

# CS 410 Unity Documentation

By Joey Le

Supervised by Eric Wills

## Change Log

Date	Name	Update
4/21/2024	Joey Le	<ul style="list-style-type: none"><li>• Created the document</li><li>• Created introduction section</li><li>• Created Errors and Controls Section</li><li>• Wrote Addforce() Question</li></ul>
4/25/2024	Joey Le	<ul style="list-style-type: none"><li>• Wrote Merge Conflict subsection</li></ul>
4/27/2024	Joey Le	<ul style="list-style-type: none"><li>• Created Common Gameplay Elements Feature</li><li>• Wrote question on Camera subsection</li></ul>
4/30/2024	Joey Le	<ul style="list-style-type: none"><li>• Began writing Inputs subsection</li></ul>
5/2/2024	Joey Le	<ul style="list-style-type: none"><li>• Began writing “Which approach should I use?” for the Inputs subsection</li></ul>
5/4/2024	Joey Le	<ul style="list-style-type: none"><li>• Filled out “Which approach should I use?” for the Inputs subsection</li></ul>
5/19/2024	Joey Le	<ul style="list-style-type: none"><li>• Revised Errors section structure</li><li>• Added Movement subsection in Controls section</li><li>• Added Character Controller Subsection in Controls Section</li></ul>
5/25/2024	Joey Le	<ul style="list-style-type: none"><li>• Added Collision and Trigger Detection Subsection</li></ul>
6/1/2024	Joey Le	<ul style="list-style-type: none"><li>• Added virtual camera questions under Camera Subsection</li></ul>
6/2/2024	Joey Le	<ul style="list-style-type: none"><li>• Revised Duplication Error and Missing Prefab Error subsections</li><li>• Added Terminology Section</li></ul>
6/8/2024	Joey Le	<ul style="list-style-type: none"><li>• Added photos</li><li>• Italicize words that would be used in code</li><li>• Minor revisions across all sections</li><li>• Added WebGL question (source: Chroma Collapse)</li></ul>

# Introduction

This is a project created to assist CS 410: Game Programming students in working with Unity. For most students, they are introduced to Unity in this course. In the first two weeks of class, students are assigned two Unity game tutorials to become familiar with Unity. Afterwards, students are expected to work in groups and experiment with Unity to create a game from scratch during the eight remaining weeks of the term. While the class does dive into elements and techniques used in Unity such as linear algebra and patterns, students are expected to learn Unity on their own. This document is designed for students to understand basic scenarios they may or will come across.

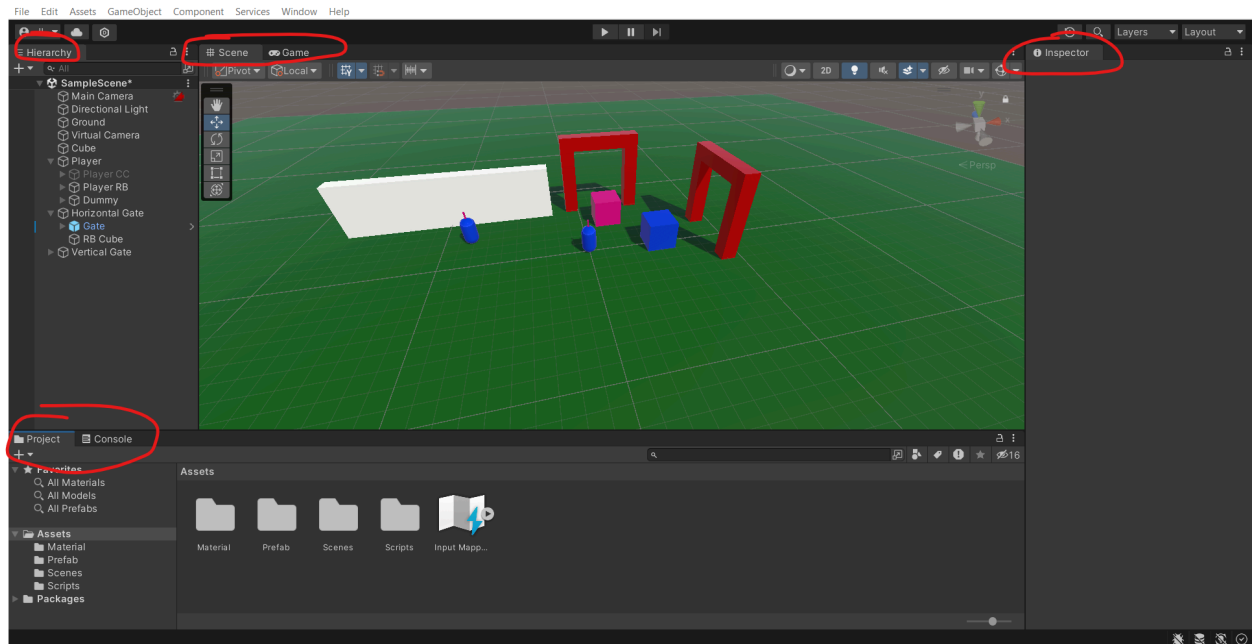
The goal of the document is to catalog different resources and situations to help students become familiar with Unity. Unity issues will be common, and it can be frustrating to not find an answer online or after spending a long period of time on it. The responses will be from students who took the class (just Joey for now) to help provide answers designed to current students as opposed to a broad online audience. Hopefully, future students can take on this project to help further the documentation after the current author(s).

To note, the concepts described here were done on version 2022.3.22f1 and on a Windows laptop and these should apply to other versions of Unity and different operating systems.

## Terminology

This section provides definitions for Unity.

- Scene - A set environment and space where you design your gameplay within Unity.
- GameObject - individual units or groups of objects used for designing the game. These are the building blocks for designing the game. This term is used interchangeably with “object”.
- Prefab - A predefined collection of GameObjects to form one single cohesive object to be used.
- Windows - The different tabs placed throughout the user interface where one can view information about particular elements of their game.



**Figure:** A sample scene with the different windows circled in red.

- Hierarchy - Usually on the left of the side of the screen that displays all the GameObjects in the scene.
- Scene - Lets you interact and design your game.
- Game - Shows gameplay when playtesting.
- Inspector - Usually on the right side of the screen that lets you view all the properties of GameObjects
- Project - Lets you see all your files and their hierarchy structure.
- Console - Where errors, warnings, and outputs are displayed.
- Component - Add-ons to GameObject that let you customize the properties of GameObjects. Found when inspecting GameObjects in the Inspector window.
  - Transform - Refers to the physical placement of GameObjects in regards to the scene.
  - Scripts - C# code that you write that lets you design how a GameObject operates.
  - Rigidbody - Offers support and usage for Unity's built-in physics system.
  - Collider - Provides physical "hitboxes" in interacting with an object. This includes box, sphere, and capsule colliders. Mesh colliders work a bit differently than these colliders, so only some discussions regarding colliders apply to mesh colliders..
  - Virtual Camera - A component that comes from the Cinemachine package that lets you design how the game is viewed when playing.
- Action - A defined type of input that a player can perform to interact in the game.

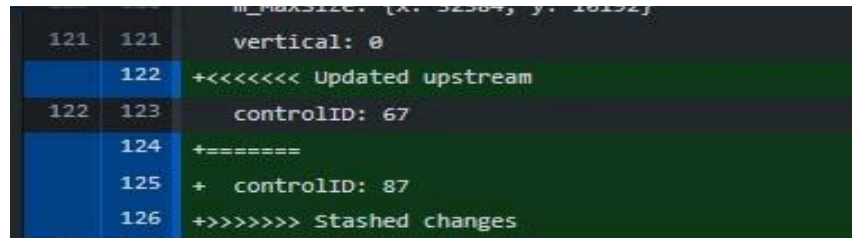
# Errors

This subject focuses on bugs that would appear on the console window and with Git.

## Merge Conflict General Information

- Why do I get merge conflicts when trying to push my work onto Git?
  - Merge conflicts can be the result of trying to incorporate the project's repo from Git into your project. This can affect the following below. It does this by having conflicts with their unique IDs.
    - Scripts
    - Scenes
    - Prefabs
  - In any of these events, there is usually some contradictory information with your current set-up when trying to import someone else's set-up.
    - Ex.) Your version of a prefab has a different GUID (in .Meta file) compared to another group member's.
  - Avoid merge conflicts with the following strategies
    - Communicate which scenes are being worked on. Ideally, do not have many people affect the same script, scene, or prefab properties and at the same time.
      - If this is unavoidable (ex. There is only one scene), only alter different objects/prefabs and scripts. Ideally, working more with different prefabs will minimize conflicts.
    - Keep git commits and pushes as small as possible. By submitting changes as small as possible, this makes it easier for others to identify issues.
    - Always pull the latest changes from other's work before you get started.
  - References
    - <https://www.youtube.com/watch?v=YgoCp2tzRh0>
    - <https://forum.unity.com/threads/github-scene-merging.1034548/>
    - <https://forum.unity.com/threads/how-can-i-resolve-github-desktop-unity-merge-conflicts.1195684/>
    - <https://docs.unity3d.com/2020.1/Documentation/Manual/UnityCollaborateResolvingConflicts.html>
- Dealing with merge conflicts
  - Inspect the differences between your file and the file on GitHub; the GitHub desktop app allows you to see the differences in files that cause a merge conflict.
    - If there are any file differences such as different GUID values in .meta files, these may be displayed as stashed or updated. If that's

the case, then edit your file to keep one or the other (ex. match GitHub's GUID and how it's formatted by keeping the "stashed" or "updated" information)



```
121 121      vertical: 0
122 122      +<<<<<<< Updated upstream
123 123      controlID: 67
124 124      +=====
125 125      + controlID: 87
126 126      +>>>>>>> Stashed changes
```

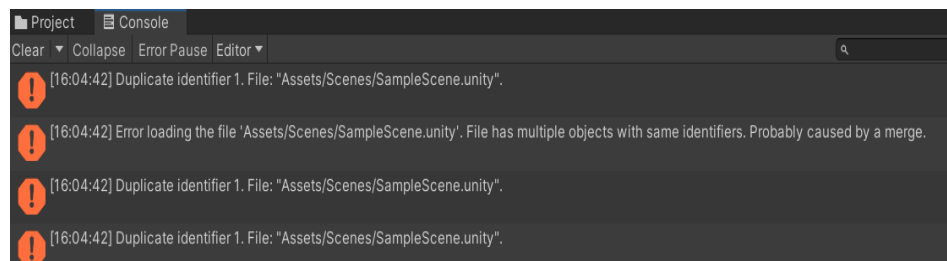
**Figure:** A representation of how file differences can be represented in the file. In this case, you would need to keep one of the controlID values. (Source:

<https://stackoverflow.com/questions/72857783/unity-github-conflict-ng-dwlt-file>)

- Ensure that your GUID is consistent with the GUID found on GitHub and vice versa.
- If the merge conflict is greatly significant, you may have to consider omitting or deleting parts of your work to allow for the merge/duplicate object to be implemented first.
  - This can involve replacing an entire file's information with the file information stored on GitHub

## Duplication Error

- I got a duplicate identifier error, how do I deal with it?
  - These usually come as the consequences from resolving merge conflicts. The merge conflict supposedly gets resolved, but this results in some values (lines of information) being duplicated or altered in files that end with .unity or .prefab.
  - To fix this error, edit the error file(s). The Console window will display what files contain the duplicate identifier error.



```
Project Console
Clear Collapse Error Pause Editor
[16:04:42] Duplicate identifier 1. File: "Assets/Scenes/SampleScene.unity".
[16:04:42] Error loading the file 'Assets/Scenes/SampleScene.unity'. File has multiple objects with same identifiers. Probably caused by a merge.
[16:04:42] Duplicate identifier 1. File: "Assets/Scenes/SampleScene.unity".
[16:04:42] Duplicate identifier 1. File: "Assets/Scenes/SampleScene.unity".
```

**Figure:** Example of a duplicate identifier error on the console. The console identifies that it's coming from SampleScene.unity

- For the .unity files, these may open up the project itself so you'll need to open these with a text editor.
- If you see something like the three lines below appear more than once, delete them and all the information below it until one copy remains. These usually appear at the top of the file.

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!29 &1
```

**Figure:** Data corresponding to a .unity file; yours may look slightly different especially if it's a .prefab file.

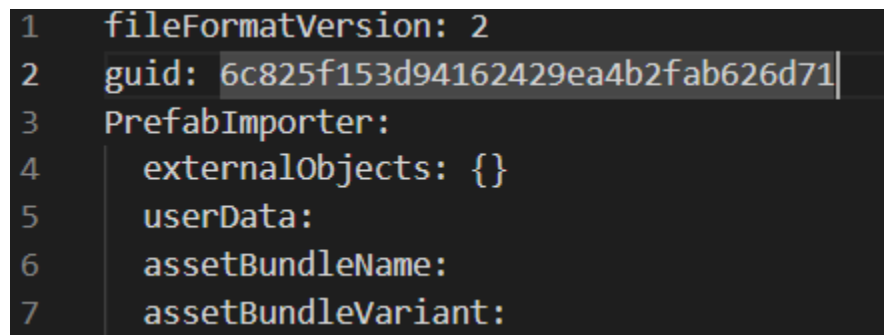
- This provides a built-in ID for Unity to recognize these files and the info below describes inspector qualities and other scene information for that file, so duplicate errors result in a single file having multiple sets of information when they should only have one.
- Essentially, a duplicate identifier error file can involve multiple IDs for that file, so Unity gets confused on how to recognize a file.
- If that doesn't work or the duplicate information can't be found, replace all the text in that file with what is found on GitHub or another person's respective copy of the file.
- References
  - <https://issuetracker.unity3d.com/issues/duplicate-identifier-error-can-report-the-wrong-object-identifier#:~:text=In%20the%20case%20of%20merge%20problems%20some%20portion,a%20message%20like%3A%20Duplicate%20identifier%20624274415.%20File%3A%20%22Assets%2FDuplicateFileIDs.unity%22>.
  - <https://forum.plasticscm.com/topic/23421-merge-bug-leads-to-unopenable-and-irrevertable-scene/>

## Null Reference Error

- When I start a game or while playing, some NullReferenceException error in the console?
  - Null references result from a script that requires a GameObject or another script in its code.
    - In *Roll-a-Ball*, a null reference will occur if you take out CountText or WinText from the PlayerController script attached to the player.
    - The console window will display the script that is missing some dependency and the line where it would need that dependency.

## Missing Prefab Error

- For some reason, Unity says that a prefab is missing even though we still have the prefab to use?
  - This usually results from pulling your project from Git repo. Each prefab and scene has a .meta file with a GUID value to identify the object. Sometimes the .meta is lost or overwritten, and usually this gets the GUID to be replaced.
  - Solution 1:
    - Edit the prefab's .meta file by opening it in a text editor.
    - Find the GUID value ("guid: <guid value> ") usually at the top of the file.
    - Replace the current GUID with the GUID found in GitHub.



```
1 fileFormatVersion: 2
2 guid: 6c825f153d94162429ea4b2fab626d71
3 PrefabImporter:
4   externalObjects: {}
5   userData:
6   assetBundleName:
7   assetBundleVariant:
```

**Figure:** What a normal .meta contents file looks like. GitHub will have its own copy in the repo.

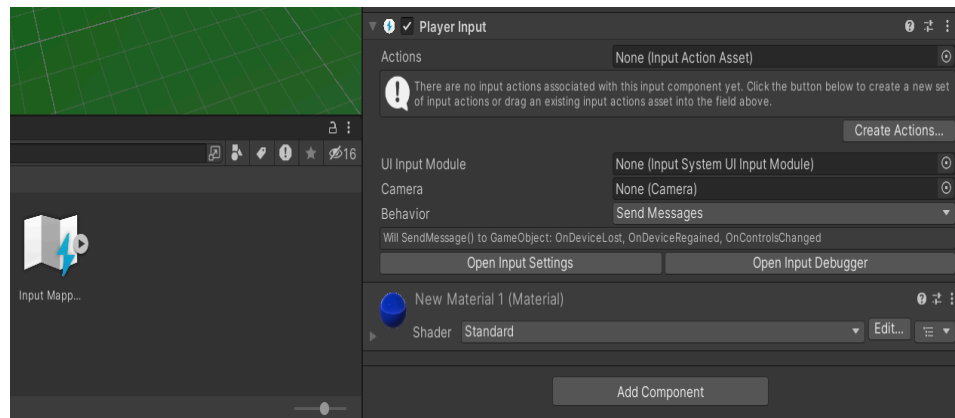
- Solution 2 (the GUID can be found in the object's name)
  - Copy its GUID found in the name. This can be done by looking at the object's name in the Inspector window at the top.
  - Access the .meta file and replace its GUID with the copied GUID.
  - For reference, see the first link under the References points.
- Solution 3 (the problem involves duplication identification error)
  - Perform the solution described in the Duplication Error section.
- References
  - <https://docs.google.com/document/d/1GJODylszZKtniA9l4bcbh-WyBQ8JAqznT-TkHJvG8zM/edit>
  - <https://medium.com/codex/solving-the-missing-prefab-issue-in-unity-3d-ae5ba0a15ee9>
  - <https://www.dragonflydb.io/error-solutions/unity-missing-prefab>

# Controls

This subject focuses on common actions in games (moving, jumping, attacking) and how Unity would take in inputs.

## Inputs

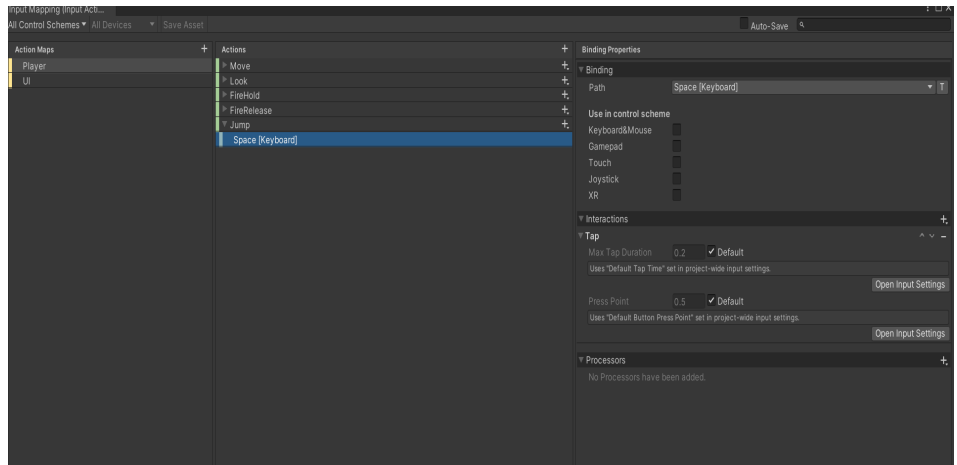
- What are the different forms of input?
  - Player Input/New Input System
    - This involves using a “Player Input” component on the Player object and creating a new Input Action (a mapping between controller buttons to in-game actions) or using a pre-established Input Action.



**Figure:** Player Input component that is attached to a GameObject. The “Create Actions” button would need to be pressed to create an Action for the component to use. It will require saving a new file (into the Asset file by default). It will appear in the Project window to view.

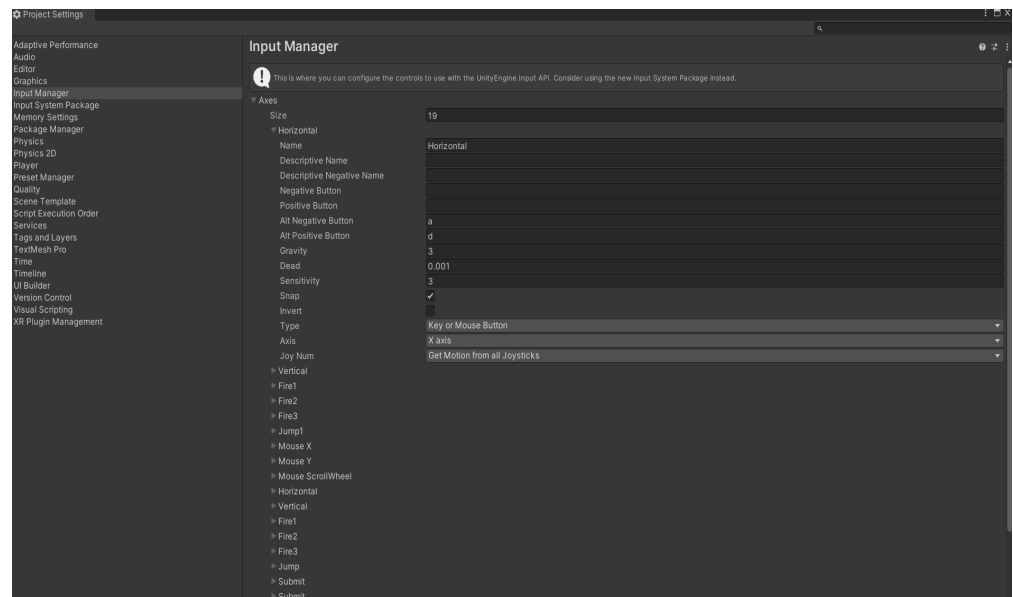
- An Input Action is a UI in your assets folder where you define the name of actions and how to trigger them (more on this later) and you write code to handle the specific action.
  - Each Action is stored in its own function and starts with “On” (Ex. An Action named Jump would be called “OnJump()” in the code)





**Figure:** An example of what the UI looks like for an Input Action. While it's hard to see, there are a couple of Actions listed with Jump being an example and is set to use the space button.

- This is what *Roll-a-Ball* used.
- Input Manager/Old Input System
  - This is an input setting built in your Project.
  - To access its UI, Edit (top of Unity) → Project Settings (towards the bottom) → Input Manager (left-hand side and towards the top)



**Figure:** The layout of the Input Manager. Each action has a drop-down menu that provides more details for that action. In the image, the Horizontal Action/axis's drop-down menu is present.

- The Axes section displays the names of each Action with their own drop down menu to specify how to activate them.

- Each Action can be recorded as a float value between -1 to 1 through the “Negative Button” and “Positive Button” text boxes respectively. The value is used in code.
  - Access the value with “*Input.GetAxis(String axisName)*” where *axisName* is the Action name.

```
void move() {
    movX = Input.GetAxis("Horizontal");
    movY = Input.GetAxis("Vertical");
    mov = new Vector3(movX, 0.0f, movY) * spd;
```

**Figure:** Example of using *GetAxis()* by getting the Horizontal and Vertical axes to set a Vector3 for movement.

- Each Action can be recorded as a Bool based on the button press. The value would be put in a conditional statement to perform the action.
  - *Input.GetButton(String axisName)* - Remains true as long as the player holds the key down
  - *Input.GetButtonDown(String axisName)* - True only when the player presses the key, but will need to release and press again to get another True response.
  - *Input.GetButtonUp(String axisName)* - True only when the player releases their finger from the button.

```
if (Input.GetButtonDown("Jump")) {
    doJump = true;
}
```

**Figure:** Example of using *GetButton()* to perform a jump.

- There are alt buttons for the Negative and Positive Button to provide additional buttons to perform the same Action
- This allows multiple buttons to be associated with a single Action.
- Example: The default values defined in the “Horizontal” Axis can be applied to move left and right.
  - The negative buttons (‘a’ and left arrow key) allow the value to be towards -1
  - The positive buttons (‘d’ and right arrow key) allow the value to be towards 1.
  - The code would apply the value in a Vector3 (x-axis) to specify the direction.

- Gather with `Input.GetAxis("Horizontal");`
    - This is used in *John Lemon's House of Jaunt*.
- `Input.GetKey(KeyCode key)`
  - This involves directly specifying which keyboard character is pressed in the code.
    - Unity has given names for each keyboard button which gets specified in `key`.
    - For a list of KeyCodes, see here: <https://docs.unity3d.com/ScriptReference/KeyCode.html>
  - Can put this in a conditional statement where the action is performed.
  - This is a function that returns a bool.
  - Variations include:
    - `Input.GetKey(KeyCode key)` - Remains true as long as the player holds the key down
    - `Input.GetKeyDown(KeyCode key)` - True only when the player presses the key, but will need to release and press again to get another true response.
    - `Input.GetKeyUp(KeyCode key)` - True only when the player releases their finger from the button.
  - Example: `Input.GetKeyDown(KeyCode.Space)` - allows the code to check if the player has pressed the Space button
- Which approach should I use?
  - This can vary based on personal preferences and how you would like to organize your code.
  - Here are some notes for each approach:
    - Player Input/New Input System
      - Requires inputs that influence actions to be isolated from `Update()` and easily identified with "On" in their function name.
      - Multiple assets can be made to allow a different set of controls for the same object. Also, many buttons can be assigned into a single action and as part of a hierarchy.
      - The Interactions section in the UI allows specification on how the player should interact with the button to register the action (hold, tap, tap and release, etc.)
        - For more information: <https://www.youtube.com/watch?v=rMlcwtoui4I&t=373>

- Can record various types of variables beyond just float when the player performs an input.
- If its code component is disabled and the Player Action component is enabled, the fun will run but the rest of the code won't. (ex. If the function has a *Debug.Log()* in it, then it will still print to the console despite disabling the code component).
- Potentially harder to use and less intuitive
- Overall, this approach has a higher skill ceiling and is tougher to find solutions online, but can allow more versatility for designing inputs.
- For more information:  
<https://docs.unity3d.com/Packages/com.unity.inputsystem@1.8/manual/PlayerInput.html>
- Input Manager / Old Input System
  - Has a more intuitive UI to configure controls
  - Gathering only float and bool values provides a more simple process of getting player inputs, but limits other value alternatives.
  - Controls are stored within the engine itself rather than in code or a separate asset.
  - Acquiring inputs can be structured similarly to the new input system, so that separate functions are designed to collect player inputs like the On-functions.
  - Allowing players to remap controls may be complicated.
  - Decent documentation on what each Axis quality does.
  - Overall, this approach is simple to learn and understand. It strikes a fine balance between having Unity's built-in system and your code to read on player inputs. It does struggle with allowing adjustments if you want to give players remapping options or multiple buttons for a single action, but there are potential workarounds such as having the code change what Axis gets used instead. However, it is easy for developers to change the control themselves.
  - For documentation information:  
<https://docs.unity3d.com/Manual/class-InputManager.html>
- *Input.GetKey(KeyCode key)*
  - This is essentially hard-coding the player controls in the scripts.

- Simple for the most part, and usually requires being put into conditional statements to perform the action.
- Allow many buttons to be assigned to a particular action with “or” or “else if” in the conditional statements.
- Adjusting the control involves the need to change the code itself rather than through an asset or in the settings.
- Overall, this can be considered the most simplest approach. It does all of its operations in the code, so this involves more work on your end to make adjustments.
- Alternatively, you can combine parts of each approach.
  - Ex. Using “*OnMove()*” to perform movement, *GetAxis(“Jump”)/GetButton(“Jump”)* for jumping, and *GetKey(<Keycode>)* to perform some other commands.
  - It can be more common to have the old input system and *Input.GetKey()* be used together.
  - Can offer flexibility in your development process, but can be harder to keep track of every control.

## Movement

- What are different ways to move?
  - Abbreviations and key for discussion below:
    - rb: rigidbody component
    - cc: Character Controller component
    - <movement vector>: a Vector3 of the collected movement
    - <speed value>: a float or int to determine how fast the object moves
  - If you want to affect the gameobject’s position in the world, you can affect its position component (*transform.position* or *movePosition()*). Setting the positions is readjusting the object’s position in the game’s world space while *Translate()* makes the object move across the world.
    - *transform.position = transform.position + <movement vector> \* Time.deltaTime \* <speed value>;*
    - *transform.Translate(<movement vector> \* Time.deltaTime \* <speed value>, Space.World);*
  - If the gameobject has a rigidbody, you can affect it’s rigidbody (<rigidbody name>.position or *movePosition()*) or apply a force (*addForce()*)
    - *rb.position = rb.position + <movement vector> \* Time.deltaTime \* <speed value>;*
    - *rb.MovePosition(rb.position + <movement vector> \* Time.deltaTime \* <speed value>)*

- `rb.Addforce(<movement vector> * <speed value>)`
- If you wish to use a character controller, you can use `<character controller name>.Move()`
  - `cc.Move(<movement vector> * <speed value> * Time.deltaTime);`

```
void move() {
    movX = Input.GetAxis("Horizontal") ;
    movY = Input.GetAxis("Vertical");
    mov = new Vector3(movX, 0.0f, movY);

    Vector3 desiredForward = Vector3.RotateTowards (transform.forward, mov, turnSpd * Time.deltaTime, 0f);
    // Opt 1: rigidbody.movePosition(<vector3 destination>)
    if (useMove == 0)
        rb.MovePosition(rb.position + mov * Time.deltaTime * spd);
    else if (useMove == 1)
        // Opt 2: rigidbody.position
        rb.position = rb.position + mov * Time.deltaTime * spd;
    // Opt 3: transform.Translate(<Vector3 increment>, <relative space>)
    else if (useMove == 2)
        transform.Translate(mov * Time.deltaTime * spd, Space.World);
    // Opt 4: transform.position
    else if (useMove == 3)
        transform.position = transform.position + mov * Time.deltaTime * spd;
    // Op 5: rigidbody.velocity (Not the most ideal)
    else /
        rb.velocity = mov * spd;

    // Rotation
    if (mov != Vector3.zero)
        rb.MoveRotation (Quaternion.LookRotation(desiredForward));
}
```

**Figure:** Example of a function that moves an object through its rigidbody or transform property based on an int variable *useMove*.

- How do all the movement adjustments with rigidbody work?
  - Setting the position just sets up where the object is located in space; this essentially involves teleporting the object to a different point in the game space. Using `Time.deltaTime` allows position-setting to feel like movement. This applies to `transform.position` as well.
  - `rb.movePosition()` invokes the object to move across the game's space and factors in the rigidbody's interpolation.
    - Here, `movePosition()` is more built to perform movement than just setting the rigidbody position because this function allows a more smooth transition of the object and accounts for interpolation (rigidbody setting to address potential jittering)
  - `rb.Addforce()` involves pushing the object with some force (either invisible or visible) onto the object. The Jumping section describes more information on how this works.

- This can invoke a little bit of “after-movement” if the object is applying physics like friction. (Ex. The ball continues to slightly move after moving in *Roll-a-ball*)
- `rb.velocity` is another approach to consider, but the documentation considers using *Addforce()* instead because velocity is more designed for immediate changes rather than gradual movement.
- References:
  - <https://docs.unity3d.com/ScriptReference/Rigidbody-position.html>
  - <https://docs.unity3d.com/ScriptReference/Rigidbody.MovePosition.html>
  - <https://docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html>
  - <https://docs.unity3d.com/ScriptReference/Rigidbody-velocity.html>

## Jumping

- What do I need to consider when making a successful jump
  - Gravity - if not specified in a rigidbody or elsewhere, the object will not come down
    - Keep in mind that RigidBodies have a “Use Gravity” checkbox and if checked, the game will incorporate this into your object with all other forces applied to it.
    - This works similar to *rb.AddForce(Physics.gravity, ForceMode.Acceleration);*
  - Ground state - the ground places a role in enabling a jump if you wish to limit jumps (ie. prevent infinite jumping)
    - Usually, you can create a function or statement to check whether your object is touching the ground or not
    - The Character Controller has a built-in function to check if the object is touching the ground
  - How the jump works - this can involve making double jumps, variable jump (height of jump changes based on how long the button was held), and the object’s properties while mid-air.
- What are different options to jump with?
  - *Rigidbody.AddForce()*
  - Setting *Rigidbody.velocity()*
    - Ex) *rb.velocity = new Vector3(0, <jump height float>, 0)*
  - *transform.Translate(Vector3 Direction)*
    - Direction - A vector3 that specifies where the object will go
    - This works best if you merge your (horizontal) movement along with the jump into the same *Translate()* call.
  - *CharacterController.Move()*

- Even with a rigidbody, you'll need to incorporate gravity into the code.

```
float gravityVal = -9.81f;
Vector3 playerVelocity;
playerVelocity.y += gravityVal * Time.deltaTime;
cc.Move(playerVelocity * Time.deltaTime);
```

**Figure:** Example of code that would be set up to use *CharacterController.Move()*; Character Controller (cc) is defined elsewhere.

- Incorporate *Time.deltaTime* to make the *Move()* call feel like movement.
- How does *AddForce()* work with a rigidbody trying to jump?
  - *AddForce()* takes two optional parameters as input
    - *Vector3 force* - this is the direction you want the object to move, normally you can do *Vector3.up* to specify a consistent upwards direction
      - You can think of this as like the game object's initial position being a start point and <force> as a destination. The function moves the object to <force>.
    - *ForceMode mode* - this specifies how the force should be applied to the object.
      - There are four different modes to type and vary based on its reliance on DeltaTime (DT) and the object's mass (set by the rigidbody).
        - *ForceMode.Force* - relies on both DT and mass.
          - $AddForce() = (force * DT) / mass$
        - *ForceMode.Acceleration* - relies on just DT
          - $AddForce() = (force * DT)$
        - *ForceMode.Impulse* - relies on just mass
          - $AddForce() = (force / mass)$
        - *ForceMode.VelocityChange* - does not rely on DT or mass
          - $AddForce() = force$
- force = <force> as described above;
- Think of DT as <1 for now.
- Each coordinate interacts with DT and mass individually to get a three coordinate (Vector3) answer
  - ie.  $(x, y, z) * DT / mass = (x * DT / mass, y * DT / mass, z * DT / mass)$ .



- References

1. <https://docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html>
2. <https://docs.unity3d.com/ScriptReference/ForceMode.html>
3. [https://www.reddit.com/r/Unity3D/comments/uxcviq/how\\_exactly\\_do\\_es\\_unity\\_apply\\_gravity\\_to/](https://www.reddit.com/r/Unity3D/comments/uxcviq/how_exactly_do_es_unity_apply_gravity_to/)

## Character Controller

- I notice a Character Controller component, how does this work compared to the input systems, rigidbodies, and movement?
  - Information on this is spread throughout this section, so this is to group all that information here.
  - This is a component like a rigidbody and it specializes in providing a simple approach to move the player.
    - It may have some weird interactions with rigidbodies, so it may be worth having one or the other.
  - This component provides a simplified way to move the player and can use any of the input approaches to move.
    - You can use the built in *CharacterController.Move(Vector3 movementVector)* function to move
      - Useful for horizontal and vertical movement
      - Can separate horizontal and vertical movement in different calls.
    - Has a built-in *isGrounded* variable to check if the object is contacting a surface
      - Ex) *cc.isGrounded == True;*
  - Can specify how the object can interact with slopes and stairs.
  - Does not allow interactions with Unity's physics system like rigidbody
  - References
    - <https://www.youtube.com/watch?v=e94KggaEAr4&list=PLwyUzJbFNeQrlxCEjj5AMPwawsw5beAy>
    - <https://docs.unity3d.com/ScriptReference/CharacterController.html>

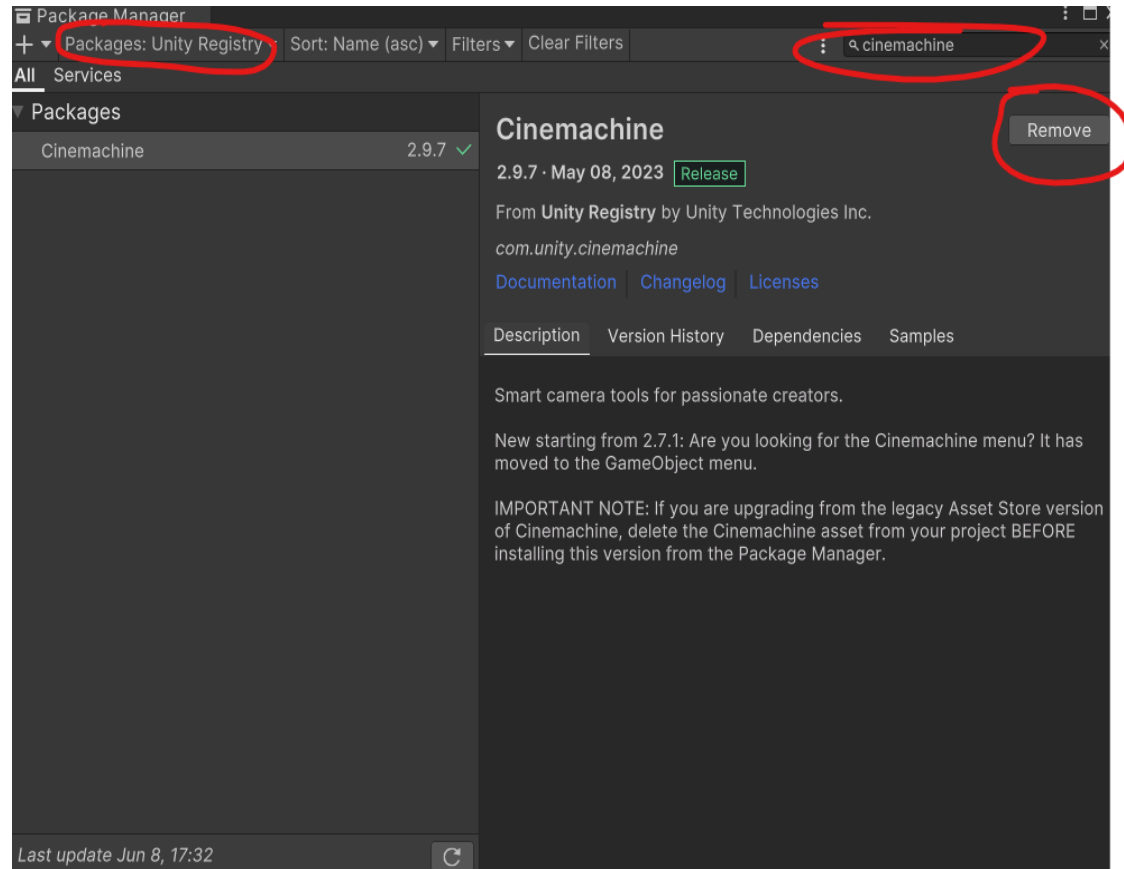
## Common Gameplay Elements

This section emphasizes common features when designing a game such as using a camera and interactions through detections.

### Camera

- What is the relationship between the virtual camera to the Main Camera object?

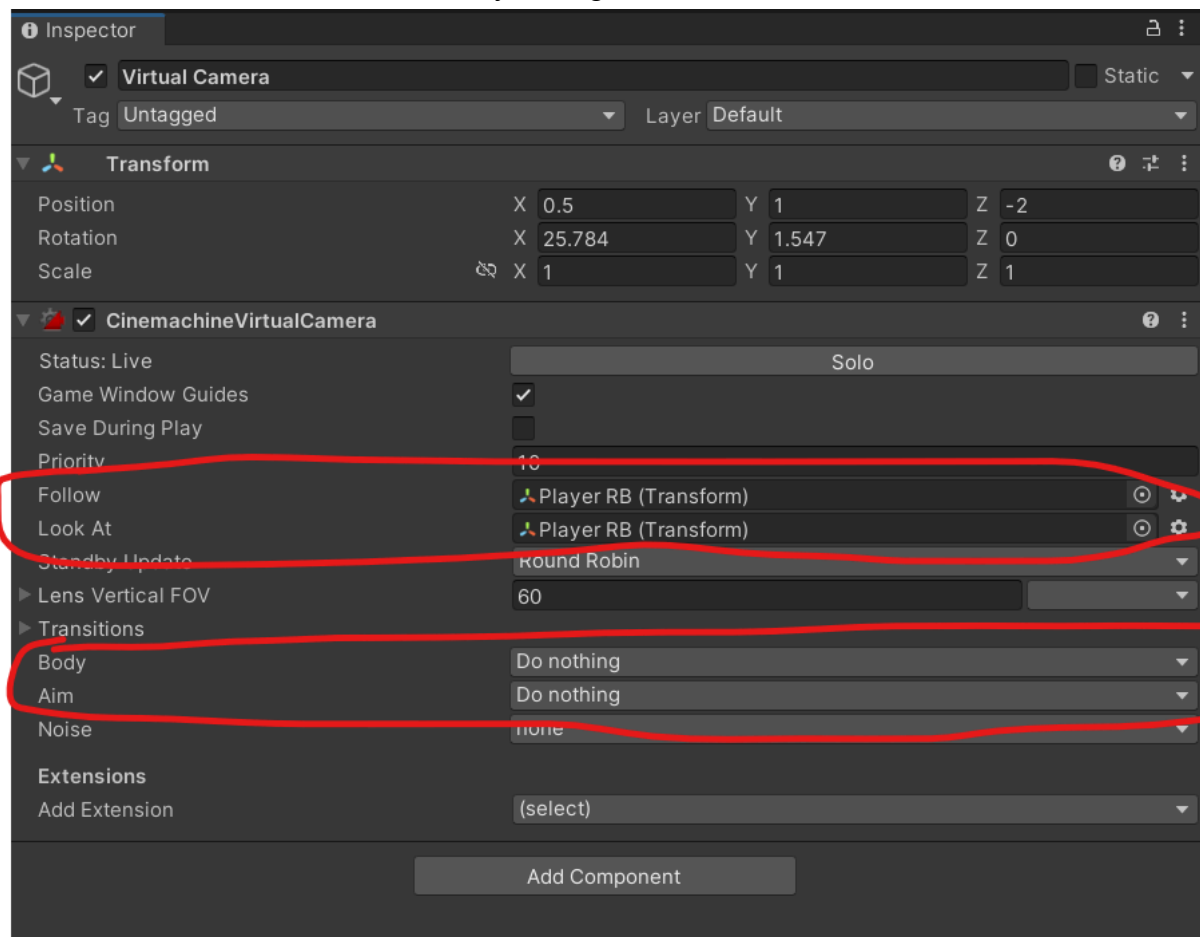
- The virtual camera is dependent on the Main Camera, so it cannot function without the Main Camera object.
- A virtual camera object is dependent on the Main Camera without being a child to that object similar to attaching a script to another script.
- Overall, a virtual camera attaches itself to the Main Camera and offers unique features and options in designing the game's camera.
- Why would I use a virtual camera?
  - A virtual camera offers different ways to view the game as opposed to the stationary Main Camera. This may not be as important if designing a game that involves infinite running like *Temple Run*
  - In theory, you can adjust the Main Camera with a script to do what you want. However, a virtual camera allows dynamic adjustments in regard to what it's trying to view.
  - A virtual camera offers “templates” when set to view specific objects. These templates define how the camera follows or adjusts its view of the player.
- I would like to access the cinemachine virtual camera, but I can't find it?
  - It usually is not included when you install Unity. However, it can be downloaded from the Package Manager
    1. From the top of the screen, click Window -> Package Manager
    2. Make sure the tab labeled “Packages:” with the drop down arrow has “Unity Registry” selected such that the tab is named “Packages: Unity Registry”
    3. In the search bar on the top right corner, search “cinemachine” and then select it.
    4. Click “Install” at the top right.
    5. Once it's downloaded, you can add it to the hierarchy like any other GameObject by right-clicking the hierarchy and seeing the “Cinemachine” option towards the bottom



**Figure:** The installation page for Cinemachiene after performing step 4. The areas circle in red involve setting “Packages” to “Unity Registry,” searching for “Cinemachine,” and the “Install” button being replaced with “Remove” after clicking on that button.

- How do I use a virtual camera?
  - After adding the virtual camera into the hierarchy, you can view the inspector window to see the transform properties and CinemachineVirtualCamera component
    - You can adjust the position and rotation of the camera in the Transform section similar to the Main camera. Essentially, the virtual camera becomes the new way to configure the camera while the Main Camera exists to maintain the virtual camera.
    - Under CinemachineVirtualCamera, you can set the object for the camera to follow under the “Follow” and “Look At” fields.
      - “Follow” defines what GameObject’s position for the virtual camera to pay attention
      - “Aim” defines what GameObject’s rotation for the virtual camera to pay attention to.
  - At first, nothing will change if an object is selected for either “Follow” or “Aim”. This is because the “Body” and “Aim” are set to “Do Nothing”

- Body defines how the virtual camera adjusts its position in relation to the GameObject's position
- Aim defines how the virtual camera adjusts its rotation in relation to the GameObject's rotation.
- “Body” and “Aim” have different options, or templates, to allow the virtual camera to dynamically view an object and any potential transform changes. When using any option besides “Do Nothing,” “Body” and “Aim” lock the virtual camera's position and rotation respectively as in those transform values cannot be manually changed.



**Figure:** The virtual camera component and where the “Follow,” “Look At,” “Body,” and “Aim” fields are.

- What are the different “Body” and “Aim” options?
  - Body
    - Do Nothing: The Virtual Camera will not adjust its position to follow the player. However, this lets the transform be manually adjusted.
    - 3rd Person Follow: The Virtual Camera will adjust to follow the player from behind. The view is initially set to be in the object as if it were 1st person. However, adjustments to the “Shoulder Offset”

coordinates can allow the camera's view to view the GameObject. This does lock the camera's rotation.

- Framing Transpose: The object will always be in the center of the camera's view; if the object moves, the camera will slightly lag to catch up so that the object becomes the center of the view again.
- Hard Lock To Target: Similar to the 3rd Person Follow, it follows the object in a first person perspective but this cannot be changed. This does not lock the rotation unlike 3rd Person Follow. This doesn't really interact with any "Aim" options besides POV
- Orbital Transposer: Allows the virtual camera to circle around the object and can initially be adjusted by moving the mouse around.
- Tracked Dolly: This involves creating a Cinemachine Path Base in the hierarchy, and this options allows the virtual camera to be confined in a specified path that can follow the GameObject if in range of the path, but will lose the GameObject if's far enough from it.
- Transposer: Relies on its "Binding Mode" field to decide how it follows the player. Essentially, it "drags itself" along with the player and how it does so is based on the "Binding Mode" option. For more info see here:

<https://docs.unity3d.com/Packages/com.unity.cinemachine@2.2/manual/CinemachineBodyTransposer.html>

○ Aim

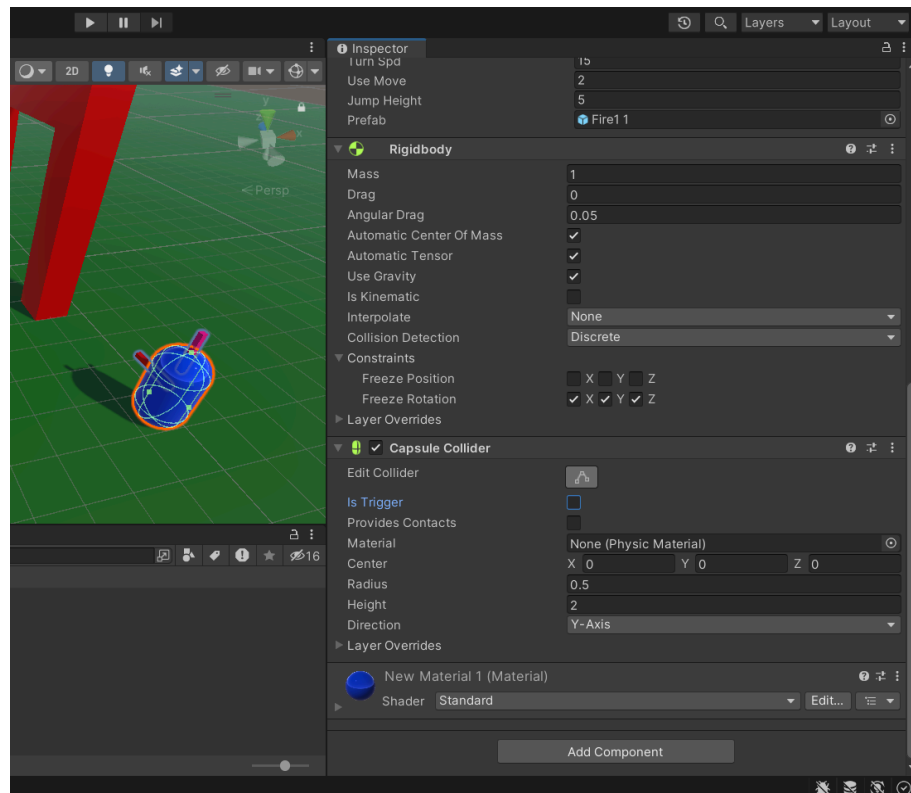
- Composer: Works similar to Framing Transpose where it makes sure the object is at the center of the view and the camera will adjust its rotation to allow that.
- Group Composer: The camera will rotate to fit multiple targets in its view - needs the target to contain a group of objects otherwise it will work just like a composer.
- Hard Look At: The camera will spin so that it is facing the target object
- POV: Allows the mouse to move the camera around as if in a first-person perspective even if the camera is stationary while the object is moving. On its own, it won't actually follow the player's rotation and it works best when paired with a "Body" option
- Same as Follow Target: The Camera's rotation will match the object's rotation.

○ References:

- <https://www.youtube.com/watch?v=asruvbmUyw8&t=520s>

## Collision and Trigger Detection

- What is the difference between collision and trigger detection?
  - Collision involves at least two GameObjects rigidbodies making physical contact with each other.
    - This highlights that each GameObject must have a rigidbody to be a collision.
    - Examples
      - Taking damage when an enemy makes physical contact with the player's model.
      - Walking into a wall non-stop without going through the wall.
  - Triggers are associated with colliders (Box, Sphere, and Capsule) under the “Is Trigger” and turns a GameObject into a sensor that waits for some sort of stimuli to contact the collider's range.
    - To use, add a Box, Sphere, or Capsule Collider component on a Gameobject. Click on the checkbox labeled as “Is Trigger”.
    - Enabling the “Is Trigger” allows other objects to phase right through it as if it were a ghost.



**Figure:** A capsule collider attached to a GameObject on the left. The “Is Trigger” can be selected under the component.

- Examples

- Getting a game over when near an enemy or winning the game by reaching the (invisible) finish line in *John Lemon's Haunted Jaunt*.
  - Triggering an alarm when being in certain areas in stealth games.
- Both have their own set of functions that allow scripts to determine what happens when a collision occurs or when a trigger is activated. These need to be defined in the code, and you can decide whatever happens in the function body.
  - Collisions
    - *OnCollisionEnter(Collision other)* - Occurs when another object makes the initial physical contact. Does not occur again until the collision stops and reoccurs.
    - *OnCollisionStay(Collision other)* - Occurs constantly until the collision stops.
    - *OnCollisionExit(Collision other)* - Occurs once an object withdraws from the collision.
  - Trigger
    - *OnTriggerEnter(Collider other)* - Occurs when an object is in the collider. Does not occur again until the object leaves the collider's range.
    - *OnTriggerStay(Collider other)* - Occurs constantly whenever an object is in the collider.
    - *OnTriggerExit(Collider other)* - Occurs once an object exits the collider's range.
- How can I define what causes a collision or a trigger? In a specific example, how can I cause an object on the ground to not produce a collision or trigger effect just by touching the surface?
  - In *Roll-a-Ball*, it uses tags to filter out what allows the pickups to be collected.
    - Tags provide a label or set that a GameObject can be recognized as.
    - In *Roll-a-Ball's* case, it has the player be its own tag and have the code specify that only objects with the "player" tag can have a trigger/collision.
    - In code, this would be a conditional statement like *"if (other.gameObject.CompareTag("<Tag Name>"))"* inside of a collision or trigger function.

```

void OnCollisionEnter(Collision other) {
    if (other.gameObject.CompareTag("Wall")) {
        Debug.Log("Wall: Collision Enter");
    }
    else if (other.gameObject.CompareTag("Player")) {
        Debug.Log("Player: Collision Enter");
    }
}

void OnTriggerEnter(Collider other) {
    if (other.gameObject.CompareTag("Wall")) {
        Debug.Log("Wall: Trigger Enter");
    }
    else if (other.gameObject.CompareTag("Player")) {
        Debug.Log("Player: Trigger Enter");
    }
}

```

**Figure:** Example of using OnCollisionEnter() and OnTriggerEnter() with tags.

- Alternatively, you can specify the name of the GameObjects from the Hierarchy window that are allowed to cause a collision
  - In code, this would be a conditional statement like “*if (other.gameObject.name == <Name>)*” inside of a collision or trigger function.
- There is a lot going on when I’m trying to produce collisions and trigger responses but not having any luck, so what is all going on?
  - In a setting with two GameObjects that each have their own coded collision/trigger response to the other.
    - Both GameObjects’ responses will occur regardless of who initiates it.
    - An object without a rigidbody but has a collider cannot cause a collision to occur. It can still have its *OnCollision()* function occur when another object with a rigidbody causes the collision.
      - In other words, an object without a rigidbody can react to a collision but not cause a collision.
  - For colliders,



- They are invisible borders around an object or however you customize the collider. These are physical and cannot be passed through unless you enable the “Is Trigger” option.
- These are useful for setting invisible boundaries when you don’t want the player to fall off the map.
- If you notice that an object is partially clipping or has sunk into a plane due to having a rigidbody, try putting a collider on it. This allows the object to be “physical” in the world.
- Even with a rigidbody, objects can still be passed through unless a collider is provided.
  - Rigidbodies act as a ticket to engage with Unity’s built in physics system such as *AddForce()* or built-in gravity.
- If an object lacks a collider, no collisions or trigger interactions can occur with that object. It also allows the object to be phased through too.
- Trigger responses are not dependent on rigidbodies but still need colliders.
  - Only one object needs to have the “Is Trigger” enabled. This allows all objects to respond with their *OnTrigger()* function.
- Collisions need both rigidbodies and colliders.
- In the relationship between collision and triggers
  - Triggers will be prioritized over collisions, so collisions may not occur if a trigger occurs instead.
  - However, it is possible to have both collisions and triggers in the same object
    - Ex) A gate can have a trigger when entering it and a collision by touching the support beams. This can be done by setting the collider to be the interior of the gate while having rigidbodies on the object(s).

## WebGL

This section briefly discusses a troubleshooting scenario with WebGL.

- Our shader is not showing up properly on our WebGL build, but it works when testing it in our Unity project?
  - Certain shaders may not work on specific operating systems or build means (ex. Windows, Mac, WebGL).
  - In this case, make sure to verify if there are any restrictions on your shader when you are looking for existing ones online.

