Unity for Midlifes

JANUARY 07, 2021

- Learning Objectives: Pathfinding Using Navmesh
- Install NavMeshPlus and NavMeshComponents
- Basics of Navigation System
  - NavMesh Generation
  - NavMesh Obstacle
  - Traversing Between NavMeshSurfaces
    - OffMeshLinks
    - NavMeshLinks
  - NavMeshSurface
- NavMesh for 2D Games
  - Create a TileMap
  - Create NavMeshSurface2D and NavMeshModifier
  - Generate NavMeshSurface2D from Polygon
    - NavMeshLink
- NavMesh Data
- Finding “Click” Location in the World Coordinates
  - 2D Click Location
  - 3D Click Location
- Navigating Between Waypoints
- Next

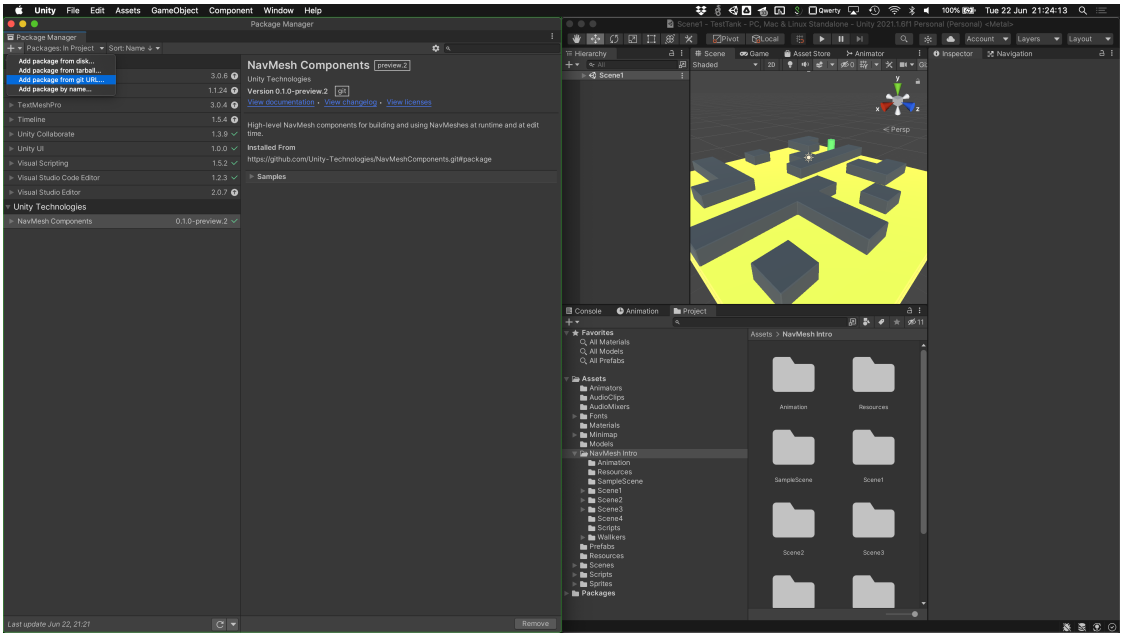
Learning Objectives: Pathfinding Using Navmesh

- **NavMesh Basics:**
  - Creating a click-to-move game
  - Connecting navmeshes together
  - Trigger animation using Offmeshlink
  - Determining area and costs
- Using **NavmeshComponents**
  - Install navmeshcomponents package
  - Creating Navmeshlinks
  - Creating Navmesh agents and baking Navmesh surfaces
  - Orient agents on walls
  - Generating Navmesh obstacles at runtime
- Using **NavMeshPlus** to create NavMeshSurface on 2D Sprites or TileMap

Install NavMeshPlus and NavMeshComponents

Navigation System is a part of Unity.AI module, and is extremely useful to perform pathfinding in pre-determined (baked) maps automatically. To learn about this, **create a new 3D project** in Unity, and **import** the `navmesh3dintro` asset given in the course handout. This asset requires the `NavMeshComponents` package. Unity already has built-in Navigation System but this package has additional functionalities that will make your life way easier.

Go to Window » Package Manager as usual, and click *add package from git URL*:



You can then paste the following URL for NavMeshComponents only:
`https://github.com/Unity-Technologies/NavMeshComponents.git#package`
When the import is done, there should be no more error messages in the Console.
The manual for this package can be found here.

Basics of Navigation System

NavMesh Generation

To allow automatic pathfinding, you need to first generate (*bake*) a **NavMesh** from all objects in your scene.

> From Unity Documentation: **NavMesh** (short for Navigation Mesh) is a data structure which describes the walkable surfaces of the game world and allows to find path from one walkable location to another in the game world. The data structure is built, or baked, automatically from your level geometry.

Unity's built in NavMesh will **automatically** detect all 3D objects (mesh) that lie on the **XZ** plane.

> We can create NavMesh on surfaces lying on other planes, such as the common XY plane in 2D games using `NavMeshComponents's Surface` (later).

We need to also use **3D meshes** (mesh renderer) in order for Unity to *bake* the NavMesh, so 2D colliders can't be used to define "*walkable areas*".

> There's plenty of workarounds that you can find online, or paid assets to help you.

The rest of this tutorial will just introduce you how to use standard 3D Navigation System using Mesh Renderer (as per Unity base implementation). The 2D workarounds are built upon this knowledge.

Import the asset from the Course Handout, then **open** the folder NavMeshIntro » Scene1 and open the Scene Scene1.

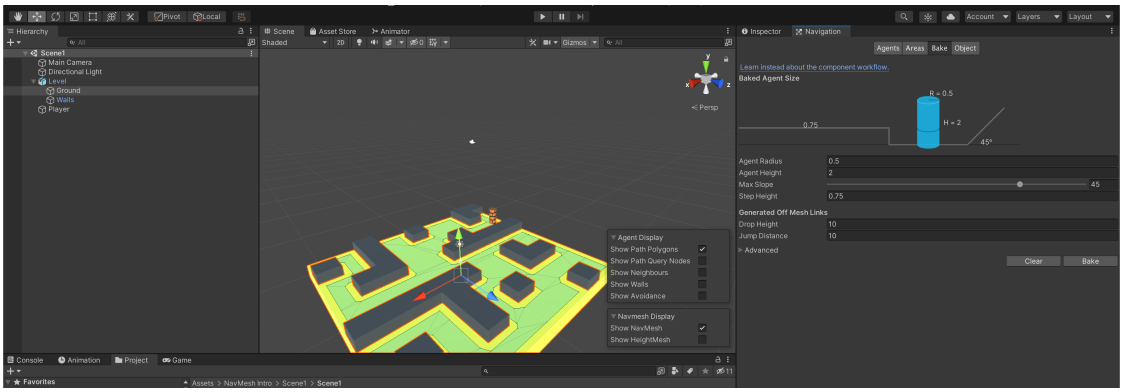
Add the Navigation Window from Window » AI » Navigation. There's four tabs in the Navigation Window:

- **Agents:** You can define NavMesh Agents properties such as its step height (agents cannot "go over" steps more than this height), and max slope (agents cannot climb anything steeper than this slope)
- **Areas**: Similar to "layers", you can name different areas and its cost (second column). The cost will be used for pathfinding computation later on. Areas with lower cost is always more preferable. More about Navigation Area and cost here.
- **Bake**: This is where you generate the NavMesh with the respective agent size. You need to bake different NavMesh per agent.

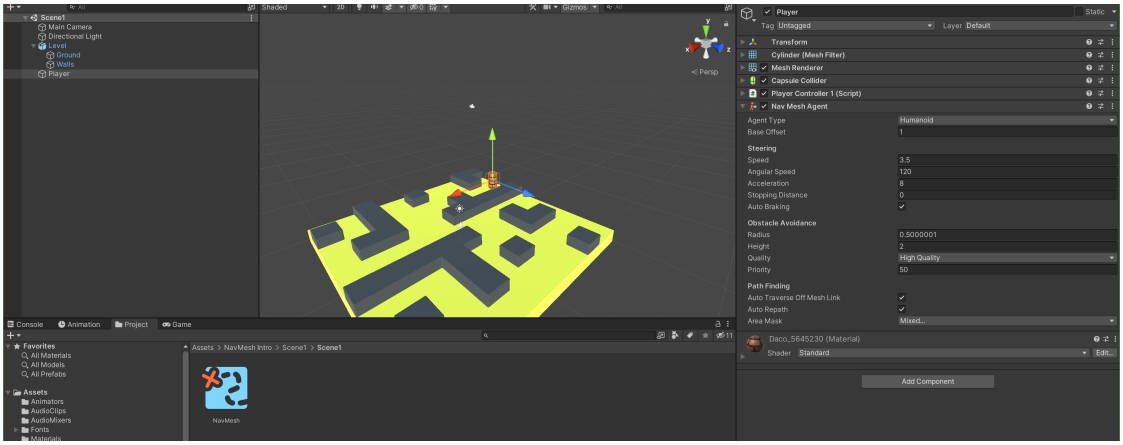
- **Object**: You can define the properties about the Object you clicked on the Scene:
  - Navigation Static: means this object will be used to compute NavMesh
  - Navigation Area: set the area for this object.

Click on the Ground GameObject on the left, and **go** to the Object Tab on the Navigation Tab. **Tick** the Navigation Static and **set** its Navigation Area into "Walkable". This turns the mesh of the Ground into something that will be considered for baking NavMesh.

Then, **click** on the Bake tab and click "*Bake*". You should see the following in your Scene. That blue overlay is the baked NavMesh. During runtime, any NavMesh agent will use it to traverse the area. Note that anything that lies on the XZ plan is **automatically detected** as *bake-able area*.

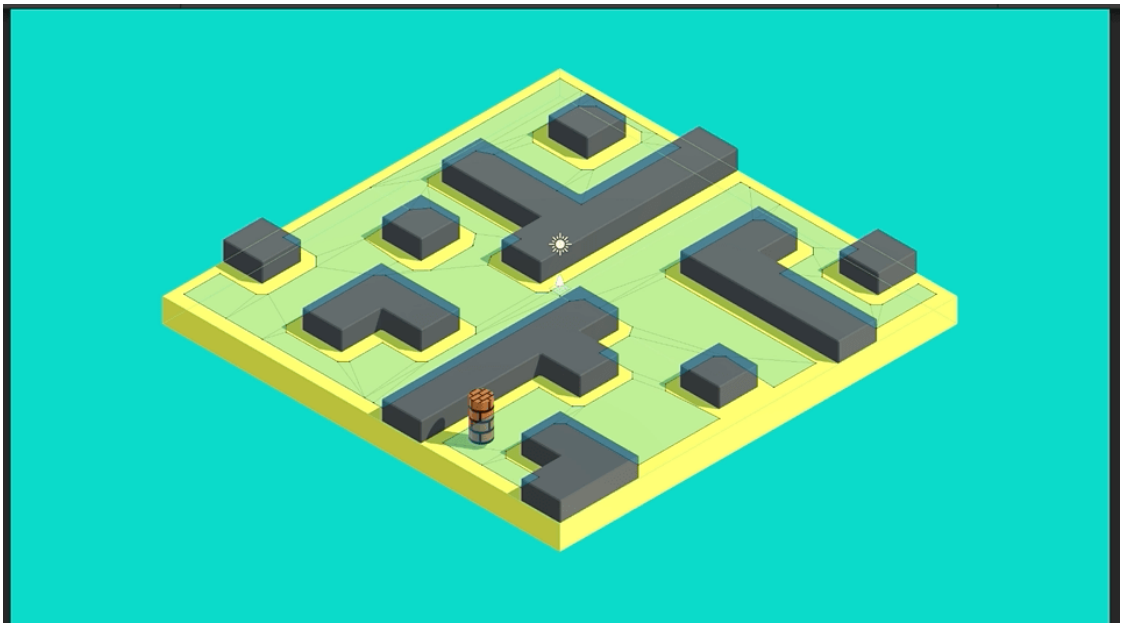


Now click on the Player GameObject and notice that there's already a Navigation Agent component added to it, which means it can interact on the baked NavMesh. Ensure that the Player GameObject is nicely placed above the Ground NavMesh.



The NavMesh you just baked matches the Agent's specification. If you were to create new agent type in the Navigation » Agent Tab, then you can't use that new agent in this NavMesh baked for "Humanoid". We will get to other agent creation later on.

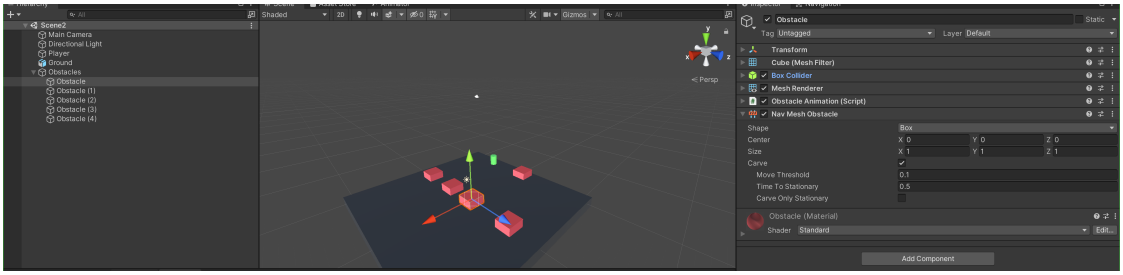
Test run and click anywhere on the Game screen. The "player cylinder" will go to the place on the Screen that you click.



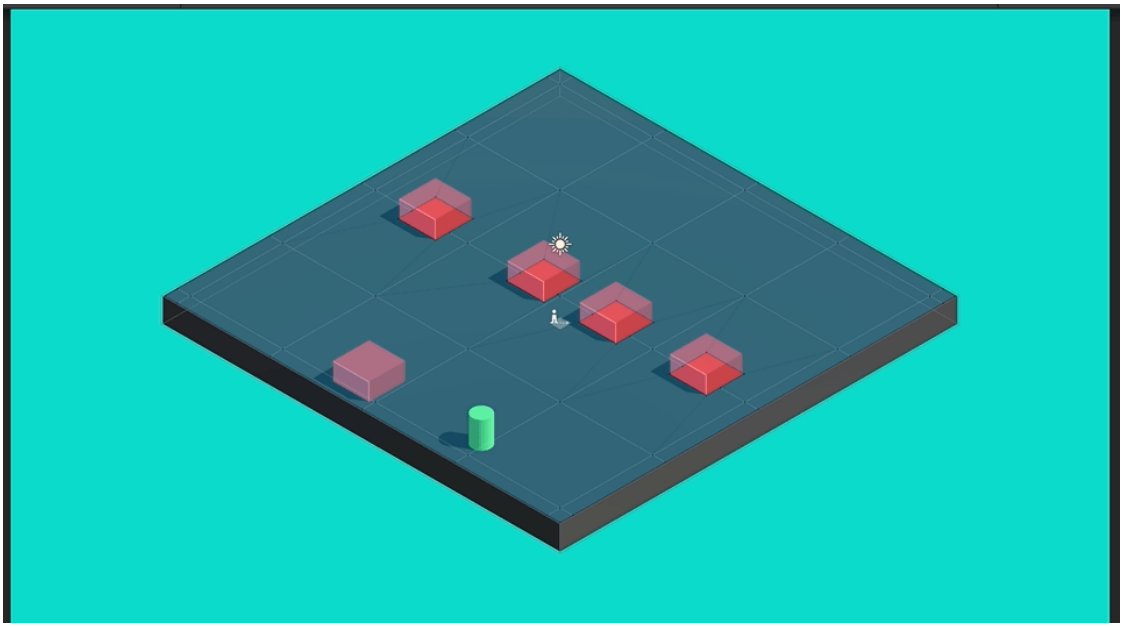
NavMesh Obstacle

As you have noticed, the NavMesh map so far is generated *before* the program is run, meaning that it is a **static** map. We can also choose to *create* obstacles at runtime, hence modifying the map as the program runs instead of only once in the beginning. To do this, we need the component **NavMesh Obstacle**.

Open the scene inside NavMeshIntro » Scene2 folder and click on one of the Obstacles gameobject:



Observe the component NavmeshObstacle that can be added to any gameobject with a MeshRenderer. The property **carve** is ticked, which means that if you move this obstacle at runtime, the Navmesh will be updated accordingly.



You can also *generate* NavMesh Obstacle at runtime. You'll need a reference to the gameobject with a MeshRenderer (as an obstacle to generate), and the NavMeshSurface that you want to instantiate this obstacle on.

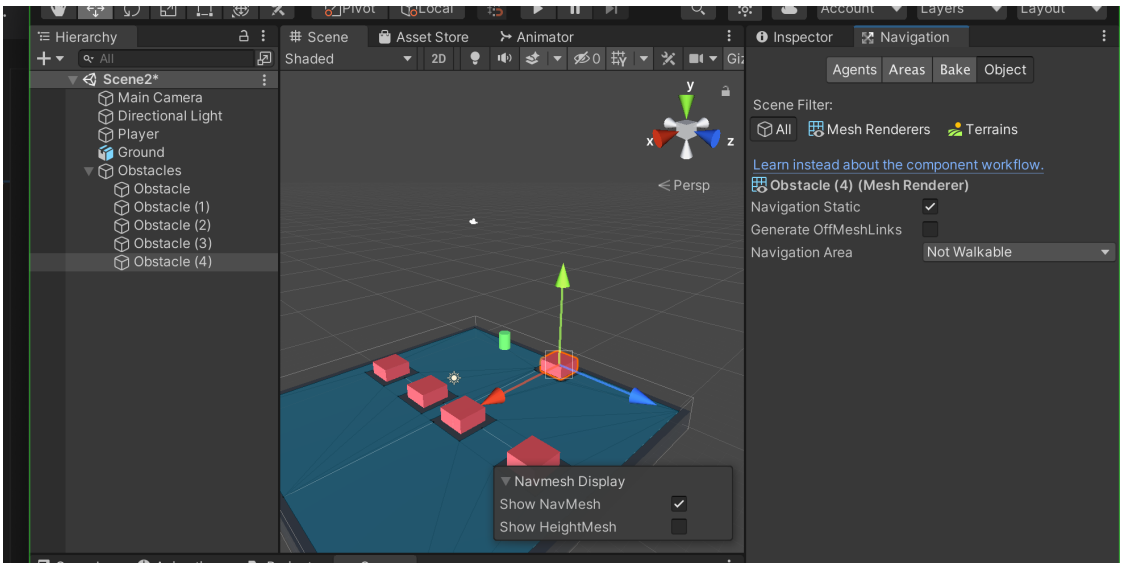
Open the Script `Generate.cs` inside the Ground gameobject in Scene2, and look at the following method:

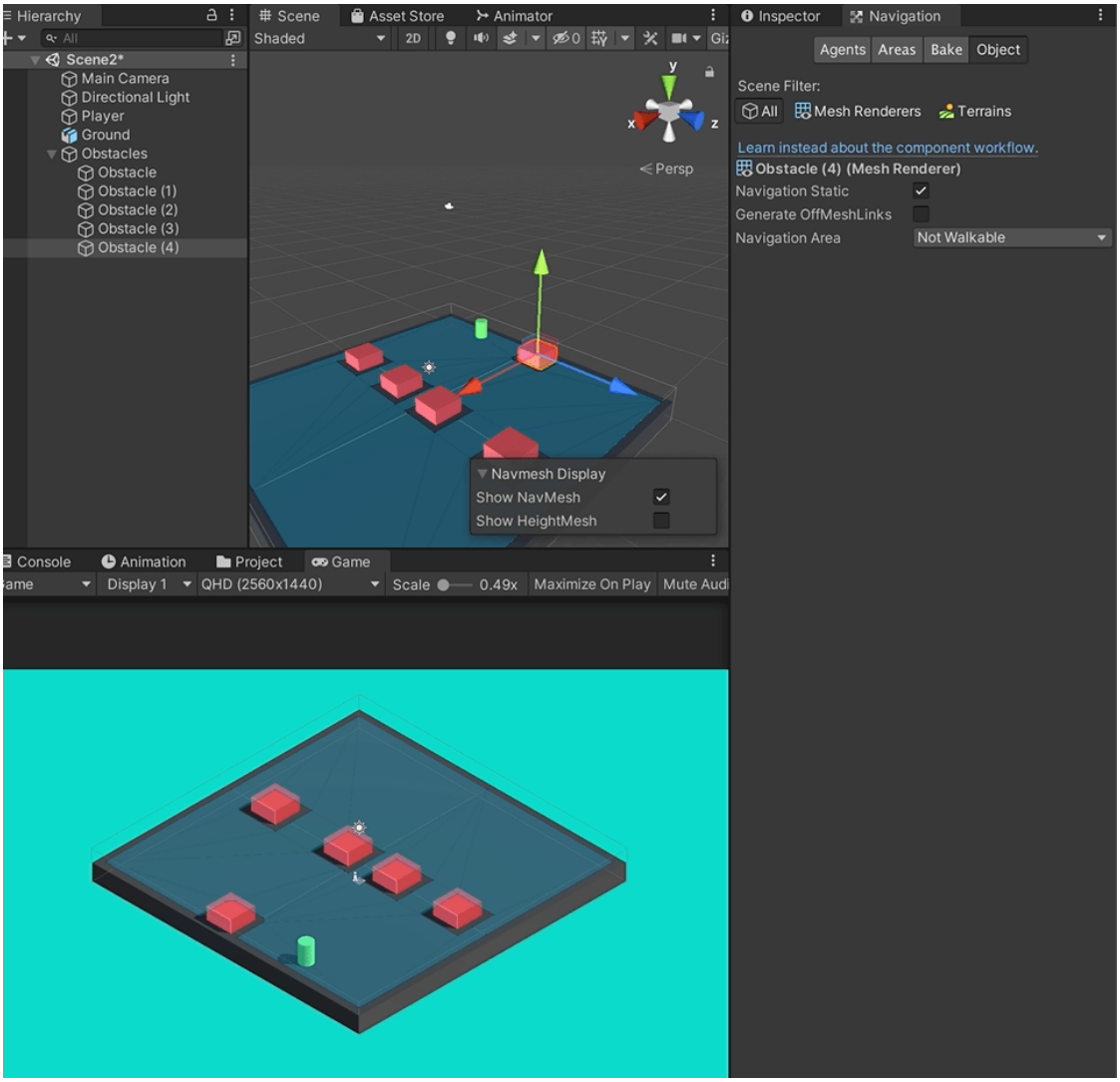
```
void GenerateWall()
{
    float x = (Random.value - 0.5f) * 20.0f;
    float z = (Random.value - 0.5f) * 20.0f;
    Vector3 pos = new Vector3(x, 1f, z);
    GameObject instance = Instantiate(wall, pos, Quaternion.identity, transform)
    instance.AddComponent(typeof(NavMeshObstacle));
    instance.GetComponent<NavMeshObstacle>().carving = true;
}
```

To make any newly instantiated object an **obstacle** on the NavMeshSurface, we simply need to just add the NavMeshObstacle component to the game object, and

enable `carving`. We can create a prefab with this specification so that we don't have to have boilerplate code.

If we want to remove NavMeshObstacle at runtime, we simply just **destroy** or **disable** the obstacle gameobjects. However, if we remove `static` obstacle like this Obstacle(4) during runtime, we are left with a "hole" in our NavMeshSurface:





If you want to remove static obstacles, you need to erase the existing data and rebuild the NavMesh. Of course by definition, static means **STATIC**, and you should use **OBSTACLE** if you'd like to modify stuff at runtime.

Although we are not supposed to change static (baked) objects at runtime, of course theres *workaround*. To remove static "not walkable" areas at runtime, we need to Destroy or disable that gameobject, delete the NavMeshSurface and then rebuild our NavMeshSurface using the following Coroutine in the script:

```
IEnumerator resetNavmesh()
{
    while (staticObject != null) yield return new WaitForSeconds(1);
    surface.RemoveData();
    surface.BuildNavMesh();
}
```

Notice that weird check on that `staticObject` reference. This is because you don't really know when the `staticObject` is actually deallocated when you call `Destroy(staticObject)`. You can use `DestroyImmediate` but it is not recommended for build. Therefore you either need to **skip a frame** or you need to force wait for a little bit using a Coroutine before removing the existing NavMeshSurface data and rebuilding it. The same applies for `SetActive(false)`, the effect will only be observed in the **next frame** although the object is already flagged as inactive the moment you call the `SetActive(false)` instruction.

> If you feel that this is too hacky, that's because it is. It is probably best not to modify Navigation Static meshes during runtime.

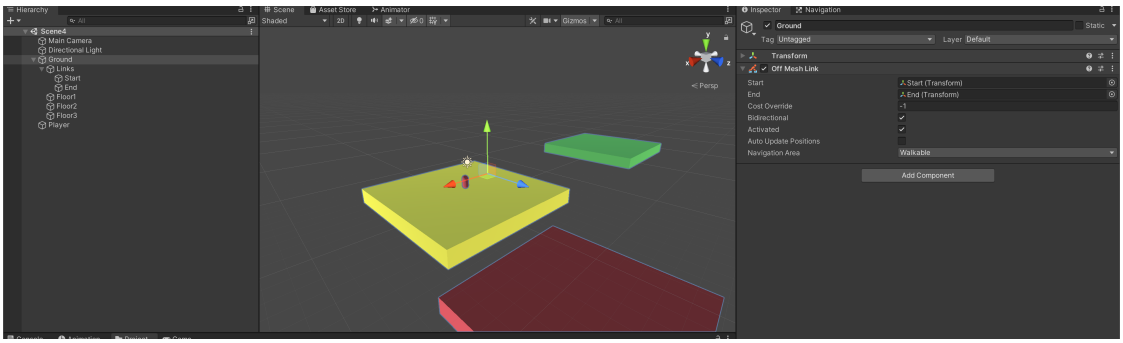
Traversing Between NavMeshSurfaces

OffMeshLinks

Sometimes we need to *connect* between two or more NavMeshSurfaces, and we can do this using OffMeshLinks (Unity default) or NavMeshLinks.

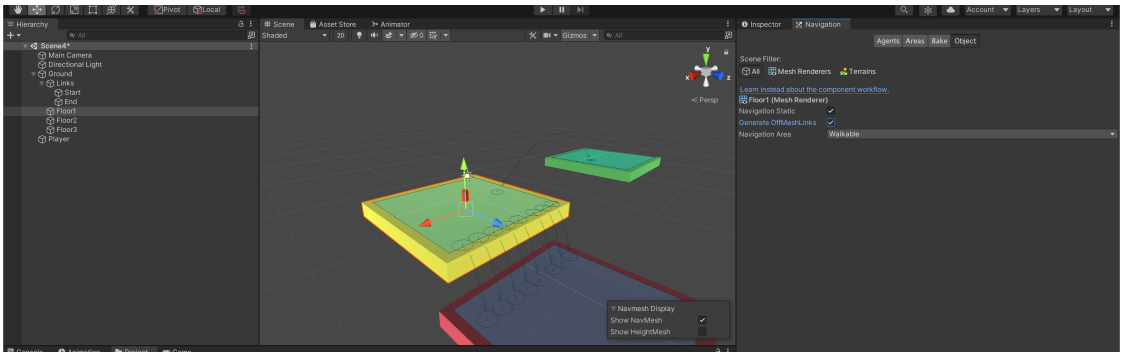
> From Unity's Official Documentation: OffMeshLink component allows you to incorporate navigation shortcuts which cannot be represented using a walkable surface. For example, jumping over a ditch or a fence, or opening a door before walking through it, can all be described as Off-mesh links.

Open the Scene inside NavMeshIntro » Scene4. **Observe** the GameObject "Start" and "End" under Ground » Links. These two indicate the location of "entry and exit" point between the green and yellow surfaces. Click on the Ground and realise that there's a component called `Off Mesh Link`, where Start and End properties contain references to the Start and End gameobjects:

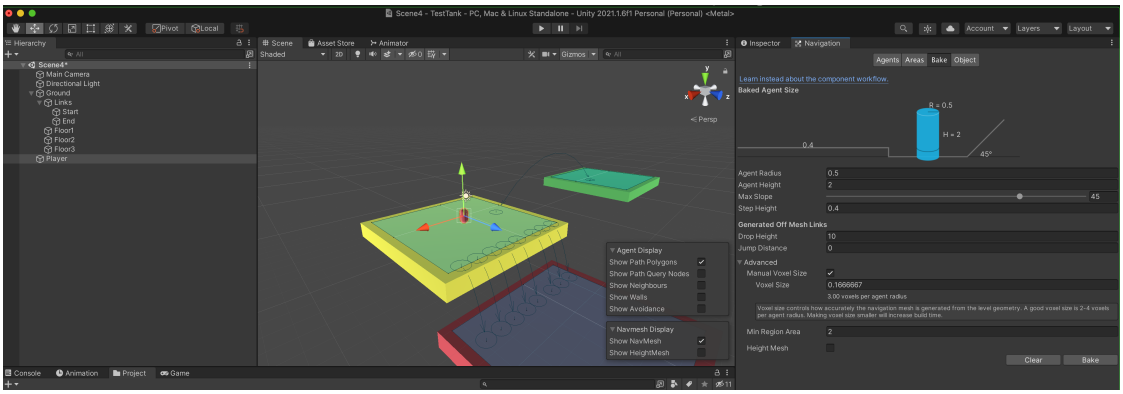


You can read more details about the rest of OffMeshLink's properties here.

Now Floor1 (yellow) is placed slightly above Floor3. We can *automatically* generate OffMeshLink by enabling the Generate OffMeshLinks property on the Floor1 Mesh Renderer. If this property is not ticked, then the OffMeshLink generated is only the one between Floor1 and Floor2 which we added earlier. Click "Bake" under the Bake tab and you shall see the NavMeshSurface generated as follows:



It is *possible* to auto-generate OffMeshLink as long as the "Drop Height" property inside the Bake tab is obeyed between Floor1 and Floor3:



You can also **animate** the Player gameobject when on OffMeshLink using `agent.isOnOffMeshLink` check. Open the script attached to the Player in Scene4, and read the following code:

```
if (agent.isOnOffMeshLink && gameObject.tag == "customplayer" && !isPlaying)
{
    gameObject.GetComponent<Animation>().Play();
    isPlaying = true;
}

else if (!agent.isOnOffMeshLink && gameObject.tag == "customplayer" && isPla
    if (isPlaying){
        isPlaying = false;
        gameObject.GetComponent<Animation>().Stop();
    }
}
```

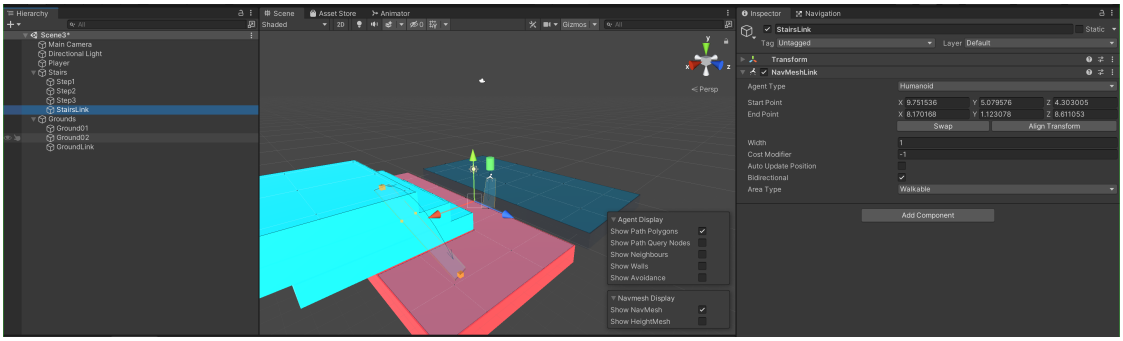
There's already an *animation* component attached on the Player gameobject (just a single animation, and we will explicitly play it from the script so there's no need for an Animator). We can easily check if the agent is on or off the OffMeshLink and we can play or stop the Animation accordingly.

NavMeshLinks

Another alternative is to use NavMeshLink (part of NavMeshComponents package we installed earlier). Neither is favoured over the other, and it depends on your usage. For example, in NavMeshLink, you can set the **link width**, whereas OffMeshLink is just one single path.

Open the Scene in Scene3 folder, and click on the gameObject StairsLink. Notice that it has a component called NavMeshLink. Similar to OffMeshLink, you can define a start and end point on separate NavMeshSurface. You can also adjust the

width of the link:

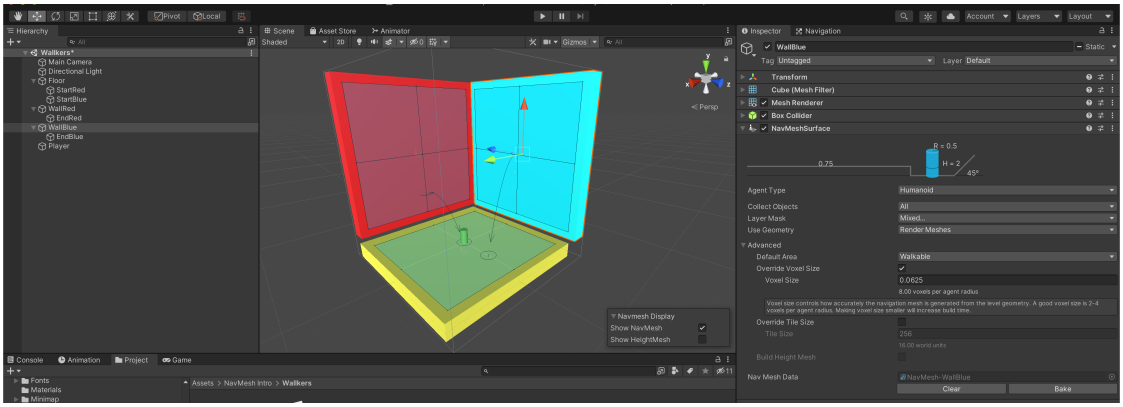


More details about its other properties can be found here.

NavMeshSurface

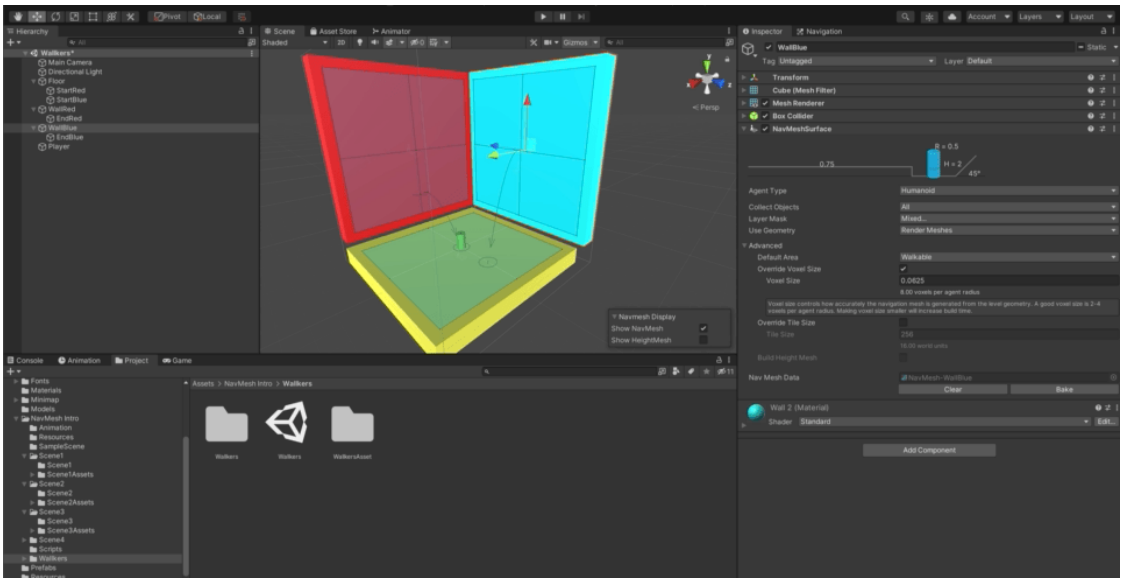
Finally, if we want to create a *walkable* surface on vertical walls, we need to add the NavMeshSurface Component. This feature comes from the NavMeshComponent asset we installed earlier (doesn't come with standard Unity installation). Standard Unity installation only allows us to create NavMesh out of meshes that are *aligned* with the **world** XZ plane, or more precisely: the XZ plane on the positive global Y-axis side.

Open Scene4 (inside NavMeshIntro » Scene4). Notice how we have NavMeshSurface component attached to WallRed and WallBlue. If we click *Bake*, this will generate a NavMesh for each wall, at its XZ plane on the positive Y-axis side:



We can add a couple of OffMeshLink to connect each surface, hence allowing the Agent on the Player gameobject to travel between the three surfaces. One thing to

note is to align the *local* Y-axis orientation of the player to be the same as the surface it is on as shown:



This can be done by enabling the `updateUpAxis` property:

```
NavMeshAgent agent = gameObject.GetComponent<NavMeshAgent>();
agent.updateUpAxis = true;
```

NavMesh for 2D Games

As mentioned previously, NavMesh works on *Mesh Renderer*, which means that it won't work by itself on regular Sprites or TileMap. This asset called NavMeshPlus is a great workaround if you want to create NavMesh on 2D surfaces. You can add the package from Git URL: `https://github.com/h8man/NavMeshPlus.git#master`

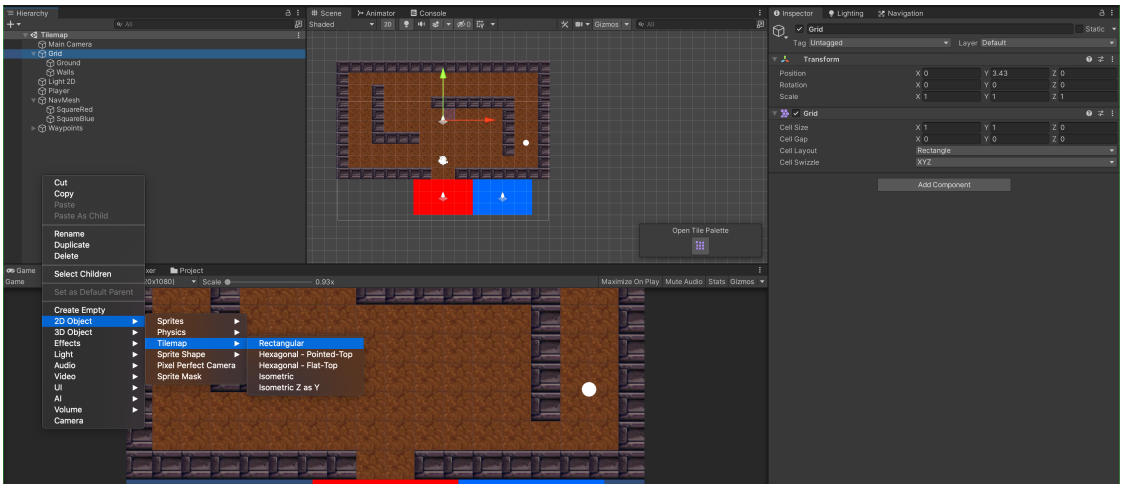
> NavMeshPlus comes with NavMeshComponents, therefore to avoid **conflicts**, you may remove NavMeshComponents from the package registry.

Then import the asset `navmesh2dintro` from the course handout. to your project.

> NavMeshPlus comes with NavMeshComponents, and you may be faced with conflicts. To avoid **conflicts**, you may remove NavMeshComponents from the package registry.

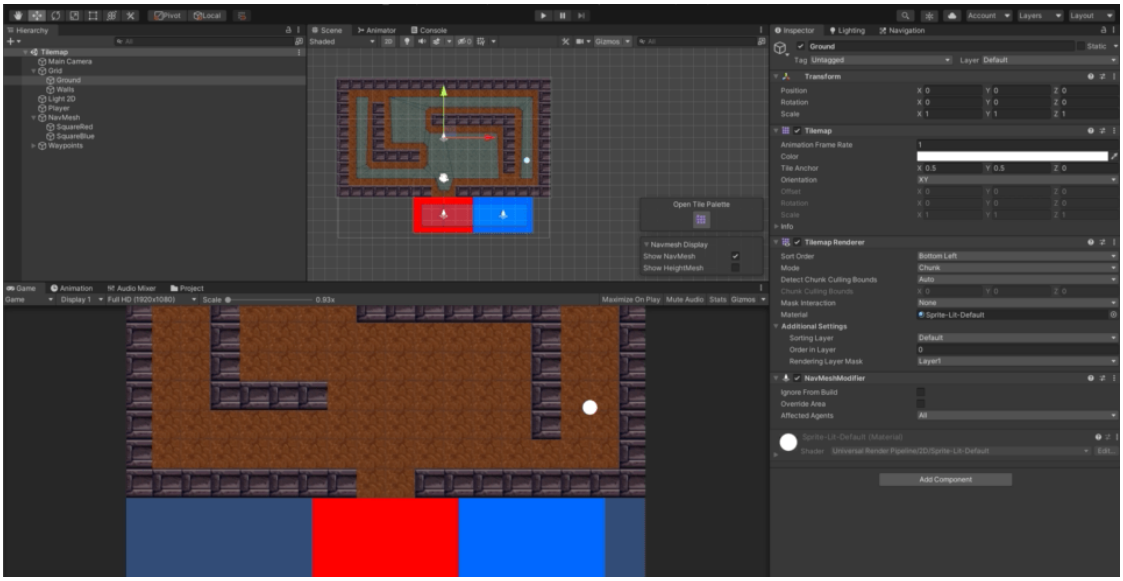
Create a TileMap

Tilemap is another handy tool to create 2D maps, especially for top-down games which levels or world would otherwise be too cumbersome to be created as separate Sprites. You need to add the 2D Tilemap Editor from the **package manager.** Once downloaded, open the Scene Tilemap (inside NavMeshIntro » TileMap) and click on the Grid gameobject. This gameobject was created using Create » 2D » Tilemap » Rectangular.



You'll have a Grid gameobject with one child gameobject for you to place your tilemaps. You can continue **adding** more Rectangular TileMap as a child of Grid. **Click** on Grid and then click on **Open Tile Palette** in the Scene. This allows you to create new **Palettes** which you can use to **paint** the Tilemap. You can drag **tile sprites** onto an empty palette. Once loaded, you can start using the Tile Sprites to pain the tilemaps. You can select the **Sorting Layer** and set the value of **Order in Layer** properties in the **TileMap Renderer** to determine which layer should be rendered on top.

The gif below summarises the entire process:

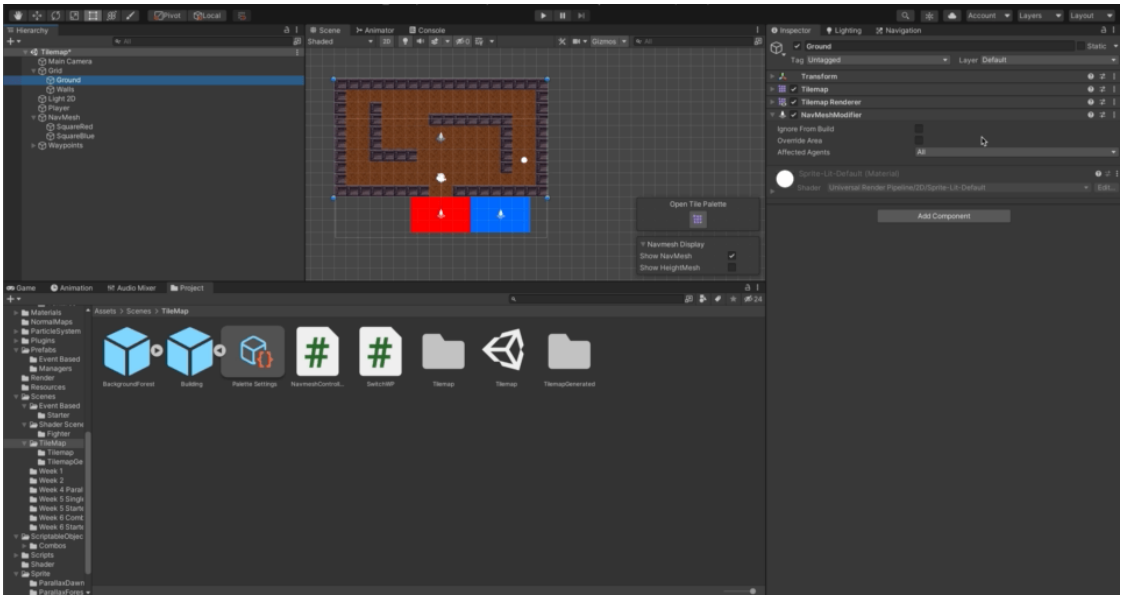


Create NavMeshSurface2D and NavMeshModifier

Click on the NavMesh gameobject, and notice there's two NavMeshSurface2D components placed on it:

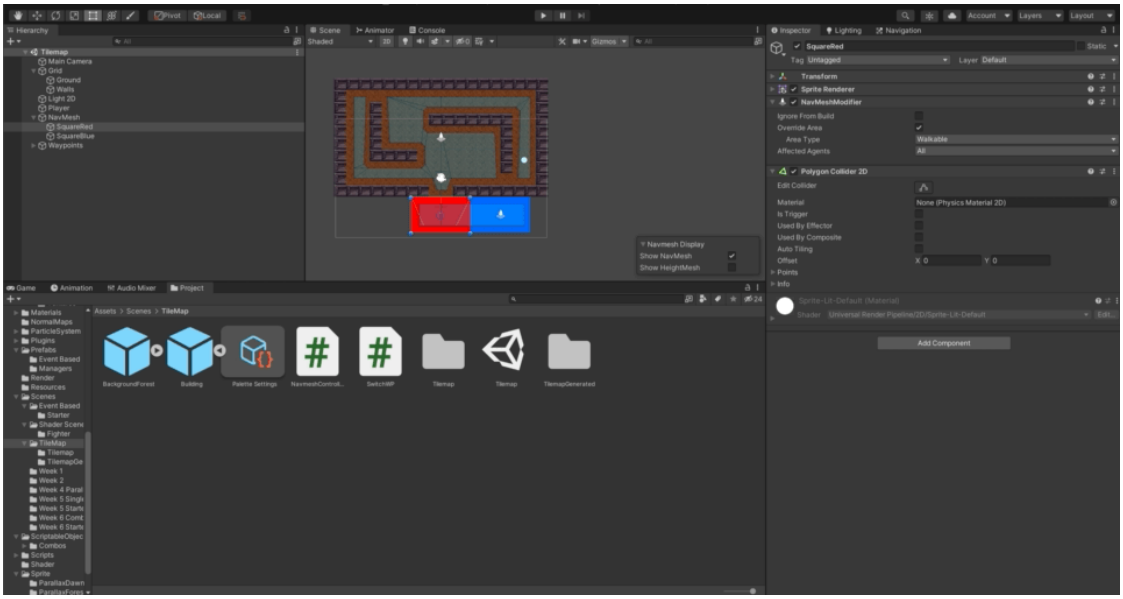
- The first one monitors its "children", which is a collection of Sprites (**collect object** property set to Children)
- The second one monitors the "world" (**collect objects** set to All)

When you click "Bake", this generates Navmeshes on corresponding sprites or tilemap. Of course **NOT** all Sprites or Tilemap are considered into NavMesh baking by default. You'd need to **add** NavMeshModifier component to tell the system that "this object" should be considered during baking. You can even define whether it is Walkable or not Walkable as usual:



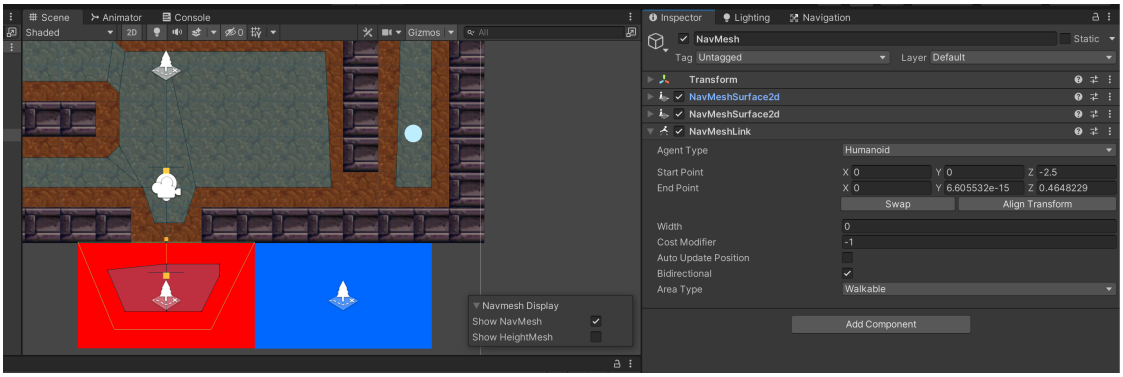
Generate NavMeshSurface2D from Polygon

You can also change the NavMeshSurface2D property: Use Geometry to use Physics Colliders instead so that we can generate NavMesh following a defined 2D collider instead of the Sprite Renderer.



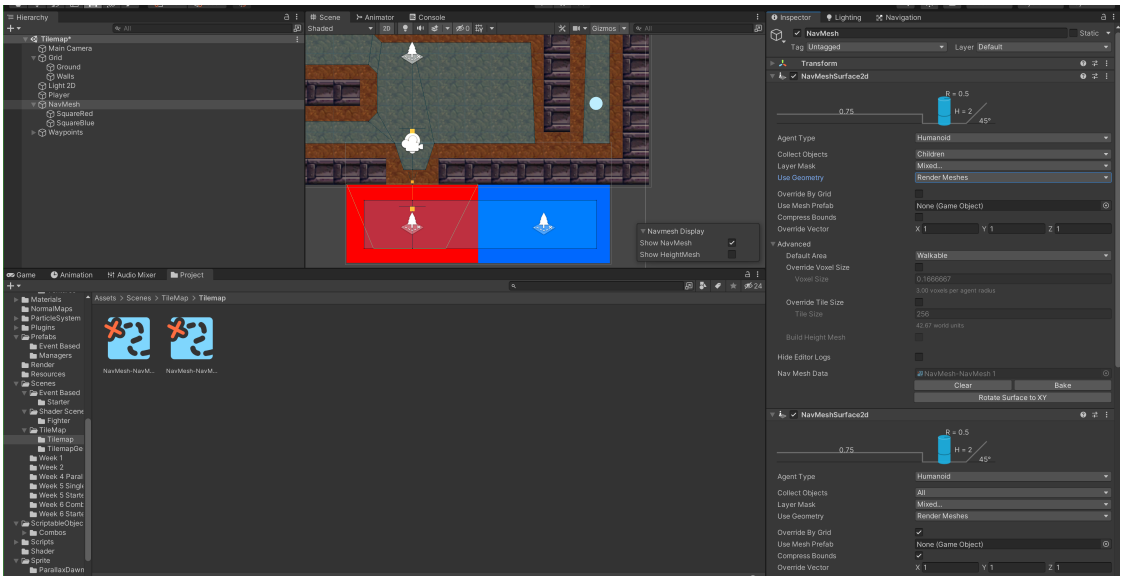
NavMeshLink

Similar to NavMeshes in 3D, we can set NavMeshLinks as usual to connect two baked NavMeshSurface2Ds together:

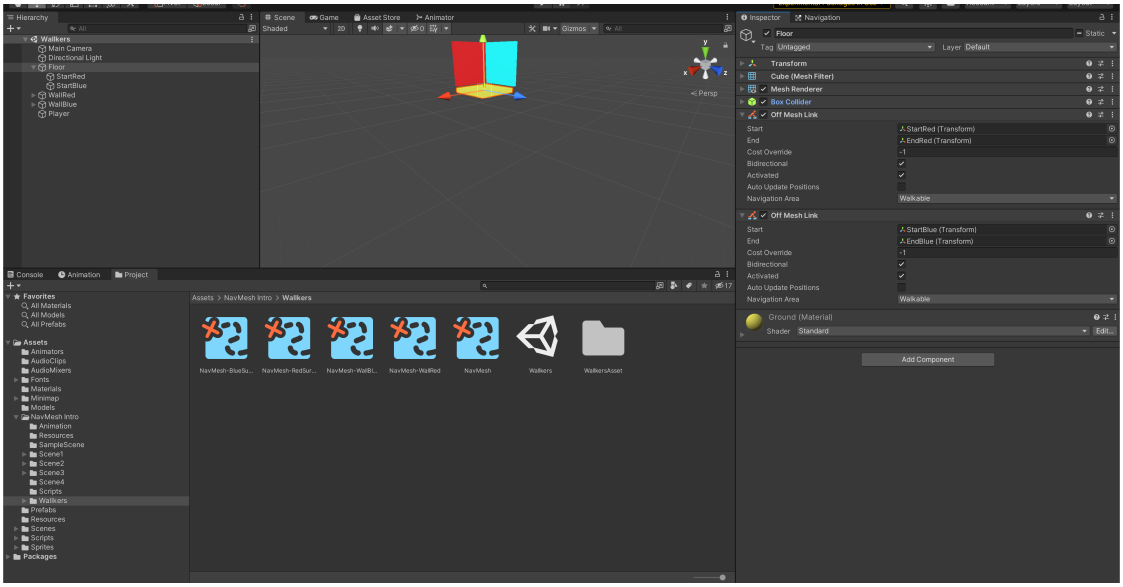


NavMesh Data

If you haven't realised already, the data for NavMesh (2D or 3D, doesn't matter what it is) is stored typically at the directory where the Scene is, inside a folder with the same name as the Scene:



Here's the data for the Wallkers Scene:



Finding "Click" Location in the World Coordinates

Notice how in both 2D and 3D Navmesh scene demos, we can always click to screen and the Player is able to navigate to that location.

2D Click Location

In 2D setup with **Orthographic** camera, we need to transform the screen coordinate into the world coordinate using `Camera.main.ScreenToWorldPoint`. Open the script `NavmeshController.cs` and observe the following instructions:

```
if (Input.GetMouseButtonDown(0))
{
    Vector3 mousePos = Input.mousePosition;
    mousePos.z = Camera.main.nearClipPlane;
    worldPosition = Camera.main.ScreenToWorldPoint(mousePos);
    // Debug.Log(worldPosition);
    agent.destination = new Vector3(worldPosition.x, worldPosition.y, zpos);
}
```

3D Click Location

In 3D setup with **Perspective** camera, we use **raycasting** instead. That is, we will cast a Ray originated from the main camera towards the direction of mouse click on screen, and find if any the **Hit** location on **any 3D Physics Collider** in the Scene. This method wouldn't have worked in 2D games because *obviously* we **do not use 3D colliders** there, which belongs to the Physics Engine and not 2D Physics Engine.

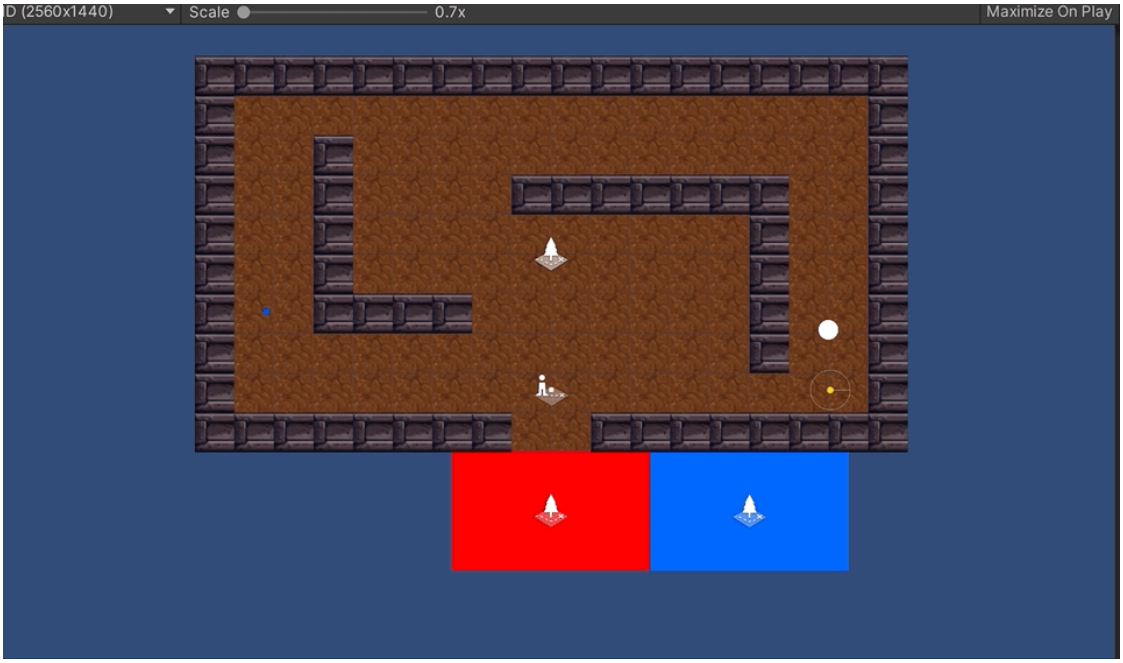
Open any PlayerController script used in Scene1 to Scene4, or Wallkers, and you should see the following code that computes the exact World location for the agent to go to on Mouse Click:

```
if (Input.GetMouseButtonDown(0))
{
    Ray ray = cam.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;

    if (Physics.Raycast(ray, out hit))
    {
        agent.SetDestination(hit.point);
    }
}
```

Navigating Between Waypoints

We often need to define a specific path for bots to patrol, and we can do that by creating a few gameobjects as WayPoints, and then ask the Navigation Agents to patrol between the waypoints as such:



This is done in a simple script `SwitchWP` placed inside the gameobjects WP0 and WP1 in Tilemap Scene:

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

public class SwitchWP : MonoBehaviour
{
    public int destination;
    void OnTriggerEnter2D(Collider2D other)
    {
        Debug.Log("collided");
        if (other.tag == "Player")
        {
            other.GetComponent<NavmeshController>().goToWP(destination);
        }
    }
}
```

You can define a more complex patrol path using some kind of State Machine, which we will cover in the next tutorial.

Next

There’s no checkoff with this lab. In the next lab, we will utilise our knowledge on NavMesh and Scriptable Object to create patrolling bots.

PREVIOUS POST
Unity for Elderlies

NEXT POST
Unity for Adults