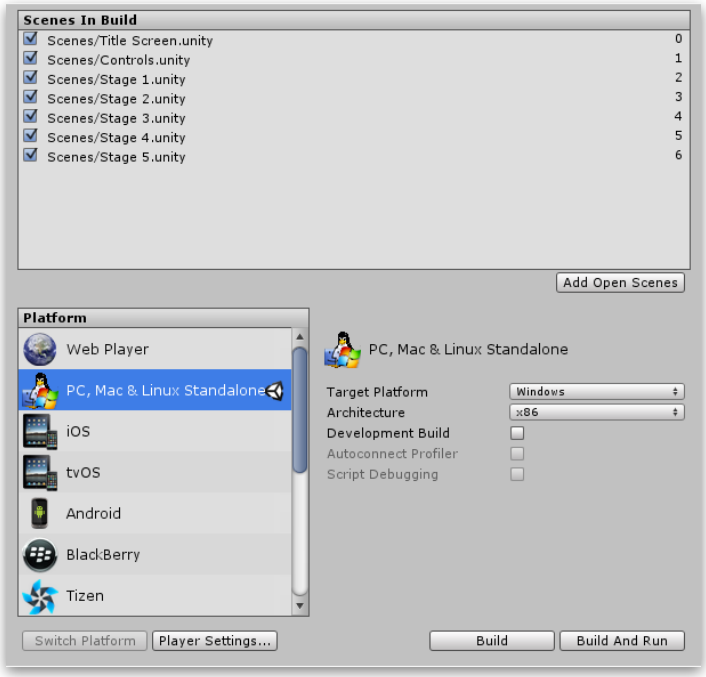
你可以使用 DEVELOPMENT_BUILD 來識別你的腳本是否在啟用 Development Build 選項構建中運行。

你還可以根據腳本後端選擇性地編譯代碼。

2. 測試預定義編碼

以下是如何使用預編譯編碼的範例。它列出一個取決於你為你的目標構建選擇的平台的訊息。

首先，選擇要測試程式碼的平台，到 File > Build Settings 開啟 Build Settings 視窗；從這裡選擇目標平台。



選擇要測試你的預編譯編碼的平台，然後點擊 Switch Platform，告訴 Unity 你想要要指向的平台。

使用以下程式碼建立腳本：

```
public class PlatformDefines : MonoBehaviour {

    void Start () {

        #if UNITY_EDITOR
        Debug.Log("Unity Editor");
        #endif

        #if UNITY_IOS
        Debug.Log("Iphone");
        #endif

        #if UNITY_STANDALONE_OSX
        Debug.Log("Stand Alone OSX");
        #endif

        #if UNITY_STANDALONE_WIN
        Debug.Log("Stand Alone Windows");
        #endif
    }
}
```

要測試程式碼，請點擊播放模式。根據你選擇的平台，在 Unity Console 視窗中檢查相關消息，確認程式碼的工作原理 - 例如，如果你選擇 iOS，則會在 Console 視窗中顯示訊息「Iphone」。

請注意，在 C# 中，可以使用一個 Conditional 屬性，他是更乾淨、更不容易出錯的剝離功能的方法。 有關詳細信息，請參閱 http://msdn.microsoft.com/en-us/library/4xssyw96(v=vs.90).aspx。

```
public class PlatformDefines : MonoBehaviour {

    void Start () {

        EditorMessage ();
        IOSMessage ();
        StandaloneOSXMessage ();
        StandaloneWinMessage ();
    }
```

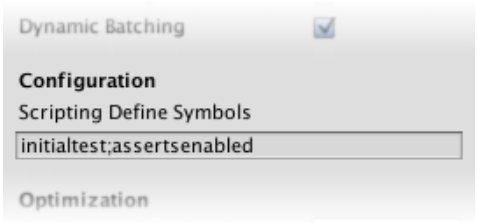
```
    [System.Diagnostics.Conditional("UNITY_EDITOR")]
    private void EditorMessage(){

        Debug.Log("Unity Editor");
    }

    [System.Diagnostics.Conditional("UNITY_IOS")]
    private void IOSMessage(){

        Debug.Log("Iphone");
    }

    [System.Diagnostics.Conditional("UNITY_STANDALONE_OSX")]
    private void StandaloneOSXMessage(){

        Debug.Log("Stand Alone OSX");
    }

    [System.Diagnostics.Conditional("UNITY_STANDALONE_WIN")]
    private void StandaloneWinMessage(){

        Debug.Log("Stand Alone Windows");
    }
}
```

除了基本的 #if 編譯器指令之外，還可以在使用多向測試：

```
public class PlatformDefines : MonoBehaviour {

    void Start () {

        #if UNITY_EDITOR
        Debug.Log("Unity Editor");

        #elif UNITY_IOS
        Debug.Log("Unity iPhone");

        #else
        Debug.Log("Any other platform");

        #endif
    }
}
```

3. 平台自定義

還可以為內建的定義指令提供自己定義的指令。打開 Player Settings 的 Other Settings 面板，並找到 Scripting Define Symbols 文字欄位。



輸入要為該特定平台定義的符號名稱，以分號分隔。這些符號可以用於 #if 指令的條件，就像內建的一樣。

XLIV.特殊資料夾與編譯順序

Unity 保留一些專案資料夾名稱，以表明內容具有特殊用途。這些資料夾名稱為：

- Assets
- Editor
- Editor default resources
- Gizmos
- Plugins
- Resources
- Standard Assets
- StreamingAssets

其中一些資料夾對腳本編譯的順序有影響。

腳本編譯有四個不同的階段。編譯腳本的階段由其父資料夾決定。

在腳本必須參考其他腳本中定義的類的情況下，這是很重要的。基本規則是在當前編譯之後的任何內容都不能被參考到。在當前階段或更早階段編譯的任何內容都是完全可用的。

當用一種語言編寫的腳本必須參考另一種語言定義的類（例如，有個 UnityScript 檔案所聲明變數的類別定義於 C# 腳本中）時，會發生另外的情況。這裡的規則是被參考的類必須在較早的階段被編譯。

編譯階段如下：

- 第 1 階段：名為 Standard Assets、Pro Standard Assets 和 Plugins 的資料夾中的執行期腳本。
- 第 2 階段：位於名為 Standard Assets、Pro Standard Assets 和 Plugins 的頂級文件夾中任何位置的名為 Editor 的資料夾中的編輯器腳本。

- 第 3 階段：不在名為 Editor 的資料夾內的所有其他腳本。
- 第 4 階段：所有剩餘的腳本（位於名為 Editor 的資料夾內）。

當 UnityScript 檔案需要參考 C# 檔案中定義的類時，會發生此順序的重要性的常見例子。要實現這一點，你需要將 C# 檔案放在 Plugins 資料夾中，將 UnityScript 檔案放在非特殊資料夾中。如果你不這樣做，會出現錯誤指出找不到 C# 類。

注意：Standard Assets 僅在 Assets 根資料夾中運作。

XLV.腳本序列化

序列化是將資料結構或物件狀態轉換為 Unity 可以在以後儲存和重建的格式的自動過程。Unity 的一些內建功能使用序列化；如儲存和載入、Inspector 視窗、實例化以及 Prefab。

如何在 Unity 專案中組織資料會影響 Unity 如何序列化資料，並可能對專案的效能產生重大影響。以下是 Unity 中序列化的一些指導，以及如何最佳化你的專案。

1. 了解熱重新載入

- 熱重新載入

  熱重新載入是編輯器開啟期間創建或編輯腳本並立即套用腳本行為的過程。你不必為了使改變生效而重新啟動遊戲或編輯器。

  當你更改並儲存腳本時，Unity 熱重新載入所有當前已載入的腳本資料。它首先儲存所有已載入的腳本中的所有可序列化的變數，並且在載入腳本後，還原它們。所有不可序列化的資料在熱重新載入後都會丟失。

- 儲存與載入

  Unity 使用序列化從你的電腦硬碟中載入和儲存場景、資源以及 AssetBundle。這包括被儲存在你自己的腳本 API 物件中的資料，如 MonoBehaviour 組件和 ScriptableObject。

  Unity 編輯器中的許多功能構建在核心序列化系統之上。使用序列化所特別要注意到的兩個部分是 Inspector 視窗和熱重新載入。

- Inspector 視窗

  當你在 Inspector 視窗中查看或更改 GameObject 組件欄位的值時，Unity 將對此資料進行序列化，然後將其顯示在 Inspector 視窗中。當 Inspector 視窗顯示欄位的值時，不會與 Unity Scripting API 相通。

  如果在腳本中使用屬性，則當查看或更改 Inspector 視窗中的值時，任何屬性 getter 和 setter 都不會被呼叫，因為 Unity 直接序列化了 Inspector 視窗欄

位。這意味著：雖然 Inspector 視窗中的欄位值代表腳本屬性，但是 Inspector 視窗中值的更改不會在腳本中呼叫任何屬性 getter 和 setter。

2. 序列化規則

Unity 中的序列化器運行於即時遊戲環境中。這對效能有重大影響。因此，Unity 中的序列化與其它程式設計環境中的序列化不同。下面列出了有關如何在 Unity 中使用序列化的一些提示。

- 如何確保腳本中的欄位是序列化的

  確保：

  - 是公開的，或者有 SerializeField 屬性。
  - 不是靜態的。
  - 不是常數。
  - 不是唯讀。
  - 具備可被序列化的欄位類型。

- 可被序列化的簡單欄位類型
  - 具備 Serializable 屬性的自定義非抽象、非泛型類別。
  - 具備 Serializable 屬性的自定義結構（Struct）。
  - 參考繼承自 UnityEngine.Object 的物件。
  - 原始資料型態（int、float、double、bool、string 等）。
  - Enum 類型。
  - 某些 Unity 內建的類型：Vector2、Vector3、Vector4、Rect、Quaternion、Matrix4x4、Color、Color32、LayerMask、AnimationCurvr、Gradient、RectOffset、GUIStyle 等。

- 可被序列化的欄位類型容器
  - 可被序列化的簡單欄位類型的陣列。
  - 可被序列化的簡單欄位類型的 List<T>。

注意：Unity 不支援多層次類型（多維陣列、不規則陣列和巢狀類型）的序列化。

如果要序列化這些，有兩個選項：將巢狀類型包裝在類或結構中，或使用序列化回調 ISerializationCallbackReceiver 來執行自定義序列化。

- 如何確保自定義類可被序列化

  確保：

  - 具有 Serializable 屬性。
  - 不是抽象類。
  - 不是靜態的。
  - 不是泛型，儘管可能從泛型類繼承。

3. 序列化器何時可能出現意外？

- 表現像結構的自定義類

  使用不是繼承自 UnityEngine.Object 的自定義類，Unity 將它們透過實值來串聯序列化，與序列化結構的方式類似。如果在多個不同的欄位中儲存對自定義類實例的引用，則在序列化時它們將成為單獨的物件。然後，當 Unity 反序列化欄位時，它們會包含具有相同資料的不同物件。

  當你需要使用參考序列化複雜物件圖時，請勿讓 Unity 自動序列化物件。而是使用 ISerializationCallbackReceiver 手動序列化它們。這樣可以防止 Unity 從物件參考中創建多個物件。

  這只適用於自定義類。Unity 將自定義類「內聯」序列化，因為它們的資料成為其使用的 MonoBehaviour 或 ScriptableObject 的完整序列化資料的一部分。當欄位參考某個繼承自 UnityEngine.Object 的類（例如 public Camera myCamera）時，Unity 將序列化實際參考到 Camera 的 UnityEngine.Object。如果腳本來自於 MonoBehaviour 或 ScriptableObject，這兩個腳本都是從 UnityEngine.Object 繼承的，那麼腳本的實例也是如此。

- 不支援自定義類的 null

  考慮在使用以下腳本的 MonoBehaviour 反序列化時，需要多少分配。

```
public class Example : MonoBehaviour {

    public Trouble t;
}

[System.Serializable]
class Trouble{

    public Trouble t1;
    public Trouble t2;
    public Trouble t3;
}
```

預期一個分配並不奇怪：Example 物件。預期有兩個分配也是不奇怪的：一個用於 Example 物件，一個用於 Trouble 物件。

然而，Unity 實際上已經有超過一千個分配。序列化器不支援 null。如果序列化一個物件，並且欄位為空，則 Unity 將實例化該類型的新物件，並將其序列化。顯然這可能導致無限循環，所以有七層的深度限制。此時，Unity 停止序列化具有自定義類、結構、列表或陣列類型的欄位。

由於這麼多 Unity 的子系統構建在序列化系統之上，所以 Example MonoBehaviour 的這個意想不到的大型序列化流，使所有這些子系統的運行速度較慢。

- 不支援多型

  如果你有個 public Animal[] animals，並且在序列化後，將狗、貓和長頸鹿的實例放在一起，將有三個 Animal 實例。

  處理這個限制來實現的一種方法是只適用於被內聯序列化的自定義類。參考到其他 UnityEngine.Object 得到的被序列化做為實際參考，對於那些，多型確實有效。你將創建一個繼承 ScriptableObject 的類或另一個繼承 MonoBehaviour 的類，並參考它。這樣做的缺點是你需要將 Monobehaviour 或 ScriptableObject 的物件存儲在某處，並且無法有效地將其序列化。

  這些限制的原因是序列化系統的核心基礎之一是提前知道物件的資料流佈局：它取決於類的欄位類型，而不是欄位中儲存的內容。

4. 改善序列化的使用

你可以組織資料，以確保你從 Unity 的序列化獲得最佳使用。

- 組織你的資料，目的是讓 Unity 序列化最小的資料集。 這樣做的主要目的不是為了節省電腦硬碟空間，而是要確保與以前版本的專案保持向後相容。 如果你使用大量的序列化資料，向後相容性在後續的開發中變得更加困難。
- 組織你的資料，從不將 Unity 序列化為副本資料或快取資料。這會導致向後相容性的重大問題：它具有高風險的錯誤，因為資料太容易失去同步。
- 在你參考其他類的地方，避免巢狀、遞歸結構。序列化結構的佈局總是需要相同；資料的獨立，僅依賴於腳本中暴露的內容。參考其他物件的唯一方法是透過從 UnityEngine.Object 繼承的類。 這些類是完全分開的；他們只是互相參考，不嵌入內容。

5. 使編輯器程式碼可熱重載

當重新載入腳本時，Unity 將所有被載入的腳本中的所有變數序列化並儲存。重新載入腳本後，Unity 將其恢復為原始的預序列化值。

當重新載入腳本時，即使變數沒有 SerializeField 屬性，Unity 也會恢復滿足序列化要求的所有變數（包括私有變數）。在某些情況下，你特別需要防止修復私有變數：例如，如果你希望在從腳本重新載入後參考為 null。 在這種情況下，請使用 NonSerializable 屬性。

Unity不會恢復靜態變數，所以不要在重新載入腳本後需要保留的狀態使用靜態變數。

6. 自定義序列化

序列化是將資料結構或物件狀態轉換為 Unity 可以在之後儲存和重建的格式的自動處理。

有時你可能想序列化 Unity 的序列化器不支援的東西。在許多情況下，最好的方法是使用序列化回調（Callback）。

序列化回調允許你在序列化器從欄位讀取資料之前以及在寫入完畢之後通知它們當你實際序列化時，可以使用序列化回調在運行時為你的難以序列化的資料提供不同的表示形式。

要做到這一點，要在 Unity 序列化之前，將你的資料轉換成 Unity 所理解的東西。然後，在 Unity 將資料寫入你的欄位之後，可以將序列化格式轉換為你想要在運行時具有資料格式。

例如：你想要有個樹資料結構。如果讓 Unity 直接序列化資料結構，那麼「不支持 null」的限制會導致資料流變得非常大，導致許多系統的效能下降。如下所示：

```
public class Example : MonoBehaviour {

    [System.Serializable]
    public class Node {
        public string interestingValue = "value";

        //下面的欄位使序列化資料變得巨大，因為它引入了「類循環」。
        public List<Node> children = new List<Node>();
    }

    //這被序列化了
    public Node root = new Node();

    void OnGUI() {

        Display (root);
    }

    void Display(Node node) {

        GUILayout.Label ("Value: ");
        node.interestingValue =
            GUILayout.TextField(node.interestingValue,
                GUILayout.Width(200));

        GUILayout.BeginHorizontal ();

        GUILayout.Space (20);

        GUILayout.BeginVertical ();

        foreach (Node child in node.children)
            Display (child);

        if (GUILayout.Button ("Add child"))
            node.children.Add (new Node ());

        GUILayout.EndVertical ();
        GUILayout.EndHorizontal ();
    }
}
```

相反，告訴 Unity 不要直接序列化樹，並且製作一個單獨的欄位以適合 Unity 序列化器的序列化格式來儲存樹。如下所示：

```
public class Example : MonoBehaviour ,
                       ISerializationCallbackReceiver{

    // 在運行時使用的 Node 類。這是 Example 類的內部，沒有序列化。
    public class Node {

        public string interestingValue = "value";
        public List<Node> children = new List<Node>();
    }

    // 將用於序列化的 Node 類。
    [System.Serializable]
    public struct SerializableNode {

        public string interestingValue;
        public int childCount;
        public int indexOfFirstChild;
    }

    // 用於運行時表示樹的根節點，沒有序列化。
    private Node root = new Node();

    // 這是我們給 Unity 進行序列化的欄位。
    public List<SerializableNode> serializedNodes;

    public void OnBeforeSerialize() {

        // Unity 即將讀取 serializedNodes 欄位的內容。
        // 正確的資料必須現在寫入到該欄位。
        if (serializedNodes == null)
            serializedNodes = new List<SerializableNode> ();

        if (root == null)
            root = new Node ();

        serializedNodes.Clear();

        AddNodeToSerializedNodes(root);
        // 現在，Unity 可以自由地序列化這個欄位，當我們稍後反序列化時，
        // 我們應該會收回預期的資料。
    }
```

```
private void AddNodeToSerializedNodes(Node n) {

    SerializableNode serializedNode = new SerializableNode () {
        interestingValue = n.interestingValue,
        childCount = n.children.Count,
        indexOfFirstChild = serializedNodes.Count+1
    };

    serializedNodes.Add (serializedNode);

    foreach (var child in n.children)
        AddNodeToSerializedNodes (child);
}

public void OnAfterDeserialize() {

    // Unity剛剛將資料寫入 serializedNodes 欄位。
    // 讓我們用這些新值填充我們實際的運行時資料。
    if (serializedNodes.Count > 0) {
        ReadNodeFromSerializedNodes (0, out root);
    } else
        root = new Node ();
}

private int ReadNodeFromSerializedNodes(int index, out Node node) {

    SerializableNode serializedNode = serializedNodes [index];

    // 將反序列化資料傳到內部 Node 類
    Node newNode = new Node() {
        interestingValue = serializedNode.interestingValue,
        children = new List<Node> ()
    };

    for (int i = 0; i != serializedNode.childCount; i++) {
        Node childNode;
        index = ReadNodeFromSerializedNodes (++index, out childNode);
        newNode.children.Add (childNode);
    }

    node = newNode;
    return index;
}
```

```
    void OnGUI() {

        if (root != null)
            Display (root);
    }

    private void Display(Node node) {

        GUILayout.Label ("Value: ");

        node.interestingValue =
            GUILayout.TextField(node.interestingValue,
                GUILayout.Width(200));

        GUILayout.BeginHorizontal ();

        GUILayout.Space (20);

        GUILayout.BeginVertical ();

        foreach (var child in node.children)
            Display (child);

        if (GUILayout.Button ("Add child"))
            node.children.Add (new Node ());

        GUILayout.EndVertical ();
        GUILayout.EndHorizontal ();
    }
}
```

XLVI.UnityEvent 欄位的應用

UnityEvent 是一種允許使用者從編輯期持續到執行期驅動回調的方式，而不需要額外的程式編寫和腳本配置。

UnityEvent 對於一些事情很有用：

- 內容驅動回調
- 去耦系統
- 持續回調
- 預配置的呼叫事件

UnityEvent 可以添加到任何 MonoBehaviour，並從程式碼裡像標準的 .net 委派那樣執行。當 UnityEvent 添加到 MonoBehaviour 時，它會顯示在 Inspector 視窗中，並且可以添加持續回調。

UnityEvent 對標準委派具有類似的限制。也就是說，它們保持對目標元素的引用，並且停止目標被垃圾回收。如果你有一個 UnityEngine.Object 作為目標，而本體表示消失，回調則不會被調用。

1. 使用 *UnityEvent*

要在編輯器中配置回調，需要執行以下幾個步驟：

(1) 確保你的腳本有使用 UnityEngine.Events。

(2) 選擇加號（+）圖標添加回調的插槽。

(3) 選擇你希望接收回調的 UnityEngine.Object（可以使用物件選擇器）。

(4) 選擇你要呼叫的功能。

(5) 你可以為事件添加多個回調。

在 Inspector 視窗中配置 UnityEvent 時，有兩種類型的函數呼叫被支持：

- 靜態：靜態呼叫是預配置的呼叫，具有在 UI 中設置的預配置值。這意味著當回調被調用時，目標函數使用已經被輸入到 UI 中的參數來調用。
- 動態：動態呼叫使用從程式碼中發送的參數來調用，並將其綁定到正在調用的 UnityEvent 類型。UI 過濾回調，並只顯示對 UnityEvent 有效的動態呼叫。

2. 泛用 *UnityEvent*

預設情況下，MonoBehaviour 中的 UnityEvent 動態綁定到一個 void 函數。這不一定是這種情況，因為 UnityEvent 的動態調用支持綁定到最多 4 個參數的函數。為此，你需要定義一個支持多個參數的自定義 UnityEvent 類。這很容易做到：

```
using UnityEngine;
using UnityEngine.Events;

public class ExampleUnityEvent : MonoBehaviour {

    [System.Serializable]
    public class StringEvent : UnityEvent <string> {}

    public StringEvent show;

    void Update(){

        if(Input.GetButtonDown ("Fire1")){

            show.Invoke (Time.time.ToString ());
        }

    }
}
```

透過將這個實例添加到你的類取代基本的 UnityEvent，它將允許回調動態綁定到字串函數。

這可以通過使用字串作為參數呼叫 Invoke() 函數來調用。

UnityEvent 可以使用在它們的泛型定義中的最多 4 個參數來定義。

3. *UnityEvent* 功能

- Invoke：調用所有執行時或編輯時所註冊的回調。

```
show.Invoke (Time.time.ToString ());
```

- AddListener：添加監聽器給 UnityEvent，使用它來添加運行時的回調。

```
public void AddShowMessage(){

    show.AddListener (ShowMessage);
}

private void ShowMessage(string message){

    Debug.Log (message);
}
```

- RemoveListener：從 UnityEvent 移除監聽器，使用它來移除運行時的回調。

```
public void RemoveMessage(){

    show.RemoveListener (ShowMessage);
}
```

4. UnityEventBase 功能

- GetPersistentEventCount：取得於 Inspector 視窗上註冊的持續監聽器的數量。
- GetPersistentMethodName：使用索引取得 Inspector 視窗上註冊的持續監聽器的目標功能名稱。
- GetPersistentTarget：使用索引取得 Inspector 視窗上註冊的持續監聽器的目標組件。

```
public void ShowPersistent(){

    int count = show.GetPersistentEventCount ();

    for(int i = 0 ; i < count ; i++){

        string name = show.GetPersistentMethodName (i);
        Object target = show.GetPersistentTarget (i);

        Debug.Log (name);
        Debug.Log (target);
    }
}
```

- RemoveAllListeners：從事件中移除所有非持續監聽器（從腳本創建），所有的持續監聽器（從 Inspector 視窗創建）不受影響。

```
public void RemoveAllListeners(){

    show.RemoveAllListeners ();
}
```

```
using UnityEngine;
using UnityEngine.Events;

public class ExampleUnityEvent : MonoBehaviour {

    [System.Serializable]
    public class StringEvent : UnityEvent <string> {}

    public StringEvent show;

    void Update(){

        if(Input.GetButtonDown ("Fire1")){

            show.Invoke (Time.time.ToString ());
        }

    }

    public void AddShowMessage(){

        show.AddListener (ShowMessage);
    }

    private void ShowMessage(string message){

        Debug.Log (message);
    }

    public void RemoveMessage(){

        show.RemoveListener (ShowMessage);
    }

    public void ShowPersistent(){

        int count = show.GetPersistentEventCount ();

        for(int i = 0 ; i < count ; i++){

            string name = show.GetPersistentMethodName (i);
            Object target = show.GetPersistentTarget (i);

            Debug.Log (name);
            Debug.Log (target);
        }
    }

    public void RemoveAllListeners(){

        show.RemoveAllListeners ();
    }
}
```

XLVII.Null 參考例外

當你嘗試訪問未參考任何物件的參考變數，會發生 NullReferenceException 異常。如果參考變數未參考物件，那麼它將被視為 null。運行時會告訴你，你正在嘗試訪問一個物件，此時變數為 null，發出一個 NullReferenceException。

C# 和 JavaScript 中的參考變數在概念上與 C 和 C++ 中的指針相似。參考類型預設為 null，表示它們未參考任何物件。因此，如果你嘗試訪問被參考的物件，並且沒有一個物件可供訪問，則會得到 NullReferenceException。

當你的程式碼中出現 NullReferenceException 時，這意味著你在使用該程式碼之前忘記了設置變數。錯誤消息將如下所示：

```
NullReferenceException: Object reference not set to an instance of an object at
Example.Start () [0x0000b] in /Unity/projects/nre/Assets/Example.cs:10
```

此錯誤訊息表示 NullReferenceException 發生在腳本檔案 Example.cs 的第 10 行。此外，該訊息表示異常發生在 Start() 函數內。這使得 Null Reference Exception 易於查找和修復。在這個例子中，程式碼是：

```
public class Example : MonoBehaviour {

    void Start () {
        GameObject go = GameObject.Find("wibble");
        Debug.Log(go.name);
    }
}
```

程式碼只是尋找一個名為「wibble」的遊戲物件。在這個例子中，沒有該名稱的遊戲物件，所以 Find() 函數返回 null。 在下一行中，我們使用 go 變數，並嘗試列印出參考的遊戲物件的名稱。 因為我們正在訪問不存在的遊戲物件，所以運行時給我們一個 NullReferenceException。

1. *Null* 檢查

雖然這種情況可能令人沮喪，但這只是意味著腳本需要更加小心。在這個簡單的例子中的解決方案是改變這樣的程式碼：

```
public class Example : MonoBehaviour {

    void Start () {

        GameObject go = GameObject.Find("wibble");

        if (go) {

            Debug.Log(go.name);

        } else {

            Debug.Log("No game object called wibble found");
        }
    }
}
```

現在，在我們嘗試使用 go 變數執行任何操作之前，我們檢查它不是 null。如果它是 null，則顯示一條訊息。

2. *Try / Catch* 區塊

另一個導致 NullReferenceException 的原因是使用的變數應該在 Inspector 中初始化。如果你忘記這樣做，那麼變數將為 null。處理 NullReferenceException 的另一種方法是使用 try / catch 區塊。例如，這段程式碼：

```
public class Example : MonoBehaviour {

    public Light myLight;

    void Start () {

        try {

            myLight.color = Color.yellow;

        }catch (UnassignedReferenceException ex) {

            Debug.Log("myLight was not set in the inspector");
        }
    }
}
```

在此程式碼範例中，名為 myLight 的變數是應在 Inspector 視窗中設置的 Light。如果未設置此變數，那麼它將預設為 null。嘗試更改 try 區塊中的光的顏色會導致 UnassignedReferenceException 透過 catch 區塊被拾取。catch 區塊顯示一個可能對美術和遊戲設計師更有幫助的訊息，並提醒他們在 Inspector 中設置燈光。

3. 總結

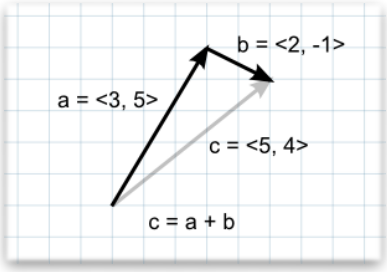
- 當你的腳本程式碼嘗試使用未設置（參考）和不是物件的變數時，NullReferenceException 會發生。
- 出現的錯誤訊息告訴你很多關於問題發生在程式碼中的何處。
- 可以通過編寫在訪問物件之前檢查 null 的程式碼來避免 NullReferenceException，或者使用 try / catch 區塊。

XLVIII.了解向量計算（一）

向量計算是 3D 圖學、物理和動畫的基礎，深入了解它可以充分利用 Unity。以下是主要操作的描述和關於它們可以使用的許多事情的一些建議。

1. 加法

當兩個向量相加在一起時，結果相當於將原始向量「逐步」一個接一個相加。請注意，兩個參數的順序並不重要，因為結果是一樣的。



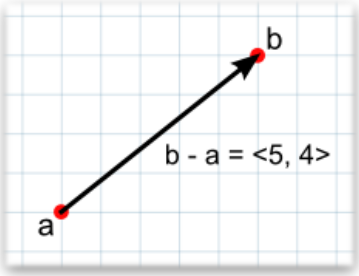
如果第一個向量做為空間中的一個點，那麼第二個向量可以被解釋為與該位置的偏移或「跳躍」。例如，要在地面之上找到一個 5 單位的點，你可以使用以下計算：

```
Vector3 pointInAir = pointOnGround + new Vector3(0, 5, 0);
```

如果此向量代表力量，那麼使用它們的方向和大小（表示力量的大小）方面考慮它們是更直覺的。相加兩個力量向量會產生一個相當於力量組合的新向量。當使用幾個單獨的組件一次作用的力（例如，向前推進的火箭也可能受側風影響）時，這個概念通常是有用的。

2. 減法

向量相減最常用於獲取從一個物件到另一個物件的方向和距離。請注意，兩個參數的順序與減法無關：



```
// 向量 d 的大小與 c 相同，但指向相反的方向。
Vector3 c = b - a;
Vector3 d = a - b;
```

與數字一樣，加上向量的負數與減去正數相同。

```
// 兩者產生相同結果
Vector3 c = a - b;
Vector3 d = a + -b;
```

向量的負數與原始點並沿相同直線相反的方向具有相同的大小。

3. 純量乘法和除法

當討論向量時，通常將普通數值（例如，浮點值）稱為純量。其含義是純量只具有大小，而向量具有大小和方向。

將一個向量乘以一個純量，得到一個與原始方向相同方向的向量。 然而，新向量的大小等於原始大小乘以純量值。

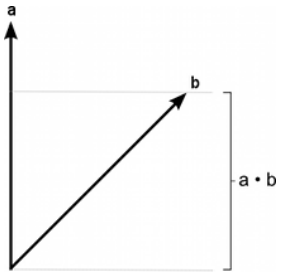
同樣，純量除法將原始向量的大小除以純量。

當向量代表運動偏移量或力量時，這些操作很有用。它們讓你能夠改變向量的大小而不影響其方向。

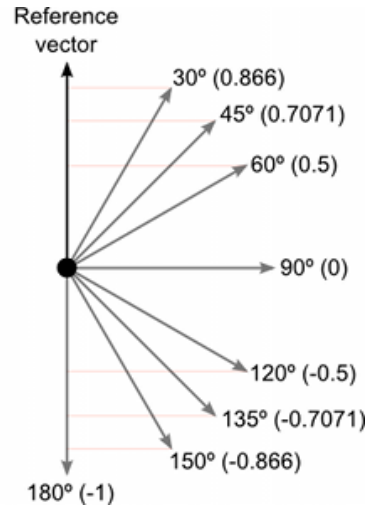
當任何向量除以其自身的大小時，結果是大小為 1 的向量，其被稱為歸一化（Normalized）向量。如果歸一化向量乘以純量，則結果的大小將等於該純量值。當力量的方向是恆定的但強度是可控的（例如，來自汽車的車輪的力量總是向前推，但是動力由駕駛員控制）時，這是有用的。

4. 點積

點積取用兩個向量並返回一個純量。該純量等於兩個向量的大小相乘，其結果乘以向量之間角度的餘弦。當兩個向量被歸一化時，餘弦本質上說明了第一個向量在第二個的方向上延伸多遠（反之亦然 - 參數的順序並不重要）。



用角度來思考，然後使用計算機找到相應的餘弦是很容易的。然而，直覺地了解一些主餘弦值是有用的，如下圖所示：



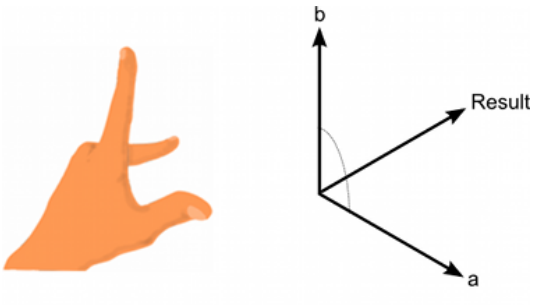
點積是一種非常簡單的操作，可以在某些情況下使用向量大小操作代替 Mathf.Cos 函數（它不完全相同，但有時效果是相同的）。然而，計算點積函數花費的時間要少得多，因此它可以是有價值的優化。

```
public class Example : MonoBehaviour {

    public Transform targetA;
    public Transform targetB;
    public bool normalized;
    public float result;

    void Update(){

        if(normalized){

            result = Vector3.Dot
                (targetA.position.normalized,
                    targetB.position.normalized);

        }else{

            result = Vector3.Dot
                (targetA.position, targetB.position);
        }
    }
}
```

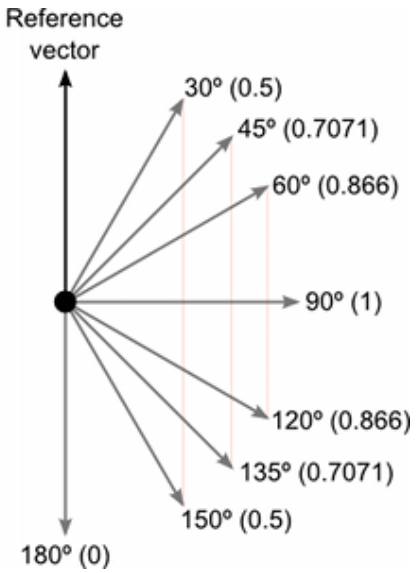
5. 叉積

其它操作是針對 2D 和 3D 向量定義的，並且實際上是具有任意大小的向量。相比之下，叉積只對 3D 向量有意義。它需要兩個向量作為輸入，並返回另一個向量做為其結果。

結果向量垂直於兩個輸入向量。「左手定則」可用於根據輸入向量的順序來控制輸出向量的方向。如果第一個參數與手的拇指匹配而第二個參數相對於食指，則結果將指向中指的方向。如果參數的順序相反，則所得到的向量將指向完全相反的方向，但是具有相同的大小。



結果的大小等於輸入向量的大小相乘在一起，然後該值乘以它們之間的角度的正弦。正弦函數的一些有用值如下所示：



叉積看起來很複雜，因為它在返回值中結合了幾個有用的訊息。然而，像點積一樣，它是效率高的演算方式，可以用於改善程式，否則將依賴於緩慢的超越數（Transcendental）功能。

```
public class Example : MonoBehaviour {

    public Transform targetA;
    public Transform targetB;
    public bool normalized;
    public Vector3 result;

    void Update(){

        result = Vector3.Cross
            (targetA.position, targetB.position);

        if (normalized)
            result = result.normalized;
    }

    void OnDrawGizmos(){

        Gizmos.color = Color.red;
        Gizmos.DrawLine (Vector3.zero , result);
        Gizmos.DrawSphere (result , .25f);
    }
}
```

6. 從一個物件到另一個物件的方向和距離

如果從空間中的一個點減去另一個點，那麼結果就是從一個物件「對」到另一個物件的向量：

```
// 獲取從玩家的位置指向目標的向量。
Vector3 heading = target.position - player.position;
```

除了指向目標物件的方向之外，該向量的大小等於兩個位置之間的距離。通常需要一個歸一化的向量，給出目標的方向以及目標的距離（比如說用於引導射彈）。物體之間的距離等於指向向量的大小，該向量可以通過將其除以其大小來進行歸一化：

```
float distance = heading.magnitude;
Vector3 direction = heading / distance;
```

這種方法優於分別使用 magnitude 和 normalized 屬性，因為它們都會讓 CPU 相當忙碌（它們都涉及計算平方根）。

如果只需要使用距離進行比較（針對接近檢查），則可以避免 magnitude 計算。sqrMagnitude 屬性提供 magnitude 的平方，並且像 magnitude 一樣計算，但沒有耗時的平方根操作。你可以針對平方距離來比較平方大小，而不是針對已知距離比較大小：

```
if (heading.sqrMagnitude < maxRange * maxRange) {

    // 目標在範圍內
}
```

這比使用真實大小進行比較更有效率。

有時，所需要得目標點需要在地面。例如，想像一個站在地上的玩家需要接近浮在空中的目標。如果你從目標位置減去玩家的位置，則所得到的向量將向上朝向目標。這不適合於定位玩家的 Transform，因為它們也將向上指向；真正需要的是從玩家的位置到目標的正下方的位置的向量。通過取減法的結果並將 Y 坐標設置為零可以很容易地獲得。

```
public class Example : MonoBehaviour {

    public Transform target;
    public Transform player;
    public float maxRange = 2;
    public bool within;

    void Update(){

        Vector3 heading = target.position - player.position;
        heading.y = 0;

        within = heading.sqrMagnitude < maxRange * maxRange;
    }
}
```

未完，不續....

連志偉

Facebook

https://www.facebook.com/profile.php?id=100000066674252

粉絲專頁

https://www.facebook.com/RandomExp/

Unity 應用領域

社團

https://www.facebook.com/groups/UnityFrontier/

專頁

https://www.facebook.com/UnityFrontier/