UNIVERSITY OF WOLVERHAMPTON
KNOWLEDGE ▪ INNOVATION ▪ ENTERPRISE

Unity Game Engine Activity

Student Workbook

Activity Title:

Programming Finite State Machines in the Unity Game Engine.

Activity Description:

In this activity, you will learn the basics of programming finite state machines in Unity. You will also learn further programming in the Unity games engine and be able to prototype decision-making elements.

Please Read:

- This student activity will check that you have understood and can apply the knowledge you have gained during the talks / presentation.
- Please read each activity task carefully and when required please type out all code. Attempting to copy and paste code *may* cause errors during compilation. Also writing out the code helps you understand how to program C# and Unity.

Document Version:

V4.0.

Game Engine Version:

This student workbook was checked using **Unity 2022.-.--f1**.

1. Start the Unity Hub and Create a New Project

Start the **Unity Hub** by double clicking on the “Unity Hub” shortcut in the Windows start menu. The Unity Hub icon looks like the screenshot below.

Image description: *The Unity hub Windows icon.*

In the Unity Hub, go to the Projects tab and click the (blue) New Project button

The Unity Hub window should look like the screenshot below. The blue New Project button is on the top right of the window.

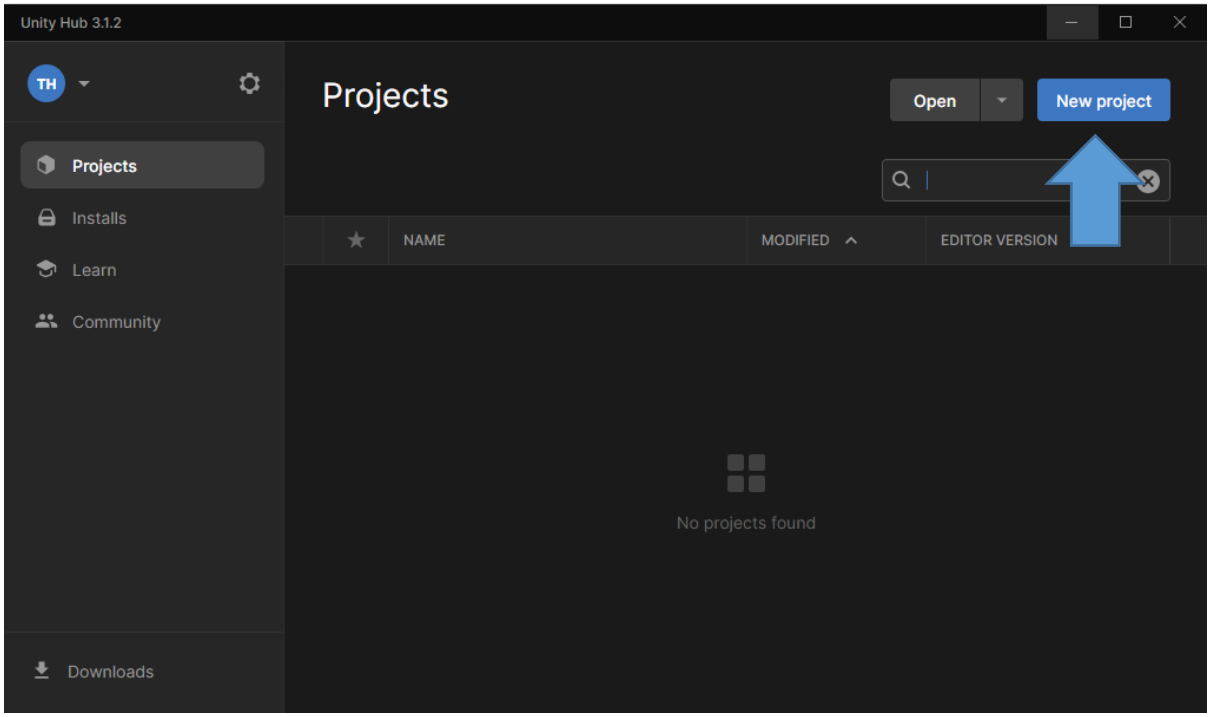


Image description: *The Unity hub with an arrow pointing to the New Project button.*

When you click the New Project button a “New project” screen will appear.

The window looks like the screenshot below.

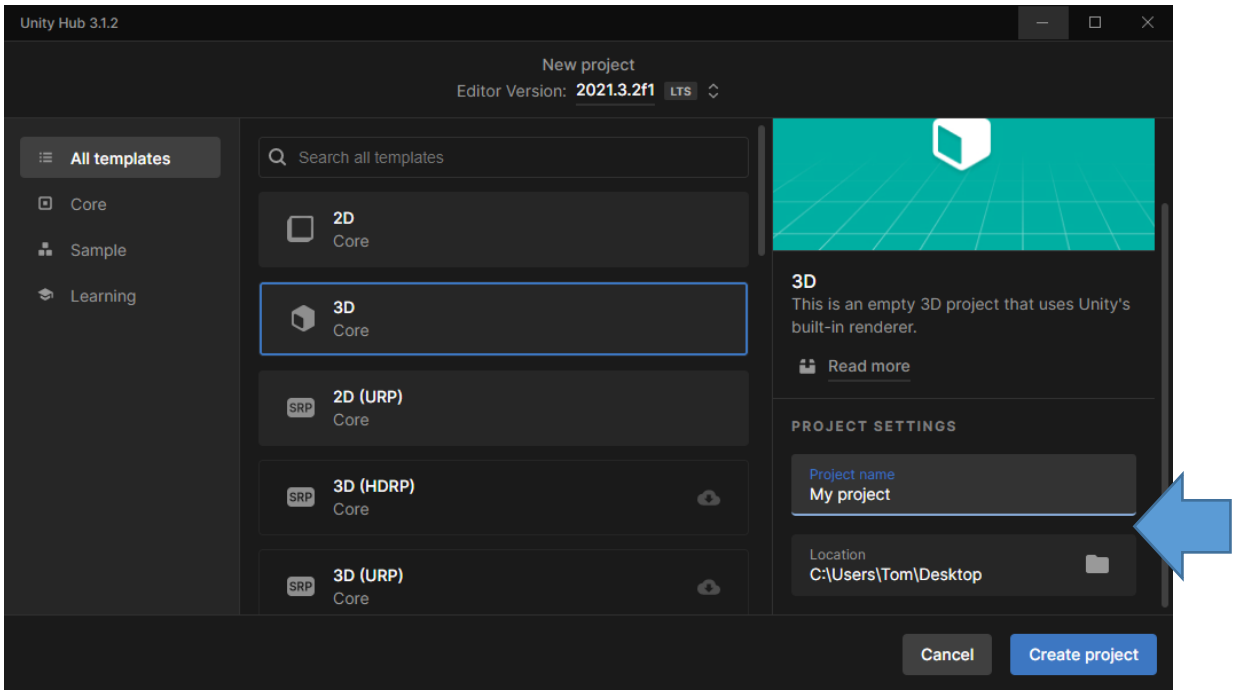


Image description: *The Unity hub “New project” screen.*

Set the following settings in the Unity hub “New project” screen.

Template: Select the 3D Core template. This is a basic empty 3D template.

Project Name: Give your project the name **FSMExample1**.

Location: Select a location to store your project. Below is some additional information about where to store your projects.

Useful Project Save Location Information:

- Any save location should be fine if you have enough disk space. Your will probably find project load and compile times are quicker if you save your projects to your computer’s hard drive.
- **Don’t forget to back-up your Unity projects.** I would recommend keeping at least one USB memory stick or USB hard drive for backup of all your university work. It would be better if you could maintain two backup devices.

Make sure you are creating a project in the correct version of the game engine. You can see the game engine version next to the “Editor Version:” text at the top of the Unity hub “New project” screen. **See the title page of this document for the version of the game engine we are using.**

We are now ready to create the project. <u>Click the “Create project” button.</u>

2. Importing Assets into your Project

We will now import some assets that we will use in this workbook.

Unity has an Asset Store that is home to thousands of free and priced assets. We will use **free** assets on the Unity Asset Store to help us create games or interactive 3D applications.

We are going to add the following free assets to our project:

- Starter Assets - First Person Character Controller

Before you can add the assets to your Unity project you need to first add them to your Unity account.

Step 1: Adding Assets to Your Unity Account

If needed, add the Unity assets to your Unity account. See the "Introduction to Unity" workbook for detailed guidance on how to add assets to your Unity account via the Unity asset store.

In a web browser, go to the web site: https://assetstore.unity.com/

Log into your Unity account. If you do not have a Unity account, you need to create one.

Search for the each of the assets listed above, select the asset and click the "Add to My Assets" button (it is a big blue button near the top right of the web page). Make sure you have logged into your Unity account at this point

When you have added the assets to your account, you can close the web browser.

Step 2: Importing Assets into Your Unity Project

In Unity you can import assets that have been added to your account. If you have not added the assets to your account, you need to add them before you proceed. See the previous step for more information.

In the Unity Editor, go to the Window menu and select Package Manager.

Select "My Assets" from the drop-down menu.

You may need to sign in with your Unity account to view your assets. If you are not signed-in, you should have the option to sign-in on the left panel of the Package Manager. Sign-in if needed.

A list of all your assets should appear.

Import all the assets asset pack(s) outlined above.

Follow these steps to import each asset pack:

- To import an asset pack, select it in the list.
- Next, you may need to click the **Download** button to download the asset to your computer.
- When the assets have been downloaded an Import button will appear. **Click the import button to import the asset.**
- **When you click the Import button a small "Import Unity Package" window will appear.**
- **Simply, click the Import button on the "Import Unity Package" window.**
- When you click the Import button the assets will be added to your project.

When you have imported all the assets, close the Package Manager window.

Step 2.1: Importing - Starter Assets - First Person Character Controller

Remember, the **First Person Character Controller** has some additional import steps.

Select the **Starter Assets - First Person Character Controller** from the list. If you have a lot of assets, you may need to click the **Load Next** button at the bottom of the assets list.

When you click the Import button, you will see this message.

Click the "Install/Upgrade" button. See the screenshot below for an example.

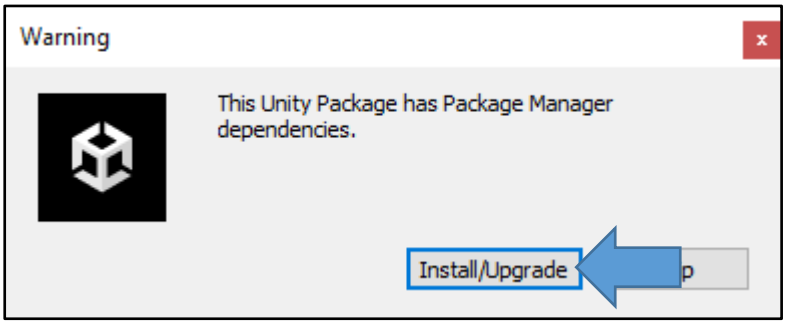


Image description: *When you click the Import button, you will see this message. Click the "Install/Upgrade" button.*

Wait for the package dependencies to be installed / upgraded.

Next, the message in the screenshot below will be displayed. Click No.

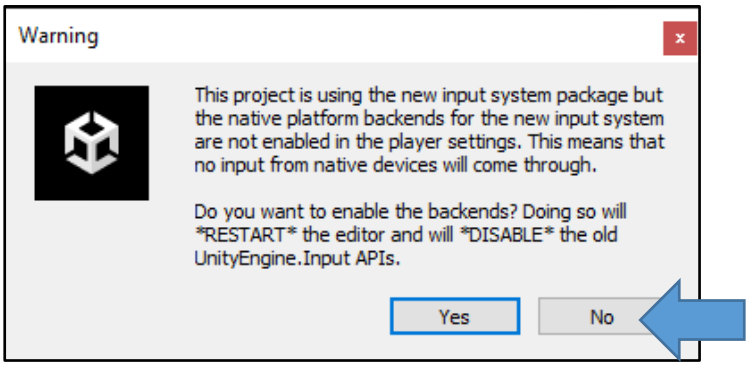


Image description: *A warning message stating that the assets being imported use the new input system package. Click no.*

Wait while more importing happens.

Then, a small "Import Unity Package" window will appear. See the screenshot below for an example.

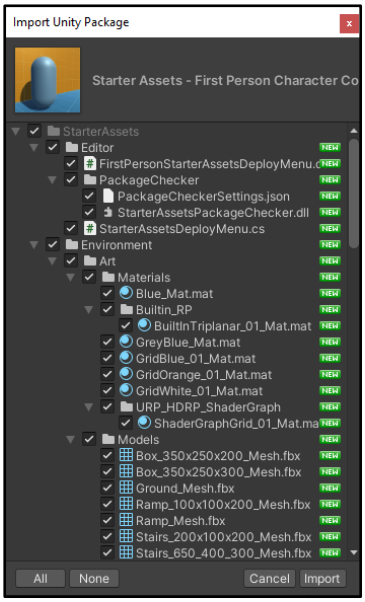


Image description: *The "Import Unity Package" window. Click Import.*

Simply, click the Import button on the "Import Unity Package" window.

When you click the Import button the assets will be added to your project. Wait while this happens.

When everything has finished importing you should close the Package Manager.

In the Unity Editor, go to the Edit drop down menu and select Project Settings. See the screenshot below for an example.

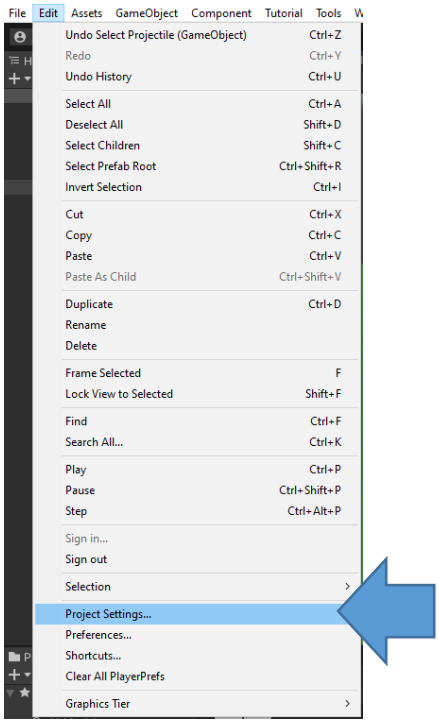


Image description: *Open the Project Settings window to set Unity project input type.*

Select Player from the list on the left and expand the “Other Settings” option. See the screenshot below for an example.

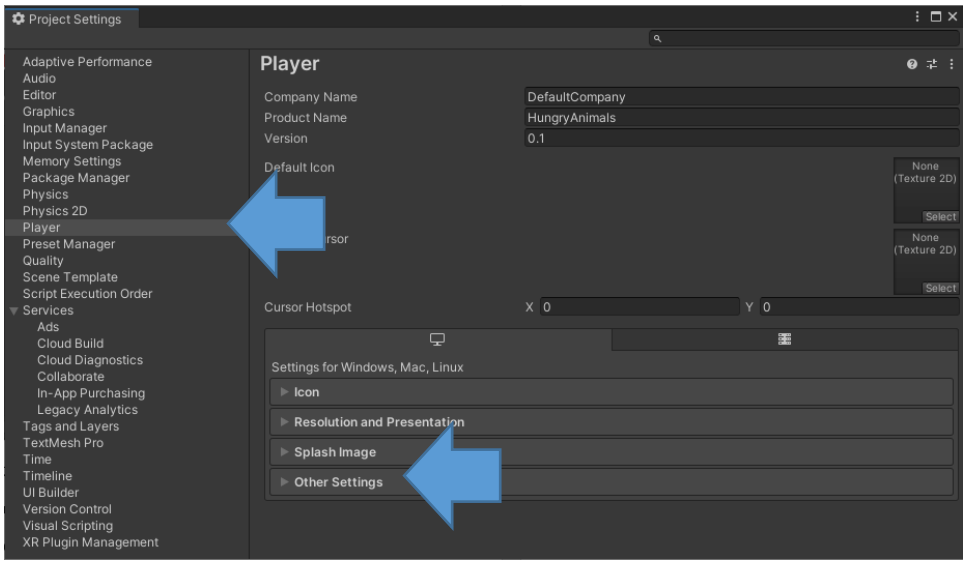


Image description: *Select Player from the list on the left and expand the “Other Settings” option.*

Scroll down “Other Settings” until you find the “Active Input Handling” option. Click the drop-down menu and select <u>Both</u>. See the screenshot below for an example.

Note: there are two input systems in Unity. An old one and a new one. In our workbooks we will use both because the old input system is very quick to get up and running.

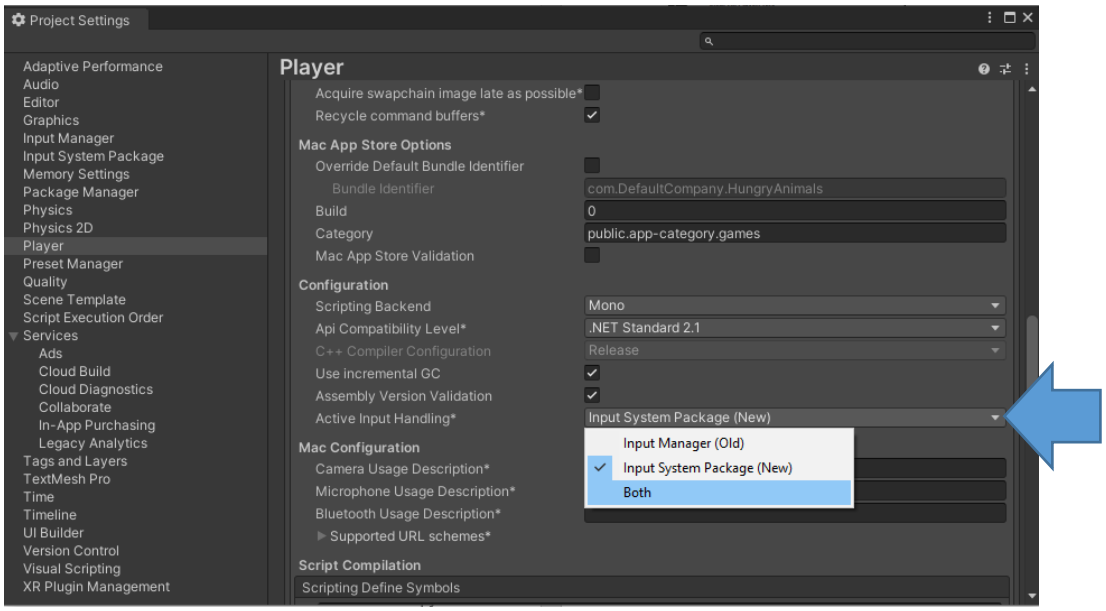


Image description: *Scroll down "Other Settings" until you find the "Active Input Handling" option. Click the drop-down menu and select Both.*

A "Unity editor restart required" message box will apear. Click Apply.

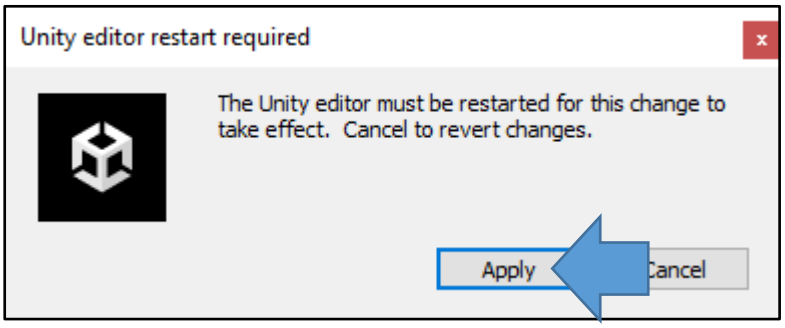


Image description: *A warning message stating that Unity must restart. Click Apply.*

Note, at this point Unity will restart. This is fine. If needed, click Save to save your scene. Unity will now restart.

When Unity restarts, close the "Project Settings" window (if needed).

When Unity has restarted we can continue to import asset packs (if needed).

3. Set External Script Editor and Regenerate Project files (*If needed*)

When our Unity projects include code, we might need to set the external script editor to Visual Studio and click the Regenerate Project files button.

This is important if you are opening a Unity project on a computer for the first time and it contains code. Regenerating project files ensures that the Visual Studio intellisense will work properly.

Therefore, it is a good idea to check the external script editor is set to Visual Studio and click the Regenerate Project files button every-time you load your Unity project on a new computer.

Tip: At home you probably do not need to do this if you are using the same computer all the time. Things should just work. You only need to do this if you are opening your Unity project on a new computer.

Follow the steps below to set Visual Studio as the external script editor and regenerate project files.

In the Unity Editor, click the **Edit** drop-down menu, and select **Preferences.** It is the 5th option from the bottom of the menu. See the screenshot below for an example.

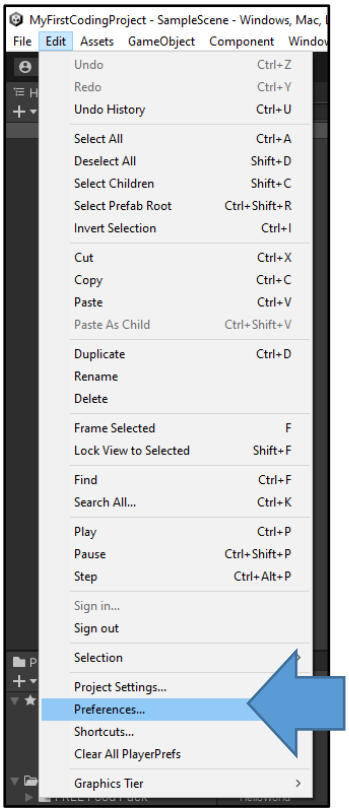


Image description: *Open the Preferences window to set Visual Studio as the external script editor and regenerate project files.*

A preferences window will load.

Select **External Tools** from the list on the left.

Then for the **External Script Editor** option and select the correct version of **Visual Studio**. If Visual Studio is already set, you do not need to do anything.

Then click the **Regenerate Project files** button.

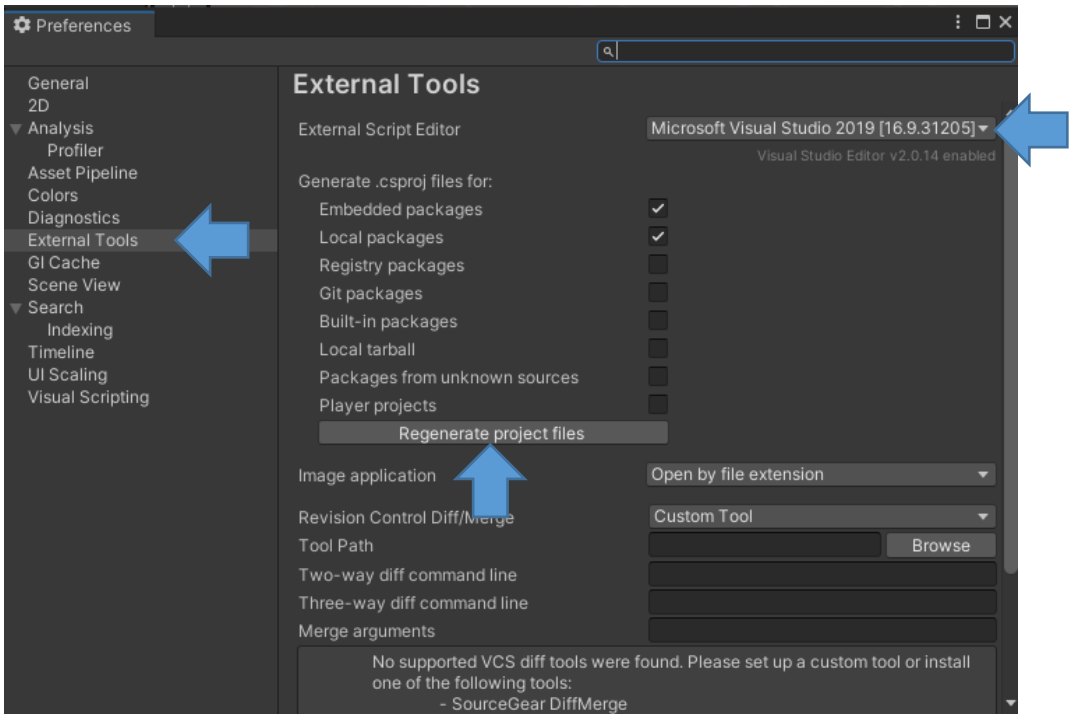


Image description: *The Preferences window. If needed, set the external script editor to Visual Studio. Then, click the regenerate project files button.*

You can now close the preferences window.

4. Creating a Simple Scene with Primitive Shapes

In this section we will create a simple scene. This illustrates how you can create a simple scene in Unity using primitive shapes. This is an alternative to terrains or could be used in conjunction with terrains.

The first thing we will do is create a simple ground or floor to walk on.

Go to the Hierarchy Window and click the plus (+) menu. Select 3D Object -> Cube. See the screenshot below for an example.

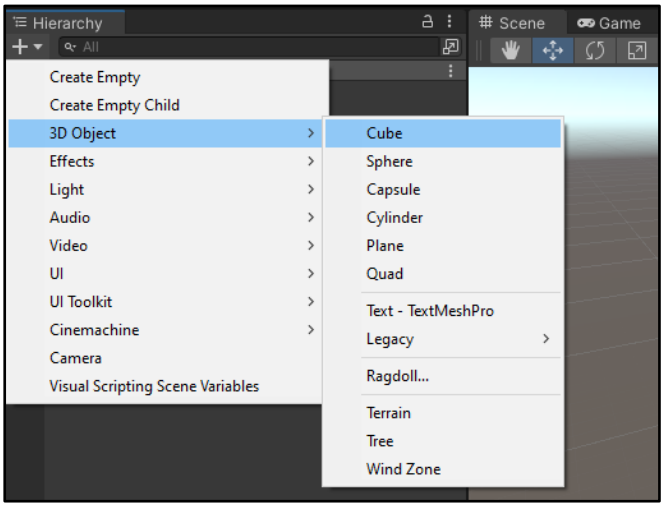


Image description: *The Hierarchy Window and the plus (+) menu. Cube has been selected in the menu.*

A cube should be created in your scene.

At this point the cube should be selected in the hierarchy window and you should be able to enter a name for the cube. Give the cube the name **Ground**.

- **Tip:** If the cube is not selected in the hierarchy window. For example, you might have clicked on something else. Go to the Hierarchy Window and slowly single click twice on the Cube text to rename it. Or you can right click and select Rename.

Select the ground cube in the hierarchy window.

Go to the inspector. Scale the cube so that it looks more like a floor. I used the following settings:

X: 100
Y: 1
Z: 100

This creates a relatively small environment. You can use larger values for the x and z axes to create a bigger environment.

Your cube should now look like the screenshot below.

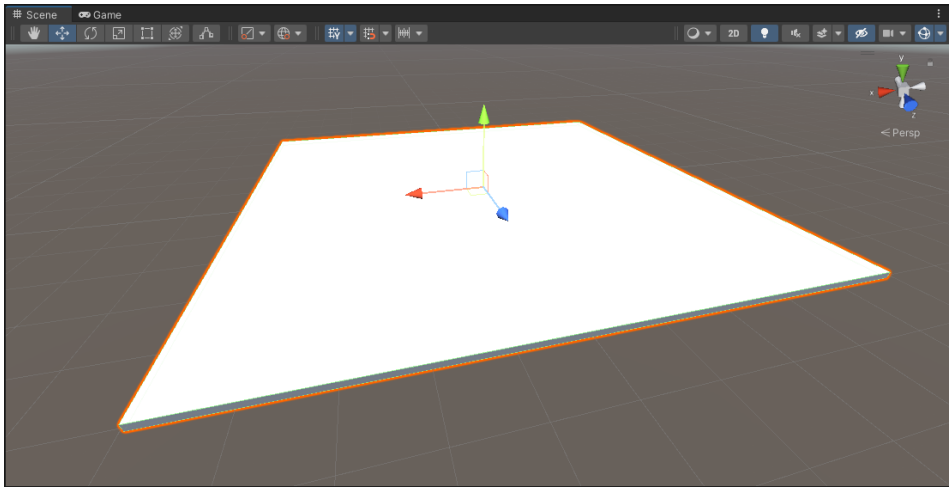


Image description: *The ground cube in the scene view.*

Next, we will give the ground some colour using a material.

Go to the Project Window. Select the Assets folder. Right click in the righthand panel of the Assets folder and select Create -> Folder.

When the folder is created give it the name **Materials**.

- **Tip:** It is a good idea to organise your assets in folders. You can make the folder structure simple or complicated. You should do what works for your project.

Double click on the Materials folder to open it.

Right click in the Materials folder and go to Create -> Material.

Give the Material the name **GrassColour**.

Select the material in the Project window. You should be able to see its properties in the inspector.

Go to the inspector, click on the block next to the Albedo property at top of the Inspector.

- **Information:** The Albedo property, pronounced, al-bee-dow, is the colour or texture map for the object.

See the screenshot below for an example.

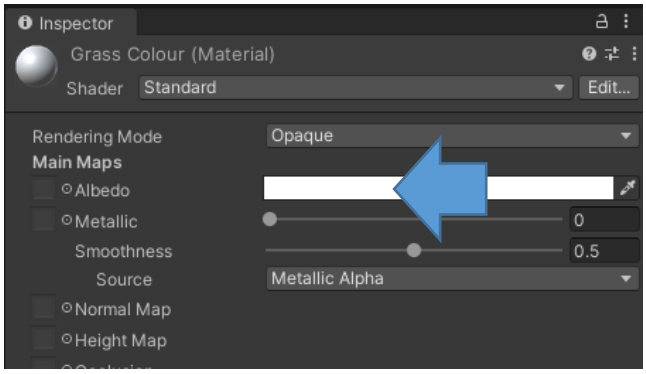


Image description: *The inspector window for the material. Click on the white box next to the Albedo property.*

Clicking on the block next to the Albedo property at top of the Inspector will open a colour picker window.

Select a green colour. I entered the hexadecimal value: 228C22.

See the screenshot below for an example.

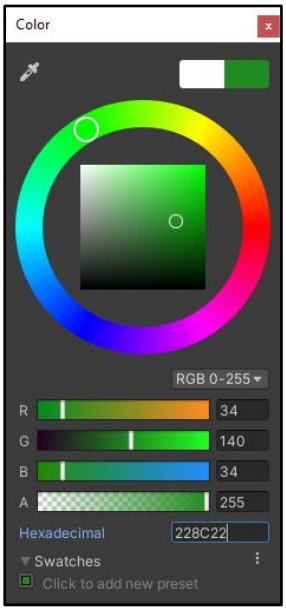


Image description: *The material Albedo property colour picker.*
Close the colour picker window.

Next, we need to assign the material to the ground cube in the scene. To do this, drag the **GrassColour** material from the Project window and drop it onto the name of the ground object in the Hierarchy OR drag it onto the ground in the scene view.

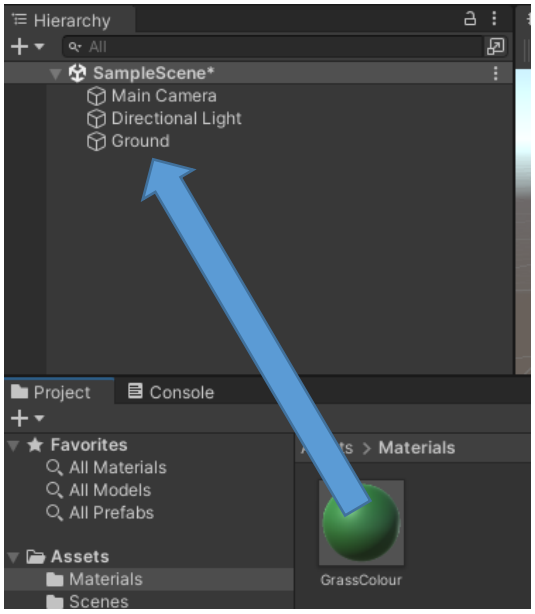


Image description: *Drag the material from the Project window to the ground GameObject.*

Your updated ground should now look like it has a green grass colour applied to it. See the screenshot below for an example.

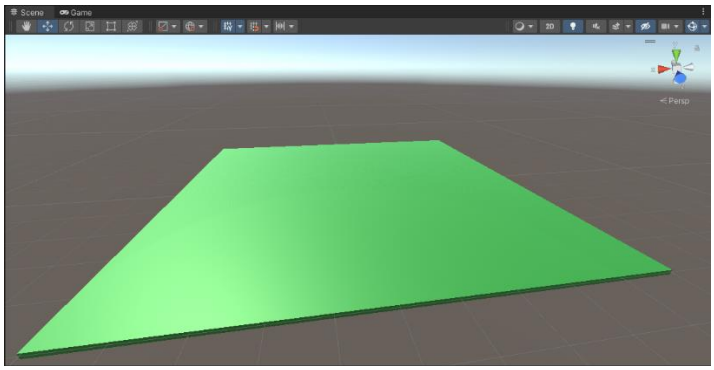


Image description: *The ground cube in the scene view. A green material has been applied to it.*

Save your scene. Go to the File menu and find the save option.

5. Adding a First-Person Camera to the Environment

We will now add a first-person camera to our scene. This camera will allow a player or user to move around our scene.

By default, Unity includes a camera in our scene. Go to the hierarchy, find the Main camera, right click on it, and select delete from the menu. See the screenshot below for an example.

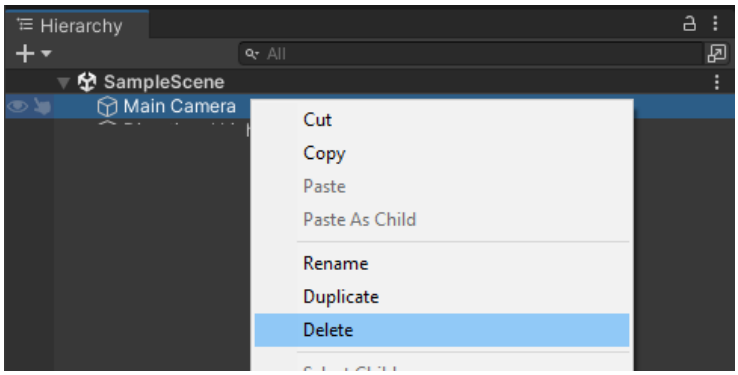


Image description: *Example of how to delete the "Main Camera" from the scene.*

Once you have deleted the main camera from the scene you can add a first-person camera.

Go to the Project window.

Navigate to the folder: StarterAssets -> FirstPersonController -> Prefabs.

You should find that some of the assets in this folder are coloured magenta (pink). This is because they are setup to use the universal render pipeline, instead of the built-in render pipeline, which this project is using.

First, we need to change the **PlayerCapsule prefab** so that is used the built-in render pipeline.

Double click on the **PlayerCapsule prefab**. This will open the prefab editor.
See the screenshot below for an example.

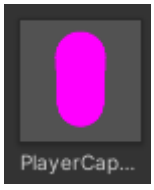


Image description: *some of the assets in the StarterAssets -> FirstPersonController -> Prefabs folder are coloured magenta (pink). This is because they are setup to use the universal render pipeline, instead of the built-in render pipeline, which this project is using. Double click on the PlayerCapsule prefab.* This will open the prefab editor.

The prefab editor should open. See the screenshot below for an example.

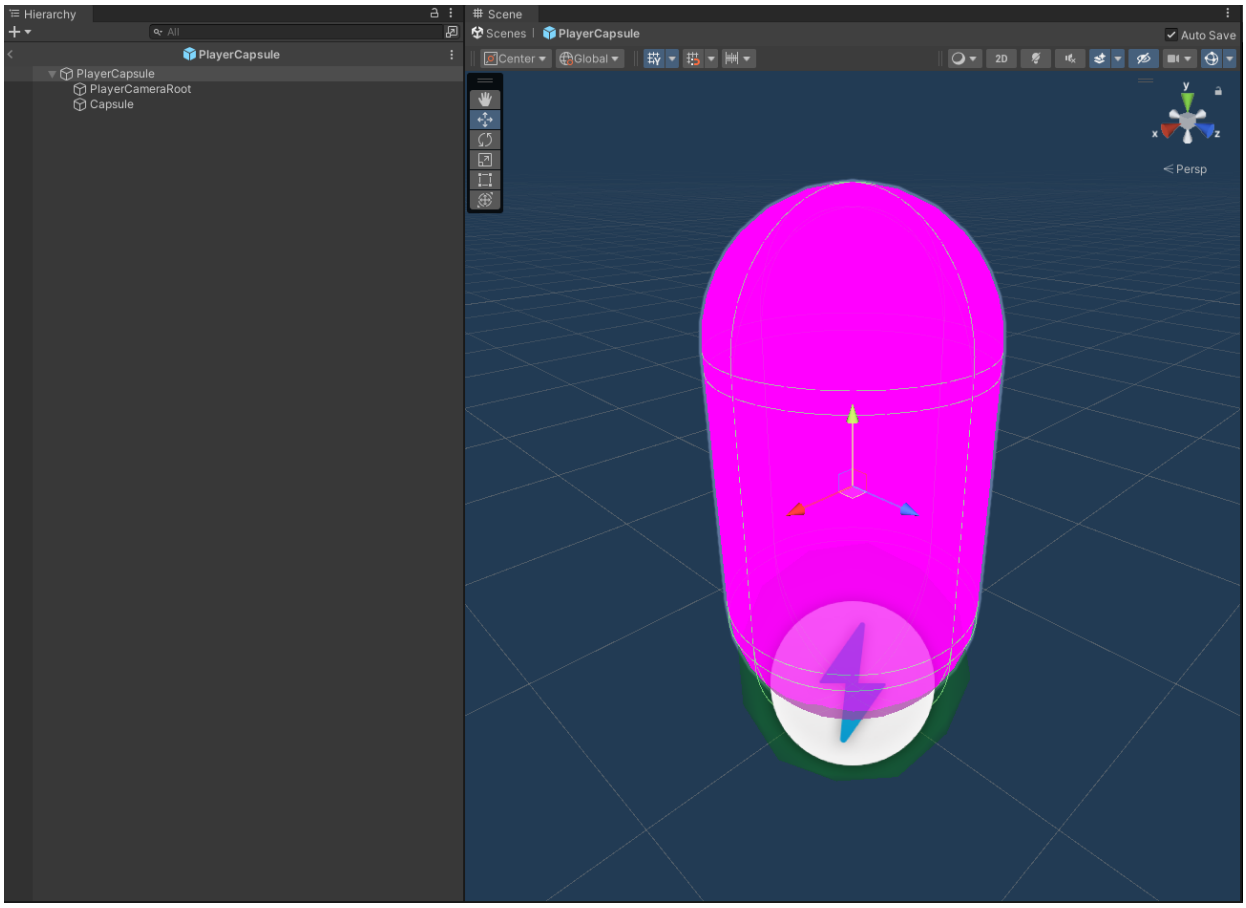


Image description: *The prefab editor with the PlayerCapsule prefab loaded.*

Click the Capsule gameObject in the Hierarchy.

Go to the Inspector and find the material for the game object. See the screenshot below for an example.

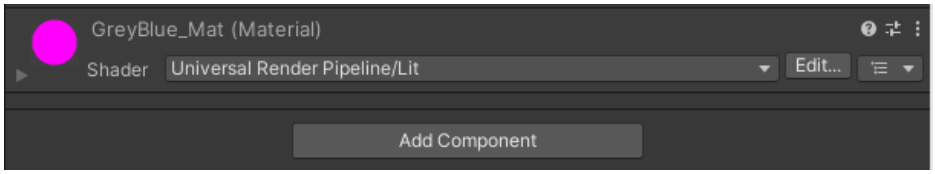


Image description: *Go to the Inspector and find the material for the game object.*

Click the drop-down menu next to the word "Shader". Select "Standard" from the drop-down menu.

The capsule should now **not** be coloured magenta (pink).

Click the back arrow near the top of the Hierarchy to move back to your scene. See the screenshot below for an example.

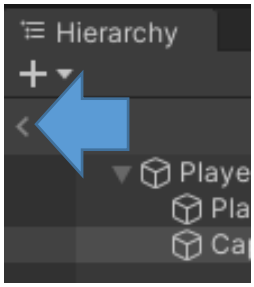


Image description: *Click the back arrow near the top of the Hierarchy to move back to your scene.*

Save your scene.

Drag the PlayerCapsule prefab into the scene and position it somewhere on the ground.

Drag the PlayerFollowCamera prefab into the scene. This prefab can be positioned anywhere; however, I would recommend you position it near the PlayerCapsule prefab.

Drag the MainCamera prefab into the scene. This prefab should automatically position itself at the PlayerFollowCamera prefab's position.

Next, go to the hierarchy, and select the PlayerFollowCamera prefab.

Go to the inspector and find the **CinemachineVirtualCamera** component.

Find the **Follow** property and click the bullet icon next to "None (Transform)". See the screenshot below for an example.

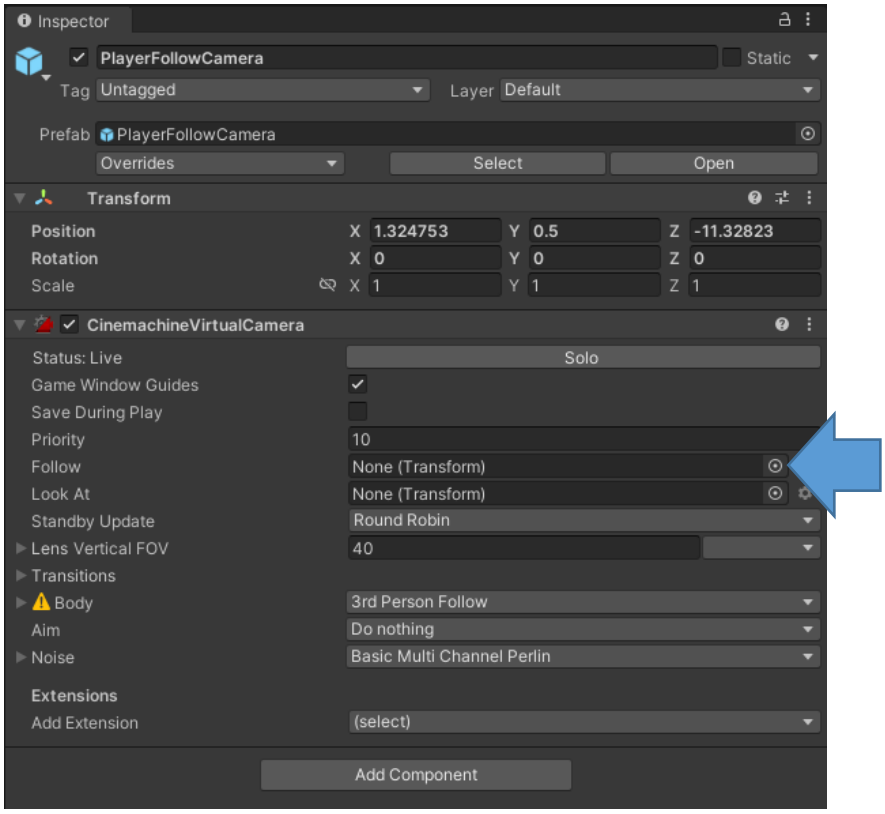


Image description: *Go to the hierarchy and select the PlayerFollowCamera prefab. Go to the inspector and find the CinemachineVirtualCamera component. Find the follow property and click the bullet icon next to "None (Transform)".*

A "Select Transform" window will appear.

Double click on **PlayerCameraRoot** (in the Scene tab) to select it.

The follow property of the CinemachineVirtualCamera component should now have the value "PlayerCameraRoot (Transform)".

The first-person camera should now be setup.

Save the scene.

🎮 **We are now ready to playtest the scene.**

Press the play button to playtest your scene. See the screenshot below for an example.

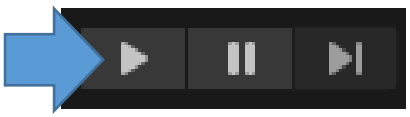


Image description: *The game / scene play controls on the main toolbar. Click the play button.*

Play Mode is a realistic test of your game.

- Note, when in Play mode you can adjust GameObjects via the Scene window. However, all adjustments made to GameObjects will be temporary and undone when play mode is stopped.

If the game view is behind the scenes view, click the Game view tab to select the Game view window.

You should be able to walk around the environment.

The default controls:

- Arrow keys or WASD to move. Spacebar to jump. Hold down shirt to sprint.
- Mouse look.

To stop playtesting the game, click the play button again.

See the screenshot below for an example. Note, in my setup I have the Scene and Game view side-by-side.

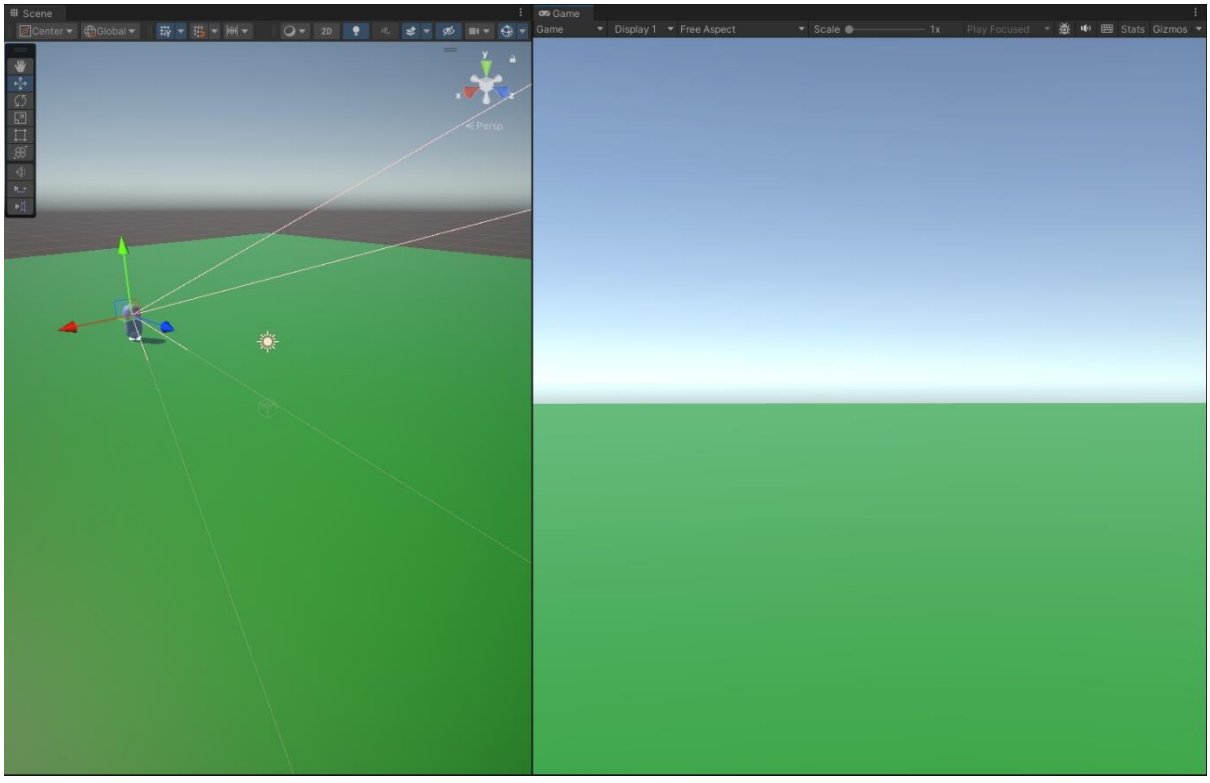


Image description: *A playtest of the scene using the first-person controller. Note, in my setup I have the Scene and Game view side-by-side.*

6. Finite State Machines (PLEASE READ NOW)

This section provides a brief introduction to Finite State Machines (FSM). Please read this section to develop your understanding of the subject.

Much AI is implemented with the equivalent of if….else rules. For example, consider the code below to handle player input [1]:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    yVelocity_ = JUMP_VELOCITY;
    setGraphics(IMAGE_JUMP);
  }
}
```

This type of code starts simple, but we will soon need to add complexity to handle more events within the environment. This complexity will make it difficult to manage and track down bugs. For example, you could end up with code like this [1]:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_ && !isDucking_)
    {
      // Jump...
    }
  }
  else if (input == PRESS_DOWN)
  {
    if (!isJumping_)
    {
      isDucking_ = true;
      setGraphics(IMAGE_DUCK);
    }
  }
  else if (input == RELEASE_DOWN)
  {
    if (isDucking_)
    {
      isDucking_ = false;
      setGraphics(IMAGE_STAND);
    }
  }
}
```

It is easy for bugs to get into the type of code above (e.g. lots of nested if.. else statements). It is very challenging to extend the type of code above.

A Finite State Machine is an abstract concept that can be used to structure decision-making in your code and avoid these types of issues.

A FSM can be implemented in various ways, e.g. by using if….else, switch statements, or objects. **In this workbook we will explore a simple if….else example. First, let explore the basics of FSMs.**

FSMs consist of:

- **States** determine an agent's current behaviour or "state of mind" [2] / mode and usually result in actions being generated.
    - A collection of actions that are used when in a particular mode [4].
- **State transition** define rules on how to move from one state to another.
- **Events / Inputs** are either externally or internally generated, may trigger rules and lead to state transitions [3].
- A start state.

The best way to visualise a FSM is as a directed graph (digraph).

- Nodes represent states.
- Vertices.
- The State transitions correspond to directed connecting edges.

An FSM will look at the state that it is currently in and any current events to decide if it needs to change to another state.

For example, in a patrolling and attacking guard agent, if the agent is in a PATROL state then there might be three relevant events that can occur:

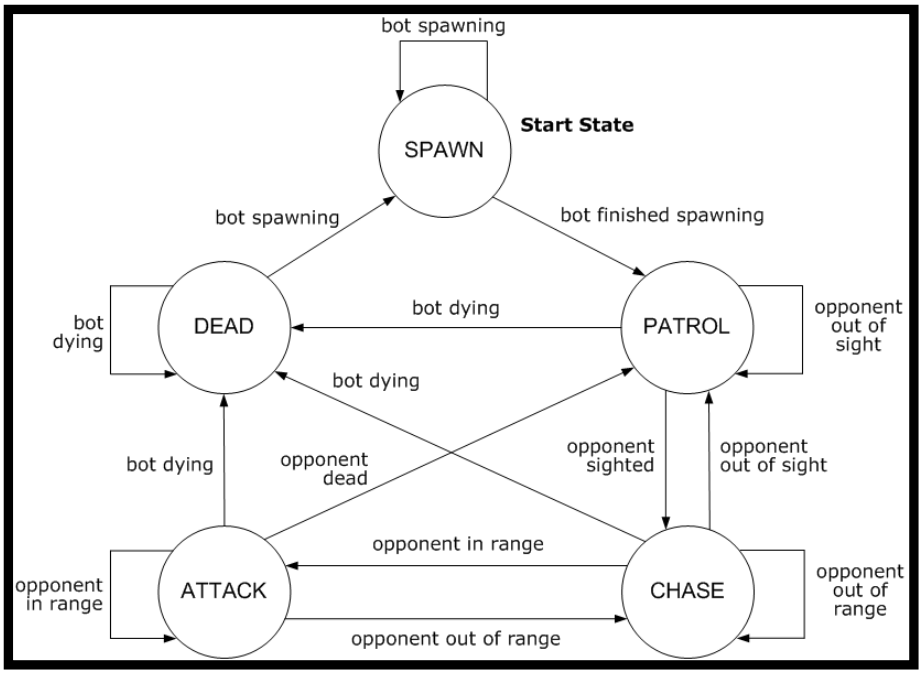
- opponent sighted.
- opponent out of sight.
- Agent dying.

If an opponent is sighted, the agent will move to a CHASE state.

If the agent is shot and dies, it will move to a DEAD state.

If no opponent is sighted, the agent will stay in the patrol state.

Example of a FSM:



Rabin[5, in 2] suggests a state machine language of sorts that implements three events for each state: OnEnter, OnExit, and OnUpdate. Therefore, each state should comprise three events. The OnEnter and OnExit events run once when the state starts and exits. The OnUpdate event is run every game loop. The OnEnter event sets properties for the state and/or game. The OnUpdate event performs actions typically found in an gameobject update function. The OnExit function clears or resets properties for the standard/or the game.

We have covered the basics of FSMs. Follow these links to explore them further:

- https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

- https://en.wikipedia.org/wiki/Finite-state_machine

Section References

[1] – Game Programming Patterns. Link - http://gameprogrammingpatterns.com/contents.html.

[2]. Byl, Penny B (2004) Programming Believable Characters for Computer Games. 1st ed., Game development Series.

[3]. Jason Brownlee (2004) Finite State Machines (FSM) <http://ai-depot.com/FiniteStateMachines/>.

[4]. John Laird – http://www.eecs.umich.edu/~soar/Classes/494/talks/AI.ppt

[5]. Rabin (2002) Game AI Wisdom.

7. Implementing a Simple FSM in Unity

This section provides an overview of how to implement a simple FSM that can be used to control the decision making of a virtual character / agent. This example implements the FSM using a single class, switch statements and functions.

We will now create a C# script to control the behaviour of a virtual agent.

Create a C# script. Go to the Project Window. Select the Assets folder. Right click in the righthand panel of the Assets folder and select Create -> Folder. When the folder is created give it the name **Scripts**.

Double click on the Scripts folder to open it.

Right click in the Scripts folder and go to Create -> C# Scripts.

When the script has been created give it the name **FSM_CON**.

Update the code in the script file to match the code below.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FSM_CON : MonoBehaviour
{
    // Enum for game states.
    private enum GameStates { IDLE, CHASEPLAYER, RETREAT };
    private GameStates State = GameStates.IDLE;

    // Enum for state events.
    private enum GameEvents { ON_ENTER, ON_UPDATE };
    private GameEvents Event = GameEvents.ON_ENTER;

    // Start is called before the first frame update
    void Start()
    {

    }

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

        // Update the FSM behaviour.
        FSMUpdate();

    }

    // FSM update method.
    private void FSMUpdate()
    {
        // --- Idle. ---
        if (State == GameStates.IDLE)
        {
            if (Event == GameEvents.ON_ENTER)
            {
                idle_Enter();
            }
            if (Event == GameEvents.ON_UPDATE)
            {
                idle_Update();
            }
        }

        // --- Move. ---
        if (State == GameStates.CHASEPLAYER)
        {
            if (Event == GameEvents.ON_ENTER)
            {
                chasePlayer_Enter();
            }
            if (Event == GameEvents.ON_UPDATE)
            {
                chasePlayer_Update();
            }
        }
```

```
        // --- Retreat. ---
        if (State == GameStates.RETREAT)
        {
            if (Event == GameEvents.ON_ENTER)
            {
                retreat_Enter();
            }
            if (Event == GameEvents.ON_UPDATE)
            {
                retreat_Update();
            }
        }

        // Process input / general events. (If needed).
        FSMProcessInput();

        //:~END: --- Process inputs ---
    }

    void OnCollisionEnter(Collision col)
    {
        Debug.Log("OnCollisionEnter - " + col.gameObject.name);

    }

    // Process input / general events.
    private void FSMProcessInput()
    {
        // Process input here / general events if needed. For example, user input.
        // Could cause a state change. Must call state change method.
        // I would only change states in state code.

    }

    // Change state method. The only places a state should change. Call this method when
you want to change state.
    private void ChangeFSMState(GameStates newState)
    {
        // Finish / exit current state.
        switch (State)
        {
            case GameStates.IDLE:
                idle_Exit();
                break;

            case GameStates.CHASEPLAYER:
                chasePlayer_Exit();
                break;

            case GameStates.RETREAT:
                retreat_Exit();
                break;
        }

        // Move to the next state.
        State = newState;
        Event = GameEvents.ON_ENTER;
    }
```

```
// --- FSM States ---


// ****************************** -Idle- ******************************

// Idle - enter.
private void idle_Enter()
{

    // Change to update at the end of enter.
    Event = GameEvents.ON_UPDATE;
}

// Idle - update.
private void idle_Update()
{
    // Implement game state here.
    // Could cause a state change. Must call state change method.

}


// Idle - Exit.
private void idle_Exit()
{

}

// ****************************** -Chase Player- ******************************

// Chase Player - Enter.
private void chasePlayer_Enter()
{

    // Change to update at the end of enter.
    Event = GameEvents.ON_UPDATE;
}

// Chase Player - Update.
private void chasePlayer_Update()
{
    // Implement game state here.
    // Could cause a state change. Must call state change method.

}

// Chase Player - Exit.
private void chasePlayer_Exit()
{

}
```

```
    // ****************************** -Retreat- ******************************

    // Retreat - Enter.
    private void retreat_Enter()
    {

        // Change to update at the end of enter.
        Event = GameEvents.ON_UPDATE;
    }

    // Retreat - Update.
    private void retreat_Update()
    {
        // Implement game state here.
        // Could cause a state change. Must call state change method.

    }

    // Retreat - Exit.
    private void retreat_Exit()
    {

    }

}
```

The code above has created a structure for the FSM and an FSM that contains three states: Idle, Chase Player and Retreat. The **FSMUpdate** function determines the current state and event using if statements.

The current state is determined using enums. There is also an event enum that determines the current event for the state.

When the current state is determined the function that implements that state is called. Each state has three functions, an **enter** function, which is called once when the state starts. An **update** function that is called every frame of the game. An **exit** function that is called once when a state is finished.

There is also a **ChangeFSMState** function that takes the new state as a parameter. The function calls the exit function for the current state and then changes the FSM state to the new state. The function also sets event to **ON_ENTER**.

The FSM also includes a function called **FSMProcessInput**. This function should process general game events that might be applicable in many states. For example, user inputs.

The FSM currently does not do anything. You can think of the FSM above as a template that can be updated to suit your needs. However, lets add some code to the FSM so that it implements simple behaviour.

We will now update the FSM to implement some simple behaviour.

Add the following class level variables to the **FSM_CON** class.

```
    // Move variables.
    // An array of gameobjects (gos).
    private GameObject[] gos;
    // Stores a reference to the player.
    private GameObject Player = null;

    private float MoveSpeed = 3;

    // Projectile hit event variables.
    private bool HitByProjectileEvent = false;
```

Add the following code to the **Start** function.

```
    void Start()
    {
        // Find the player.
        // There should be one gameobejct tagged with player.
        gos = GameObject.FindGameObjectsWithTag("Player");
        if (gos.Length > 0)
        {
            Player = gos[0];
        }
        if (Player == null)
        {
            Debug.Log("No Player found in the scene!!!");
        }
    }
```

Add the following code to the **OnCollisionEnter** function.

```
    void OnCollisionEnter(Collision col)
    {
        Debug.Log("OnCollisionEnter - " + col.gameObject.name);

        if (col.gameObject.tag.Contains("Projectile"))
        {
            Destroy(col.gameObject);
            HitByProjectileEvent = true;
        }
    }
```

The function above is called on a collision. We will setup the collision properties later. The function sets the HitByProjectileEvent to true and deletes the projectile.

Add the following code to the **idle_Update** function.

```
    // Idle - update.
    private void idle_Update()
    {
        // Implement game state here.
        // Could cause a state change. Must call state change method.

        // Spin for idle.
        transform.Rotate(0, 1, 0, Space.World);

        // FSM transition rule.
        if(Vector3.Distance(transform.position, Player.transform.position) < 10)
        {
            // Change the state.
            ChangeFSMState(GameStates.CHASEPLAYER);
        }
        // FSM transition rule.
        if (HitByProjectileEvent == true)
        {
            HitByProjectileEvent = false;
            // Change the state.
            ChangeFSMState(GameStates.RETREAT);
        }

    }
```

The function above adds some simple idle behaviour; namely, spinning the object. The state also implements two transition rules that initiate a change of state based on rules. There is one transition that moves the FSM from Idle to Chase Player when the player is close. There is another transition that moves the FSM from Idle to Retreat when the object the FSM is attached to is hit.

Add the following code to the **chasePlayer_Update** function.

```
    // Move - Update.
    private void chasePlayer_Update()
    {
        // Implement game state here.
        // Could cause a state change. Must call state change method.

        // Move to a target.
        transform.position = Vector3.MoveTowards(transform.position,
Player.transform.position, (MoveSpeed * Time.deltaTime));

        // FSM transition rule.
        if (HitByProjectileEvent == true)
        {
            HitByProjectileEvent = false;
            // Change the state.
            ChangeFSMState(GameStates.RETREAT);
        }

    }
```

The function above adds code to chase the player. The state also implements a transition rule that moves the FSM from Chase Player to Retreat when the object the FSM is attached to is hit.

Add the following code to the **retreat_Update** function.

```
    // Retreat - Update.
    private void retreat_Update()
    {
        // Implement game state here.
        // Could cause a state change. Must call state change method.

        // Move away from the target.
        Vector3 Direction = Player.transform.position - transform.position;
        // Clear y to avoid going up and down.
        Direction.y = 0;
        Vector3 Position = transform.position - Direction;
        transform.position = Vector3.MoveTowards(transform.position, Position, (MoveSpeed
* Time.deltaTime));

        // FSM transition rule.
        if (Vector3.Distance(transform.position, Player.transform.position) > 20)
        {
            // Change the state.
            ChangeFSMState(GameStates.IDLE);
        }

    }
```

The function above adds code to retreat from the player. The state also implements a transition rule that moves the FSM from Retreat to Idle when the object the FSM is attached to away from the player.

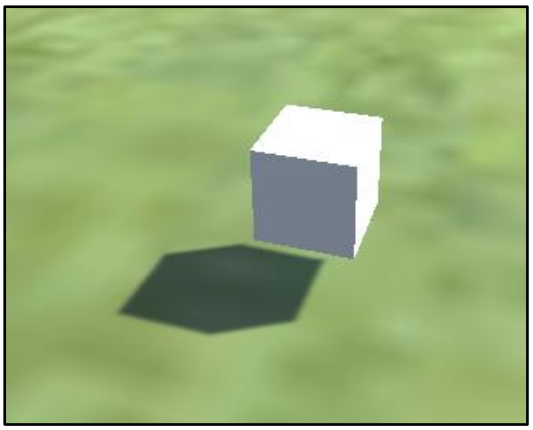
Save the script and go back to Unity.

Next, we need to attach the script to a gameobject. We will use a simple cube in this simple example.

Go to the Hierarchy window in the Unity editor.

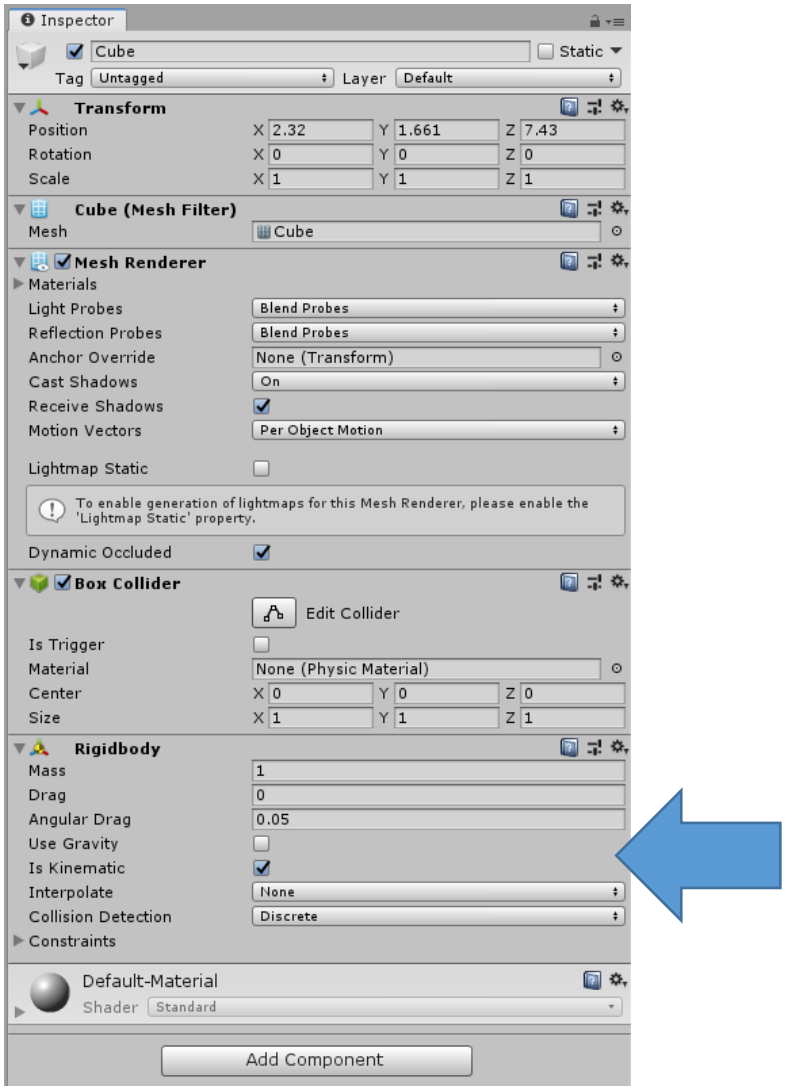
Click the Create button at the top of the Hierarchy window and select Cube in the 3D Object submenu.

Select the Cube object in the Hierarchy, move the Cube so that it is just above the terrain / floor you have created. You cube should look like the screenshot below.



Select the Cube object in the Hierarchy, go to the Inspector.

Add a **Rigidbody** to the Cube object. **Uncheck the Gravity property** of the Rigidbody component and **check the Is Kinematic property**. The properties for your Cube should look like the screenshot below.



Next, we will add the FSM to our cube.

Select the Cube object in the Hierarchy, go to the Inspector.

Add the **FSM_CON** script to the Cube.

Your **FSM_CON** script component should look like the screenshot below.

8. Adding a Crosshair (i.e., target) to the First-Person Camera

We will now add a target to the first-person camera. This will allow a player to select things more easily in the scene or target when firing projectiles.

Go to the Hierarchy Window.

Right-click on the MainCamera asset. Go to the 3D Object submenu and select plane.

See the screenshot below for an example.

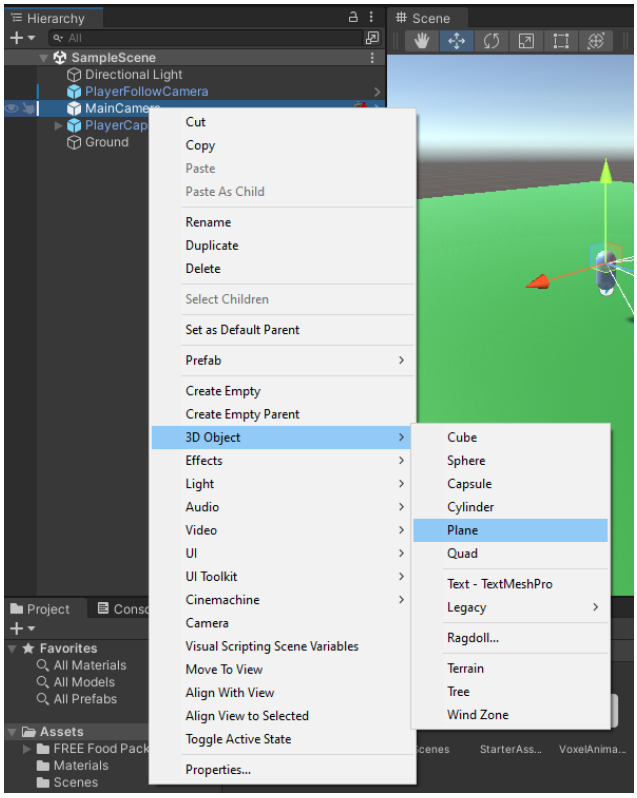


Image description: *Right-click on the MainCamera asset. Go to the 3D Object submenu and select plane.*

A plane should be created as a child object of the **MainCamera** asset.

Note, in maths, a plane is a flat, two-dimensional (2D) surface.

Give the plane the name Crosshair.

Rotate the plane -90 on the x axis, scale it to 0.008 on the x, y and z axis and set the z axis position to 1. Your transform component for the plane should now have the following position, rotation, and scale properties:

Position:

X: 0
Y: 0
Z: 1

Rotation:

X: -90
Y: 0
Z: 0

Scale:

X: 0.008
Y: 0.008
Z: 0.008

Also, remove the mesh collider component from the plane. Do this by going to the inspector and finding the Mesh Collider component.

Then, click the three dots on the right side of the mesh collider text. Then click Remove Component. See the screenshot below for an example.

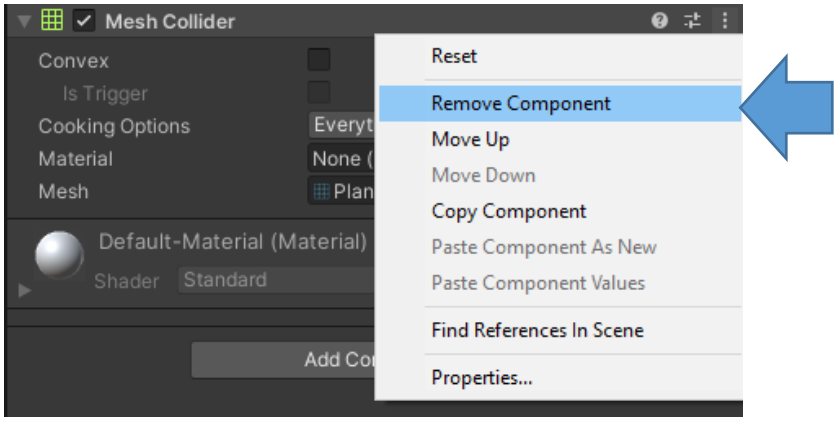


Image description: *Find the Mesh Collider component on the crosshair plane. Click the three dots on the right side of the mesh collider text. Then click Remove Component.*

You should now have a plane in the middle of the screen that is facing the camera. See the screenshot below for an example.

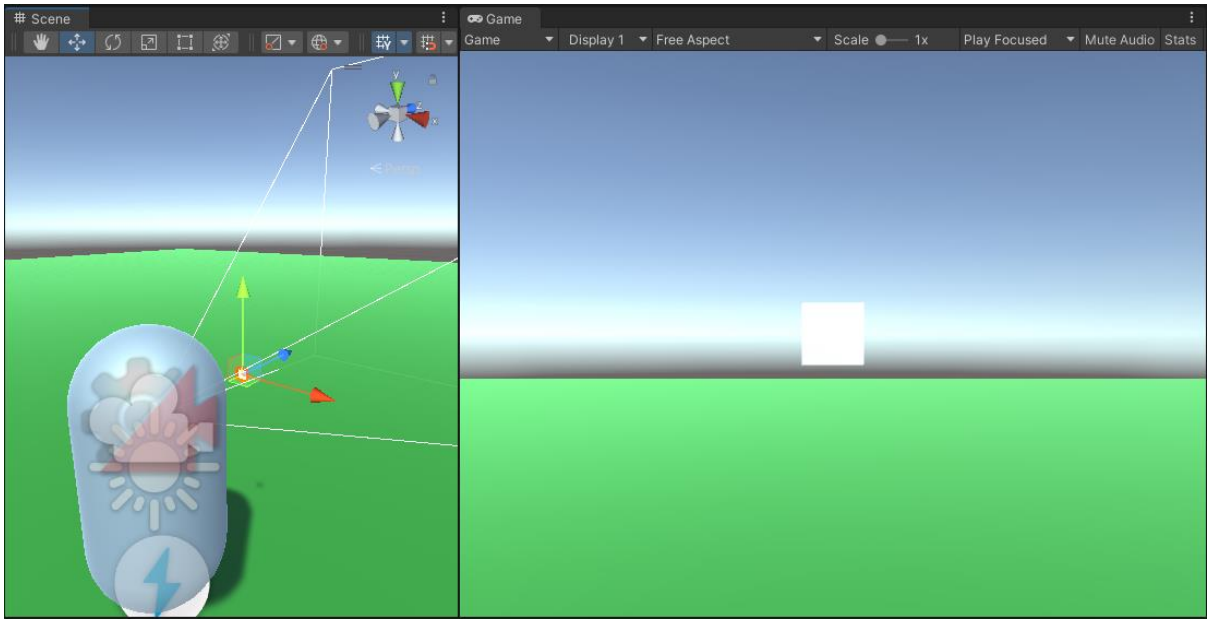


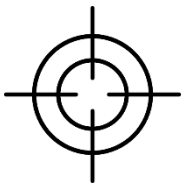
Image description: *A plane in the middle of the screen that is facing the camera. In the Scene window on the left of the image we can see the plane position in front of the camera and first-person capsule. The plane is selected in the scene.*

Next, we need to add a crosshair texture.

You can search the internet for a crosshair. In general, you need a crosshair with a transparent background.

Here is the crosshair I found:

> https://www.flaticon.com/free-icon/crosshair_865405
> *Crosshair icons created by Good Ware.*



> I downloaded the 512px png.

You can use this crosshair too or you can find your own on the internet.

Download your crosshair texture.

Go to the Unity Editor.

Go to the Project Window. Select the Assets folder. Right click in the righthand panel of the Assets folder and select Create -> Folder.

When the folder is created give it the name **Textures**.

Double click on the Textures folder to open it.

Next, we will import the image you have downloaded into your Unity project by dragging it from the saved location to the Assets window (which should be open in the Texture folder).

Your Texture folder should look like the screenshot below.

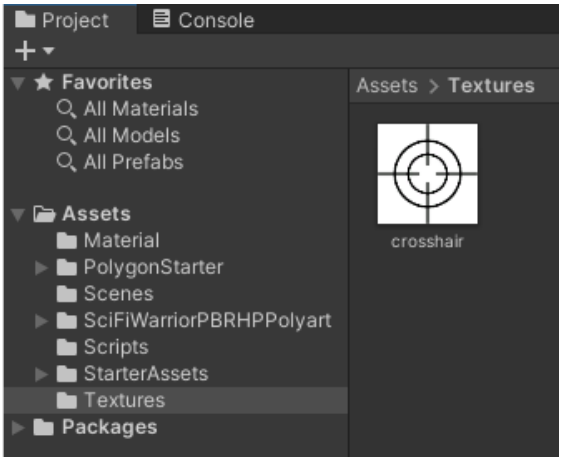


Image description: *The project window. A Textures folder has been created and opened. A texture has been added to the folder.*

Drag your crosshair texture from the project window onto the crosshair plane in the hierarchy.

When you drag your crosshair texture from the Project window onto the crosshair plane in the hierarchy Unity will automatically create a material for the crosshair in a Materials folder.

The crosshair texture should now be applied to the crosshair plane. See the screenshot below for an example.

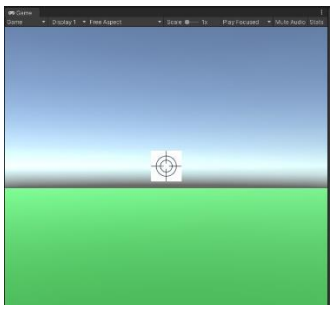


Image description: *The game window showing the crosshair texture applied to the crosshair plane.*

You will probably have a white border around your crosshair. We want to remove this.

Go to the hierarchy and click on the Crosshair plane.

Go to the inspector and find the Crosshair material.

Click on the triangle on the left side of the word “Shader”. This will expose more material properties.

Find the Rendering Mode dropdown list and select Cutout from the list.

See the screenshot below for an example.

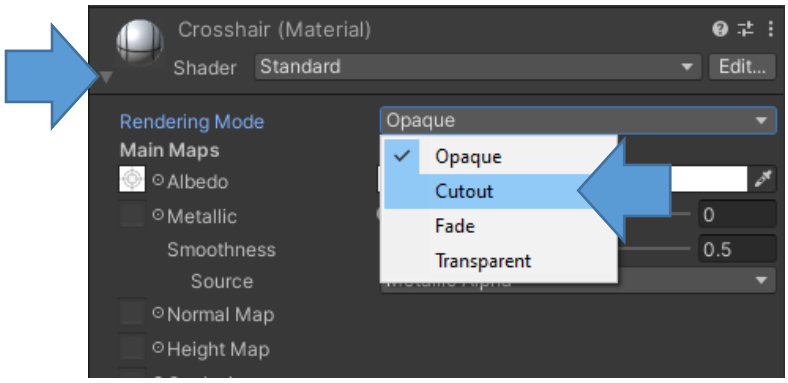


Image description: *The material component for the crosshair plane. Click on the triangle on the left side of the word “Shader”. Find the Rendering Mode dropdown list and select Cutout from the list.*

The background to the crosshair should now be transparent. See the screenshot below for an example.

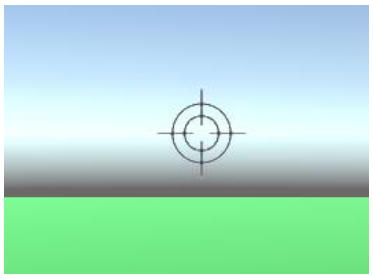


Image description: *The game window showing the crosshair texture applied to the crosshair plane. The background is now transparent.*

Save the scene.

🎮 **We are now ready to playtest the scene.**

Press the play button to playtest your scene.

You should be able to walk around the environment. You should have a crosshair at the centre of your camera in the game window.

The default controls:

- Arrow keys or WASD to move. Spacebar to jump. Hold down shift to sprint.

- Mouse look.

See the screenshot below for an example.

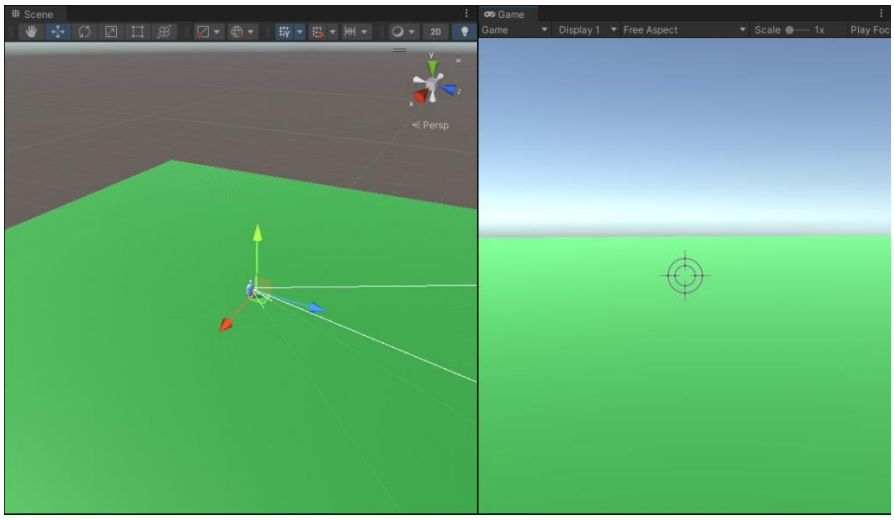


Image description: *A playtest of the scene using the first-person controller. Note, in my setup I have the Scene and Game view side-by-side.*

When you have finished walking around stop playtesting your scene.

9. Creating a Simple Projectile - Sphere

We will now add a projectile to the first-person controller. This will allow us to explore further coding in Unity.

Go to the hierarchy window and click the plus (+) menu. Select 3D Object -> Sphere. See the screenshot below for an example.

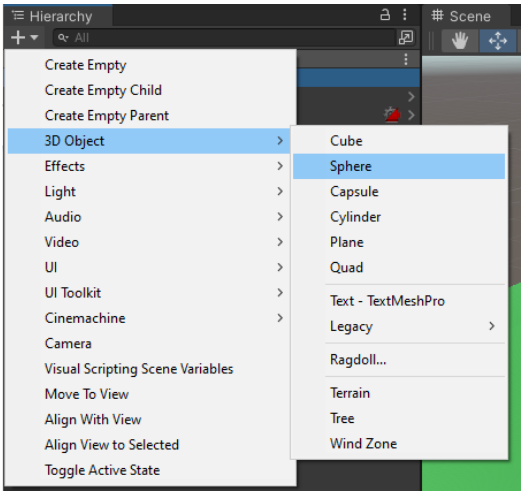


Image description: *The Hierarchy Window and the plus (+) menu. Sphere has been selected in the menu.*

Call the sphere **Projectile**.

Select the sphere object in the hierarchy, move the cursor over the scene window and press F. This should focus you view on the sphere. See the screenshot below for an example.

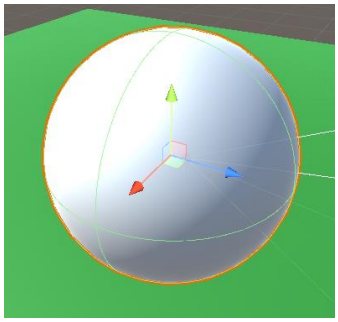


Image description: *The sphere projectile has been selected. I have pressed the F key to focus on the sphere. The scene view camera moves close to the sphere.*

Next, we will add a material to the sphere.

Go to the Assets -> materials folder.

Right click in the Materials folder and go to Create -> Material.

Give the Material the name **ProjectileColour**.

Select the material in the Project window. You should be able to see its properties in the inspector.

Go to the inspector, click on the block next to the Albedo property at top of the Inspector.

See the screenshot below for an example.

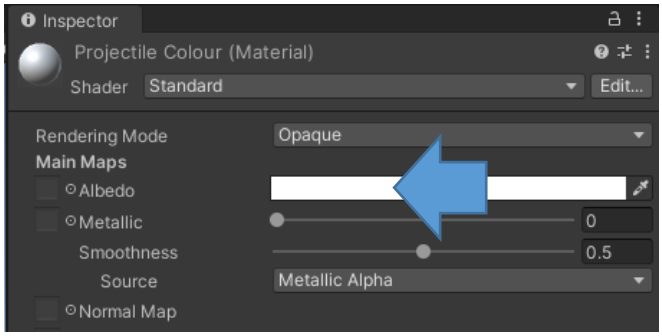


Image description: *The inspector window for the material. Click on the white box next to the Albedo property.*

Clicking on the block next to the Albedo property at top of the Inspector will open a colour picker window.

Select a red colour. I entered the hexadecimal value: FF0000.

To assign the material, drag the **ProjectileColour** material from the Project window and drop it onto the name of the sphere object in the hierarchy.

Your sphere should now match the colour of your material. See the screenshot below for an example.

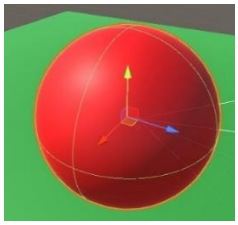


Image description: *The sphere projectile has been selected. The projectile colour material has been applied to the sphere.*

Next, we will set the physics engine to control the projectile by <u>adding a rigidbody component to the sphere</u>.

Select the Sphere in the hierarchy.

Go to the Inspector and click the **add component** button. Search for the rigidbody. Select rigidbody from the list to add the component.

<u>Next, we will give the projectile a tag of Projectile.</u>

Make sure the projectile sphere is still selected in the hierarchy.

Go to the Inspector. Click the tag dropdown menu and select “Add Tag…”.

The Inspector will change to a “Tags & Layers” window.

Click the plus (+) icon in the Tags section. See the screenshot below for an example.

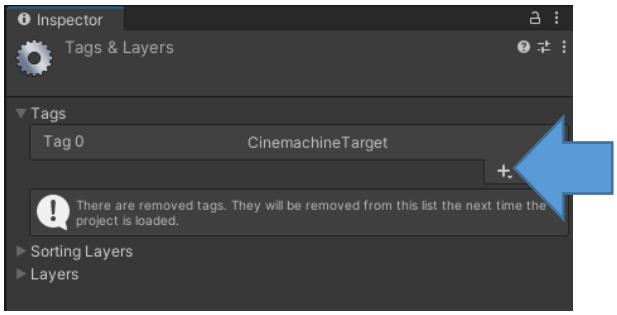


Image description: *When you click the “Add Tag…” option the Inspector will change to a “Tags & Layers” list. Click the plus (+) icon in the Tags section.*

Enter the name **Projectile** and **click save**.

“Tags & Layers” window should now have the **Projectile** tag in it.

Select the sphere projectile again in the Hierarchy.

The inspector should now have the sphere’s properties again.

Go to the inspector. Click the tag dropdown menu and select the **Projectile** tag.

Next, we want the projectile to be stored and instantiated when a key is pressed. We do not want the projectile in the scene by default.

Therefore, we need to store the object as a prefab and instantiate it when a key is pressed.

Go to the Project Window. Select the Assets folder. Right click in the righthand panel of the Assets folder and select Create -> Folder.

When the folder is created give it the name **Prefabs**.

Next, drag the sphere from the hierarchy and drop it into the Prefabs folder in the project window.

Save the scene. Go to the File menu and find the save option.

You can now delete the original sphere object in the scene / from hierarchy window. Do not delete the Projectile in the project window.

Save the scene.

10. Creating a Projectile Launcher Script

Next, we will add a script to our Unity project to launch (e.g., fire) our projectile.

Select the Assets folder in the project window.

Go to the Assets -> Scripts folder.

Right click in the Scripts folder and go to Create -> C# Scripts.

When the script has been created give it the name **ProjectileLauncher**.

Double click on your **ProjectileLauncher** C# script to open it in Visual Studio.

Update the code in the script file to match the code below.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ProjectileLauncher : MonoBehaviour
{
    [SerializeField]
    private Rigidbody projectileRigidBody;
    [SerializeField]
    private float projectilePower = 1500;
    [SerializeField]
    private GameObject muzzle;

    [SerializeField]
    private float COOLDOWN_TIME = 0.5f;
    private float coolDown = 0;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        if (coolDown <= 0)
        {
            if (Input.GetButtonUp("Fire1"))
            {
                coolDown = COOLDOWN_TIME;

                // Instantiate the projectile.
                Rigidbody aInstance = Instantiate(projectileRigidBody,
                        muzzle.transform.position, transform.rotation) as Rigidbody;

                // Add force.
                Vector3 forward = transform.TransformDirection(Vector3.forward);
                aInstance.AddForce(forward * projectilePower);

                // Destroy the object after X seconds.
                Destroy(aInstance.gameObject, 8);
            }
        }
        else
        {
            coolDown = coolDown - Time.deltaTime;
        }
    }
}
```

In the code above we add two class level variables to the script:

private Rigidbody projectileRigidBody;
private float projectilePower = 1500;

These variables will store a reference to the GameObjects Rigidbody component and store a float value that represents the projectile power.

We also add another variable that stores the GameObject that represents the muzzle position of the weapon firing the projectile.

There are also two more variables that handle how quickly the player / user can fire a projectile. The time is set in seconds in the COOLDOWN_TIME variable.

In the update method we add code to check if the cooldown counter is equal to zero or less than zero. If this is true, we check if the Fire1 button has been pressed. If this is true, we create an instance of the project using the Rigidbody component variable, then add a force to it to make it move. Finally, we tell it to destroy itself in 8 seconds. If the cooldown counter is not equal to zero or less than zero, we decrease it by the amount of time since the update method was last called.

Save the script in Visual Studio and go back to Unity.

Next, we need to attach the script to the camera.

In the Unity Editor, go to the hierarchy window and select the MainCamera GameObject.

Go to the Inspector. In the Inspector click the **Add Component button**, which is at the bottom of the window. Search for the **ProjectileLauncher** script and then double click on it to add it.

Next, we need to set some properties in the **ProjectileLauncher** component.

In the Inspector, find the **ProjectileLauncher** component.

We also need to assign the Projectile prefab to the script variable **projectileRigidbody** and we need to assign the **muzzle** GameObject.

To do this, drag the **Projectile prefab** from the project window and drop it onto the **projectileRigidbody** variable in the inspector.

See the screenshot below for an example.

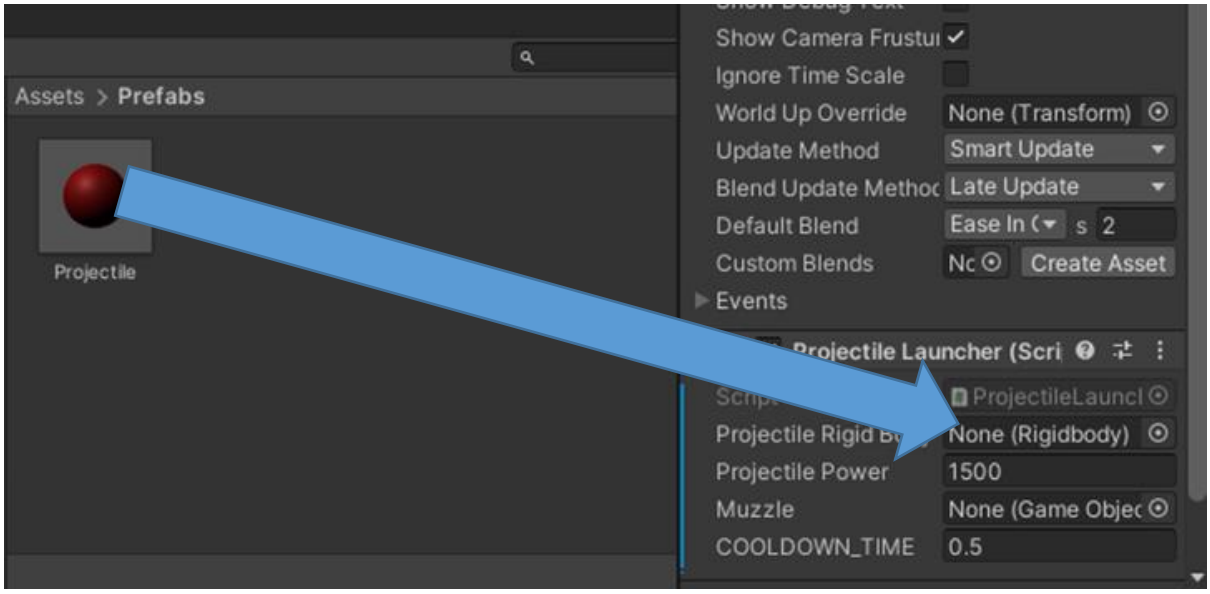


Image description: *drag the Projectile prefab from the project window and drop it onto the projectileRigidbody variable in the inspector.*

Next, we set the muzzle variable.

Go to the hierarchy, select the MainCamera GameObject.

Go to the inspector and find the **Projectile Launcher** component.

Click the bullet icon next to the Muzzle variable. See the screenshot below for an example.

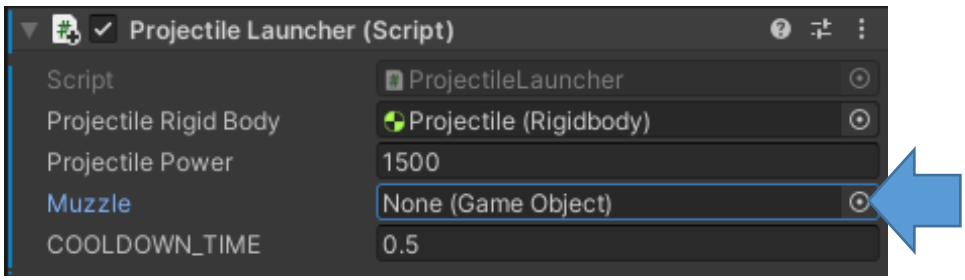


Image description: *Click the bullet icon next to the Muzzle variable.*

A “Select GameObject” window will appear. Click the Scene tab and select the Crosshair from the list. See the screenshot below for an example.

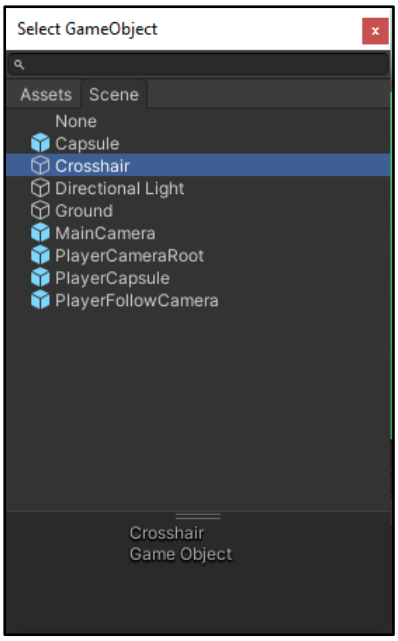


Image description: *Click the Scene tab and select the Crosshair from the list. Do this by double clicking on the Crosshair GameObject to select it.*

Double click on the Crosshair GameObject to select it.

The **Projectile Launcher** component for the **MainCamera** GameObject should now look like the screenshot below. The **projectileRigidbody** variable and **muzzle** variable have been set.

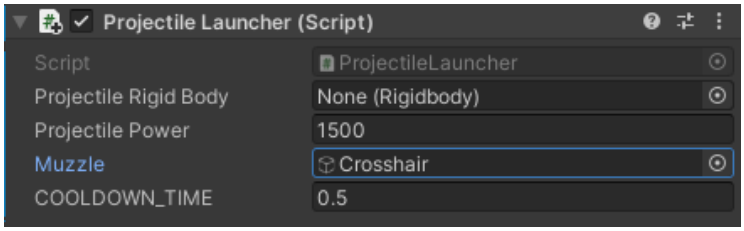


Image description: *The Projectile Launcher component for the MainCamera GameObject.*

Save the scene.

7. Testing the FSM

We will now test the FSM we have implemented.

🎮 **Playtest** - Click the **Play** button.

Use the W, A, S, D keys to move the player camera.

You should see the cube spin when the FSM is in its Idle state. If you move towards the cube it should start chasing, you / the player. If you fire a projectile and hit the cube it should move away from you / the player. When the cube is 20 units away from the player it moves back to idle state.

Your scene should look like the screenshot below.

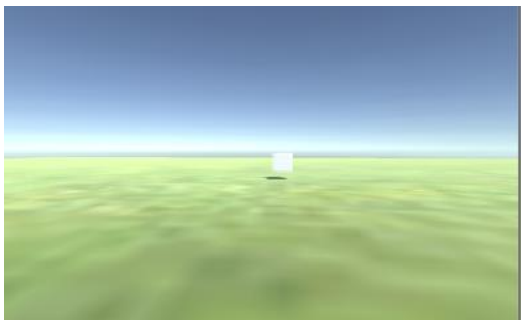


Image description: *The simple FSM in action.*

If you FSM does not work, please review your code. If you still have issues with you code, please speak to a tutor.

Below are some questions that require you to edit the program above. **Please work through these questions:**

1. We haven't implemented any code in the **FSMProcessInput** function. Add code to this function that changes the state of the FSM to chase when the c key has been pressed.

2. Make a copy of this project. Update the project so that the FSM controls the Footman animated character from the "Programming Animated Characters in the Unity Game Engine" workbook. The animations for the animated character should be aligned to the state of the FSM. The idle animation should play during the animated state and the walking animation should play during the retreat and chase states.

> END OF STUDENT WORKBOOK ▮