UNIVERSITY OF WOLVERHAMPTON
KNOWLEDGE ▪ INNOVATION ▪ ENTERPRISE

Unity Game Engine Practical Activity

Student Workbook

Workbook Title:

Pathfinding: Custom Waypoints and A* (Star) Example.

Workbook Description:

In this student activity, you will develop waypoints and a A* pathfinding example using Unity. We will code all the elements in this example from scratch.

Please Read:

- This student workbook will check that you have understood and can apply the knowledge you have gained during this week's talks.
- Please read each workbook section 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# for 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.

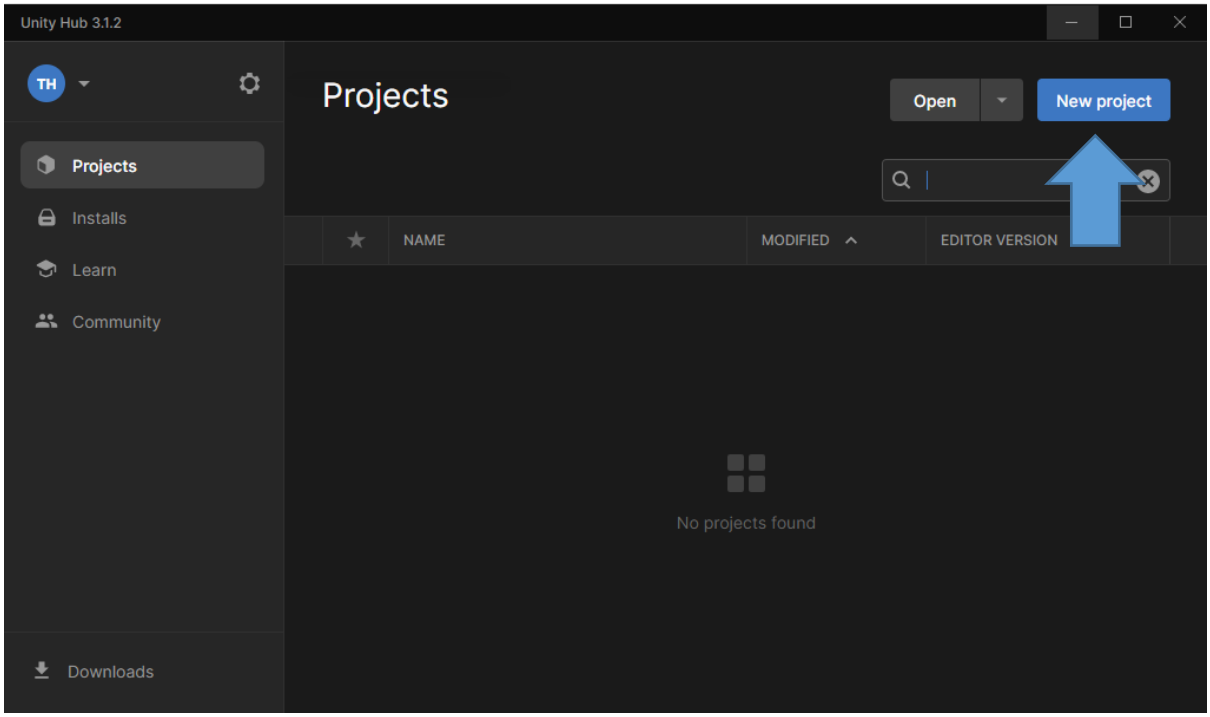

Image description: *The Unity hub with an arrow pointing to the New Project button.*

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

The window looks like the screenshot below.

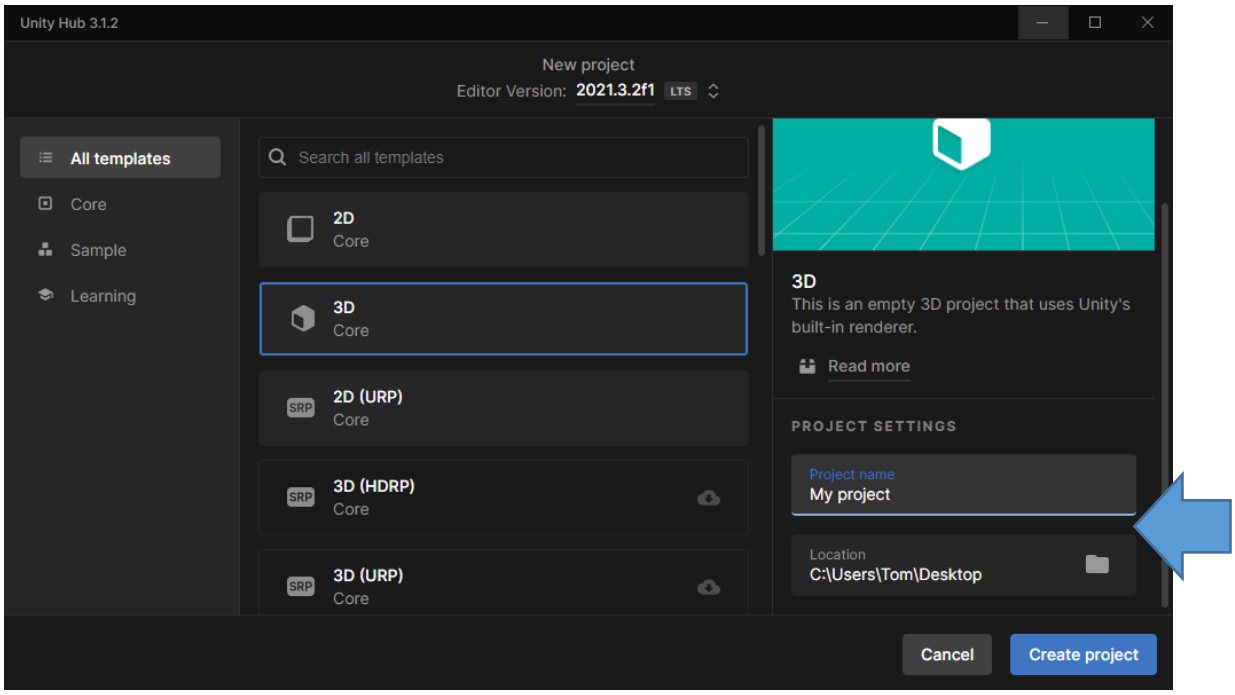

Image description: *The Unity hub “New project” screen.*

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 **Pathfinding**.

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
- POLYGON Starter Pack - Low Poly 3D Art by Synty
- 5 animated Voxel animals

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. Your Package Manager window should look like the screenshot below.

When you click the Import button a small "Import Unity Package" window will appear. See the screenshot below for an example.

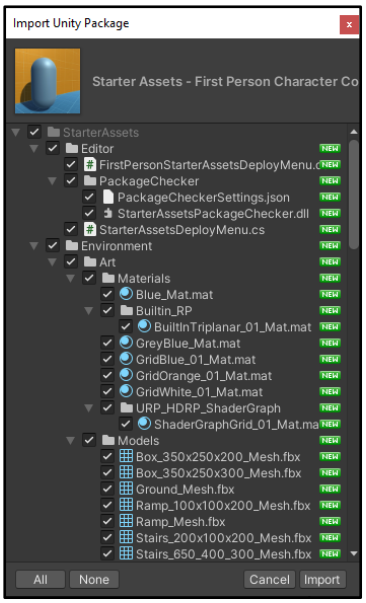

Image description: *The "Import Unity Package" window. Click Import.*

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.

As part of the process, the message in the screenshot below will be displayed. Click No.

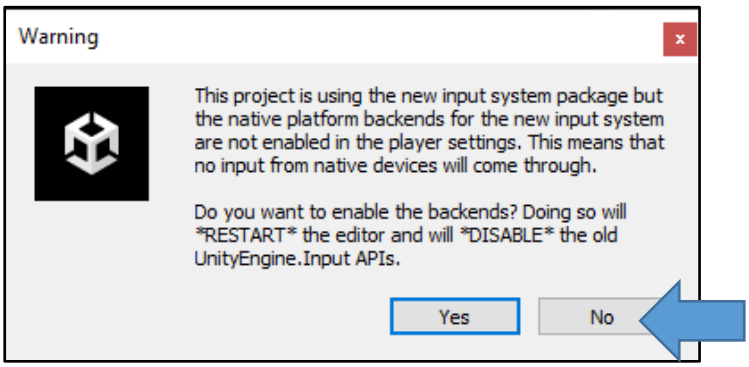

Image description: *A warning message stating that the assets being imported use the new input system package. Click no.*

Now, 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.

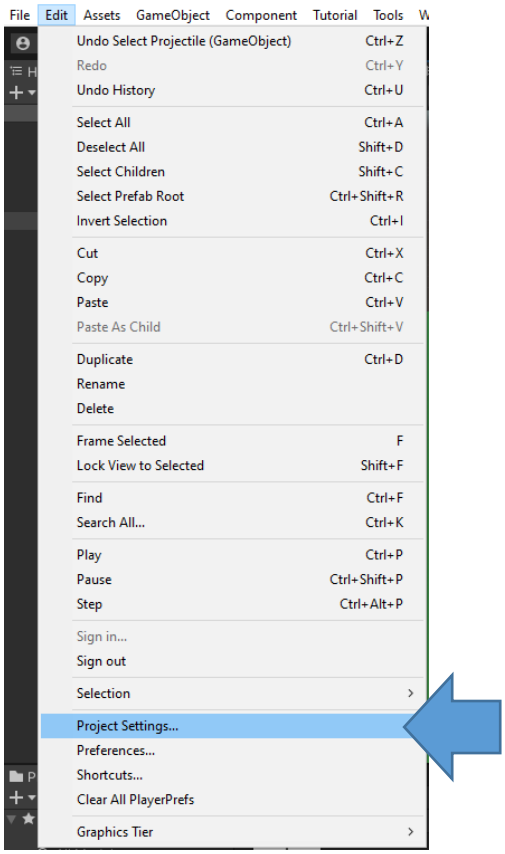

Image description: *Open the Project Settings window to set Unity project input type.*

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

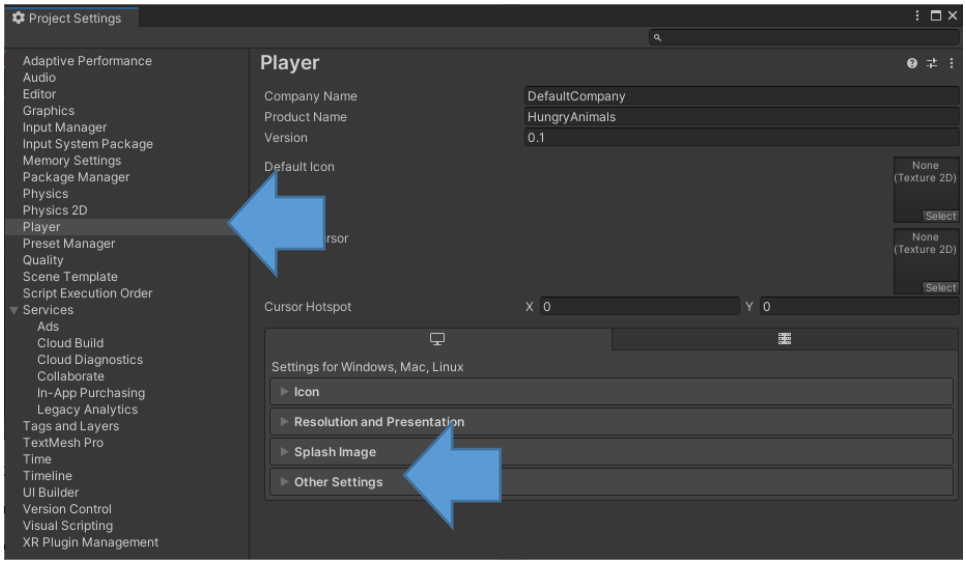

Image description: *Select Player from the list on the left and expand the “Other Settings” option.*

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.

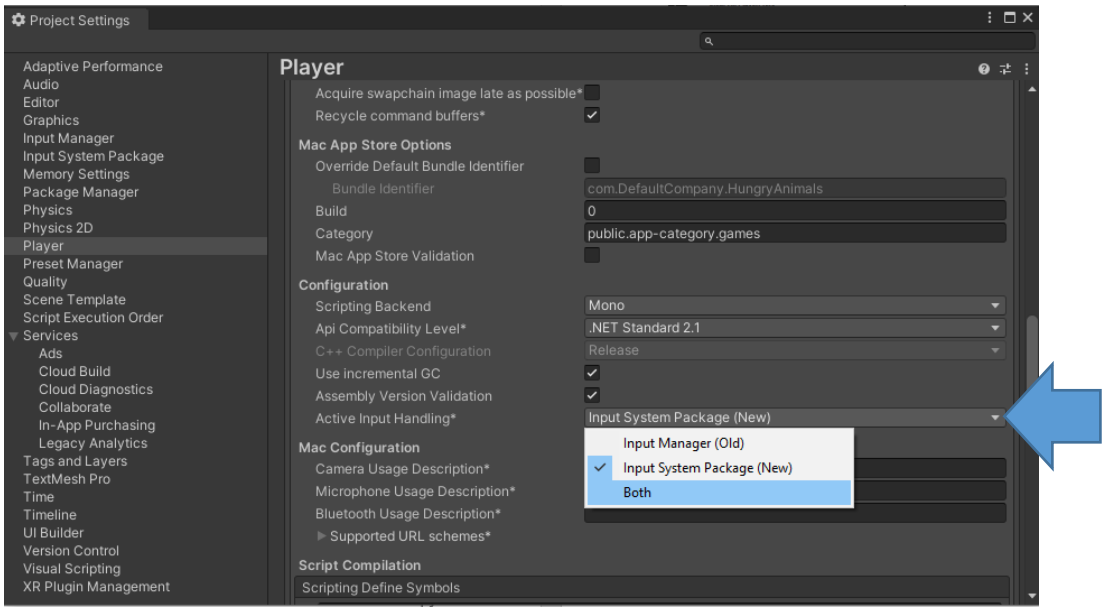

Image description: *Scroll down “Other Settings” until you find the “Active Input Handling” option. Click the drop-down menu and select <u>Both</u>.*

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

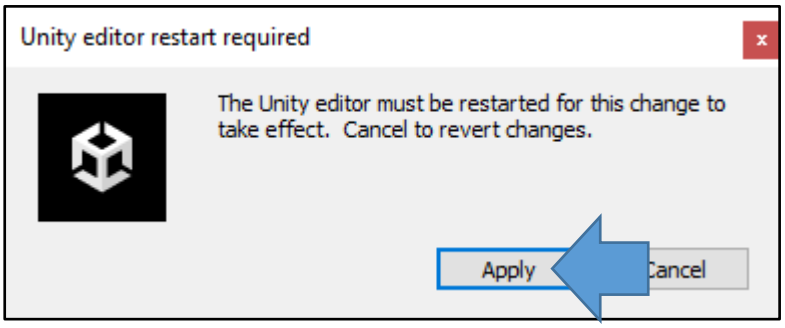

Image description: *A warning message stating that Unity must restart. Click Apply.*

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.

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.

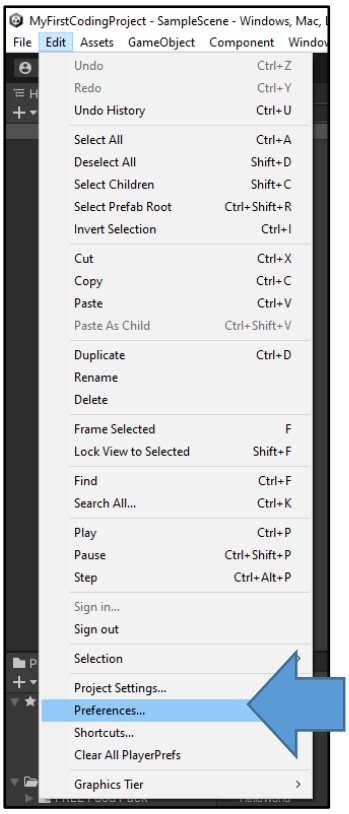

Image description: *Open the Preferences window to set Visual Studio as the external script editor and regenerate project files.*

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.

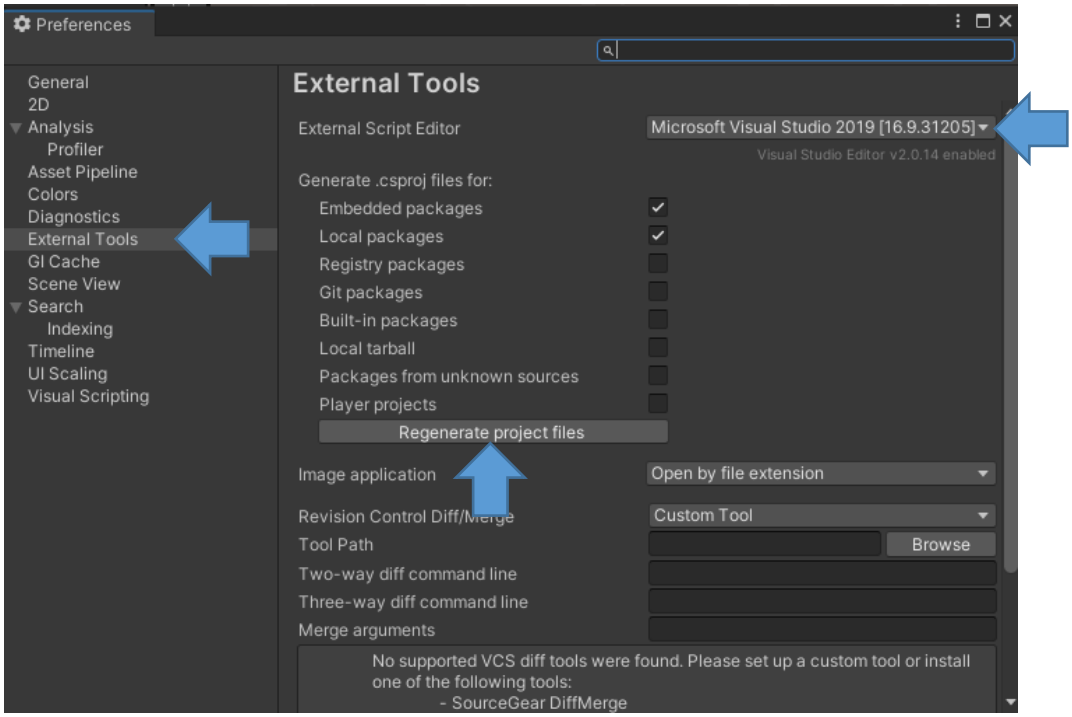

Image description: *The Preferences window. If needed, set the external script editor to Visual Studio. Then, click the regenerate project files button.*

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.

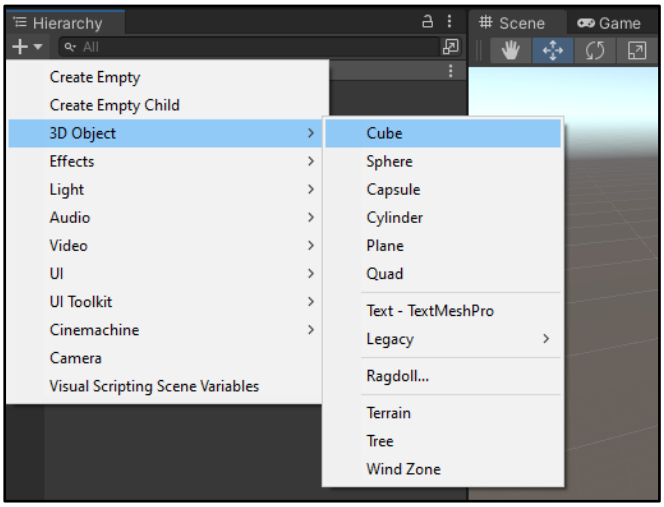

Image description: *The Hierarchy Window and the plus (+) menu. Cube has been selected in the menu.*

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.

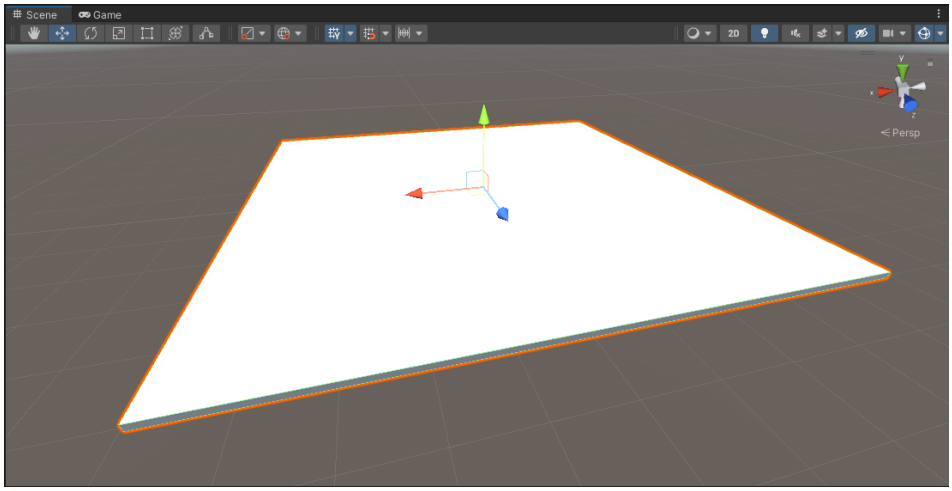

Image description: *The ground cube in the scene view.*

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.

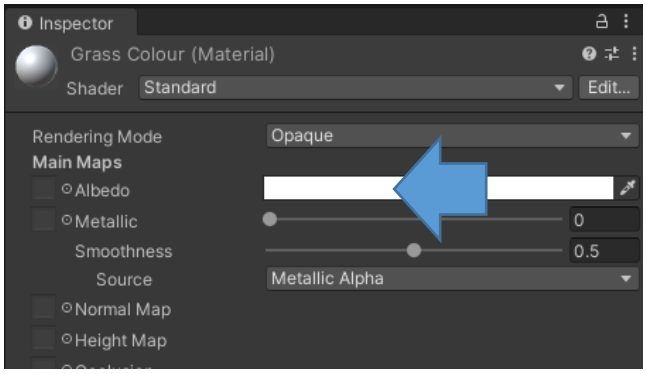

Image description: *The inspector window for the material. Click on the white box next to the Albedo property.*

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.

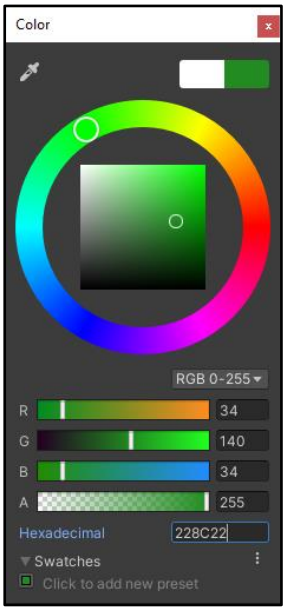

Image description: *The material Albedo property colour picker.*
Close the colour picker window.

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.

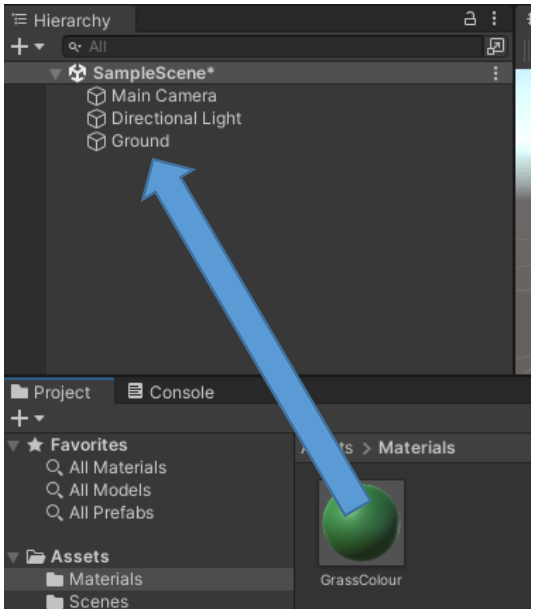

Image description: *Drag the material from the Project window to the ground GameObject.*

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

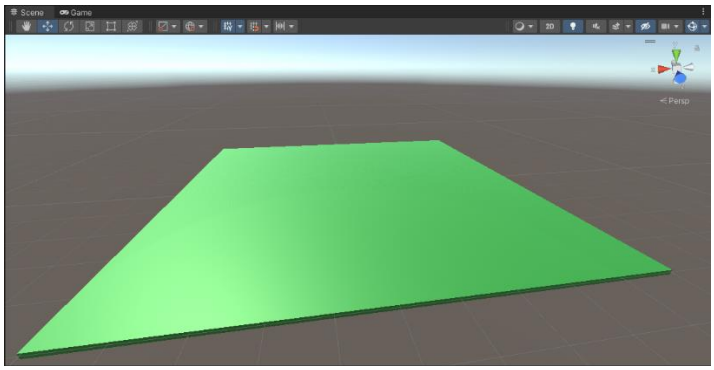

Image description: *The ground cube in the scene view. A green material has been applied to it.*

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.

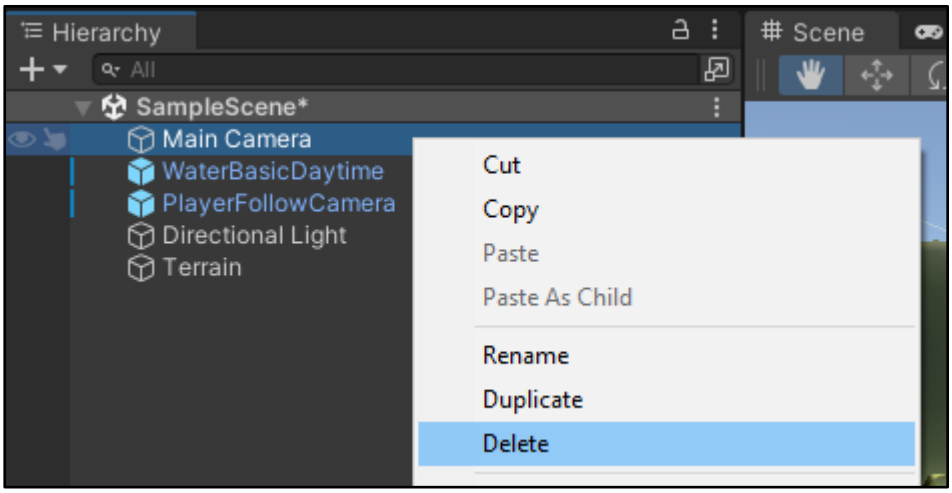

Image description: *Example of how to delete the "Main Camera" from the scene.*

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

Go to the Tools drop-down menu, select Starter Assets, then <u>select Reset First Person Controller</u>. See the screenshot below for an example.

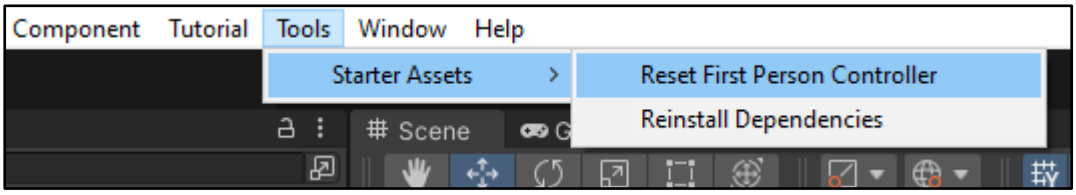

Image description: *To add a first-person camera, go to the Tools drop-down menu, select Starter Assets, then select Reset First Person Controller.*

Unity should add all the assets you need to your scene.

The first thing with need to do is move the first-person camera to a good starting position.

Go to the Hierarchy and select **PlayerCapsule** in the list.

You can now use the move tool to move the first-person camera to a good starting position. Its default position is x: 0, y: 0, z: 0. However, we can move it more quickly by entering a position in the position part of the transform component.

Select the **PlayerCapsule** in the hierarchy.

Go to the inspector, set the position of the **PlayerCapsule** to:

X: 0,
Y: 2,
Z: 0.

This should move the **PlayerCapsule** (which includes the camera) to a good starting location (e.g., over land). If the location is not good for you, use to move tool to drag the camera in the scene to a good position.

See the screenshot below for an example.

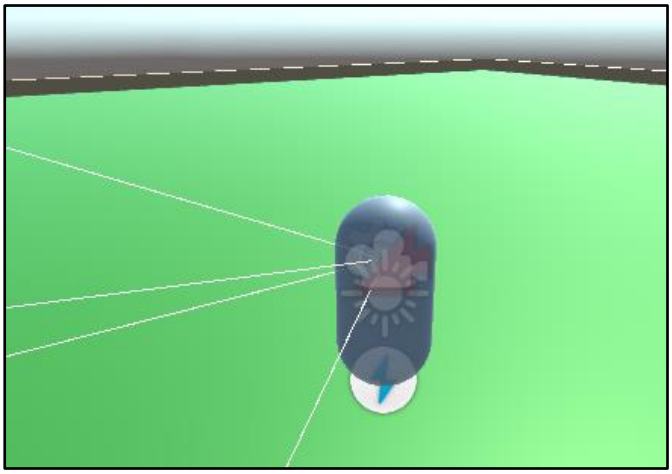

Image description: *The PlayerCapsule after it has been moved to a* good starting location.

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.

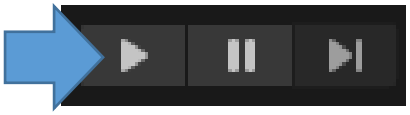

Image description: *The game / scene play controls on the main toolbar. Click the play button.*

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 shift 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.

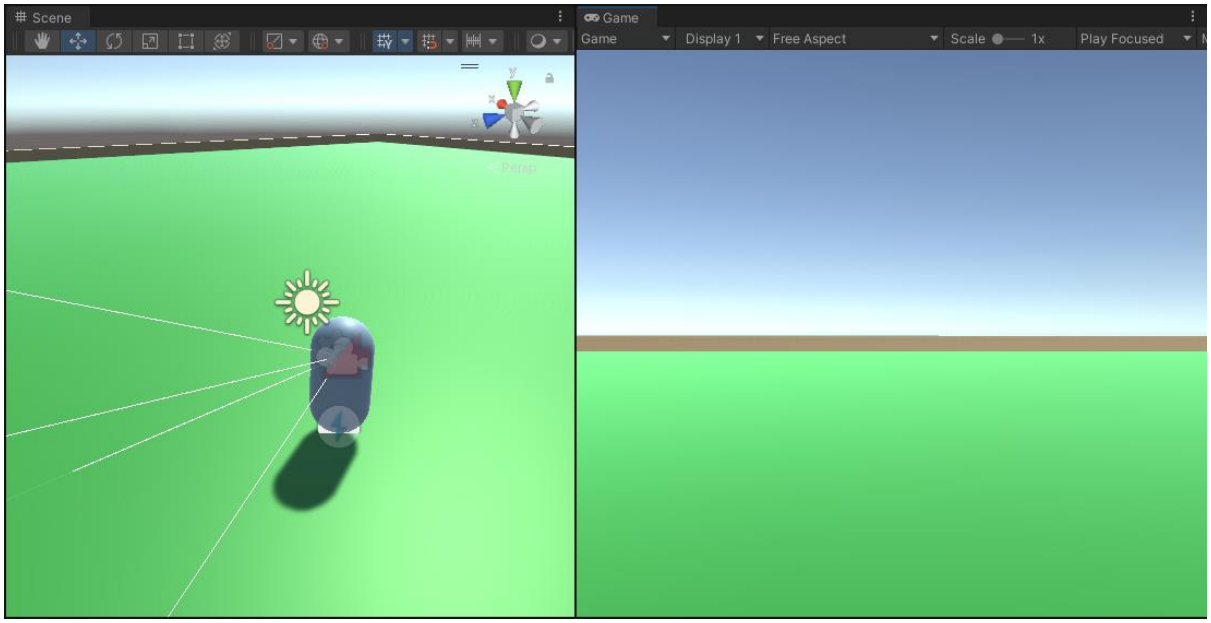

Image description: *A playtest of the scene using the first-person controller. Note, in my setup I have the Scene and Game view side-by-side.*

6. Creating a Simple Waypoint Graph – Visual Representation

Next, we will create some simple waypoints in Unity. We will do this in code. These waypoints will allow us to create a waypoint graph in the Unity editor and provide a visual representation of the waypoint.

We will then use the visual waypoints to create a waypoint graph in code. The code waypoint graph will be used by a navigation algorithm to allow a non-player character or agent to navigate around a virtual environment.

General information:

Pathfinding techniques cannot directly utilize the 3D game environment data employed to render the game world. Pathfinding techniques usually make use of a world representation and an algorithm for finding an appropriate route through the representation.

Pathfinding algorithms use simplified representations of the game world. Waypoint maps have traditionally been the most popular approach to representing game worlds. A directed non-negative weighted graph.

Below is an example of a simple waypoint graph. The image shows a top-down / birds eye view of a virtual world. The blue triangle represents an obstacle, such as a mountain.

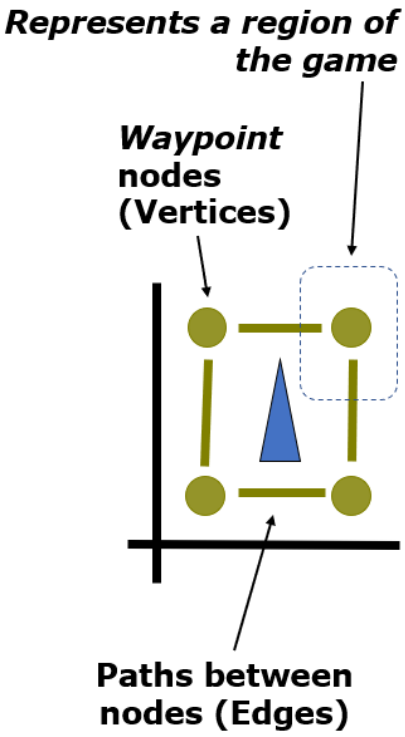

Image description: *An example of a simple waypoint graph.*

We will start by creating a C# class that represents a connection in your visual representation of our waypoint graph. The class will only be used for the visual representation. We will create a connection class for our code representation of the waypoint graph later.

We will now create a C# script to represent a waypoint node.

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 **VisGraphConnection**.

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

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

[System.Serializable]
public class VisGraphConnection
{
    // The to node for this connection.
    [SerializeField]
    private GameObject toNode;
    public GameObject ToNode
    {
        get { return toNode; }
    }
}
```

In the code above we add the [System.Serializable] attribute so that our VisGraphConnection class is visible in the inspector.

We then add one variable that stores the connection “to node”. This is the node the connection goes to. The from node will be the waypoint itself.

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

Next, we will create a C# script to represent a waypoint node.

Go to 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 **VisGraphWaypointManager**.

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

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

// Used to display text above the node.
using UnityEditor;

public class VisGraphWaypointManager : MonoBehaviour
{
    // Allow you to set the waypoint text colour.
    [SerializeField]
    private enum waypointTextColour { Blue, Cyan, Yellow };

    #pragma warning disable
    [SerializeField]
    private waypointTextColour WaypointTextColour = waypointTextColour.Blue;
    #pragma warning restore

    // List of all connections from this node.
    [SerializeField]
    public List<VisGraphConnection> connections = new List<VisGraphConnection>();
    public List<VisGraphConnection> Connections
    {
        get { return connections; }
    }

    // Allow you to set a waypoint as a start or goal.
    [SerializeField]
    private enum waypointPropsList { Standard, Start, Goal };

    #pragma warning disable
    [SerializeField]
    private waypointPropsList WaypointType = waypointPropsList.Standard;
    #pragma warning restore

    // Controls if the node type is displayed in the Unity editor.
    private const bool displayType = false;

    // Used to determine if the waypoint is selected.
    private bool ObjectSelected = false;

    // Text displayed above the node.
    private const bool displayText = true;
    private string infoText = "";
    private Color infoTextColor;

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

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

```
// Draws debug objects in the editor and during editor play (if option set).
void OnDrawGizmos()
{
    // Text displayed above the waypoint.
    infoText = "";
    if (displayType)
    {
        #pragma warning disable
        infoText = "Type: " + WaypointType.ToString() + " / ";
        #pragma warning restore
    }
    infoText += gameObject.name + "\n Connections: " + Connections.Count;

    switch (WaypointTextColour)
    {
        case waypointTextColour.Blue:
            infoTextColor = Color.blue;
            break;
        case waypointTextColour.Cyan:
            infoTextColor = Color.cyan;
            break;
        case waypointTextColour.Yellow:
            infoTextColor = Color.yellow;
            break;
    }

    DrawWaypointAndConnections(ObjectSelected);

    if (displayText)
    {
        GUIStyle style = new GUIStyle();
        style.normal.textColor = infoTextColor;
        Handles.Label(transform.position + Vector3.up * 1, infoText, style);
    }
    ObjectSelected = false;
}

// Draws debug objects when an object is selected.
void OnDrawGizmosSelected()
{
    ObjectSelected = true;
}
```

```
// Draws debug objects for the waypoint and connections.
private void DrawWaypointAndConnections(bool ObjectSelected)
{
    Color WaypointColor = Color.yellow;
    Color ArrowHeadColor = Color.blue;
    if (ObjectSelected)
    {
        WaypointColor = Color.red;
        ArrowHeadColor = Color.magenta;
    }

    // Draw a yellow sphere at the transform's position
    Gizmos.color = WaypointColor;
    Gizmos.DrawSphere(transform.position, 0.2f);

    // Draw all the connections.
    for (int i = 0; i < Connections.Count; i++)
    {
        if (Connections[i].ToNode != null)
        {
            if(Connections[i].ToNode.Equals(gameObject))
            {
                infoText = "WARNING - Connection to SELF at element: " + i;
                infoTextColor = Color.red;
            }

            Vector3 direction = Connections[i].ToNode.transform.position - transform.position;
            DrawConnection(i, transform.position, direction, ArrowHeadColor);

            if(ObjectSelected)
            {
                // Draw spheres along the line.
                Gizmos.color = ArrowHeadColor;

                float dist = direction.magnitude;
                float pos = dist * 0.1f;
                Gizmos.DrawSphere(transform.position +
                    (direction.normalized * pos), 0.3f);
                pos = dist * 0.2f;
                Gizmos.DrawSphere(transform.position +
                    (direction.normalized * pos), 0.3f);
                pos = dist * 0.3f;
                Gizmos.DrawSphere(transform.position +
                    (direction.normalized * pos), 0.3f);
            }
        }
        else
        {
            infoText = "WARNING - Connection is missing at element: " + i;
            infoTextColor = Color.red;
        }
    }
}
```

```
    // This arrow method is based on the example here: https://gist.github.com/MatthewMaker/5293052
    public void DrawConnection(float ConnectionsIndex, Vector3 pos, Vector3 direction,
        Color ArrowHeadColor, float arrowHeadLength = 0.5f, float arrowHeadAngle = 40.0f)
    {
        Debug.DrawRay(pos, direction, Color.blue);

        Vector3 right = Quaternion.LookRotation(direction) *
            Quaternion.Euler(0, 180 + arrowHeadAngle, 0) * new Vector3(0, 0, 1);

        Vector3 left = Quaternion.LookRotation(direction) *
            Quaternion.Euler(0, 180 - arrowHeadAngle, 0) * new Vector3(0, 0, 1);

        Debug.DrawRay(pos + direction.normalized +
            (direction.normalized * (0.1f * ConnectionsIndex)),
            right * arrowHeadLength, ArrowHeadColor);
        Debug.DrawRay(pos + direction.normalized +
            (direction.normalized * (0.1f * ConnectionsIndex)),
            left * arrowHeadLength, ArrowHeadColor);

    }
}
```

In the code above we include a class level variable to store a list of connections (VisGraphConnection objects) for the waypoint node. Remember, this class is designed to help use visually create a waypoint graph. We create a code representation of the waypoint graph used by the A* algorithm later.

There is also a class level variable to set the waypoint type in the editor. This might be useful if you wan to set waypoint nodes as goals or starts.

The rest of the code in the class is designed to display the position of the waypoint node and the connections for the node in the Unity editor or during editor play (if the option is set).

The code draws a line between nodes if there is a connection. It also draws an arrow hear near a node indicating there is a connection in a direction. The code also displays the number of connections above a node.

There are three helper visuals to help you determine if you have issues with a node. Firstly, the text above the node will display a warning if you have added a connection element, but not included a connection. It will warn you about the last missing connection is the list. The text above the node will also display a warning if you have added a connection to itself (e.g., a connection to the current waypoint node). It will warn you about the last connection to itself connection is the list.

Secondly, a node will add an arrowhead for each connection. Therefore, if you see multiple arrow heads on a single connection is means you have connected to another waypoint node twice. This might be intentional; however, it is useful to for you to know if this has happened.

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

Next, we need to attach the VisGraphWaypointManager script to an empty game object and create a prefab that we can use to create waypoint nodes.

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

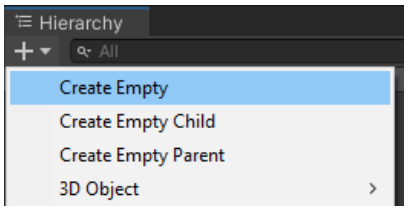

Image description: *The Hierarchy Window and the plus (+) menu. Create Empty has been selected in the menu.*

Call the empty GameObject **Waypoint**.

Make sure the empty **GameObject** called Waypoint is selected in the hierarchy.

Go to the inspector and click the **add component** button. Search for the **VisGraphWaypointManager** script. Select the **VisGraphWaypointManager** script from the list to add the component. See the screenshot below for an example.

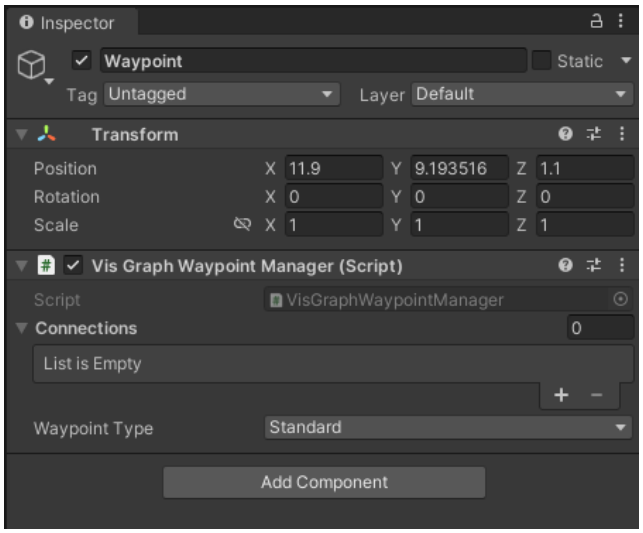

Image description: *The inspector window for the empty GameObject called Waypoint. The VisGraphWaypointManager script has been added as a component.*

Next, we will give the waypoint a tag of Waypoint.

Make sure the empty GameObject called Waypoint 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.

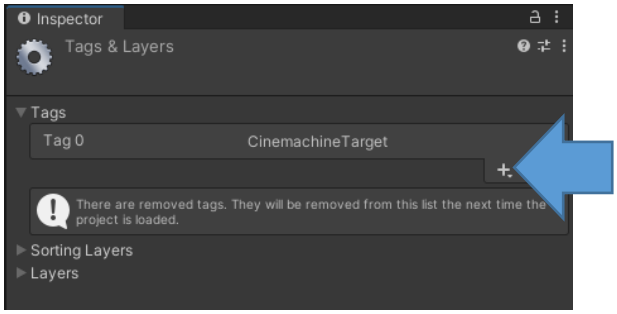

Image description: *When you click the “Add Tag…” option the Inspector will change to a “Tags & Layers” list. Click the plus (+) icon in the Tags section.*

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

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

Select the empty GameObject called Waypoint again in the Hierarchy.

The inspector should now have the empty GameObject called Waypoint properties again.

Go to the Inspector. Click the tag dropdown menu and select the **Waypoint** tag.

We want the waypoint to be stored in the asset folder and instantiated by a virtual world designer when they create the waypoint map. Therefore, we need to store the object as a prefab.

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 empty GameObject called Waypoint 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.

See the screenshot below for an example.

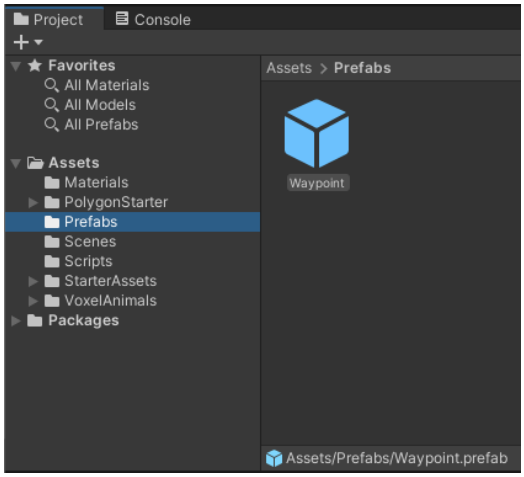

Image description: *The project window. The Prefabs folder has been opened. The empty GameObject called Waypoint has been dragged into the Prefabs folder. A prefab has been created and stored in the Prefabs folder.*

7. Adding the Waypoint Nodes to our Level and Setting Properties

Next, we will add the nodes to our level and set the connections for each node. We will visually create a waypoint graph in a Unity scene.

Waypoint graph nodes have connections that identify the links between each graph node. We will add waypoint nodes and set the connections

In general, you need to add 10 nodes to your level and set the connections so that your nodes form a waypoint graph. I would recommend you add a couple of nodes at a time and start setting the connections.

We should already have one node in the scene.

To add a waypoint node, you need to drag an instance of the "Waypoint" prefab into the scene. The "Waypoint" prefab is in the Assets -> Prefabs folder.

You need to place your waypoints so that they are just above your ground. Imagine that the waypoints float just above the ground.

In this workbook the ground is positioned at x: 0, y: 0, z: 0; therefore, **a good y location for our waypoints is y: 1**. We can have any x or z locations.

In the screenshot below I have dragged two more waypoints into the scene. I now have three waypoints in total. **I have positioned all the waypoints so that they have a y position of 1.**

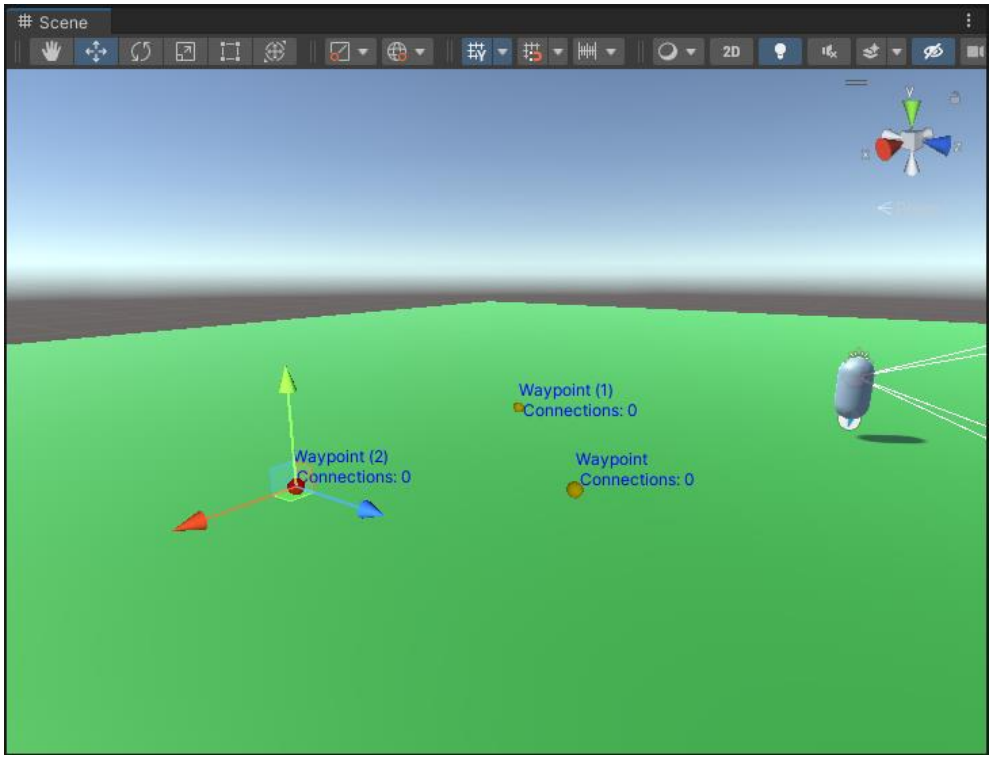

Image description: *The scene window. Three waypoint nodes have been added to the scene and all have been given the y position of 1.*

To start with I would recommend you build a waypoint map on a flat surface or terrain and keep the y position of all the waypoint nodes the same. Once you understand how to build a waypoint graph you can work with more uneven terrain.

You should do the above. Get three nodes in your scene and set all their y positions to 1.

Next, you need to give your waypoints unique names, such as Waypoint01 or Node01.

Note, giving your waypoints is not essentially for the A* algorithm to work; however, it gives you more clarity when creating your waypoint map and can help when debug; therefore, I would recommend it.

I have renamed my waypoints to Waypoint01, Waypoint02 and Waypoint03. See the screenshot below for an example.

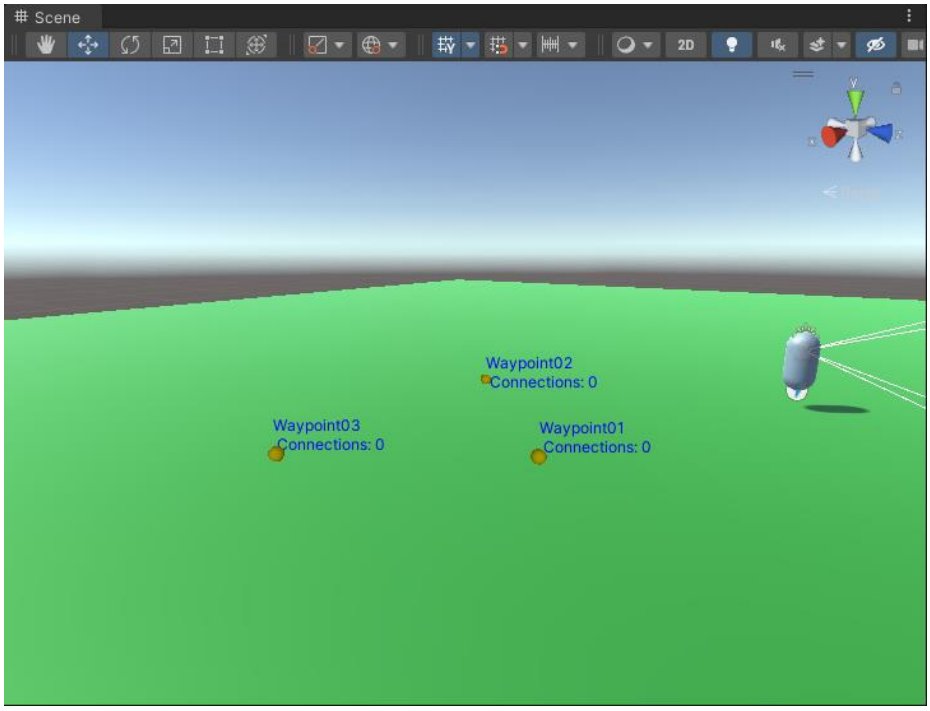

Image description: *The scene window. I have renamed my waypoints to Waypoint01, Waypoint02 and Waypoint03.*

Next, we will group our scene waypoints together to improve the organisation of our scene in the hierarchy.

Go to the Hierarchy Window and click the plus (+) menu. Select Create Empty.

Give the Empty GameObject the name **WaypointMap**.

Set the position of the WaypointMap GameObject to 0, 0, 0.

- **Information:** It is important to set the position of our empty grouping objects to the origin (0,0,0) so that the position of objects that are put inside this empty object align to world positions. When you make an object a child of another object, the child object has a position relative to the parent object. If we want the child object to have positions that align to world positions, we need the parent object to be at the origin (0,0,0).

Drag the three waypoint objects into the WaypointMap object. These objects should now be children of the **WaypointMap** object. See the screenshot below for an example.

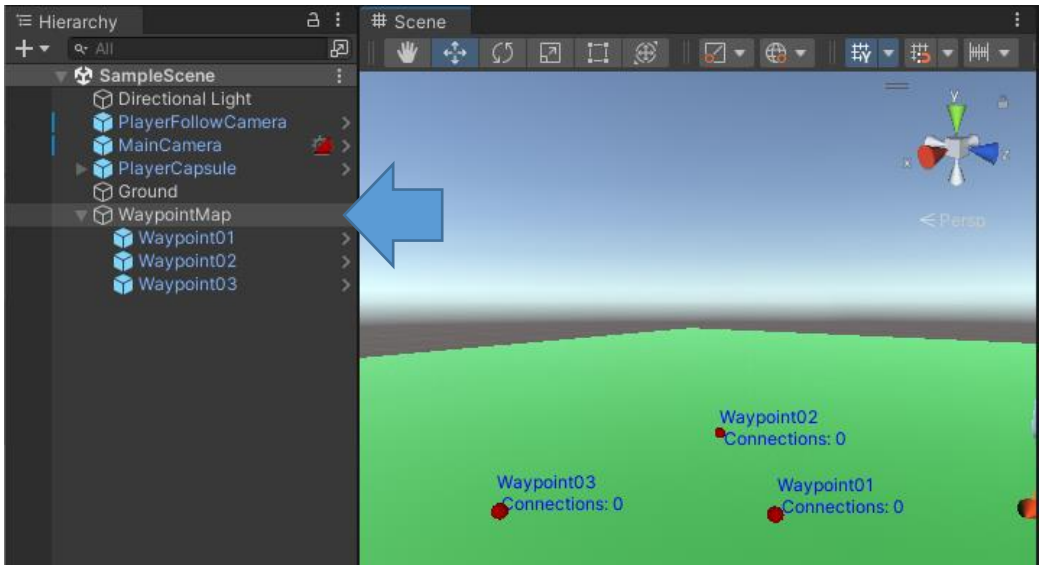

Image description: *The hierarchy window and scene window. The waypoint objects are now child objects of the empty WaypointMap object. The WaypointMap object is positioned at 0,0,0.*

You can click on the grey triangle next to the **WaypointMap** object name to close and open the grouping of objects.

Save your scene.

The next thing we need to do is set the connections for each node we added.

To do this, select the **Waypoint01** node and go to the inspector.

Go to the VisGraphWaypointManager script and enter the number of connections for the node, such as 2 or 4. Or click the plus (+) button to add a connection one at a time. See the screenshot below for an example.

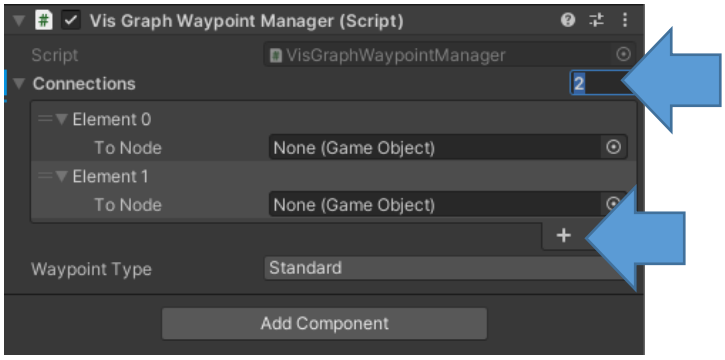

Image description: *The VisGraphWaypointManager script component in the inspector window. Enter the number of connections for the node, such as 2 or 4. Or*

click the plus (+) button to add a connection one at a time. In this example I have added two connections.
Next, you need to set the connections for the node. Each of the elements in the screenshot above (e.g., “Element 0”) represents an end node and thus a connection.

You need to drag a waypoint gameobject to an element in the inspector or click the bullet icon and select a waypoint node from the scene list.

For example, in the screenshot below I have selected Waypoint01 in the scene. I have gone to the inspector and found the VisGraphWaypointManager script component. I have added two connections. I have clicked on the bullet icon next to “Element 0”. A “Select GameObject” window has appeared. I then double click on Waypoint02 to set a link between Waypoint01 and Waypoint02.

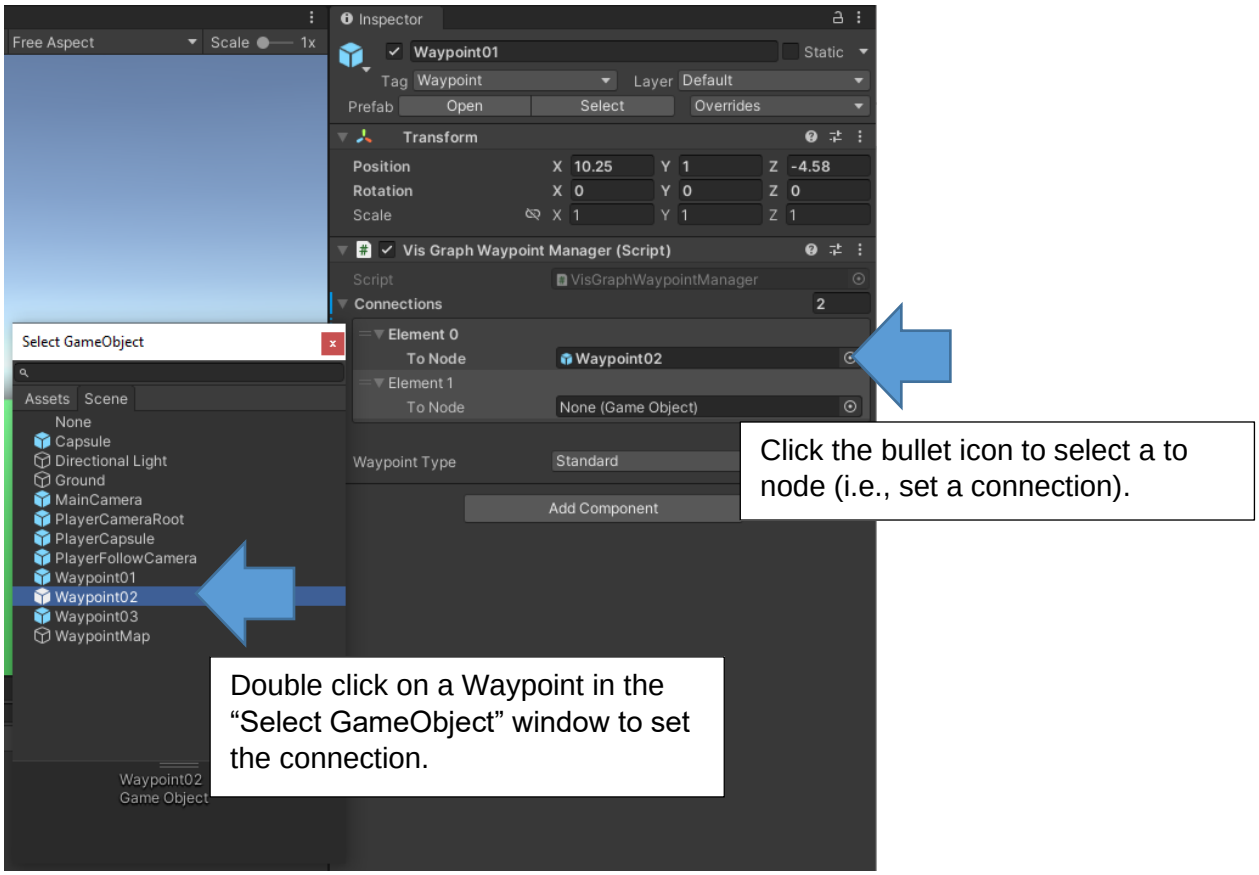

Image description: *I have selected Waypoint01 in the scene. I have gone to the inspector and found the VisGraphWaypointManager script component. I have added two connections. I have clicked on the bullet icon next to “Element 0”. A “Select GameObject” window has appeared. I then double click on Waypoint02 to set a link between Waypoint01 and Waypoint02.*

Repeat the above process to set a connection between Waypoint01 and Waypoint03 for <u>Element 1</u>.

The VisGraphWaypointManager script component should now have two connections / elements and have two GameObjects set for these connections / elements. See the screenshot below for an example.

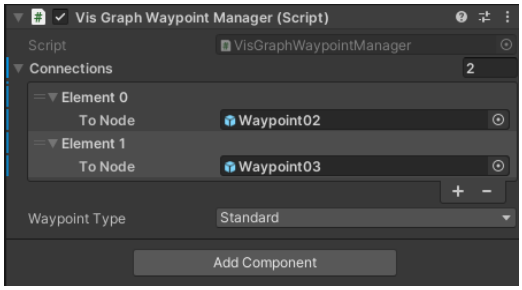

Image description: *The VisGraphWaypointManager script component in the inspector window. The two connections / elements and have two GameObjects set.*

When we set GameObjects for VisGraphWaypointManager script connections / elements line is drawn in the scene to visualise the created connection. **If the waypoint is select the arrowhead is magenta, and three spheres are draw along the first half of the connection**. If it is not selected the arrowhead is blue and there are no spheres along the line.

The arrowhead and three spheres when selected are drawn to indicate the direction of the connection. **If there is no arrowhead (or three spheres when selected) in a direction it means, there is no outgoing connection for that direction. You may need to add one if needed.**

See the screenshot below for an example.

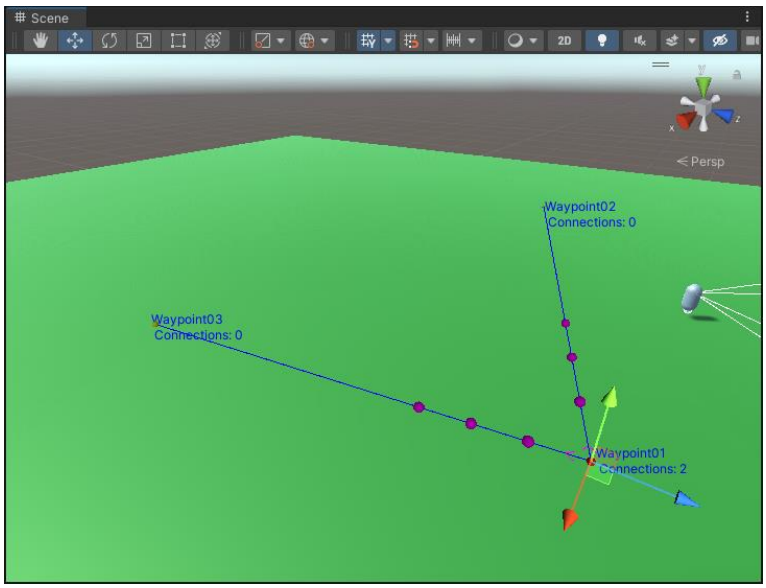

Image description: *I have selected Waypoint01. The outgoing connections arrowheads are drawn in magenta. An arrowhead is drawn indicating the direction of the connection.*

As we can see from the screenshot above, there are arrowheads pointing from Waypoint01 to Waypoint02 and Waypoint03. However, there are no arrowheads pointing from Waypoint02 and Waypoint03 to Waypoint01. This means there are connections to Waypoint02 and Waypoint03 from Waypoint01. However, there are no

connections to Waypoint01 from Waypoint02 and Waypoint03. In this situation we probably want connections in both directions; therefore, we will add them.

To do this, select the **Waypoint02** node and go to the inspector.

Go to the VisGraphWaypointManager script and add two connections. One to Waypoint01 and one to Waypoint03.

Then, select the **Waypoint03** node and go to the inspector.

Go to the VisGraphWaypointManager script and add two connections. One to Waypoint01 and one to Waypoint02.

You should now have a scene where the three waypoints are connected to each other. See the screenshot below for an example.

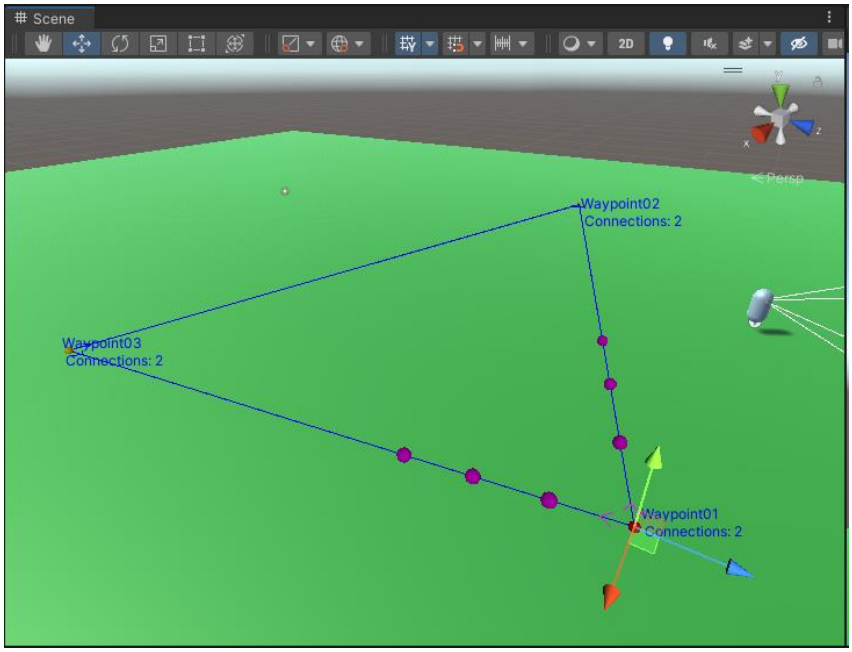

Image description: *I have selected Waypoint01. The outgoing connections arrowheads are drawn in magenta. Three spheres are draw along the first half of the connection when a node is selected. An arrowhead is drawn indicating the direction of the connection. Each waypoint node now has a connection to the two other waypoint nodes. This is indicated by the arrowheads on the connections and spheres along the first half of the connection when a node is selected.*

Add 7 more nodes to the scene and add connections between the nodes.

You only need to add connections to adjacent nodes (e.g., nodes next to each other that have line of sight.

Remember, you can drag your waypoint nodes around the scene view using the move tool.

<u>Your waypoint graph could look like the screenshot below</u> when you have added 10 or more nodes to your scene and have set the connections so that your nodes form a waypoint graph.

I have added 12 nodes and set the connections for each of the nodes.

<u>Your scene should now look like the screenshot below.</u>

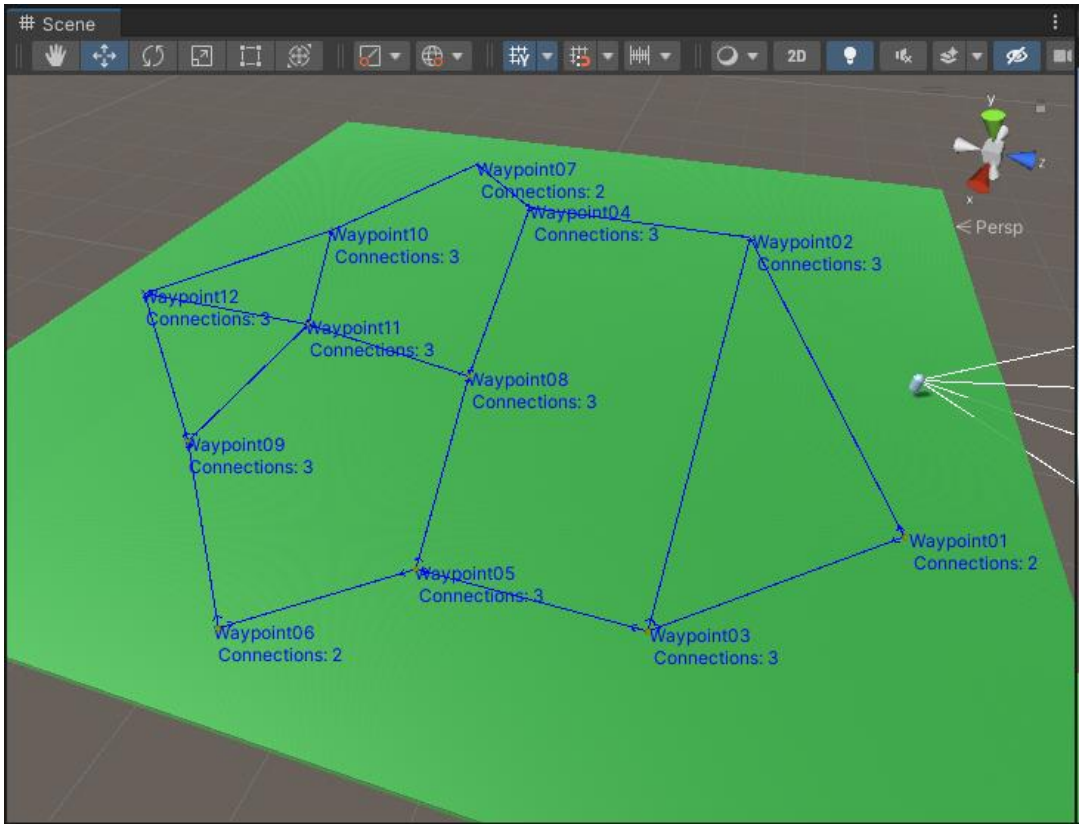

Image description: *The scene window. I have added 12 nodes and set the connections for each node.*

Save your 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 be able to see the waypoint graph in the game window.

<u>If it does not work, please review the previous steps and your code.</u>

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.

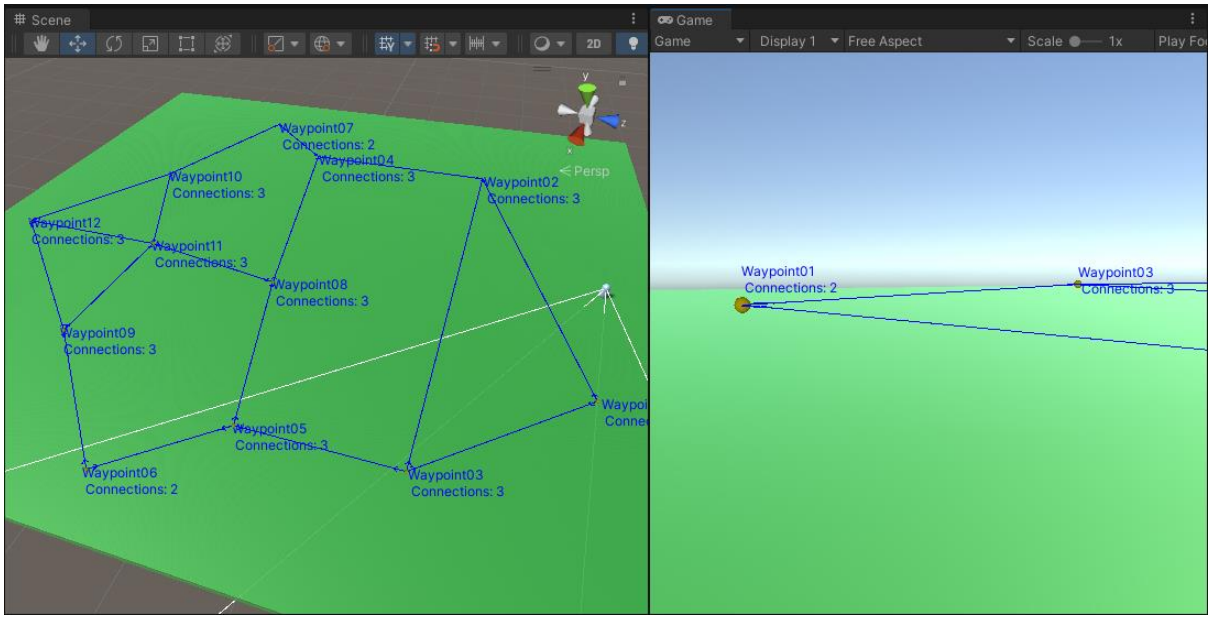

Image description: *A playtest of the scene using the first-person controller. Note, in my setup I have the Scene and Game view side-by-side.*

You might find that you do not see your waypoint graph! <u>What is the issue?</u>

If you do not see your waypoint graph, it is probably because displaying Gizmos are switch off. You need to switch it on.

In the game window make sure the Gizmos button is click. The Gizmo button is at the top right of the game window. See the screenshot below for an example.

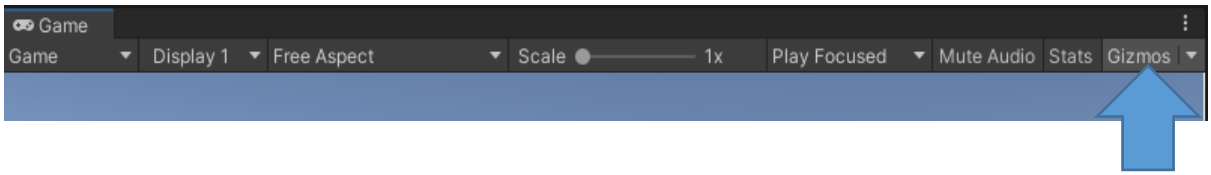

Image description: *If you do not see your waypoint map in game view, make sure the Gizmos button is clicked.*

You should also see the Gizmos in scene view. Again, you will not see the Gizmos in scene view if the Gizmos button is not clicked. Make sure it is clicked. The Gizmo button is at the top right of the scene view. See the screenshot below for an example.

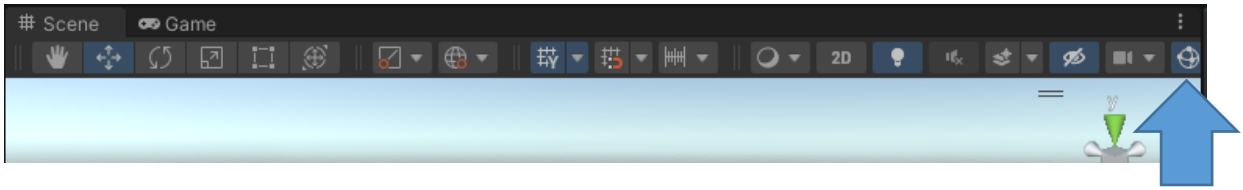

Image description: *If you do not see your waypoint map in scene view, make sure the Gizmos button is clicked.*

When you have finished walking around stop playtesting your scene.

If you do not see a waypoint map, please review your code and Unity editor work.

In the next sections we will create the classes needed to implement the A* algorithm.

8. A* Pathfinding Classes - Connections

Next, we will add a new C# class to the project. This class will represent connections in our waypoint map. The A* algorithm will use the connections class to determine routes through the waypoint map.

This connections class is different to the previous connections class, which is only used for waypoint map visualisation purposes.

Go to the Scripts folder in the project window.

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 **Connection**.

Update the code in the script file to match the code below. This class does not need to use **MonoBehaviour** so the inheritance has been removed.

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

public class Connection
{
    private float cost = 0;
    public float Cost
    {
        get
        {
            if (cost == 0)
            {
                cost = Vector3.Distance(FromNode.transform.position,
ToNode.transform.position);
            }
            return cost;
        }
        set { cost = value;  }
    }

    private GameObject fromNode;
    public GameObject FromNode
    {
        get { return fromNode; }
        set
        {
            fromNode = value;
            cost = 0;
        }
    }
```

```
    private GameObject toNode;
    public GameObject ToNode
    {
        get { return toNode; }
        set
        {
            toNode = value;
            cost = 0;
        }
    }

    // Default constructor.
    public Connection()
    {

    }

}
```

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

9. A* Pathfinding Classes - Graph

Next, we will add a new C# class to the project. This class will represent our waypoint graph. The A* algorithm will use the graph to determine connections from waypoint nodes.

Go to the Scripts folder in the Project Window.

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 **Graph**.

Update the code in the script file to match the code below. This class does not need to use **MonoBehaviour** so the inheritance has been removed.

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

public class Graph
{
    // A list of graph connections.
    private List<Connection> WaypointConnections = new List<Connection>();

    public Graph()
    {

    }

    // Add connection.
    public void AddConnection(Connection aConnection)
    {
        WaypointConnections.Add(aConnection);
    }

    // Get the connections from a node to the nodes it is connected to.
    public List<Connection> GetConnections(GameObject FromNode)
    {
        List<Connection> TmpConnections = new List<Connection>();

        foreach (Connection aConnection in WaypointConnections)
        {
            if(aConnection.FromNode.Equals(FromNode))
            {
                TmpConnections.Add(aConnection);
            }
        }

        return TmpConnections;
    }

}
```

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

10. A* Pathfinding Classes - NodeRecord

Next, we will add a new C# class to the project. This class will represent a NodeRecord in our waypoint map. The A* algorithm will use the NodeRecord to keep track of a nodes code-so-far and estimated-total-cost.

Go to the Scripts folder in the Project Window.

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 **NodeRecord**.

Update the code in the script file to match the code below. This class does not need to use **MonoBehaviour** so the inheritance has been removed.

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

public class NodeRecord
{
    private GameObject node;
    public GameObject Node
    {
        get { return node; }
        set { node = value; }
    }

    private Connection connection;
    public Connection Connection
    {
        get { return connection; }
        set { connection = value; }
    }

    private float costSoFar;
    public float CostSoFar
    {
        get { return costSoFar; }
        set { costSoFar = value; }
    }

    private float estimatedTotalCost;
    public float EstimatedTotalCost
    {
        get { return estimatedTotalCost; }
        set { estimatedTotalCost = value; }
    }

    public NodeRecord()
    {

    }

}
```

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

11. A* Pathfinding Classes - PathfindingList

Next, we will add a new C# class to the project. This class will represent a list (e.g. open and closed list) in our A* algorithm. The A* algorithm will use the PathfindingList to keep track of which nodes have been visited by the algorithm.

Go to the Scripts folder in the Project Window.

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 **PathfindingList**.

Update the code in the script file to match the code below. This class does not need to use **MonoBehaviour** so the inheritance has been removed.

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

public class PathfindingList
{
    private List<NodeRecord> NodeRecordList = new List<NodeRecord>();

    public PathfindingList()
    {

    }

    // Add NodeRecord.
    public void AddNodeRecord(NodeRecord NodeRecord)
    {
        NodeRecordList.Add(NodeRecord);
    }

    // Remove a node from the list.
    public void RemoveNodeRecord(NodeRecord NodeRecord)
    {
        NodeRecordList.Remove(NodeRecord);
    }

    // Get the size of the list.
    public int GetSize()
    {
        return NodeRecordList.Count;
    }
```

```
   // Get the smallest element.
    public NodeRecord GetSmallestElement()
    {
        NodeRecord TmpNodeRecord = new NodeRecord();
        TmpNodeRecord.EstimatedTotalCost = float.MaxValue;

        foreach (NodeRecord NodeRecord in NodeRecordList)
        {
            if(NodeRecord.EstimatedTotalCost < TmpNodeRecord.EstimatedTotalCost)
            {
                TmpNodeRecord = NodeRecord;
            }
        }

        return TmpNodeRecord;
    }

    // Returns true if a node is contained in the list.
    public bool Contains(GameObject Node)
    {
        foreach (NodeRecord NodeRecord in NodeRecordList)
        {
            if (NodeRecord.Node.Equals(Node))
            {
                return true;
            }
        }

        return false;
    }

    // Returns a node record for a node if it is contained in the list.
    public NodeRecord Find(GameObject Node)
    {
        foreach (NodeRecord NodeRecord in NodeRecordList)
        {
            if (NodeRecord.Node.Equals(Node))
            {
                return NodeRecord;
            }
        }

        return null;
    }

}
```

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

12. A* Pathfinding Classes - Heuristic

Next, we will add a new C# class to the project. This class will store our heuristic function. The Heuristic function will determine the straight-line distance between two nodes.

Go to the Scripts folder in the Project Window.

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 **Heuristic**.

Update the code in the script file to match the code below. This class does not need to use **MonoBehaviour** so the inheritance has been removed.

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

public class Heuristic
{

    public Heuristic()
    {

    }

    public float Estimate(GameObject StartNode, GameObject GoalNode)
    {
        return Vector3.Distance(StartNode.transform.position,
GoalNode.transform.position);
    }

}
```

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

13. A* Pathfinding Classes - AStar

Next, we will add a new C# class to the project. This class will store our A* algorithm. The A* algorithm will use the other classes we have created to implement the standard A* algorithm.

Go to the Scripts folder in the Project Window.

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 **AStar**.

Update the code in the script file to match the code below. This class does not need to use **MonoBehaviour** so the inheritance has been removed.

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

public class AStar
{

    public AStar()
    {

    }

    public List<Connection> PathfindAStar(Graph aGraph, GameObject start, GameObject end,
Heuristic myHeuristic)
    {
        // Set up the start record.
        NodeRecord StartRecord = new NodeRecord();
        StartRecord.Node = start;
        StartRecord.Connection = null;
        StartRecord.CostSoFar = 0;
        StartRecord.EstimatedTotalCost = myHeuristic.Estimate(start, end);

        // Create the lists.
        PathfindingList OpenList = new PathfindingList();
        PathfindingList ClosedList = new PathfindingList();

        // Add the start record to the open list.
        OpenList.AddNodeRecord(StartRecord);

        // Iterate through and process each node.
        NodeRecord CurrentRecord = null;
        List<Connection> Connections;

        while (OpenList.GetSize() > 0)
        {
            // Find the smallest element in the open list (using the estimatedTotalCost).
            CurrentRecord = OpenList.GetSmallestElement();

            // If it is the goal node, then terminate.
            if (CurrentRecord.Node.Equals(end))
            {
                break;
            }
```

```
            // Otherwise get its outgoing connections.
            Connections = aGraph.GetConnections(CurrentRecord.Node);

            // Loop through each connection in turn.
            GameObject EndNode;
            float EndNodeCost;
            NodeRecord EndNodeRecord;
            float EndNodeHeuristic;

            foreach (Connection aConnection in Connections)
            {
                // Get the cost estimate for the end node.
                EndNode = aConnection.ToNode;
                EndNodeCost = CurrentRecord.CostSoFar + aConnection.Cost;

                // If the node is closed we may have to skip, or remove it from the closed list.
                if (ClosedList.Contains(EndNode))
                {
                    // Here we find the record in the closed list corresponding to the endNode.
                    EndNodeRecord = ClosedList.Find(EndNode);

                    // If we didn't find a shorter route, skip.
                    if (EndNodeRecord.CostSoFar <= EndNodeCost)
                    {
                        continue;
                    }

                    // Otherwise remove it from the closed list.
                    ClosedList.RemoveNodeRecord(EndNodeRecord);

                    // We can use the node's old cost values to calculate its heuristic without calling
                    // the possibly expensive heuristic function.
                    EndNodeHeuristic = EndNodeRecord.EstimatedTotalCost - EndNodeRecord.CostSoFar;
                }
                // Skip if the node is open and we've not found a better route.
                else if (OpenList.Contains(EndNode))
                {
                    // Here we find the record in the open list corresponding to the endNode.
                    EndNodeRecord = OpenList.Find(EndNode);

                    // If our route is no better, then skip.
                    if (EndNodeRecord.CostSoFar <= EndNodeCost)
                    {
                        continue;
                    }

                    // We can use the node's old cost values to calculate its heuristic without calling
                    // the possibly expensive heuristic function.
                    EndNodeHeuristic = EndNodeRecord.EstimatedTotalCost - EndNodeRecord.CostSoFar;
                }
                // Otherwise we know we've got an unvisited node, so make a record for it.
                else
                {
                    EndNodeRecord = new NodeRecord();
                    EndNodeRecord.Node = EndNode;

                    // We'll need to calculate the heuristic value using the function, since we don't have
an existing record to use.
                    EndNodeHeuristic = myHeuristic.Estimate(EndNode, end);
                }
```

```
                // We're here if we need to update the node Update the cost, estimate and connection.
                EndNodeRecord.CostSoFar = EndNodeCost;
                EndNodeRecord.Connection = aConnection;
                EndNodeRecord.EstimatedTotalCost = EndNodeCost + EndNodeHeuristic;

                // And add it to the open list.
                if (!(OpenList.Contains(EndNode)))
                {
                    OpenList.AddNodeRecord(EndNodeRecord);
                }

            } //#END: Looping through Connections.

            // We've finished looking at the connections for the current node, so add it to the closed
list
            // and remove it from the open list
            OpenList.RemoveNodeRecord(CurrentRecord);
            ClosedList.AddNodeRecord(CurrentRecord);
        }

        // We're here if we've either found the goal, or if we've no more nodes to search, find which.
        List<Connection> tempList = new List<Connection>();
        if (!CurrentRecord.Node.Equals(end))
        {
            // We've run out of nodes without finding the goal, so there's no solution
            return tempList;
        }
        else
        {
            while (!CurrentRecord.Node.Equals(start))
            {
                tempList.Add(CurrentRecord.Connection);

                CurrentRecord = ClosedList.Find(CurrentRecord.Connection.FromNode);
            }

            // The path is in the wrong order. Reverse the path, and return it.
            List<Connection> tempList2 = new List<Connection>();
            for (int i = (tempList.Count - 1); i >= 0; i--)
            {
                tempList2.Add(tempList[i]);
            }

            return tempList2;
        }

    }

}
```

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

14. A* Pathfinding Classes - AStarManager

Next, we will add a new C# class to the project. This class will provide a simple interface for our A* algorithm and accompanying classes. You do not have to implement this class to use the A* algorithm; however, it provides a simple way to group things together, so we will use it in this example.

Go to the Scripts folder in the Project Window.

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 **AStarManager**.

Update the code in the script file to match the code below. This class does not need to use **MonoBehaviour** so the inheritance has been removed.

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

public class AStarManager
{
    // The a star algorithm.
    private AStar AStar = new AStar();

    // The waypoint graph.
    private Graph aGraph = new Graph();

    // The Heuristic.
    private Heuristic aHeuristic = new Heuristic();

    public AStarManager()
    {

    }

    // Add Connection.
    public void AddConnection(Connection connection)
    {
        aGraph.AddConnection(connection);
    }

    // Find path.
    public List<Connection> PathfindAStar(GameObject start, GameObject end)
    {
        return AStar.PathfindAStar(aGraph, start, end, aHeuristic);
    }

}
```

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

15. A* Pathfinding Classes - PathfindingTester

Next, we will add a new C# class to the project. This class will provide a simple interface for our A* algorithm and accompanying classes. You do not have to implement this class to use the A* algorithm; however, it provides a simple way to demonstrate the algorithm, so we will use it in this example.

Go to the Scripts folder in the Project Window.

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 **PathfindingTester**.

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

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

public class PathfindingTester : MonoBehaviour
{
    // The A* manager.
    private AStarManager AStarManager = new AStarManager();

    // List of possible waypoints.
    private List<GameObject> Waypoints = new List<GameObject>();

    // List of waypoint map connections. Represents a path.
    private List<Connection> ConnectionArray = new List<Connection>();

    // The start and end nodes.
    [SerializeField]
    private GameObject start;
    [SerializeField]
    private GameObject end;

    // Debug line offset.
    Vector3 OffSet = new Vector3(0, 0.3f, 0);

    // Start is called before the first frame update
    void Start()
    {
        if (start == null || end == null)
        {
            Debug.Log("No start or end waypoints.");
            return;
        }
        VisGraphWaypointManager tmpWpM = start.GetComponent<VisGraphWaypointManager>();
        if (tmpWpM == null)
        {
            Debug.Log("Start is not a waypoint.");
            return;
        }
        tmpWpM = end.GetComponent<VisGraphWaypointManager>();
        if (tmpWpM == null)
        {
            Debug.Log("End is not a waypoint.");
            return;
        }
```

```
        // Find all the waypoints in the level.
        GameObject[] GameObjectsWithWaypointTag;
        GameObjectsWithWaypointTag = GameObject.FindGameObjectsWithTag("Waypoint");

        foreach (GameObject waypoint in GameObjectsWithWaypointTag)
        {
            VisGraphWaypointManager tmpWaypointMan = waypoint.GetComponent<VisGraphWaypointManager>();
            if (tmpWaypointMan)
            {
                Waypoints.Add(waypoint);
            }
        }

        // Go through the waypoints and create connections.
        foreach (GameObject waypoint in Waypoints)
        {
            VisGraphWaypointManager tmpWaypointMan = waypoint.GetComponent<VisGraphWaypointManager>();
            // Loop through a waypoints connections.
            foreach (VisGraphConnection aVisGraphConnection in tmpWaypointMan.Connections)
            {
                if (aVisGraphConnection.ToNode != null)
                {
                    Connection aConnection = new Connection();
                    aConnection.FromNode = waypoint;
                    aConnection.ToNode = aVisGraphConnection.ToNode;

                    AStarManager.AddConnection(aConnection);
                }
                else
                {
                    Debug.Log("Warning, " + waypoint.name + " has a missing to node for a connection!");
                }
            }
        }

        // Run A Star...
        // ConnectionArray stores all the connections in the route to the goal / end node.
        ConnectionArray = AStarManager.PathfindAStar(start, end);
        if(ConnectionArray.Count == 0)
        {
            Debug.Log("Warning, A* did not return a path between the start and end node.");
        }
    }

    // Draws debug objects in the editor and during editor play (if option set).
    void OnDrawGizmos()
    {
        // Draw path.
        foreach (Connection aConnection in ConnectionArray)
        {
            Gizmos.color = Color.white;
            Gizmos.DrawLine((aConnection.FromNode.transform.position + OffSet),
(aConnection.ToNode.transform.position + OffSet));
        }
    }

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

    }

}
```

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

16. Adding the PathfindingTester to Our Scene and Testing the A* Algorithm

Once you have completed your **PathfindingTester** script / class we can test our A* algorithm. To test the A* algorithm we just need to add the **PathfindingTester** script to a GameObject in the scene.

We will add a vehicle asset and attach the **PathfindingTester** script to it.

In the Unity Editor, go to the **project window**.

Go to the folder: PolygonStarter -> Prefabs.

Select the **SM_PolygonCity_Veh_Car_Small_01** asset and drag it into the scene. Drag the vehicle onto the ground. Position the vehicle so that it is next to the waypoint node you would like your path to start at. See the screenshot below for an example.

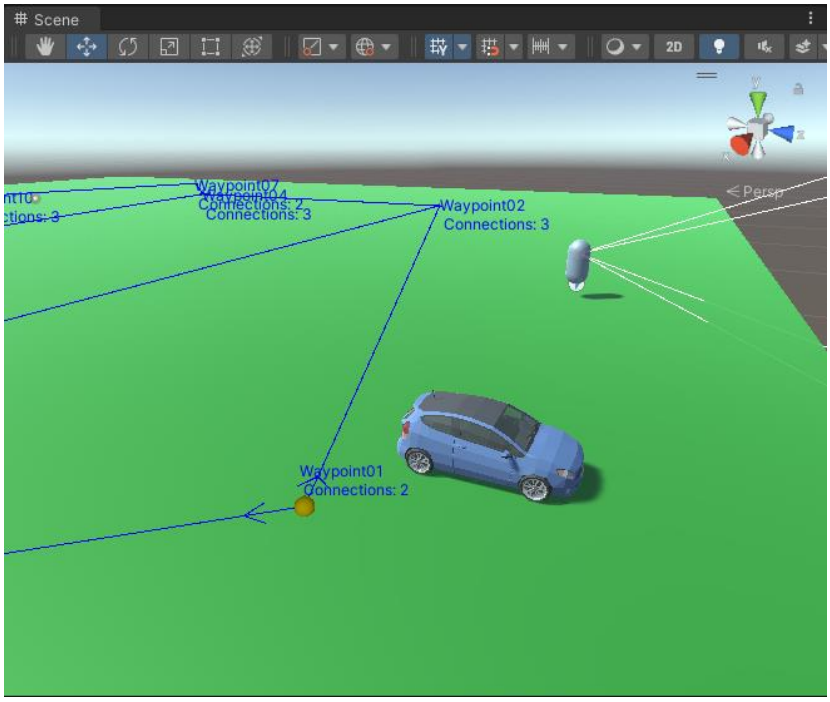

Image description: *A vehicle has been placed in the scene. I have positioned the vehicle next to Waypoint01. This will be my starting waypoint.*

We will now attach the script to our vehicle GameObject.

Select the vehicle object in the hierarchy. Go to the inspector. In the inspector click the Add Component button, which is at the bottom of the window. Search for the **PathfindingTester** script and then double click on it to add it.

When you have added the script, the inspector for the vehicle should look like the screenshot below.

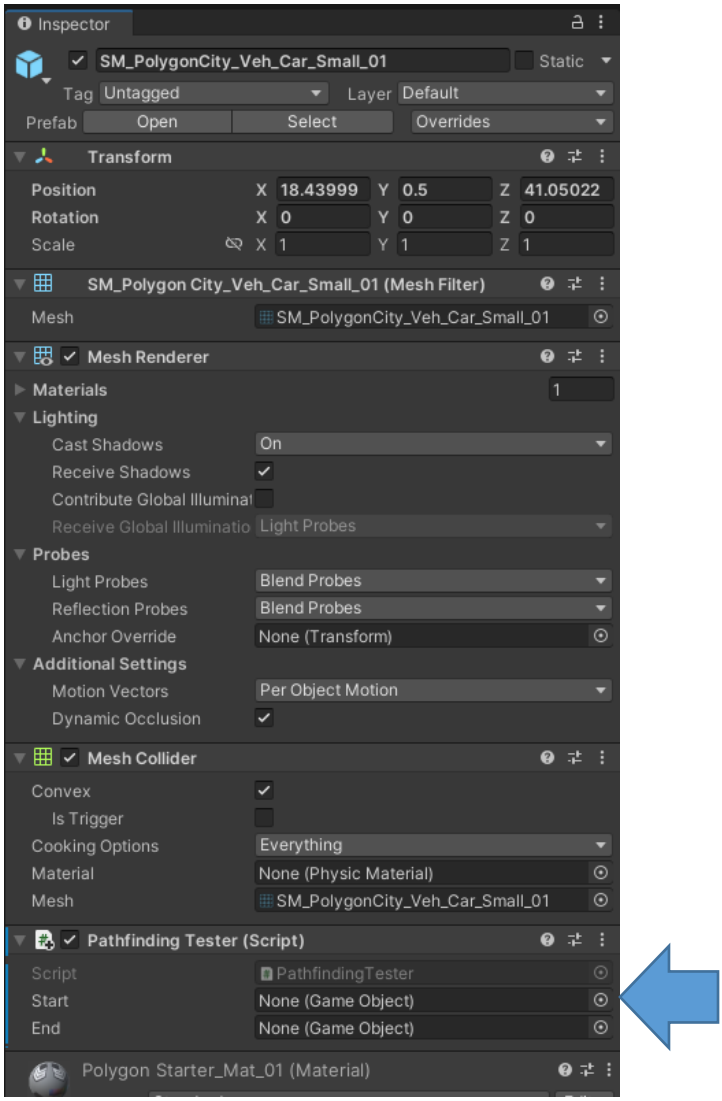

Image description: *The inspector properties of the vehicle after the PathfindingTester script has been added to it.*

Finally, we need to set a start and end node for the A* algorithm.

Click the bullet icon next to the Start and End properties in the **PathfindingTester** script component of the vehicle GameObject. Select a start and end waypoint from the scene. See the screenshot below for an example.

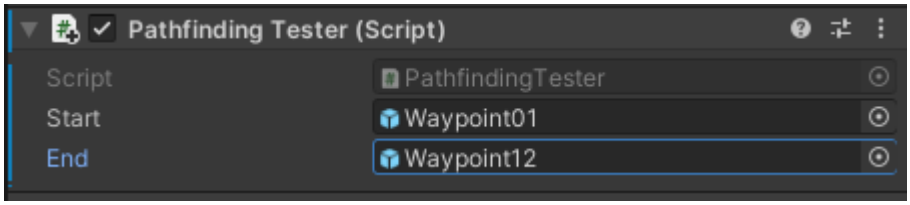

Image description: *The PathfindingTester script component of the vehicle GameObject. The Start waypoint node has been set to Waypoint01 and the end Waypoint node has been set to Waypoint12.*

Save your 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 be able to see the waypoint graph in the game window. You should be able to see a while line. This represents the path generated by the A* algorithm.

<u>If it does not work, please review the previous steps and your code.</u>

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.

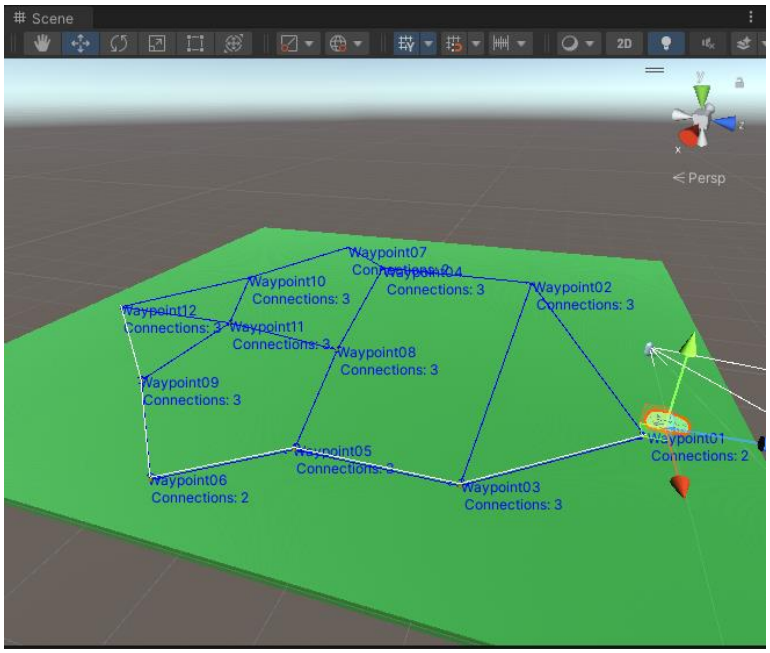

Image description: *A playtest of the scene. The while line represents the path generated by the A* algorithm.*

When you have finished walking around stop playtesting your scene.

🚀 **Further Tasks:**

1. Experiment with the A* implementation. You could start by moving the waypoint nodes around and seeing if it affects the route generated by the A* algorithm.

2. Add two more waypoint nodes to your map. See what affect is has on the route generated by the A* algorithm. Also, change the start and end nodes.

17. Moving the Vehicle - Adding Code to the PathfindingTester Update Method

So far, the A* algorithm generates a path through the waypoint map; however, it the PathfindingTester does not move the vehicle along the path. Let's do that now.

Go to the project window and the Scripts folder.

Open the **PathfindingTester** script.

Add the following class level variables near the top of the **PathfindingTester** script. Add this code just below the line "Vector3 OffSet = new Vector3(0, 0.3f, 0);".

```
    // Movement variables.
    private float currentSpeed = 8;
    private int currentTarget = 0;
    private Vector3 currentTargetPos;
    private int moveDirection = 1;
    private bool agentMove = true;
```

Next, add the following code to the **Update** method in **PathfindingTester** script.

```
    // Update is called once per frame
    void Update()
    {
        if (agentMove)
        {
            // Determine the direction to first node in the array.
            if (moveDirection > 0)
            {
                currentTargetPos = ConnectionArray[currentTarget].ToNode.transform.position;
            }
            else
            {
                currentTargetPos = ConnectionArray[currentTarget].FromNode.transform.position;
            }

            // Clear y to avoid up/down movement. Assumes flat surface.
            currentTargetPos.y = transform.position.y;

            Vector3 direction = currentTargetPos - transform.position;

            // Calculate the length of the relative position vector
            float distance = direction.magnitude;

            // Face in the right direction.
            direction.y = 0;
            if (direction.magnitude > 0)
            {
                Quaternion rotation = Quaternion.LookRotation(direction, Vector3.up);
                transform.rotation = rotation;
            }

            // Calculate the normalised direction to the target from a game object.
            Vector3 normDirection = direction / distance;

            // Move the game object.
            transform.position = transform.position + normDirection * currentSpeed * Time.deltaTime;

            // Check if close to current target.
            if (distance < 1)
            {
                 currentTarget += moveDirection;

                 if(currentTarget == ConnectionArray.Count)
                 {
                     moveDirection = -1;
                     currentTarget += moveDirection;
                 }

                 if (currentTarget < 0)
                 {
                     moveDirection = 1;
                     currentTarget += moveDirection;
                 }
            }
        }
        else
        {

        }
    }
```

The code above includes class level variables to set the agents speed, the position in the A* path array that it needs to move towards, the current position it needs to move towards, its movement direction in the array (1 for up the array and -1 for down the array), and a flag to determine if the agent should run its move code or not (true for yes, false for no).

The code in the Update methods gets a position to move towards (a target position) from the path list generated by the A* algorithm. It then clears y (sets it to the agents y position) to avoid up/down movement. It then determines the direction to the target position and the distance. It then rotates the agent to look in the direction it is moving. It then normalised the direction. By doing this we keep the direction of the vector but remove its length. The normalised direction is then used to move the agent towards the target position.

The last parts of the algorithm check if the agent is close to the current target position. If the agent is close, we move the to the next target. We also check if we have reached an end of the list. If we have, we reverse the agent's direction of movement.

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

Save your scene.

You should have positioned the vehicle next to the start node. If you have not done that, do it now.

🎮 **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 be able to see the waypoint graph in the game window. You should be able to see a while line. This represents the path generated by the A* algorithm. The vehicle should move to the first waypoint node in the path.

<u>If it does not work, please review the previous steps and your code.</u>

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.

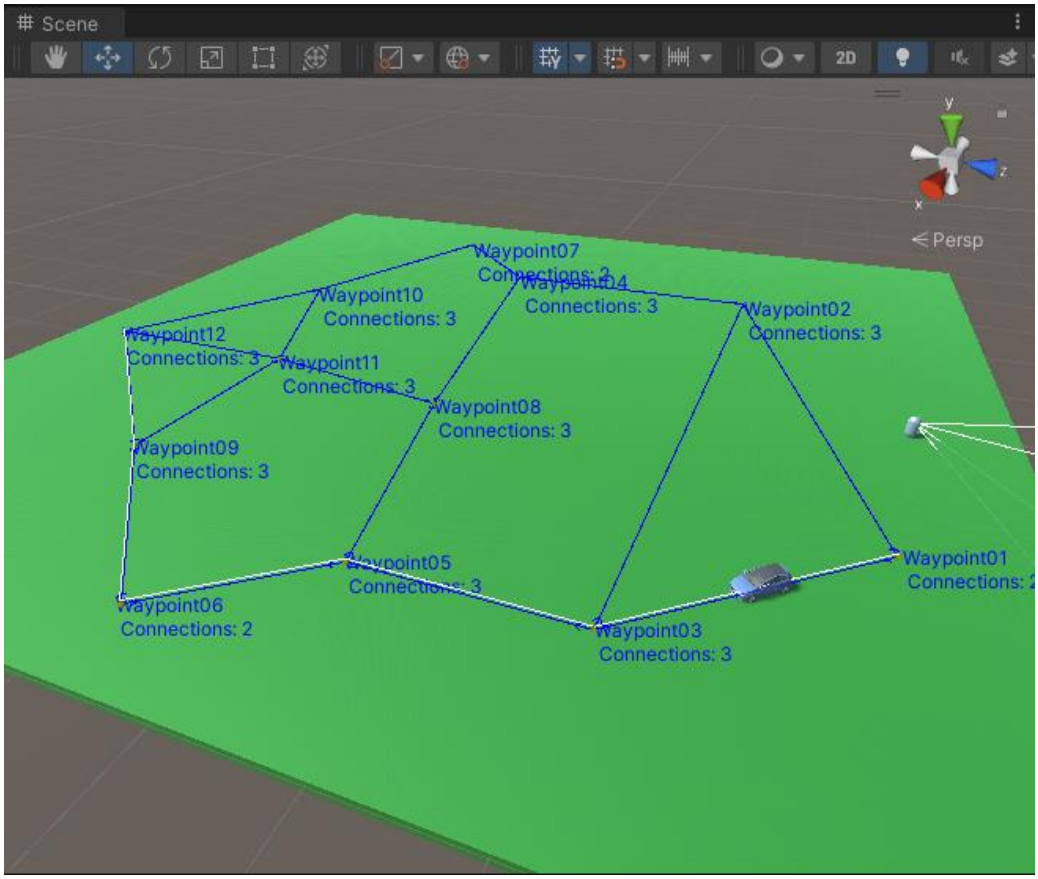

Image description: *A playtest of the scene. The while line represents the path generated by the A* algorithm. The vehicle is moving from the first waypoint in the path to the second waypoint in the path.*

11. Further Tasks

⚠ **ALERT - Useful Tip** ⚠ **These tasks and any tasks beyond this point can be considered optional if you are running behind with things (e.g., it has taken you over a week to complete this workbook). If you are behind, I would recommend you stop this workbook here and move onto the next workbook. If you are not behind, I recommend you complete the remaining tasks as they will help improve your knowledge. Please speak to a member of the module team if you have any questions.**

🚀 **Please complete these further tasks:**

1. Update the code in the PathfindingTester script update method so that the vehicle moves up and down the whole A* path.

2. Experiment with the currentSpeed value in the PathfindingTester script.

3. Experiment with different end waypoint nodes for the vehicle.

4. Add assets from the Assets -> PolygonStarter -> Prefabs assets to free space in your waypoint map. You should not block the waypoint connections. The connections represent accessible paths between nodes. The vehicle uses these connections to move through the waypoint map. See the screenshot below for an example. If you are struggling to fit an object within your waypoint map, you can also use the move tool to adjust the position of the waypoint nodes.

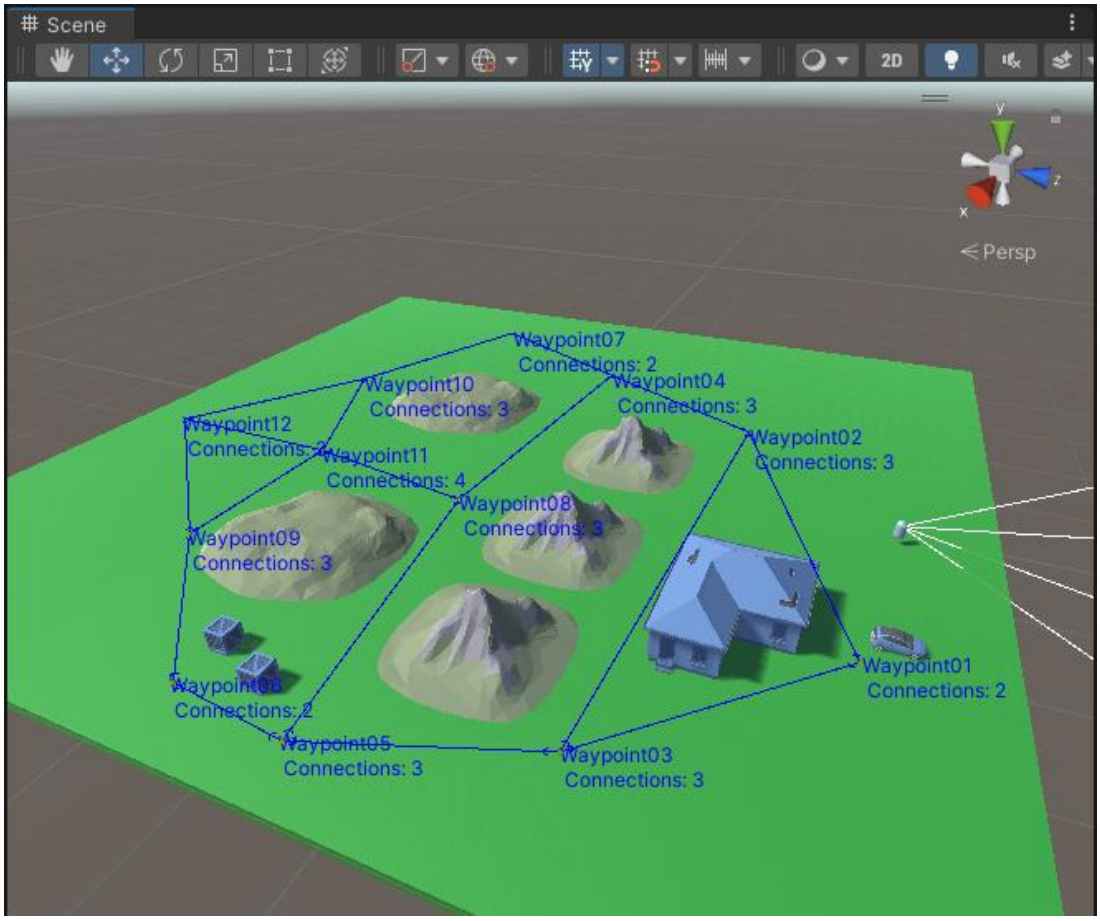

Image description: *The scene window. I have added a house, some hills and two crates to the scene. You should build a scene like this.*

5. Add a bean character from the Assets -> PolygonStarter -> Prefabs -> Characters folder. Add the PathfindingTester script to it and set a new start and end node. The PathfindingTester script should now move the bean character.

> END OF STUDENT WORKBOOK