MLH localhost

Workshop

Accelerate Your C# Learning with Unity

MLH localhost

v.0.0.1

1

Using your Web Browser,
Open this URL:

http://mlhlocal.host/lhd-resources

2

Click on the workshop you're attending, and find:

- Setup Instructions
- The Code Samples
- A demo project
- A Workshop FAQ
- These Workshop Slides
- More Learning Resources

Our Mission is to Empower Hackers.

65,000+
HACKERS

12,000+
PROJECTS CREATED

400+
CITIES

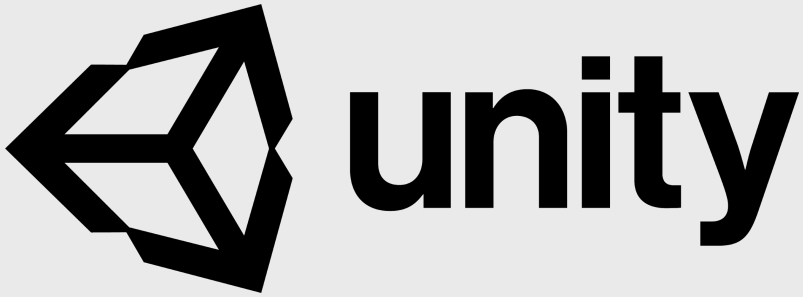
We hope you learn something awesome today!
Find more resources: http://mlh.io/

What will you learn today?

1. What you can build with Unity
2. How to build Unity projects
3. How to use Unity
4. The basics of C#

Table of Contents

What is



- Unity is a powerful development tool for building 2D and 3D games for over 25 different platforms.
- It provides a robust, real-time engine used to create fantastic, popular games on many platforms, especially mobile platforms and VR/AR experiences.
- It is also a complete development environment that uses the C# language for development!

Awesome Unity Projects

Unity is used to produce amazing games and short films.

http://mlhlocal.host/unity-examples

FAR: Lone Sails in this video is a student made game.

What are you going to do with Unity?

http://mlhlocal.host/unity-demo

You're going to write some of the code for this awesome racing game. Navigate to the URL above to test the game!

Well that was cool! Let's see what we need to do to be able to build this game!

Please note - there are many set up steps. When you're finished, you will be all set up to develop with Unity!

What do I need to create with Unity?

To complete this workshop, you need the following tools and technical specifications:

- Unity version 2018.1.1 (this version is required, please don't use any other version)
- 15GB of storage available
- Admin access to the computer you're using, especially the password
- The text editor of your choice.

How are you going to do this?

To build games with Unity, you need several applications on your computer.

1. The project folder - you either need to download this from mlhlocal.host/unity-project **or** copy it from one of the flash drives provided
2. Unity Hub - download from mlhlocal.host/unity-hub **or** copy from the flash drive
3. The Unity Editor - download version 2018.1.1 from mlhlocal.host/unity-versions **or** copy from the flash drive
4. The text editor of your choice (more on the next slide).

A Note About Text Editors:

You can use any text editor you like, but we recommend Visual Studio Code.

http://mlhlocal.host/vscode

When you install Unity, it will ask if you want to install Visual Studio. Select NO - this is NOT the same thing as Visual Studio Code!

WHOA THERE!

DELETE THIS SLIDE BEFORE PRESENTING

- **USB Flash Drives:** If you received USB Flash Drives, attendees can copy the sample code and install files from there.
- **Web / Download:** If you did not receive USB Flash Drives, attendees will need to download the sample code and install files from the web.

Downloading the Project Files

http://mlhlocal.host/unity-project

Navigate to the URL above to download the project! Note - it might take a long time to download. It's a huge file!

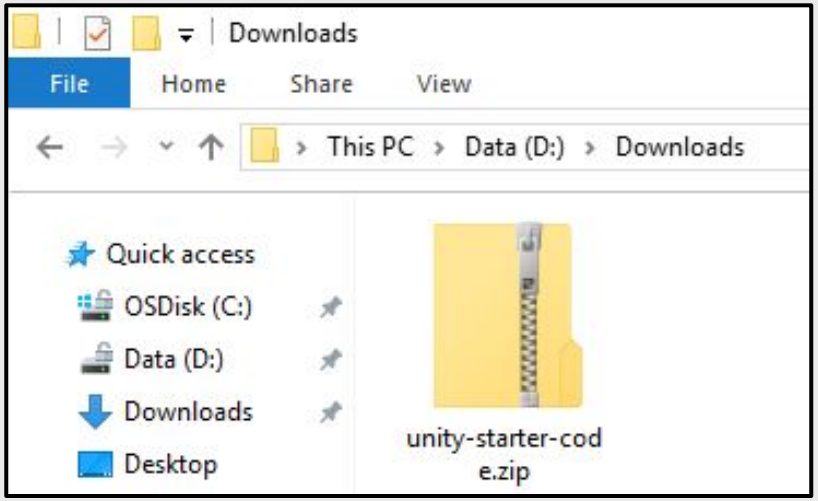
The next slides will tell you how to open the file.

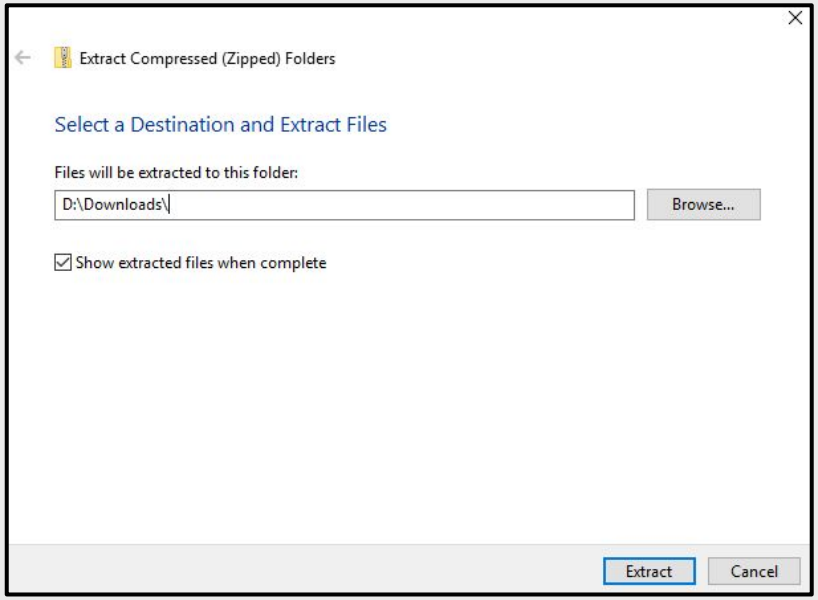
The next slides have instructions for unzipping the file.

Please note that a tool like WinRAR or 7Zip (free) will be much faster than the built-in archive tool for Windows.

Windows Extraction
(WinRAR or 7Zip Recommended for Speed)

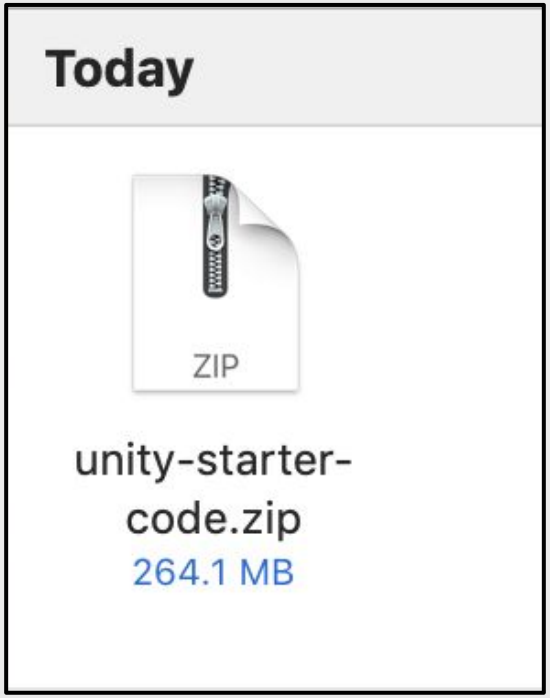
1. Open your Downloads folder.
2. Right click **unity-starter-code**.
3. Click **Extract All**.
4. Delete **`unity-starter-code`** from the end of the file name.
5. Click **Extract**.





Mac Extraction

1. Open your downloads folder.
2. Double click the zip file. It will unzip automatically.



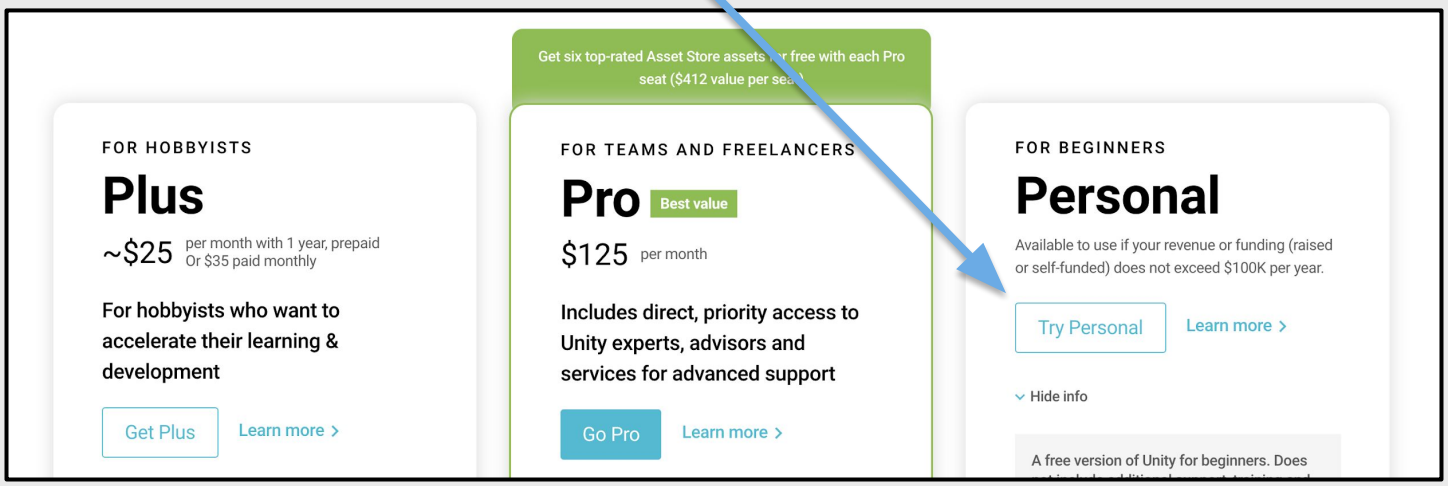
Let's get started!

Table of Contents

Downloading Unity Hub

http://mlhlocal.host/unity-hub

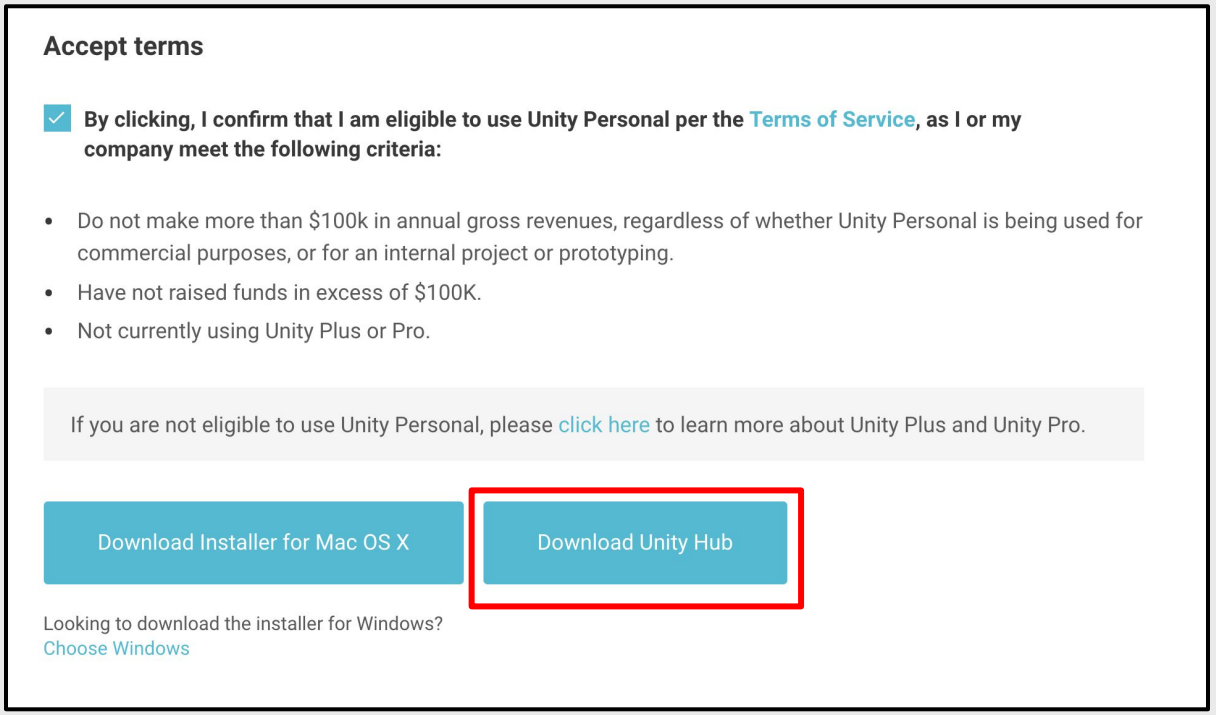
1. Navigate to the URL above.
2. Select **Try Personal**.



Downloading Unity Hub

3. Accept the terms.

4. Select **Download Unity Hub**.

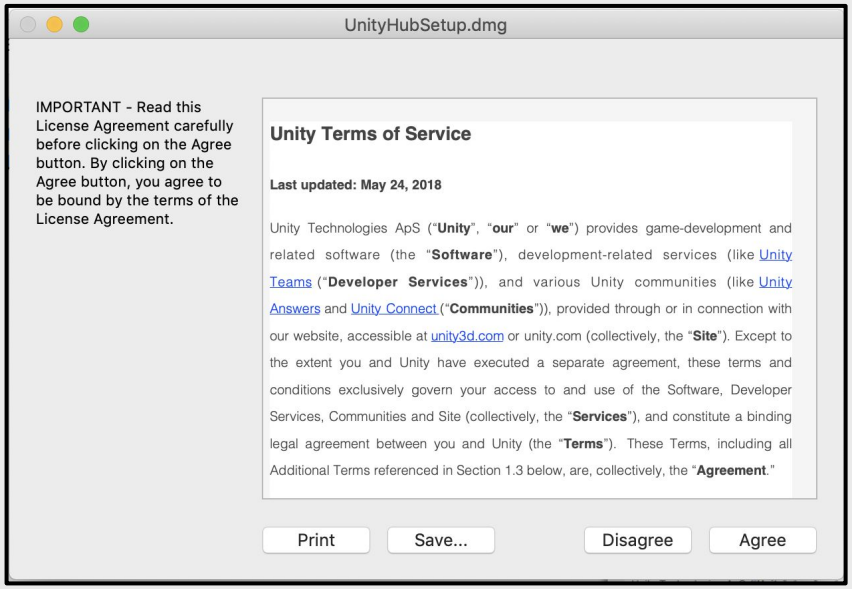


Installing Unity Hub

5. Open the installer.

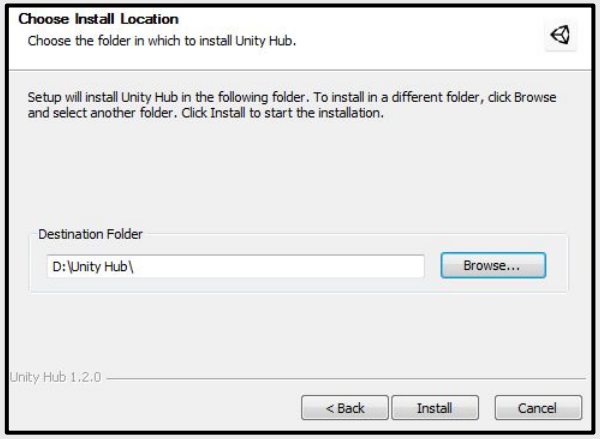
6. Running the installation will require you to agree to the Unity terms of service.

7. Once you've installed it, we can open up Unity Hub!

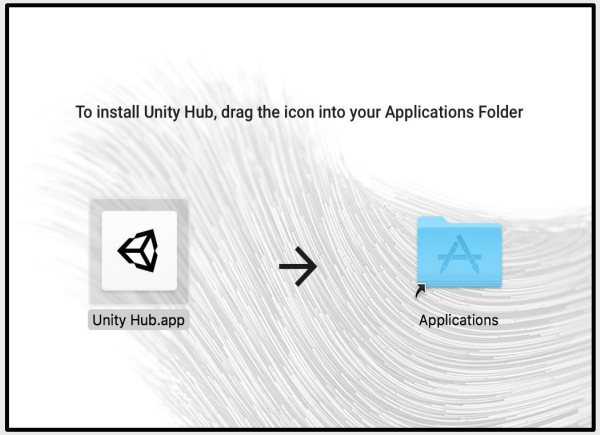


Installing Unity Hub

- The Windows installation will provide the option for selecting what disk drive you want the installation to be on. You can simply use the location it recommends.



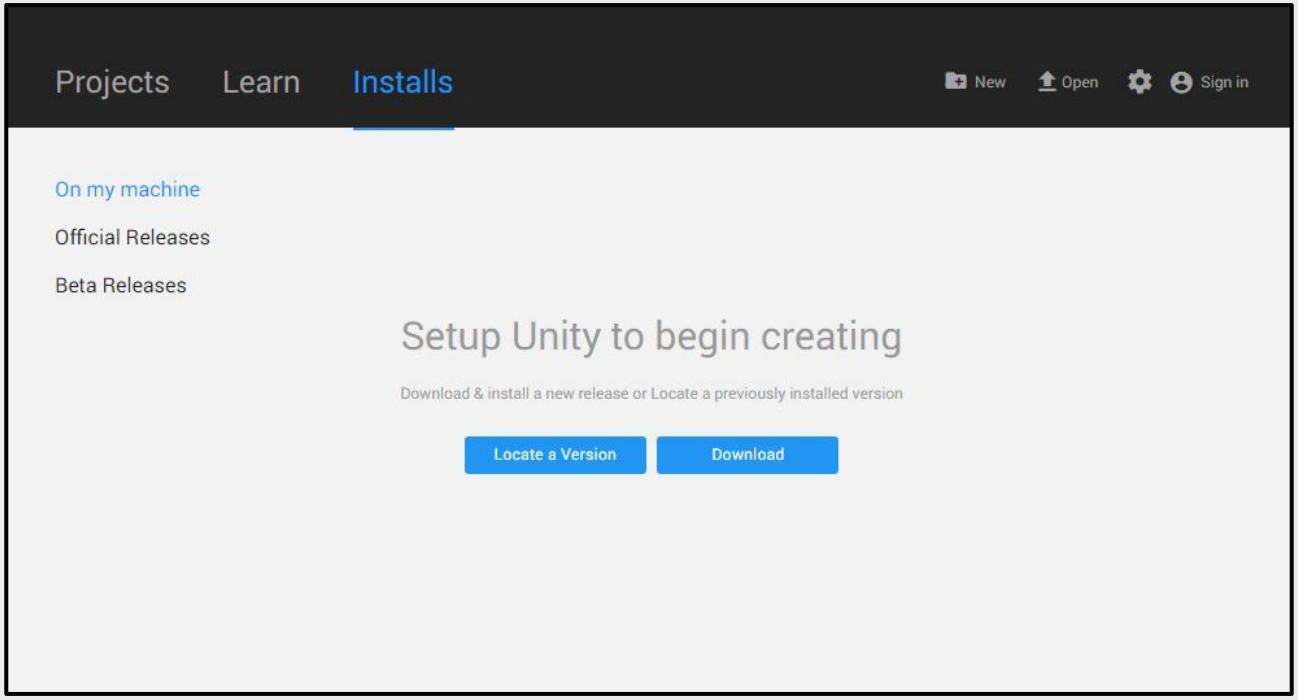
- The OS X installation will require you to drag the Unity Hub.app icon into the Applications folder



Successful Installation

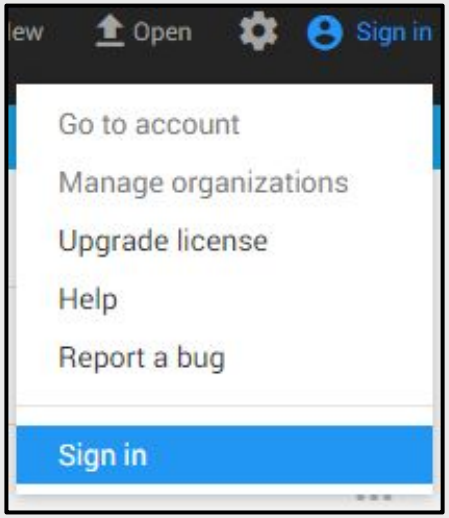
After a few minutes of installing, Unity Hub should now be installed on our system. Both Windows and OS X installers should open Unity Hub after a successful installation. If not, open it up!

Let's create a User ID and get to know the interface before we move on.



Creating a User ID

1. To create an ID we need to navigate to the Sign in menu, in the top right corner of Unity Hub.



2. When the Sign in window appears, we can click 'Create One' which will open up an account creation form.

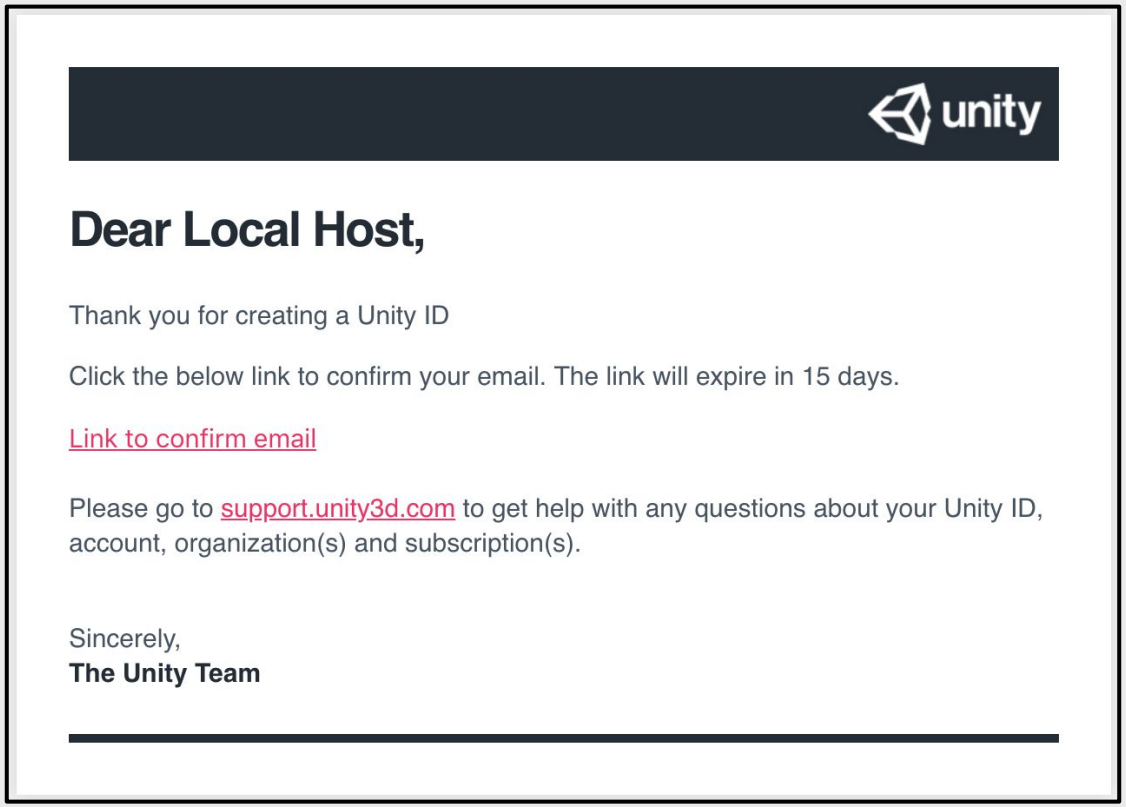
Creating a User ID

3. Follow the prompts and fill in the form to create your Unity Account.

If you choose to use Facebook or Google Accounts to establish a Unity Account, a separate Log in screen for either service will load within Unity Hub.

Creating a User ID

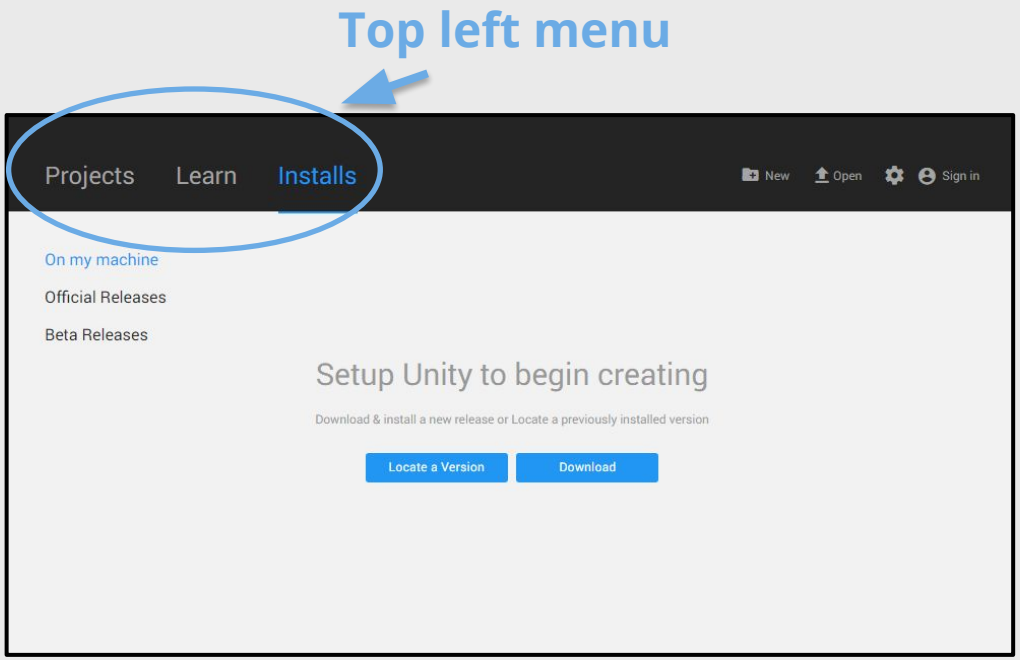
5. If you sign up with an email, you will need to confirm your email address.
6. Click the confirmation link sent to the email you provided.



Great! You've got Unity Hub installed. Let's get familiar!

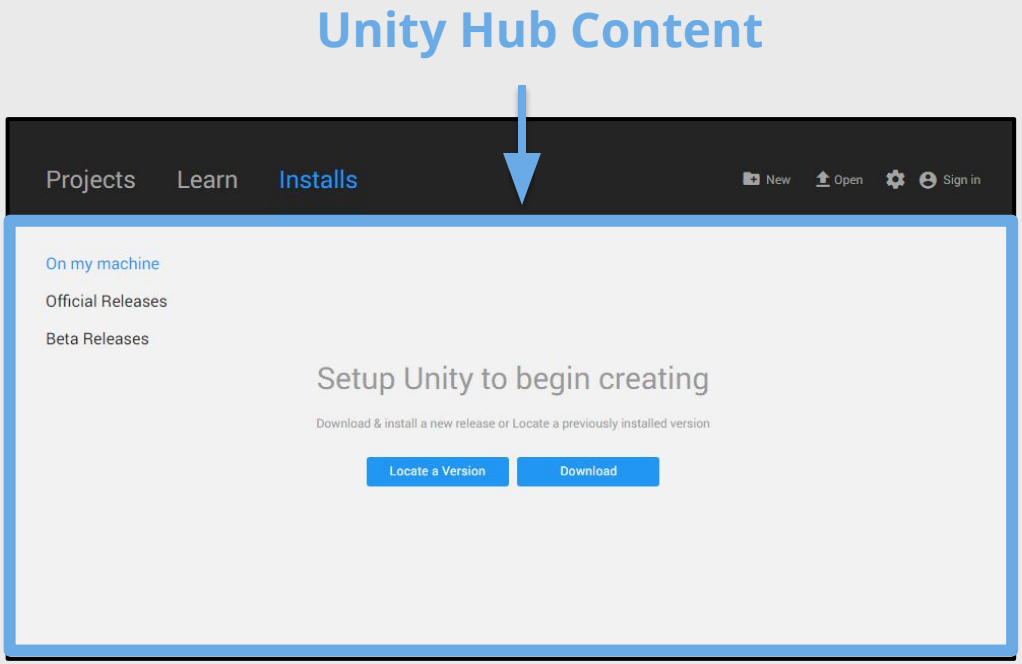
Unity Hub UI

From the Hub UI these are all the different things we have access to:



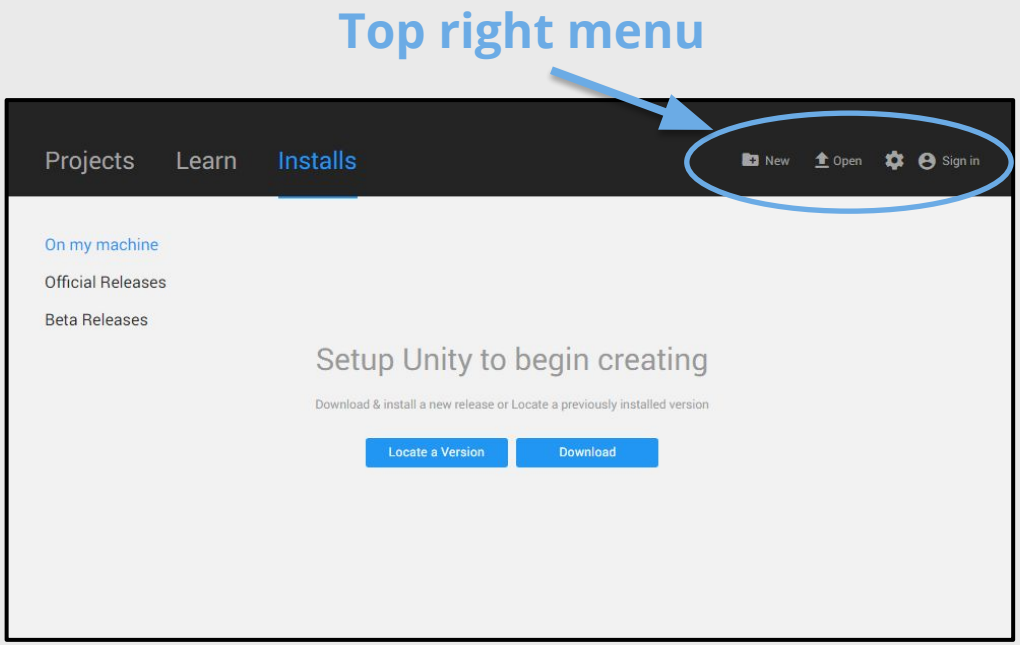
- Here we can access all of our Unity installs (the versions of Unity we have on our computer) under the Installs tab
- We can access all of our ongoing Unity projects within the Projects tab
- And within the Learn tab we can find Unity tutorials and resources

Unity Hub UI



- This area displays the Unity Hub content. Content from the Projects, Learn, and Installs tab are displayed here.

Unity Hub UI



- Users can access the Sign in tab so they can sign into Unity accounts, report program bugs or access the help menu.
- Open tab provides the ability to open pre-existing Unity projects.
- The New tab opens the menu to launch new projects within Unity.

Installing Unity

http://mlhlocal.host/unity-versions

We are able to download Unity 2018 (2018 1.1f1) through the **Unity Download Archive**.

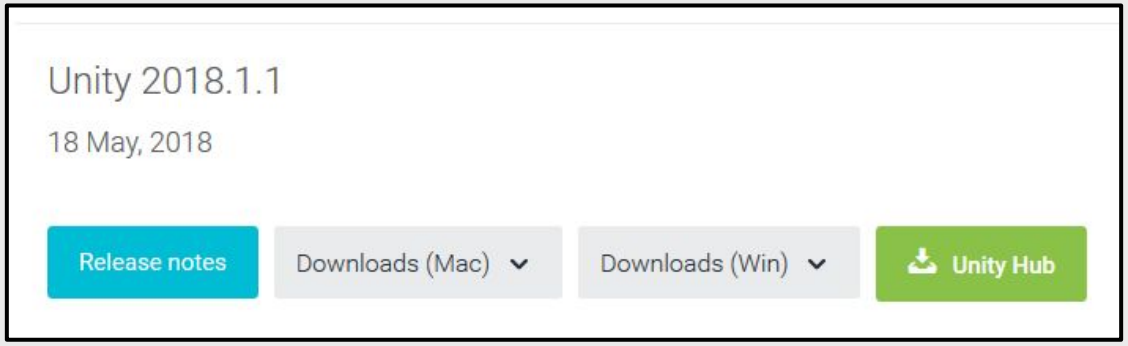
We require this version of Unity to run the project we're working with.

Navigate to the **URL** above and select the following:

Installing Unity

The version required is 2018.1.1.

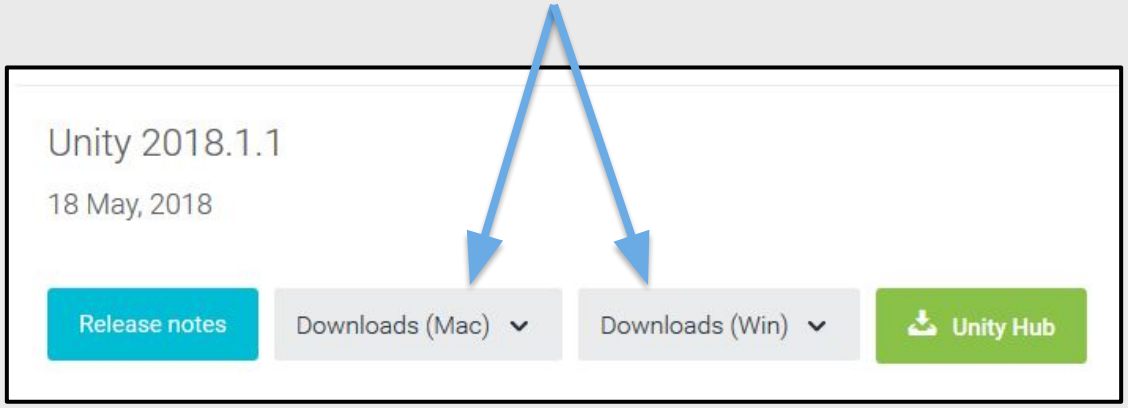
Scroll all the way to the bottom to find it.



We have two options available for downloading and installing Unity.

Installing Unity

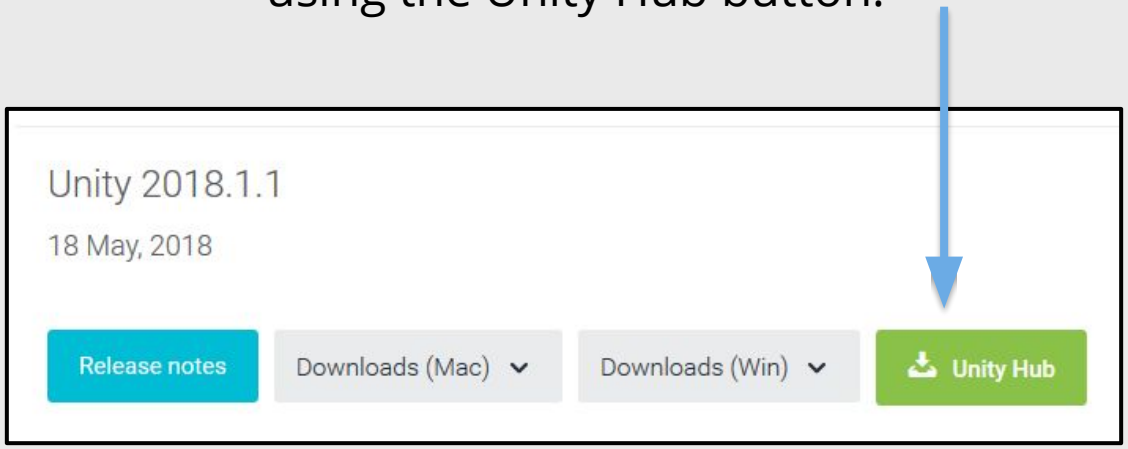
Unity can be downloaded directly to your machine with either the Downloads(Mac) or Downloads(Win) buttons.



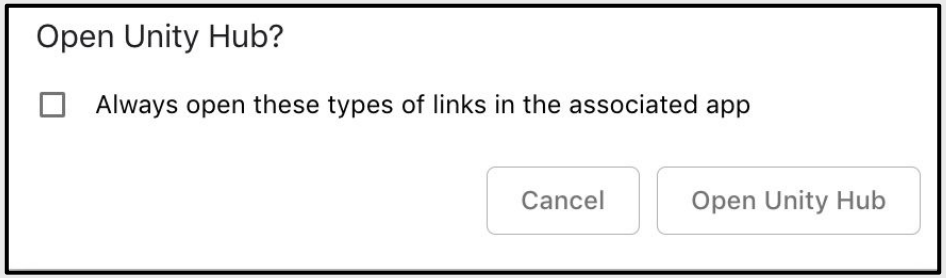
This will download the Unity Install Assistant and guide you through the installation process.

Installing Unity

It's also possible to download this version of Unity through Unity Hub by using the Unity Hub button.



This will open up the version we need, within our Unity Hub desktop application.

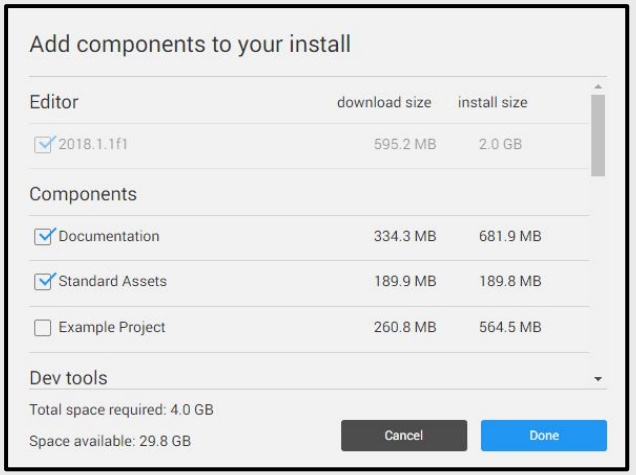


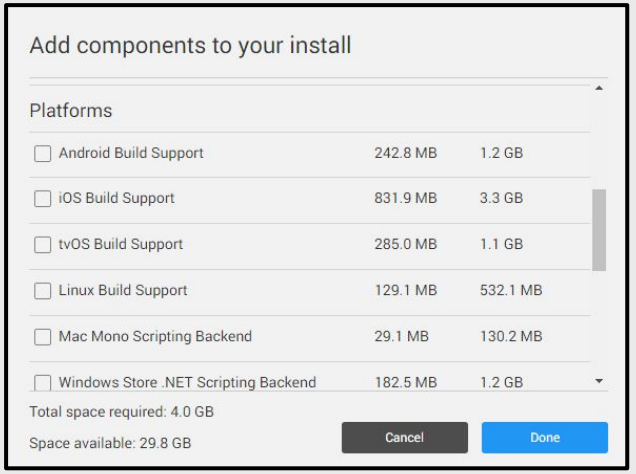
Installing Unity

- You'll see a checklist for selecting components within the installation.

- We **only** want to select
  - Unity Editor
  - Standard Assets
  - Documentation

- Extra components increase the download dramatically, but they can always be downloaded later with the Unity installer.





Installing Unity

- The Unity installer gives the option to select Microsoft Visual Studio Community as an editor for writing C#.
- Despite many Unity developers using Visual Studio to write their code, we **will not be** downloading it.
- We are going to set a code text editor as the default, once we have installed Unity completely.
- VSCode is lighter and smaller to download. Once you are familiar with Unity you can use any editor you prefer, that supports C#!

And We're Off!

You should now be able to find your newly installed version of Unity inside Unity Hub !

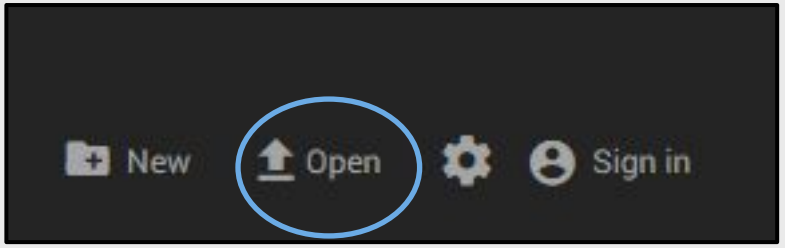
Now we can get to know C# and explore Unity!

Table of Contents

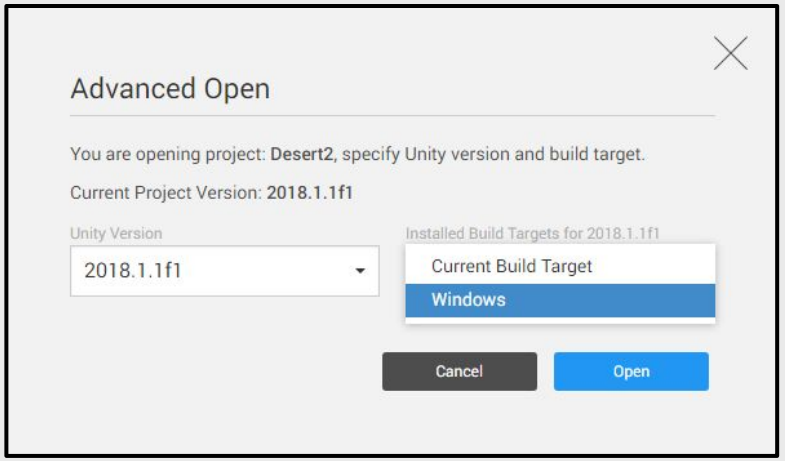
Now, we're ready to import

From Unity Hub, we can import our project to Unity. It will then load and compile everything required.

1. The Open tab will let you select a folder to load.

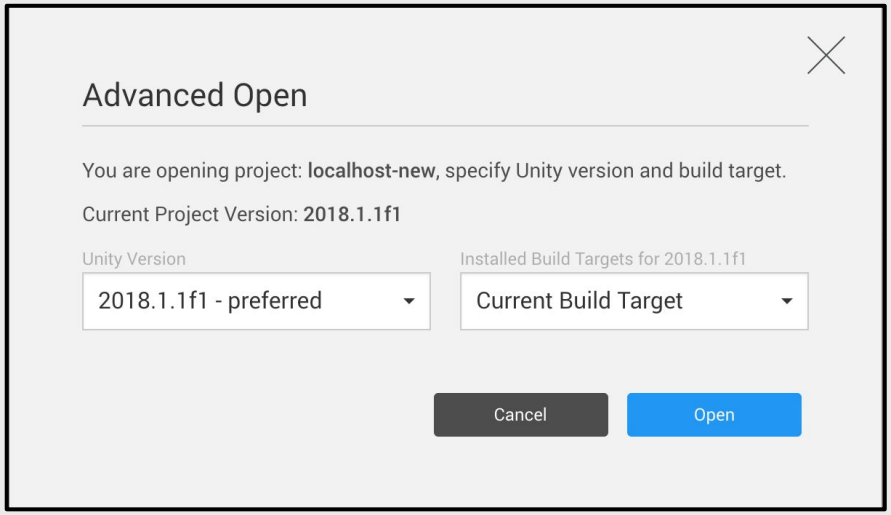


- If you're on Windows you will need the **WindowsStandaloneSupport** selected in the Installed Build Targets.
- On a Mac you will need the **MacStandaloneSupport** selected in the Installed Build Targets.



Now, we're ready to import

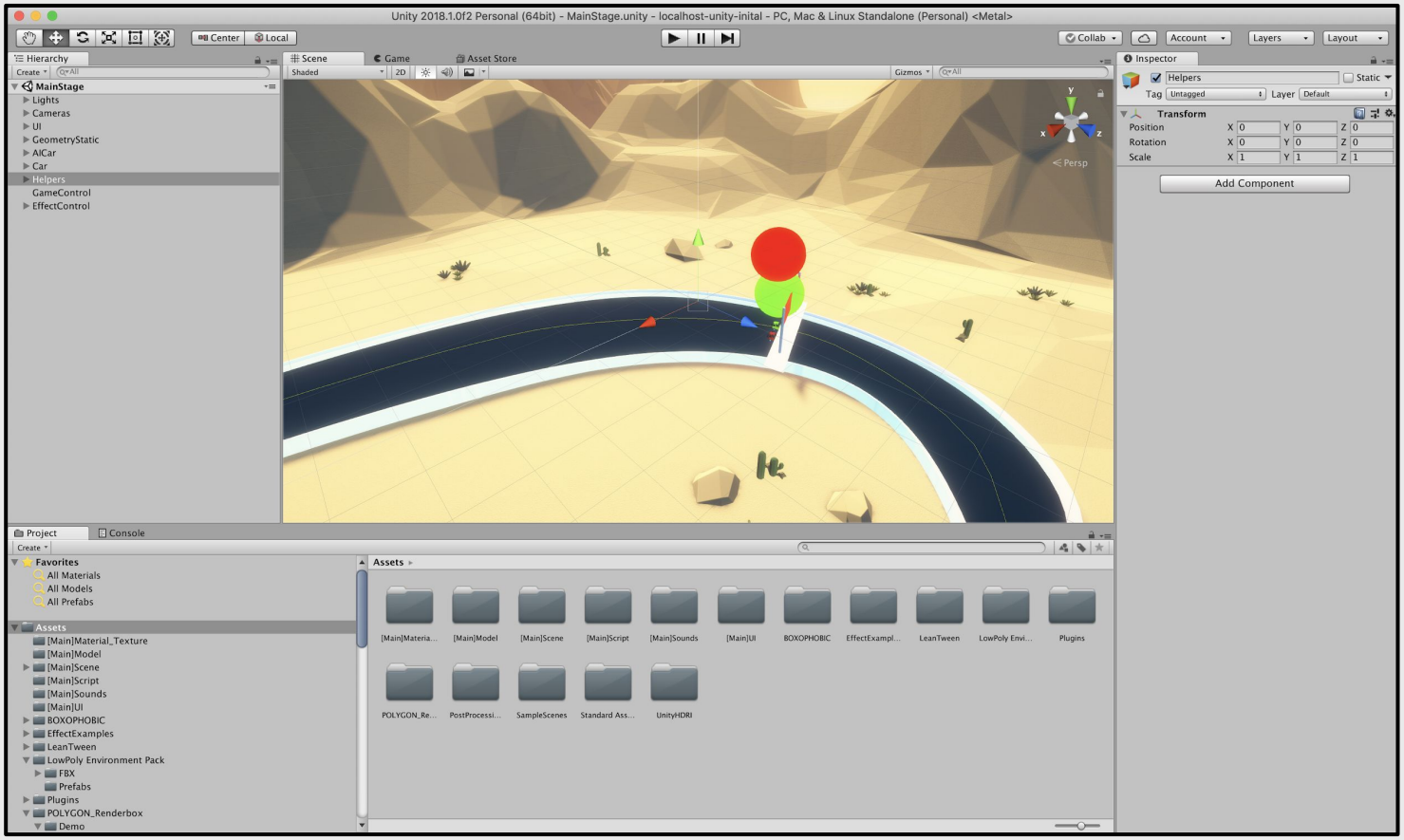
2. Once the folders and Build Target are chosen, let's click **Open** and load the game!



If you **didn't** change your Build Target you may receive this notification when you open the game in Unity. Selecting **Switch Target** will help Unity switch for you.

What are you looking at?

You opened Unity, and there is a LOT going on. Let's dive in!



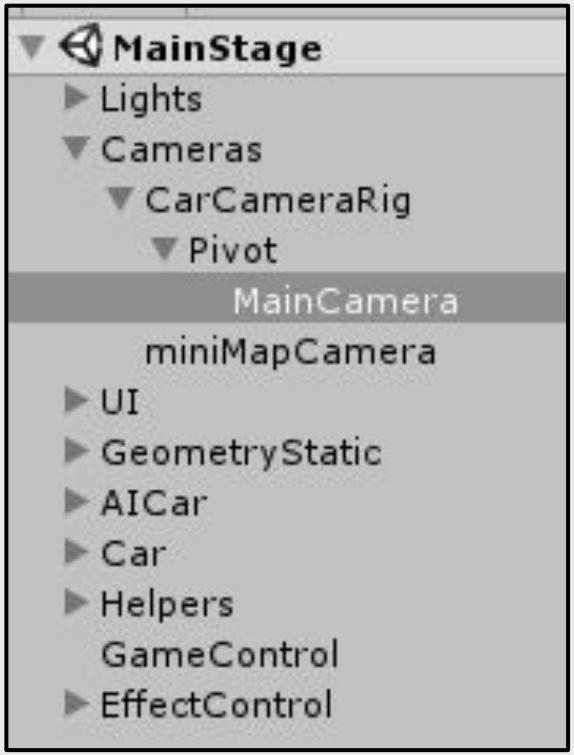
Let’s get to know Unity!

The Scene Hierarchy

The project we opened in Unity is packed full of content and the Hierarchy window is able to show us everything that is in the current scene.

We can navigate the files to see all of this.

By selecting an asset within the Hierarchy window, we can find and select objects live in the scene window.

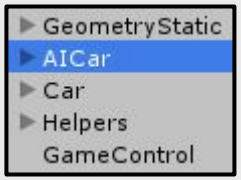


1. Try selecting the MainCamera object Cameras>CarCameraRig>Pivot>MainCamera

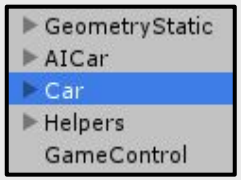
The Scene Hierarchy

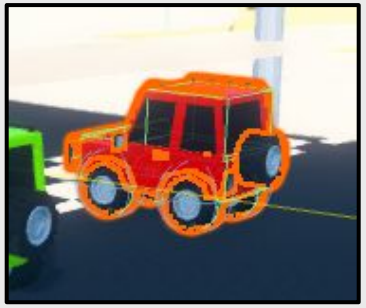
- If we select the MainCamera in the Hierarchy window, we are provided with a preview of the camera objects initial view. It shows the view of our Camera object in the scene.

2. Now, try selecting the Car object

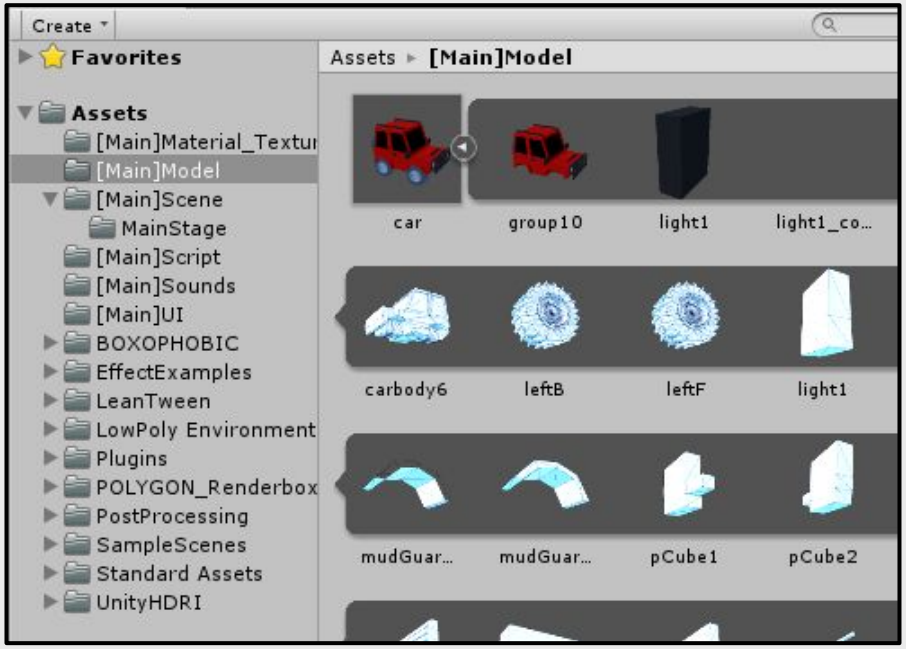


- If we choose another asset, such as the Car or AICar, we can see the objects become the area of focus and become outlined within the scene window.





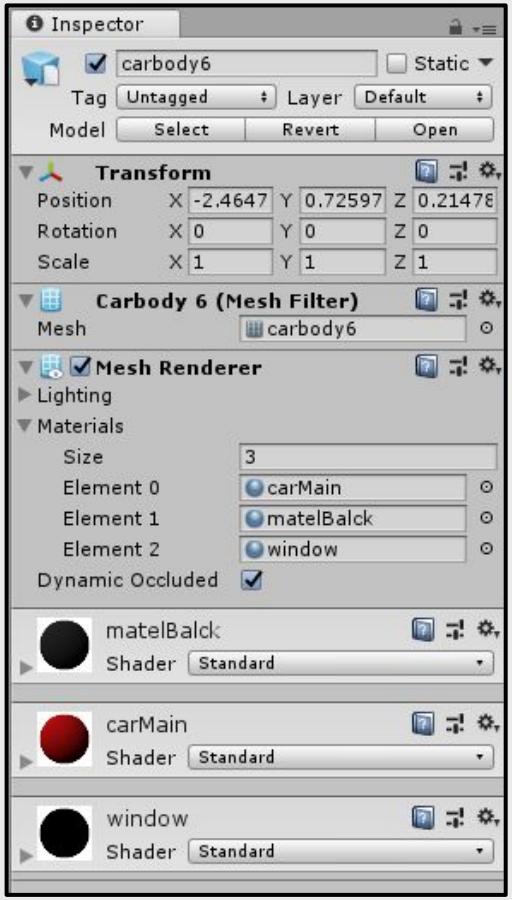
The Project Window



- The Project window provides a list of every single item within our Unity project.
- We can use this to open Scripts, find assets to add into scenes and locate anything else incorporated in our project.
- The Project window provides several abilities such favoriting items, and a text search field. These make it faster to find items or relocate frequently used content.

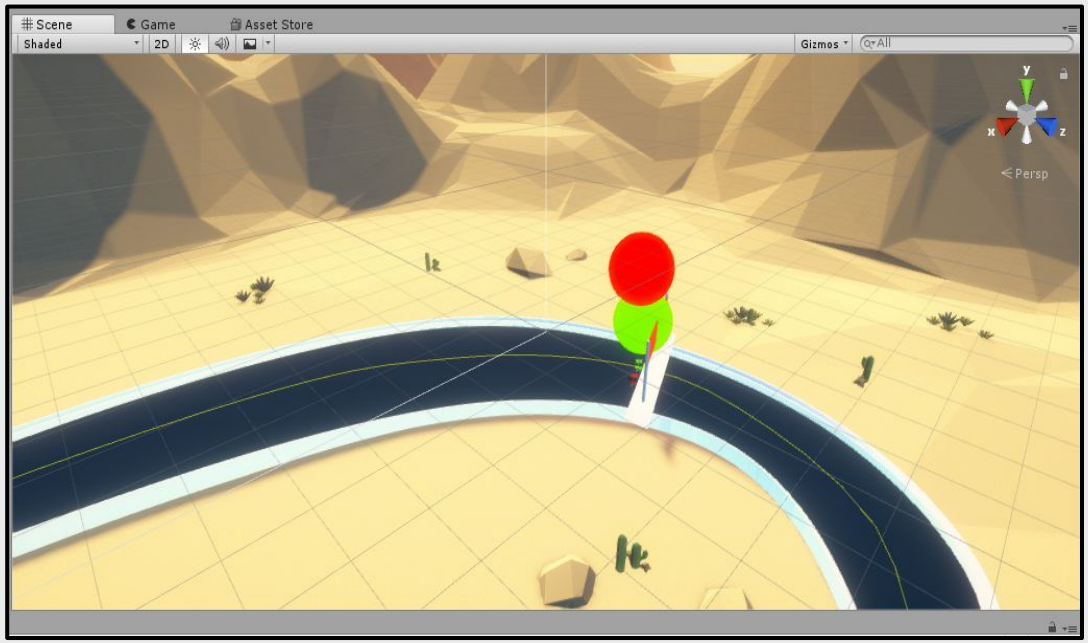
Inspector Window

- Any time we select an object in our Scene window, you might notice the Inspector window changes too.
- It loads up important details about whichever object we have selected at the time.
- If we select objects in the Scene window, Hierarchy window or Project window, the details will still display within the Inspector view.
- Within the Inspector, we can apply, remove and control the Components we apply to game objects. We will explore Components more later!



Scene Window

The Scene Window is our interactive view into the world being created.



- It's here we can add, remove and manipulate models, scenery and any other type of asset you have in the scene!
- Knowing how to control game objects inside a scene is an essential skill in Unity.

The 3D World

Unity might still look confusing, but we won't be using every tool available.

Let's try moving your field of view around the 3D world!

Some common interactions:

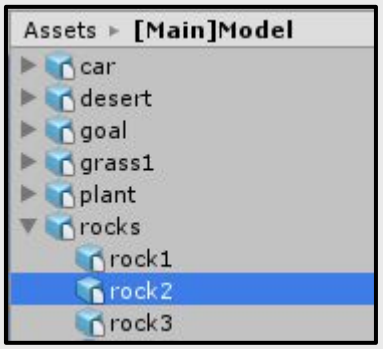
1. **Right click + drag** = Camera pivot
2. **Middle mouse click + drag** = X-axis movement
3. **Scroll wheel** = Jumping Zoom in / Zoom out
4. **Hold ALT + Right click + pan left and right** = Constant Zoom

The 3D World

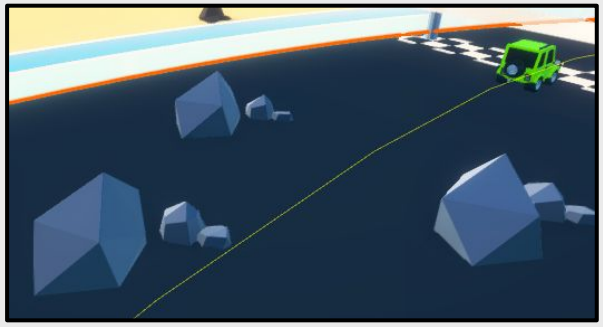
We can drag 3D Assets directly into the scene to create content! Unity provides numerous built-in assets for your project!

1. In our Project tab, let's navigate to an asset we already have. Select:

   Assets > Model > Rocks > Rock1



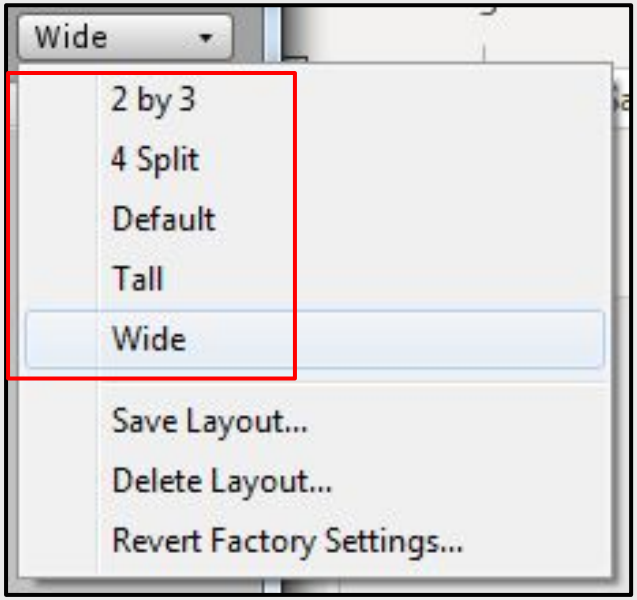
2. Then drag that file into our scene, and a new 3D object will appear! It's really that simple.



User Interface Layout

The Unity UI has many items and menus, we might find ourselves repositioning them to get a better view of other areas.

- In the top right corner of Unity, a drop down menu provides several layout configurations.
- These will reposition the entire Unity Editor layout!
- Select one and see. The changes will happen instantly.

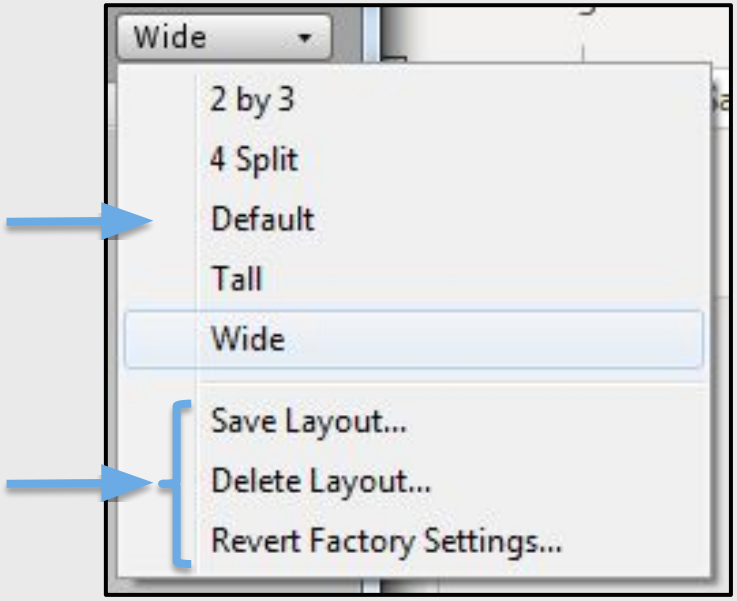


User Interface Layout

If you customise the area sizes in Unity to your liking, you can save it as a permanent layout to choose later using **Save Layout...** .

You can also delete it with **Delete Layout...**

If the windows are ever lost or moved around too much and become confusing to locate, clicking **default** will take you back the initial Unity display.

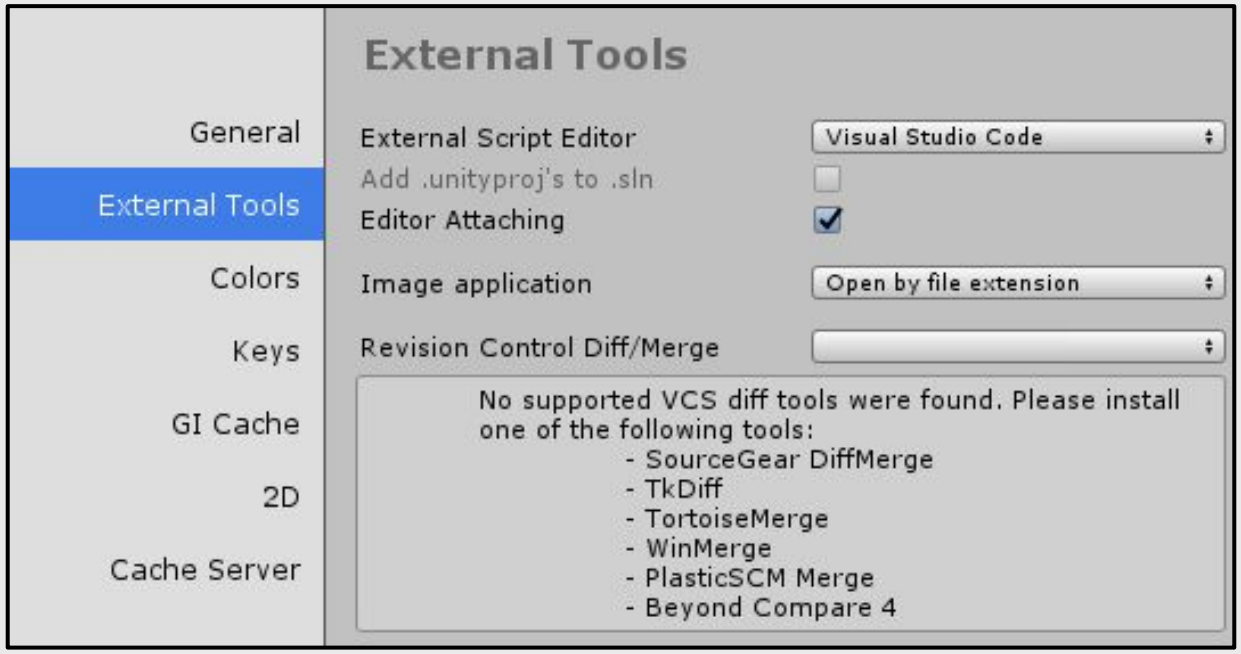


Let's set your Editor!

To edit code with Unity, a text editor or IDE is essential.

Let's set our preference now by selecting Unity on Mac or Edit on Pc then:

Preferences > External Tools > External Script Editor, then choosing your preferred text editor.



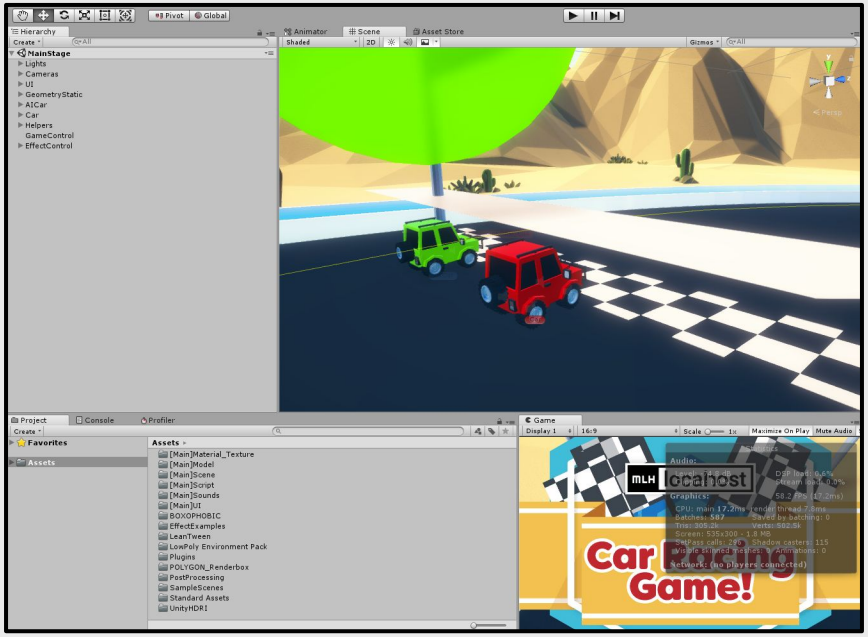
Great!
Let's get to work with this project.

Car Racing Game

The project we uploaded is a car racing game.

Here are some features already built in:

- Player VS AI
- Keyboard input controls
- Collision detection
- Interactive Menu



Let's Test This Out!

To see how the game runs, control buttons are at the top of the application.

Play the game

Pause the game

Step through the game by executing 1 frame every click

Whoa what's wrong?

A few parts of the app aren't working properly. You're going to fix them!

Problem I:

You can't move your car. Why is that?

Table of Contents

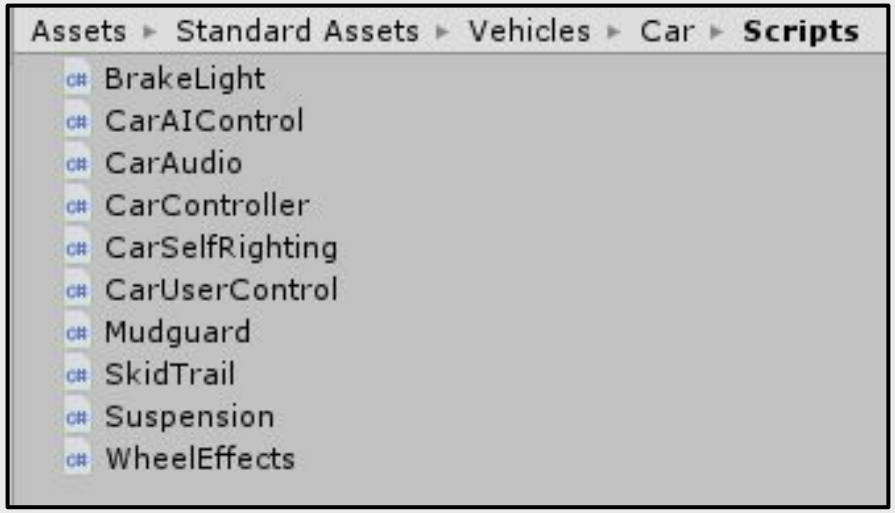
CarUserControl.CS

There is a file called CarUserControl.cs. Let's open that up.

In the **Project** tab we can follow the path:

Standard Assets > Vehicles > Car > Scripts > CarUserControl.CS

Double click the file to open it in the editor you set previously.



Missing Code

Well, that doesn't look right. This script is practically empty! It should allow the user to control their car with the **W A S D** or arrow keys.

- With missing code the game no longer has controls, and can not be played.

- We are going to repair this script, step by step with C#.

CarUserControl.cs

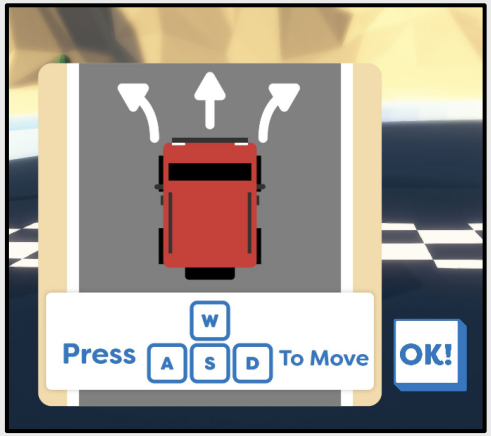
```
1   using System;
2   using UnityEngine;
3   using UnityStandardAssets.CrossPlatformInput;
4
5   namespace UnityStandardAssets.Vehicles.Car
6   {
7       [RequireComponent(typeof (CarController))]
        1 reference
8       public class CarUserControl : MonoBehaviour
9       {
10          //Write code here!
11
12
            0 references
13          private void Awake()
14          {
15
16              //Write code here!
17
18          }
19
20
            0 references
21          private void FixedUpdate()
22          {
23              //Write code here!!
24          }
25
26      }
27  }
28
```

The C# Language

- C# was created by Microsoft around the year 2000.
- It's a multi-paradigm, general purpose programming language.
- It is an extremely popular language supported within the Unity environment.

C# in Unity

- C# is not written in the same format inside Unity, compared to a stand alone C# application.
- Unity allows developers to write individual *scripts* to control objects in the game environment.
- Objects are items within the game, such as 3D models or a light source etc.



C# Variables

- Variables hold data in a computer program and allow you to access them.
- With C# we are able to declare, access and manipulate variables.
- In C# they need to be declared as a specific type to determine what values that can contain.

```
int a = 10;
double pi = 3.14;
char q = 'q';
bool trueOrFalse = False;
```

C# variable declaration follows a simple structure:

<variable_type> <variable_name> = <value>

C# Functions

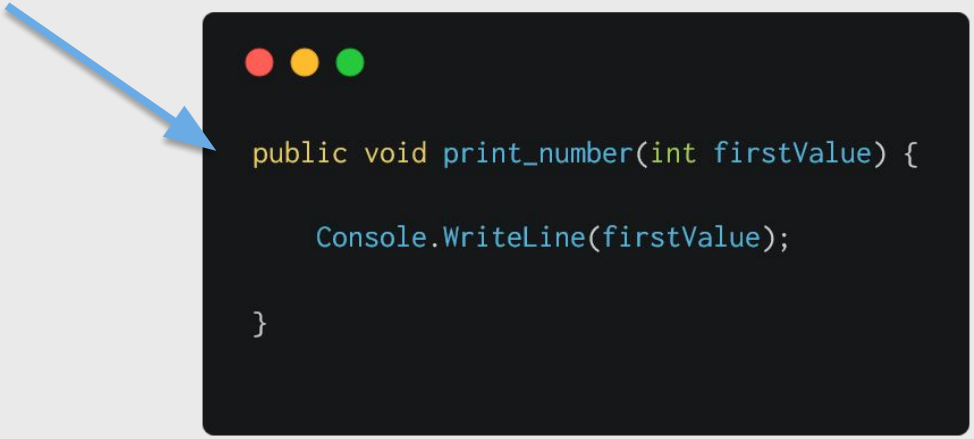
- C# Functions are more formally known as Methods.
- Functions make it possible to write code within a certain scope in any Class or file, but execute it anywhere in the applications code.
- Functions in C# have a certain structure they follow when being written.
- Functions in C# have a very similar look and feel to Functions in Java or C

<Access Specifier> <Return Type> <Method Name> (Parameter Type ; Parameter)

C# Function Structure

Access specifiers determine the scope of where a function can be accessed.

- **Public** allows the function to be accessed anywhere outside of the Class.
- **Private** prevents the function from being accessed outside the Class it's declared in.
- **Protected** is similar to **Private**, but grants access to subclasses



C# Function Structure

Return Types are the value types the function will return.

Void means a function has no return type, it will not be required to return a value at the end of its execution.

```
public void print_number(int firstValue) {

    Console.WriteLine(firstValue);

}
```

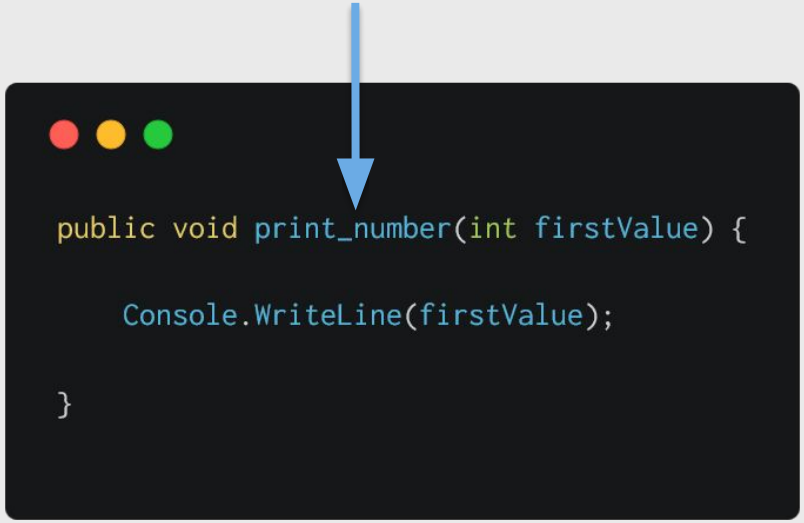
Returning data types with values, such as an **Int** or **Boolean** will require the function to perform an action in the code body that returns a value of the specified type.

```
public int return_number(int firstValue) {

    int return_value = firstValue + 10;

    return return_value;

}
```

C# Function Structure

Function Names are needed to define functions.

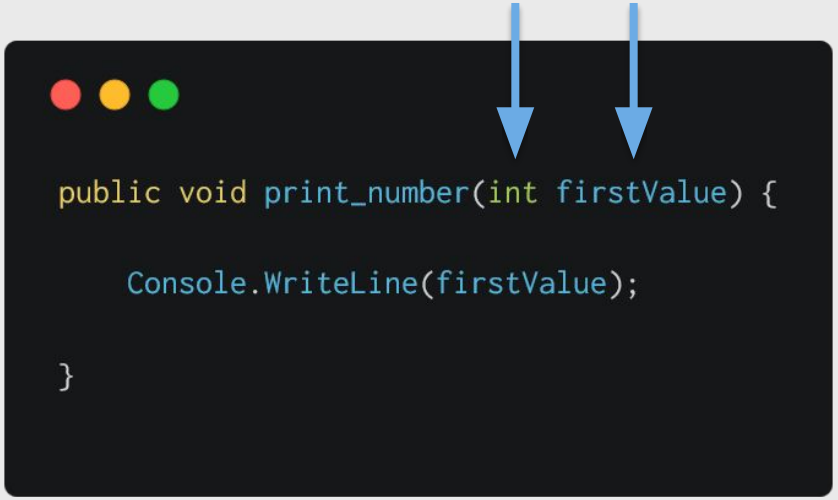
- They are used to call and execute functions from different locations in our code.
- A benefit to well thought out **Function Names** is that they can allow a user to know what a function does without reading the body of code that it executes.



C# Function Structure

Parameters provide the ability to pass values to functions.

- **Parameters** let you set what values and data types are able to be passed and accessed by the function.
- Having fixed **Parameters** also makes it simple to determine how many are able to be passed to a function.



C# File Structure

- C# makes use of **Classes** within it's file structure.
- This means that every variable initialized or Function declared will be inside a **Class**.
- Singular or multiple **Classes** can be within a single file.
- **Classes** are defined with the keyword **Class** and surround all pieces of code inside, between two curly braces {}.

```
using System;

class Program {

  public static void Main(String[] args) {

    ProgramFunctions class_object = new ProgramFunctions();
    class_object.printName("Bob");

  }

}

class ProgramFunctions {

  public void printName(string firstName) {

    Console.WriteLine(firstName);

  }
}
```

Whew! I bet that seems like a lot of information. Those are the basics of C#, which you'll need to be able to reference to fix the broken code!

Table of Contents

CarUserControl.CS

Now that you know some C#, you're going to put it to use to fix **`CarUserControl`**.

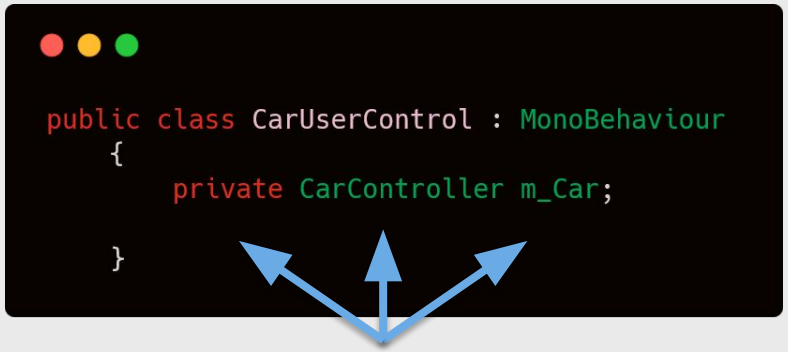
We will **create**:

- Several **variables** to store and access values
- **Instantiate** an **object** to reference another script
- Access the **Unity API**
- Write 2 **Void type functions**
- **Call methods** from other scripts

CarController Object

- To access functions from one Script inside another Script, we need to create an **Instance** of an object.
- **Instance** is a term used in programming to describe the creation of an object, which represents another code Class.
- Any object created will be able to call or access public functions and values of the Class it represents.
- We need to create an instance of **`CarController`** for our **`CarUserControl`** file to be able to use those functions.

CarController Object



1. Inside **`CarUserControl.cs`**, create a variable at the top of the **`CarUserControl`** class.

- The **Type** of our variable is the class we are creating an instance of, similar to using an Object as a type in Object Orientation code..
- We only want the reference inside the script we're currently coding, so `private` is suitable scope.

Who's Awake(); ?

2. We're going to use **`Awake()`** as a private void function
3. Using **`Awake()`** we can initialize **`m_Car`**, by executing the Unity function **`GetComponent<>().`**
4. Inside `GetComponent<>()` we will pass `CarController` as a generic parameter.

- **`Awake()`** is another pre-defined function in Unity used to initialize variables and set up references between Scripts.

```
public class CarUserControl : MonoBehaviour
    {
        private CarController m_Car;

        private void Awake()
        {

        }

    }
```

```
public class CarUserControl : MonoBehaviour
    {
        private CarController m_Car;

        private void Awake()
        {
            m_Car = GetComponent<CarController>();
        }
```

GetComponent<>

Why does this function use angle braces?

- `GetComponent` takes the variable Type as the parameter.
- The Type of that variable is a Script.
- Functions using <> are known as generic functions in C#.

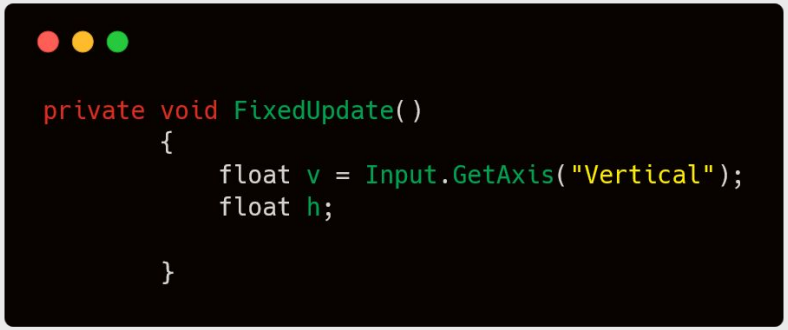
CarUserControl.CS

1. Now we will begin defining a void function called **`FixedUpdate()`**.
2. We will also need to create two Float variables inside the function body. These will hold values that contain decimal points, so Float is appropriate.

- **`FixedUpdate()`** is a predefined function inside the Unity application, meaning Unity already has set behaviour for handling this function.

CarUserControl.CS

3. We need to assign a value to our float variables. Using the Unity API we can assign an axis value to **`v`.**



```
private void FixedUpdate()
        {
            float v = Input.GetAxis("Vertical");
            float h;

        }
```

- **`Input`** is an interface created to interact with the Unity API.
- When we use **`Input`**, it's possible to call every public function it provides. One being **`GetAxis()`**;

The code written so far executes **`Input.GetAxis(“Vertical”)`** and assigns the return value to **`float v`**.

Find all Input functions @
https://docs.unity3d.com/ScriptReference/Input.html

CarUserControl.CS

We need to set the value for H in the exact same way as V.

4. We can use the exact same Input reference to complete another **`GetAxis()`** call.
5. The parameter we request this time is `Horizontal`, so both `v` and `h` are set with their respective values.

```
private void FixedUpdate()
        {
            float v = Input.GetAxis("Vertical");
            float h = Input.GetAxis("Horizontal");

        }
```

The new line written executes **`Input.GetAxis(“Horizontal”)`** and assigns the return value to `float h`.

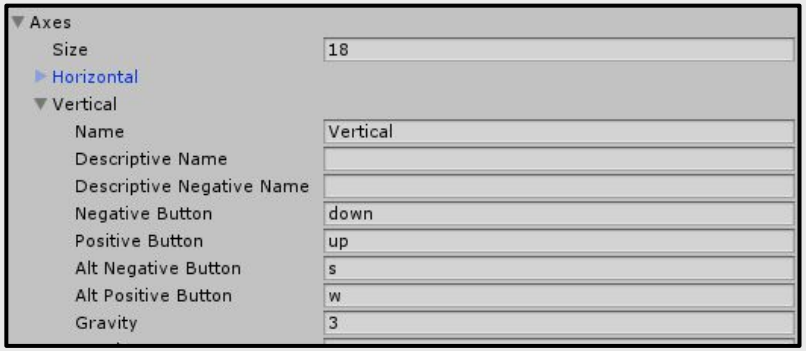
What does that code do?

- Horizontal and Vertical are axis values.
- They define our input as going forwards/backwards or left/right.
- By accessing them from the Unity API, we can tell if a user is pressing a key that should make our car go forwards, backwards or make a turn.
- The Unity Input Manager is where input axis and game actions are defined.
- This is what our **`GetAxis()`** performs for us.

Project Settings

The code makes requests to the Unity API. But we haven't set the Input values for the API to be listening for.

We need to access the Input Manager, to select certain Keys to control our game as they're pressed.



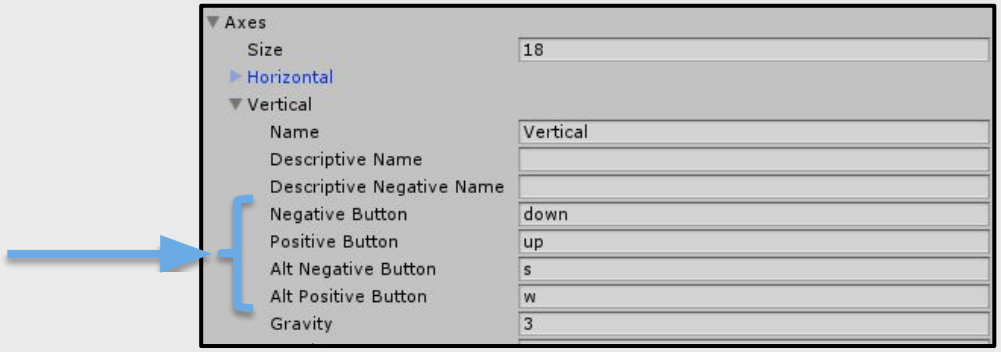
Input Manager Path:

Edit > Project Settings > Input>Axes>Vertical

Project Settings

The values we need to set are the positive and negative values:

- **Negative Button**
- **Positive Button**
- **Alt Negative Button**
- **Alt Positive Button**



Within the vertical menu setting

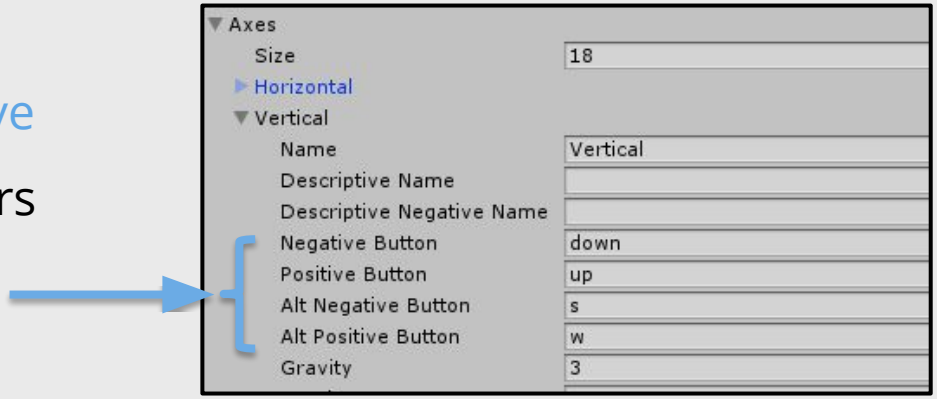
- Positive will accelerate the car.
- Negative will allow the car to reverse.

Within the Horizontal menu setting

- Positive will turn one direction.
- Negative will turn the other direction.

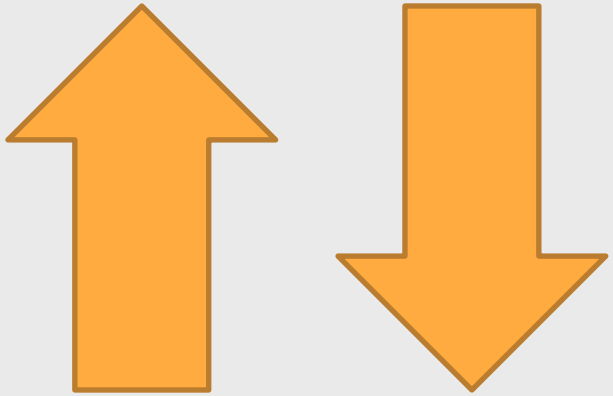
Project Settings

By setting the **Alt Negative**/Alt Positive input fields, we can input characters that will control our vehicle.



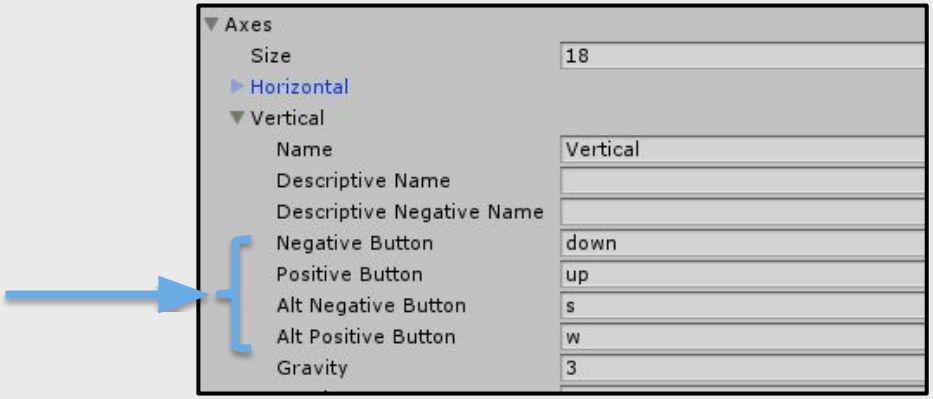
Let's set:

- **down** as **Negative**
- **up** as **Positive**
- **s** as **Alt Negative**
- **w** as **Alt Positive**



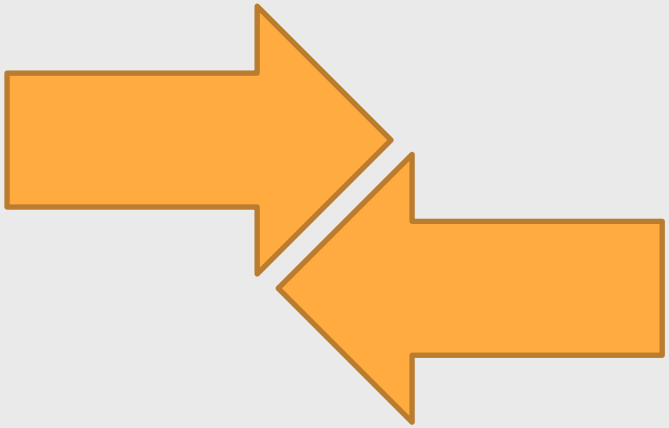
Project Settings

We have set the **Vertical** input, now the **Horizontal** values need to applied too.



Let's also set:

- **left** as **Negative**
- **d** as **Positive**
- **a** as **Alt Negative**
- **right** as **Alt Positive**



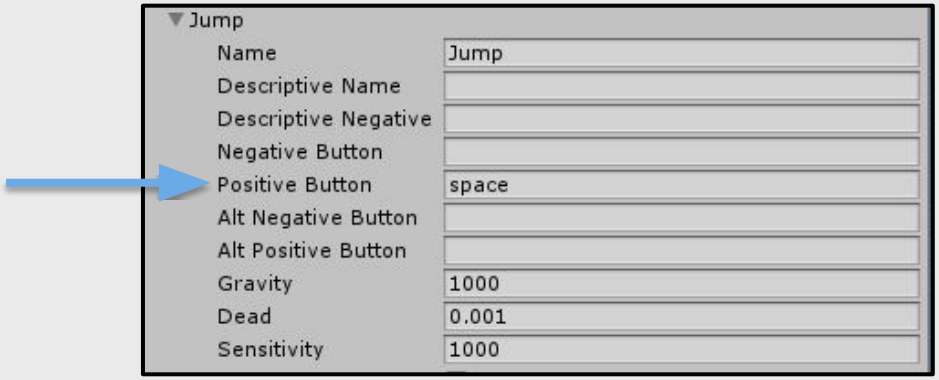
Project Settings

Despite the **Name** being **Jump,** this input will represent the vehicle handbrake.

Our Positive Button value is set to **space,** which is our keyboard spacebar.

Two keys were set for Vertical and Horizontal values because the car is able to move forwards, backwards, left and right.

This function has no alternate input, so only one key is needed to control this.



Input Manager Path:
Edit > Project Settings > Input>Axes>Jump

CarController.CS

CarController.cs is the file we use to create our object **m_Car**.

Inside **CarController.cs** there is a function named **Move()** on Line 129.

It is responsible for moving our Car. If we open it in our editor, we can see:

- It is a complex function
- It relies on many other functions
- It requires 4 float parameters

```
public void Move (float steering, float accel, float footbrake, float handbrake) {

  // Many Lines of Code In here

}
```

Preprocessor Directives

In the C# language, code statements found with a # (hashtag) at the beginning of a line are known as preprocessor directives.

These are boolean statements that can be used to include or exclude blocks of code from the application when the code executes.

There are many directives C# supports that provide different functionality to the language.

#if

#else

#endif

#define

#undef

#warning

Full list of C# pre-processor directives @
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/preprocessor-directives/

Preprocessor Directives

```
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

  }
}
```

Unity is able to use directives for platform dependent compilation.

This lets developers split scripts up into sections and compile areas of code exclusively for one supported platform.

The code is executed within the Unity Editor, this allows the code to compile specifically for any selected platform.

#if, #else and #endif

The use of Preprocessor Directives in this script is to check if the platform is or isn't mobile. Since the game makes use of a keyboard spacebar, we make sure that the key input can be assigned to a value.

1. Back in **`CarUserControl.CS`**, we're going to update the two lines we wrote previously by adding `CrossPlatformInputManager` to each line.

```
private Void FixedUpdate()
{
  // pass the input to the car!
  float h = CrossPlatformInputManager.GetAxis("Horizontal");
  float v = CrossPlatformInputManager.GetAxis("Vertical");
}
```

#if, #else and #endif

The use of Preprocessor Directives in this script is to check if the platform is or isn't mobile. Since the game makes use of a keyboard spacebar, we make sure that the key input can be assigned to a value.

2. Now we declare an **`#if`** at the very beginning of the line. The #if condition will be **!MOBILE_INPUT** meaning **not mobile input.**

```
private Void FixedUpdate()
{
  // pass the input to the car!
  float h = CrossPlatformInputManager.GetAxis("Horizontal");
  float v = CrossPlatformInputManager.GetAxis("Vertical");
#if !MOBILE_INPUT
}
```

#if, #else and #endif

3. Inside the **#if** statement scope, we need to initialise the float value `handbrake`.

3. We can do this the exact same as before, but using the variable name handbrake, and the **GetAxis()** parameter value **“Jump”**.

The code within this **#if** block will only execute if the platform is not recognized as mobile by Unity.

```
private Void FixedUpdate()
{
  // pass the input to the car!
  float h = CrossPlatformInputManager.GetAxis("Horizontal");
  float v = CrossPlatformInputManager.GetAxis("Vertical");
#if !MOBILE_INPUT
  float handbrake = CrossPlatformInputManager.GetAxis("Jump");
}
```

The float value should look like this

Float handbrake = CrossPlatformInputManager.GetAxis(“Jump”);

Call Move();

`Awake()` will create a reference to the **`CarController`** script when it is executed.

We can use our **`m_Car`** object to call public functions inside **`CarController.cs`**.

Function calls have a typical structure:

<object> <dot> <function name> <parenthesis; parameters>

Call Move();

Let's call the **Move()** function from the **CarUserControl** script. **Move()** is responsible for moving the car models in the game.

1. Inside our #if directive, after we initialise the handbrake variable, we will type our object **m_Car**.
2. Then separated by a dot **.** we can type the name of the function we want.
3. The **m_Car** object is calling a public function from the **CarController** script.

```
private Void FixedUpdate()
{
  // pass the input to the car!
  float h = CrossPlatformInputManager.GetAxis("Horizontal");
  float v = CrossPlatformInputManager.GetAxis("Vertical");
#if !MOBILE_INPUT
  float handbrake = CrossPlatformInputManager.GetAxis("Jump");
```

m_Car.Move();

We saw previously that `Move()` has 4 parameters that we have to pass as arguments.

- steering
- acceleration
- footbreak
- handbreak

```
public void Move(float steering, float accel, float footbrake, float handbrake)
        {

        }
```

Parameters [1], [2], [3]

The **`GetAxis()`** API call assigns keyboard input to our variables h and v.

Steering

- The horizontal input will control steering.
- The horizontal axis values will control the cars horizontal angle, moving forwards.

Acceleration

- Vertical input will provide an acceleration value.
- The vertical axis value will represent forwards and backwards motions.

Parameters [1], [2], [3]

4. Inside the **`Move()`** parameters, we need to pass our variables separated by a comma.

- h is passed as the steering value.
- v is passed as the acceleration value.
- v is also passed again, as the footbrake value.

```
private Void FixedUpdate()
{
  // pass the input to the car!
  float h = CrossPlatformInputManager.GetAxis("Horizontal");
  float v = CrossPlatformInputManager.GetAxis("Vertical");
#if !MOBILE_INPUT
  float handbrake = CrossPlatformInputManager.GetAxis("Jump");

  m_Car.Move(h, v, v, );

}
```

Our function call should now look like:

`m_Car.Move(h, v ,v , );`

Parameters [4]

The fourth parameter is required for **handbrake** control of the Car model.

```
private Void FixedUpdate()
{
  // pass the input to the car!
  float h = CrossPlatformInputManager.GetAxis("Horizontal");
  float v = CrossPlatformInputManager.GetAxis("Vertical");
#if !MOBILE_INPUT
  float handbrake = CrossPlatformInputManager.GetAxis("Jump");

  m_Car.Move(h, v, v, handbrake);

}
```

5. The **handbrake** variable declared above this function call is passed as the fourth parameter.

#else

6. Using the **`m_Car`** object, we will call **`Move()`** again but inside the **`#else`** directive in case handbrake is never initialised.

The parameters will be exactly the same **except** for **`handbrake`**.

This is because they were declared outside of the **`#if`** directives scope.

```
private Void FixedUpdate()
{
  // pass the input to the car!
  float h = CrossPlatformInputManager.GetAxis("Horizontal");
  float v = CrossPlatformInputManager.GetAxis("Vertical");
#if !MOBILE_INPUT
  float handbrake = CrossPlatformInputManager.GetAxis("Jump");

  m_Car.Move(h, v, v, handbrake);

#else

}
```

#else

7. The next **`m_Car.Move()`** call will be inside the scope of our **`#else`** directive.

8. Here we will pass through a value of ***`0f`***. This will meet the criteria for passing a float value but also leave the handbrake functionless, as no value is actually being passed.

```
private Void FixedUpdate()
{
  // pass the input to the car!
  float h = CrossPlatformInputManager.GetAxis("Horizontal");
  float v = CrossPlatformInputManager.GetAxis("Vertical");
#if !MOBILE_INPUT
  float handbrake = CrossPlatformInputManager.GetAxis("Jump");

  m_car.Move(h, v, v, handbrake);

#else
    m_Car.Move(h, v, v, 0f);
}
```

#endif

Anytime a condition directive uses **#if**, it needs to eventually be closed with **#endif.**

If you don't close the **#if,** then the scope will encompasses code you don't may not want inside that scope.

```
private Void FixedUpdate()
{
  // pass the input to the car!
  float h = CrossPlatformInputManager.GetAxis("Horizontal");
  float v = CrossPlatformInputManager.GetAxis("Vertical");
#if !MOBILE_INPUT
  float handbrake = CrossPlatformInputManager.GetAxis("Jump");

  m_car.Move(h, v, v, handbrake);

#else
    m_Car.Move(h, v, v, 0f);

#endif
}
```

This will complete our FixedUpdate() function!

Final Script

Including the code that was already inside `CarUserControls.cs` already, our script should now look like **this**.

```
using System;
using UnityEngine;
using UnityStandardAssets.CrossPlatformInput;

namespace UnityStandardAssets.Vehicles.Car
{
    [RequireComponent(typeof (CarController))]
    public class CarUserControl : MonoBehaviour
    {
        private CarController m_Car;

        private void Awake()
        {
            m_Car = GetComponent<CarController>();
        }

        private void FixedUpdate()
        {

            float h = CrossPlatformInputManager.GetAxis("Horizontal");
            float v = CrossPlatformInputManager.GetAxis("Vertical");
#if !MOBILE_INPUT
            float handbrake = CrossPlatformInputManager.GetAxis("Jump");
            m_Car.Move(h, v, v, handbrake);
#else
            m_Car.Move(h,v,v,0f);
#endif
        }
    }
}
```

Test Run!

When anybody writes code, they can make errors and run into problems. One way of countering this issue is by running our code, at incremental stages to see if it executes!

If the game runs, great! We're making progress.

If the code has an error, Unity will show a message at the bottom of the Unity window!

Let's test your game!

Whoa what's wrong?

A few parts of the app aren't working properly. You're going to fix them!

Problem II:

You can drive through WALLS. WHAT.

Table of Contents

Colliders

When two objects meet in a game, you might not want them to pass through each other. They need to have a physical presence in the game. Unity makes use of **colliders** for this.

Colliders can

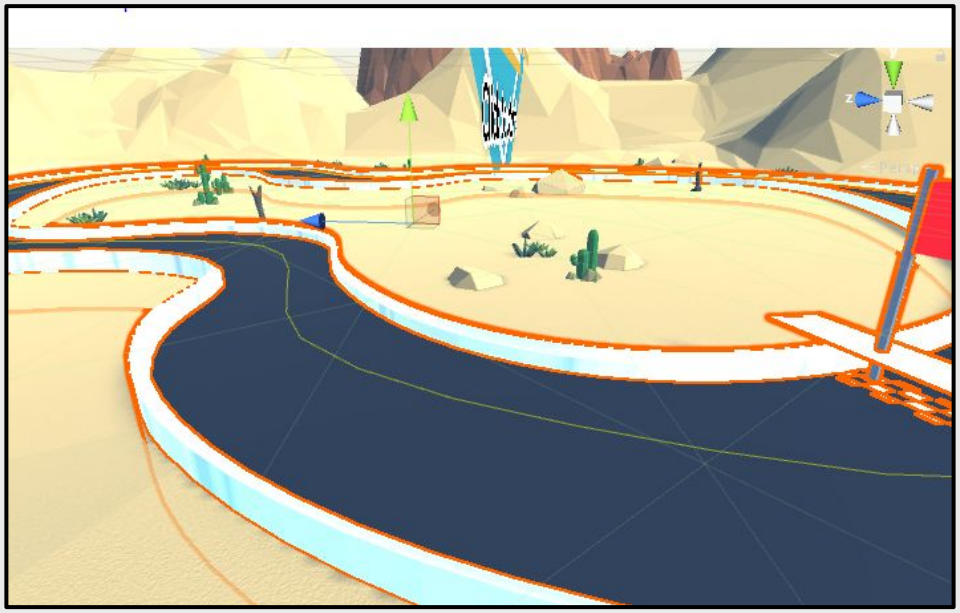
- Define the shape of an object for collision detection.
- Do not have the be the exact same shape as the model they are covering. Often a 3D model will be covered in colliders of all types of shapes!
- Unity can make use of **Static colliders** and **Rigidbody colliders**.

Colliders

1. If we **left click** the road, it will become outlined in red, showing the scale of the 3D object in the scene.

Left clicking an object inside the scene, will bring up details inside the Inspector tab.

You can do this for any object available in the scene.

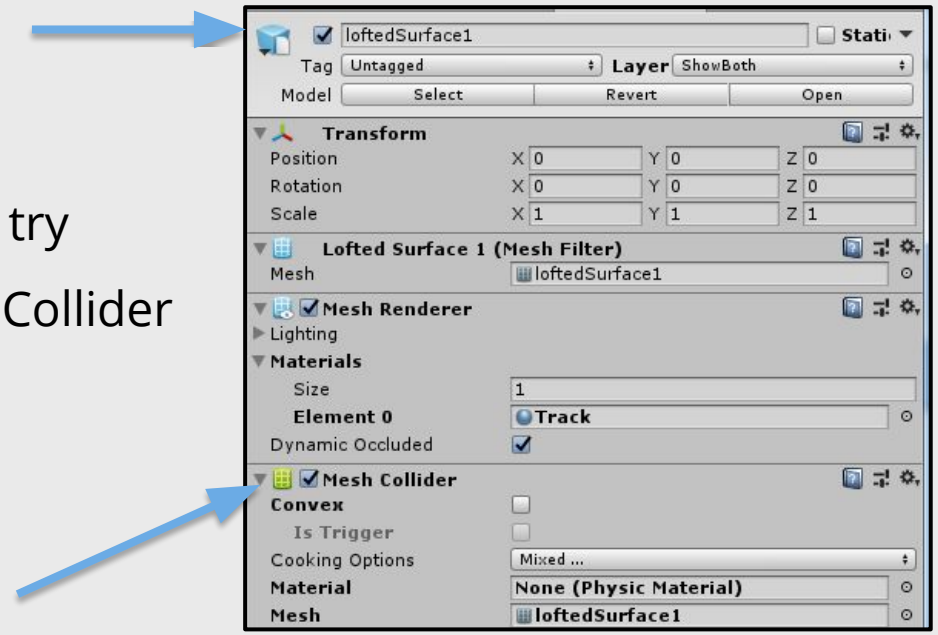


Colliders

The Inspector tab provides us with the object name '**loftedSurface1**'.

It also indicates it has a **Mesh Collider**, to make it a solid 3D object in the scene.

2. To see the difference this makes, try running the game with the Mesh Collider box **ticked** and then **unticked**.



Applying Colliders

Inspecting the walls to either side of the road shows that they have **no** colliders applied. Other moving objects like the Car would simply **pass through** them!

We currently know that **neither** wall around the track has colliders. This means:

- If either car drives into them we pass through completely
- The cars can't get back onto the track, because the road is a solid object, floating above the scene floor.

To fix this we are going to apply colliders to the necessary objects!

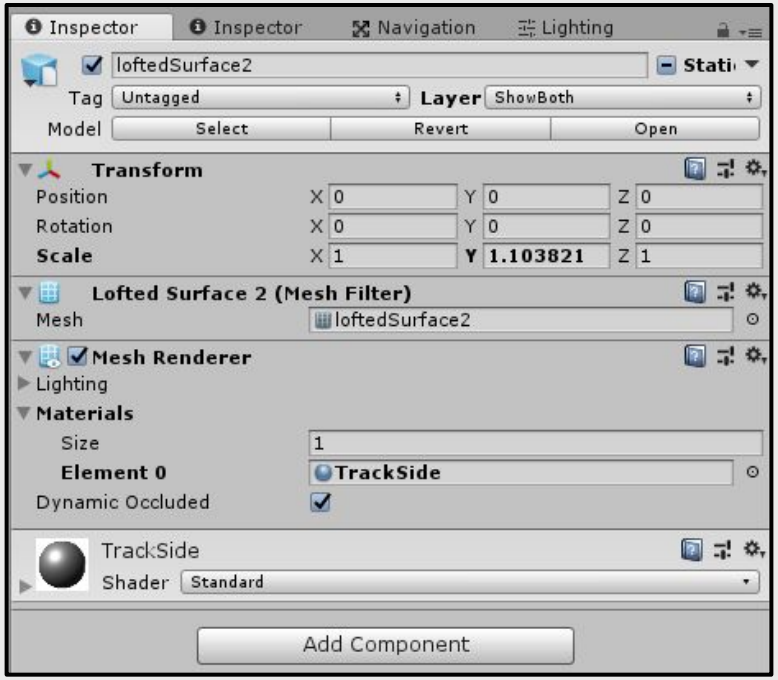
Mesh Colliders

3. Let's begin by clicking a wall object in our scene

Immediately the details in our **Inspector** tab change.

The wall is called **loftedSurfaceX** where X is a number 2 or 3.

It also has 2 components already applied, a **Mesh Filter** and **Mesh Renderer.**



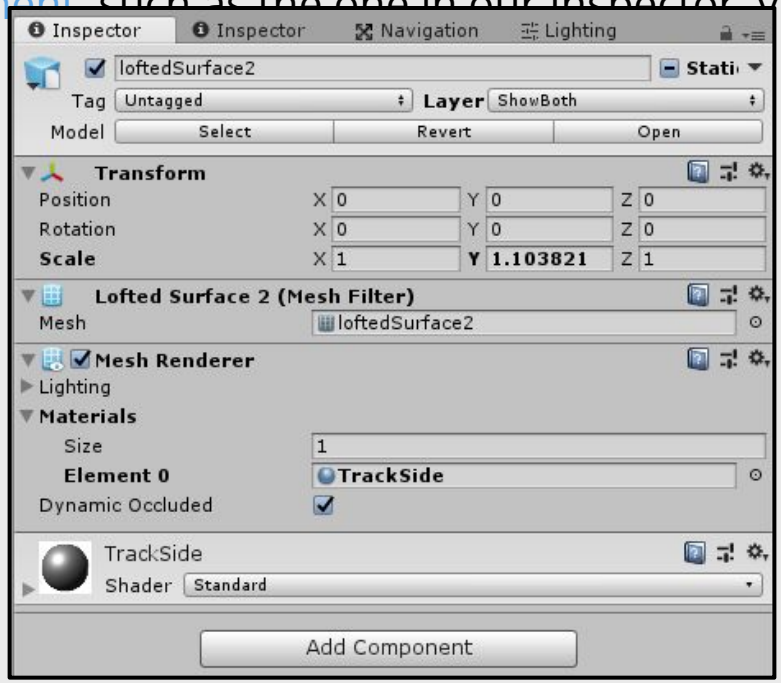
Mesh Colliders

What are these components?

A Mesh is a series of triangles that compose the 3D shapes we see in Unity Scenes.

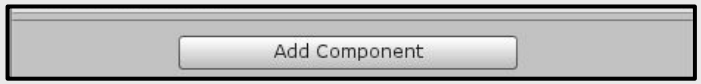
It's possible to take this data and apply textures or colors to the faces of triangles in a Mesh.

A Mesh Renderer component, such as the one in our Inspector, visually renderers the object in a Unity scene.

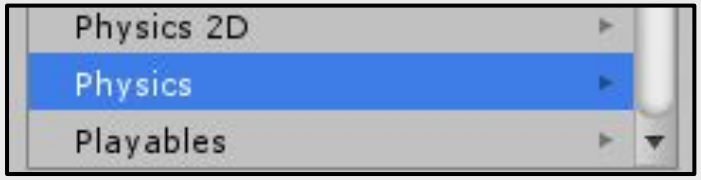


Applying Mesh Colliders

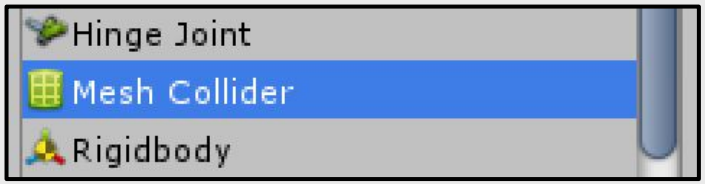
4. When we select a scene object, our Inspector will show a button saying Add Component.



Add Component will load a new menu, this will provide access to the Physics submenu.



It is within Physics, that we find Mesh Collider.



Applying a Mesh Colliders

Great! Now what do I do about the *other* wall?

We can follow the same practise to apply a Mesh Collider to that wall too!

Select the object in scene, navigate the Add Component menu

Add Component > Physics > Mesh Collider

Then let's take this game for a **Test Run!**

Test Run!

Now we have made another set of changes to the game, we should have a test run to see how they affect the game!

If our Mesh Colliders have been applied incorrectly, we **won't** receive an error message. We will just continue to pass through the scenery until we fix the issue!

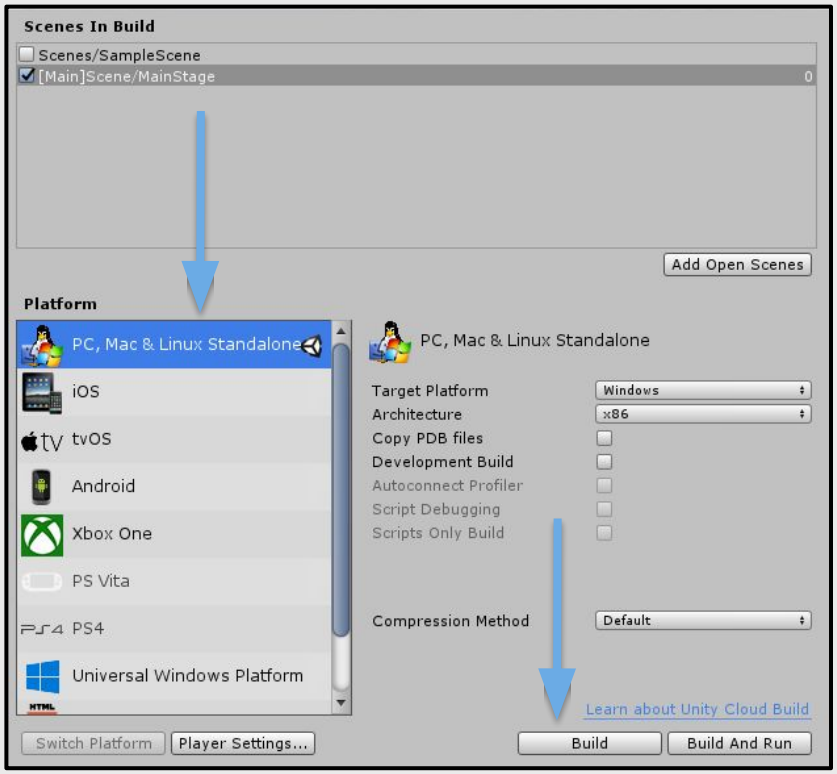
Let's build your game and try it!

What do you mean Build?

- When we develop and modify games in Unity, eventually we might want to run it outside of the Unity Editor as a standalone app.
- We are able to do this in Unity via the **Build Settings**.
- Here you are able to see every platform available to publish your game on. We are going to create a standalone for a desktop.
- We can build and export our game for multiple platforms using this menu in Unity.

Publishing Builds

1. Access the Build menu by going '**File > Build Settings**'.
2. Try choosing **'Pc, Mac & Linux Standalone'** in the platform selection.
3. Select 'Build' at the bottom of the window, a new window will open up to select where to save the build of the game.



You now have an **executable** of this project! Try run it on your machine!

Table of Contents

What did you learn today?

We created a fun quiz to test your knowledge and see what you learned from this workshop.

http://mlhlocal.host/quiz

Where to go from here...

1. Visit **mlhlocal.host/unity-learn** to learn more

2. Join the discussion at **mlhlocal.host/unity-connect**

3. Learn about Unity Hub - a new tool to make learning and collaborating easier.

Keep Learning: Practice Problems For Later

Extra Practice Problem 1:

Drag Asset models into the race track and add Mesh Filters + Colliders to them!

Extra Practice Problem 2:

Integrate actual AI into your game here:

Keep the Momentum Going!

1. Sign up to host this awesome workshop at https://localhost.mlh.io/

2. Running continuous workshops earns you points towards your Momentum Rewards

3. Redeem your Rewards Points for custom stickers, t-shirts, and pizza!

MLH localhost

Workshop

Accelerate Your C# Learning with Unity

MLH localhost

v.0.0.1