**Sprites**

**Prerequisites**

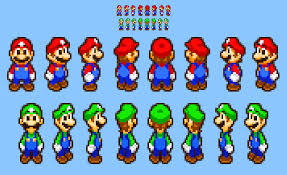
One would need to understand the prerequisites of game development to understand Sprites. This section involving the prerequisites to this tutorial will also prove to be a glossary of sorts so that whenever any of these words are used in the following content, there is clarity about what they mean. These prerequisites are as follows:-

1. 2D Game: A 2D game, as the name suggests, is a game seen, played, and developed from a 2-dimensional perspective. This implies that the game, and its elements, give no sense of depth-perception of its world to the player. Movement is restricted to 4 particular axes (technically two axes, but the animations require the developer to define it in terms of the following four axes): X axis, negative(-) X axis, Y axis, and negative (-) Y axis.
2. Model: A model is a more generic term for the element of a game. A 3-dimensional playable character is built from a model; the mobs of a game are built from a base model. Under the hood, a model is just a blueprint for a 3D element of a game.
3. 2D Core (in Unity): Similar to how we define a 3D core, as an engine/”mode” of Unity that allows 3-Dimensional game development, a 2D core helps in making 2D games, and consists of all sets of tools and animation builders that allow the development of a 2D game, and specifically speaking, the Sprites/”models” of the 2D game.
4. Asset: An “Asset”, as the name suggests, is a component of a game developed in Unity that is used in the development process. 3D models, sprites, textures, prefabs, music, scripts, etc. are all examples of assets.

With these prerequisites out of the way, we can now begin to understand what a Sprite is.

**Sprites ~ The fundamental functioning unit of a 2-D game**

What models are to 3D games, Sprites are to 2D games. Put simply, a Sprite is any 2D graphic object used for the elements of a 2D gameplay. Characters, maps, props, interactable items, projectiles, powerups- these are all portrayed in any game using dedicated Sprites. One of the earliest examples of a Sprite for any young student interested in animations and/or game design would be [Scratch Animation Software](https://scratch.mit.edu/projects/editor/?tutorial=getStarted) (Hint: One of the first things you’ll see on the workspace once you click on the link is a “Sprite1” object that shows Scratch’s mascot- Scratch Cat, which happens to be a programmable sprite as well).



Example of sprites used in Mario

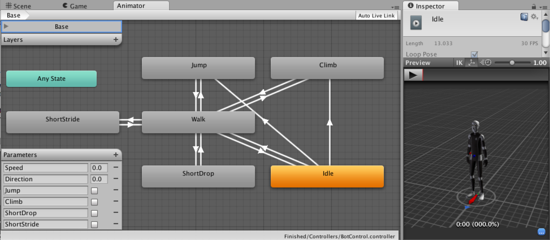
(**Observe**: There are multiple sprites for the same character, but facing different directions. As we will see later in this tutorial, these are used to animate different directional movements)

**Why Unity for 2D development?**

A Google search will reveal multiple software options (Scratch, GameMaker and GDevelop being some of them) to utilise 2D sprites and make a game, which begs the question- why even bother going ahead with a seemingly unintuitive and tough-to-use game engine such as Unity (the operative word being: “seemingly”).

The answer(s): *Scalability, versatility, and convenience.* ‘Nuff said!

Unity is probably one of the most versatile and powerful tools capable of handling sprites the way it does. Moreover, its Mecanim animation system (used to animate sprites, as we’ll see later) is highly intuitive and helps organise the animation workflow using simple mouse-and-keyboard operations, and does all of the technical work under the hood, which is quite possibly the largest icing on the cake, since developers don’t need to struggle with long-drawn codes and/or configurations for the animations.



A screenshot illustrating the Mecanim System (source: [Unity Docs](https://docs.unity3d.com/462/Documentation/uploads/Main/MecanimShowcase.png))

**Getting started with making a 2D game.**

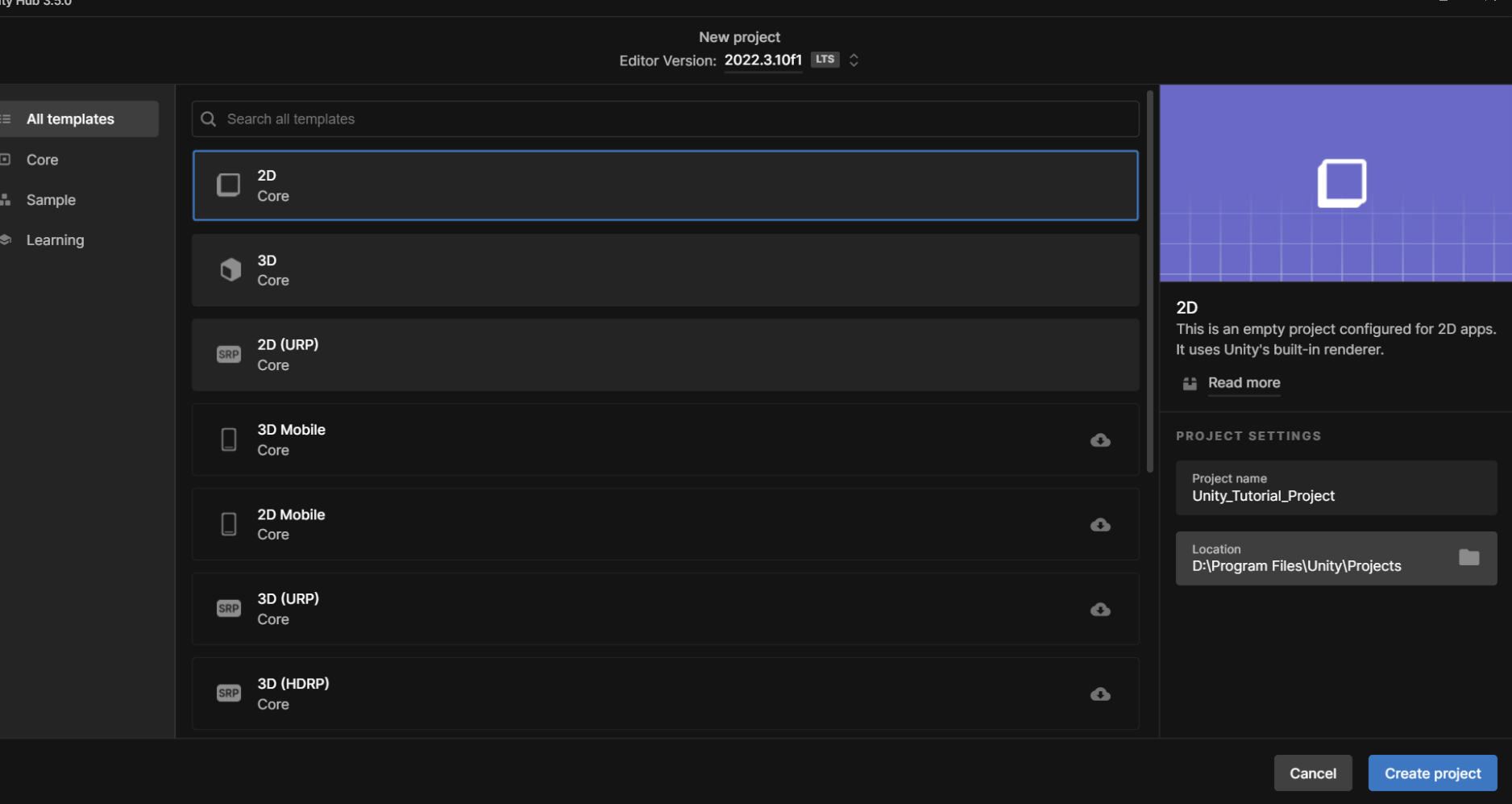
With the end of that (admittedly large) curtain-raiser to unity, there’s nothing better than getting your hands dirty and using a tool to understand it better. And Unity is no exception to this. The following sections will go over a tutorial to make an animated Pacman in a maze/map using Unity.

**Set Unity up**

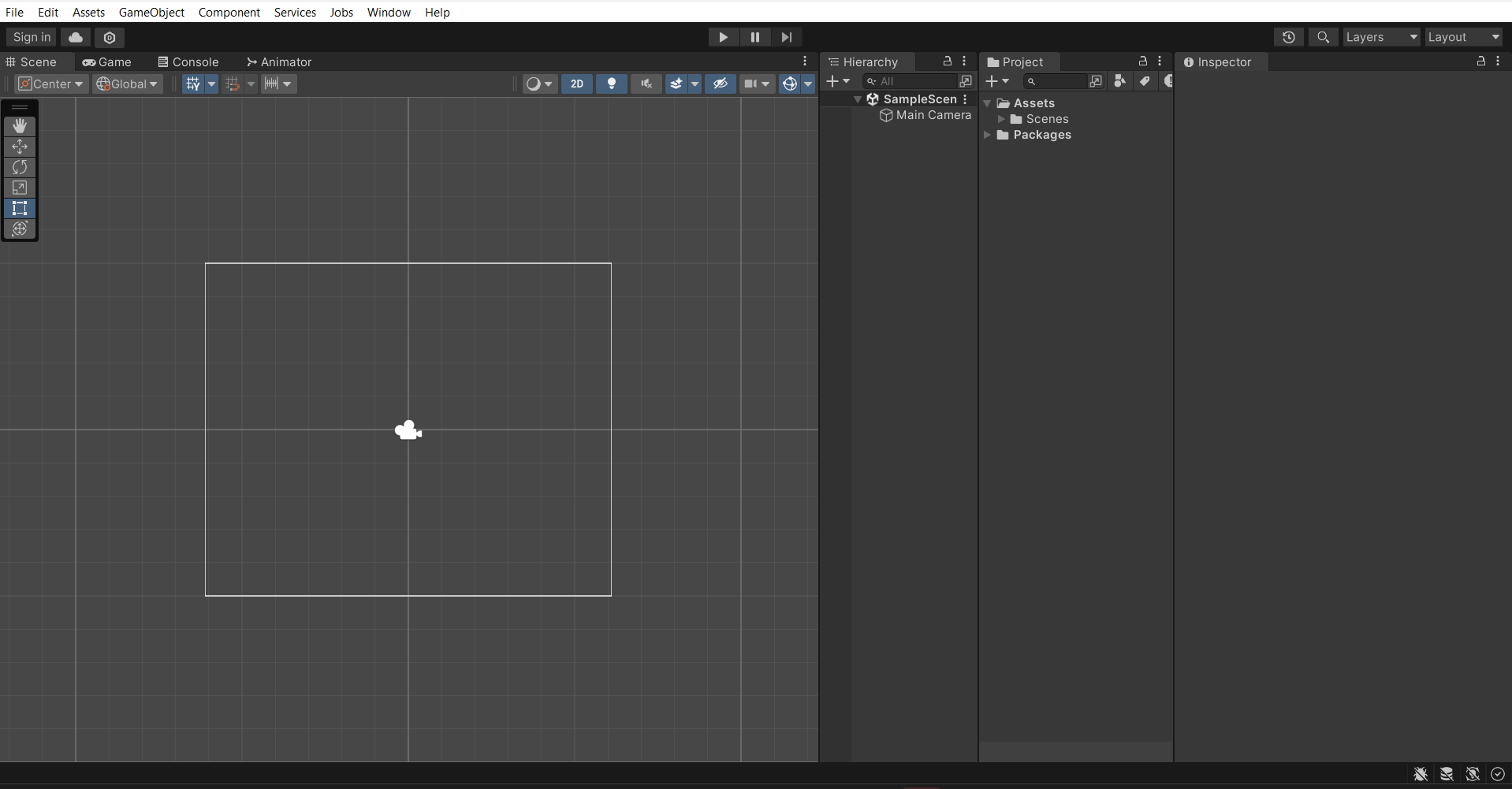
If you haven’t already, set up Unity- you can start by downloading it from the [official website](https://unity.com/releases/editor/qa/lts-releases). Going for an LTS version ensures a stable product for the next 1-2 years, hence the link referencing an LTS version. Once Unity gets downloaded, it’s time for the next step.

**New 2D Core Project**

Start a new project by clicking on **“New Project”**, “**2D Core”**, “**Create Project”** (After naming your project, of course).



If all has gone well, a Unity Workspace starts loading.

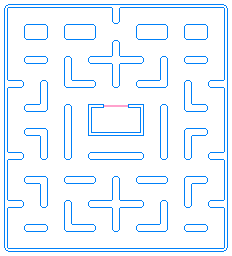


This is the workspace. Go ahead and familiarise yourself with basic navigation and UI, if you haven’t, already, before we move on to the next step. Also, give [this](https://docs.unity3d.com/2019.2/Documentation/Manual/UsingTheEditor.html) a read (by following the links) to get acquainted with basic Unity window jargon.

**Adding our first sprite: A Pacman maze**

Currently, our Project window looks a little too empty. So does our Game tab. Let’s fix that by adding our very first Sprite- a maze. If you’ve not been paying close attention, here’s a recap of one of the things we covered in the introductory sections: A sprite is a 2D graphic, that is the fundamental part of any 2D game. It forms the characters, the enemies/mobs, the map, even. And we’re going to focus on the last one: the map.

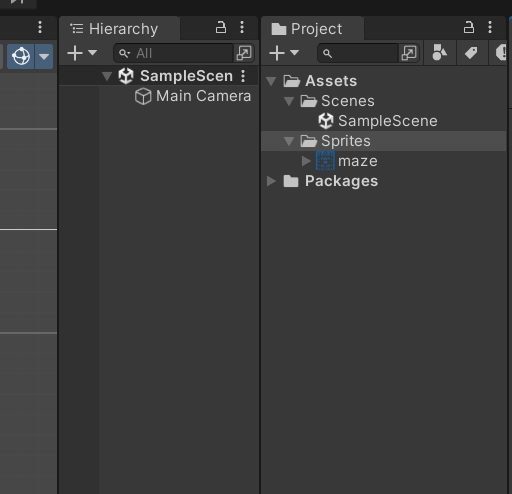
To start with, get the path of your Unity project folder, right-click the following image —> Save As - Save it in your Unity project path in the Assets folder under a new folder “Sprites”. Name it as “maze” for lesser ambiguity.



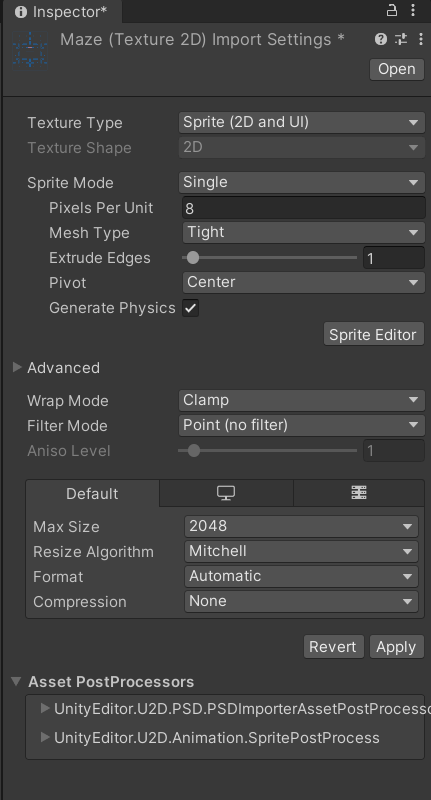
(Courtesy of <https://noobtuts.com/unity>)

(Here’s a link to the [image](https://noobtuts.com/content/unity/2d-pacman-game/maze.png) directly)

Your project folder should look something like this…

****

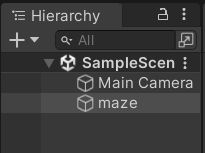
Click on the maze sprite and look at the inspector. Some settings should have come up by now. These are the import settings since Unity has already started treating your image as a Sprite since we are in the 2D core development screen (had it been a 3D core, a sprite would have to be a texture for it to work). Make sure to set the import settings exactly as the image below, and then click on Apply.



The logic for changed settings:-

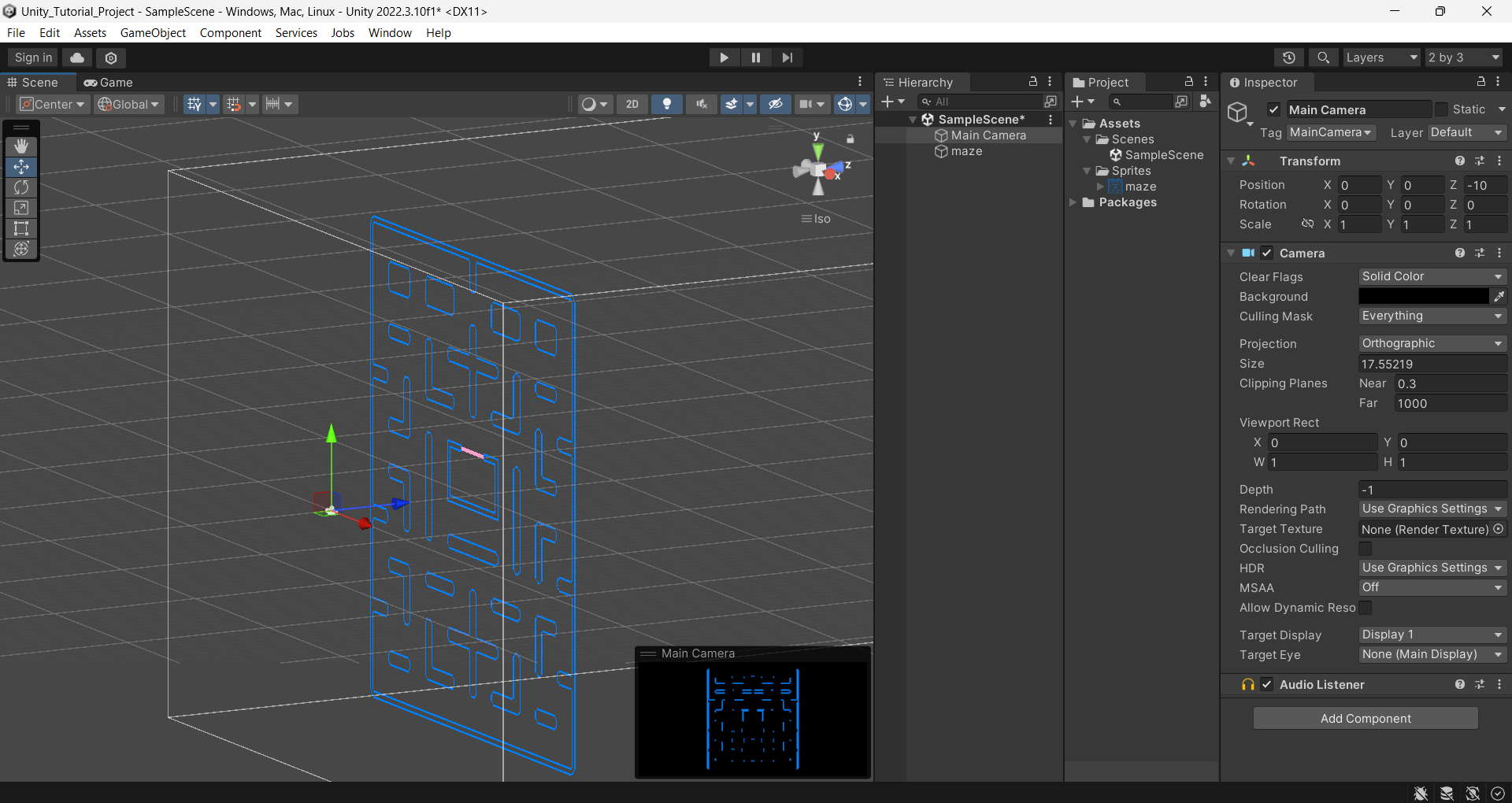
* Pixels per unit: the default is 100, which means 100 pixels is equivalent to one unit in the map. But this is way too large, and for a game like Pacman, where the collectables are supposed to be placed very close together, 100px is excessive. So we take an estimate of 8px as one unit, and now the texture treats 8 pixels worth of size as a single unit of distance. Changing this does not visually affect the game, but when 2D physics is involved, this drastically changes things for components like 2D colliders.
* Filter Mode: This is a simple one, change the setting to Bilinear or Trilinear, zoom into the map(once you drag it into the workspace), apply the settings, and see the magic happen. You’ll see that “point (no filter)” is the best choice in this case.
* Compression: None does not compress the texture and preserves the dimensions and clarity of the pixels. Does not create any visual changes for now.

Once the settings are applied, drag the “maze” sprite into the project window. Once this is done, the Hierarchy window now has a maze object. Et voila! We just used our first (of many) sprites.

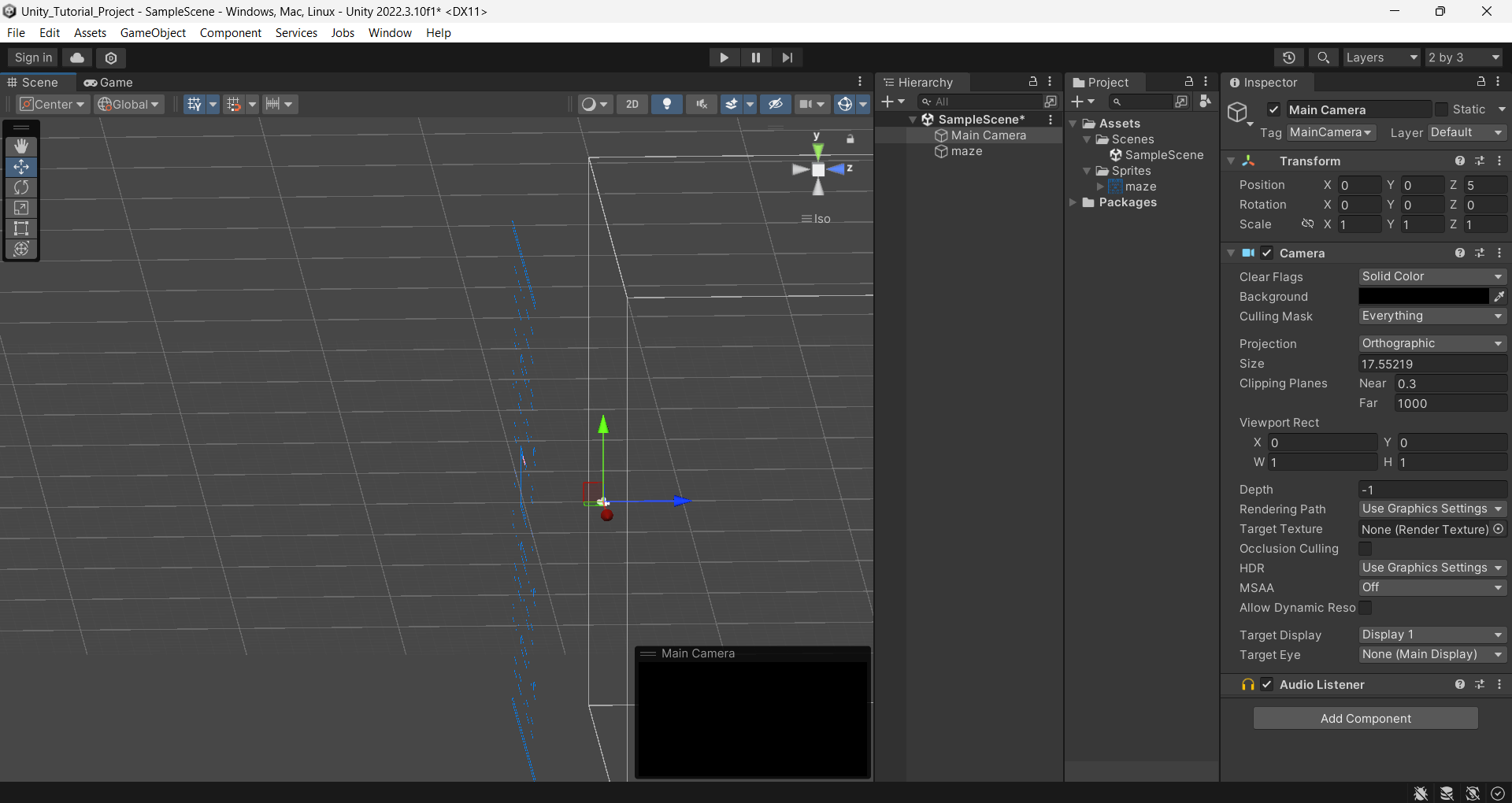


**Make changes to the maze object**

So far, so good. However, the centre of our maze and our camera object need to be in sync so that the player can see the whole map. In case you haven’t noticed already, the 2D core has a camera bang at the centre of the screen. Note that our Project Window is built with an invisible coordinate axis around it. So if you did move the map, camera or both, with your mouse, no worries! Click on the camera logo(), go to Inspector, Transform (section) and set the X, and Y coordinates to 0 (Do NOT mess with the Z coordinate for the camera; to know why it affects the game, check what happens when the camera’s Z coordinate is changed by toggling the 2D option in this tab. You’ll observe that the Z coordinate changes the distance between the camera and the maze object, and setting it to >= 0 would mean the camera goes ahead of the map, thereby resulting in an empty game screen (with the background colour you set).



(Camera at Z = -10: captures the maze in its field of view (indicated by a cuboid))

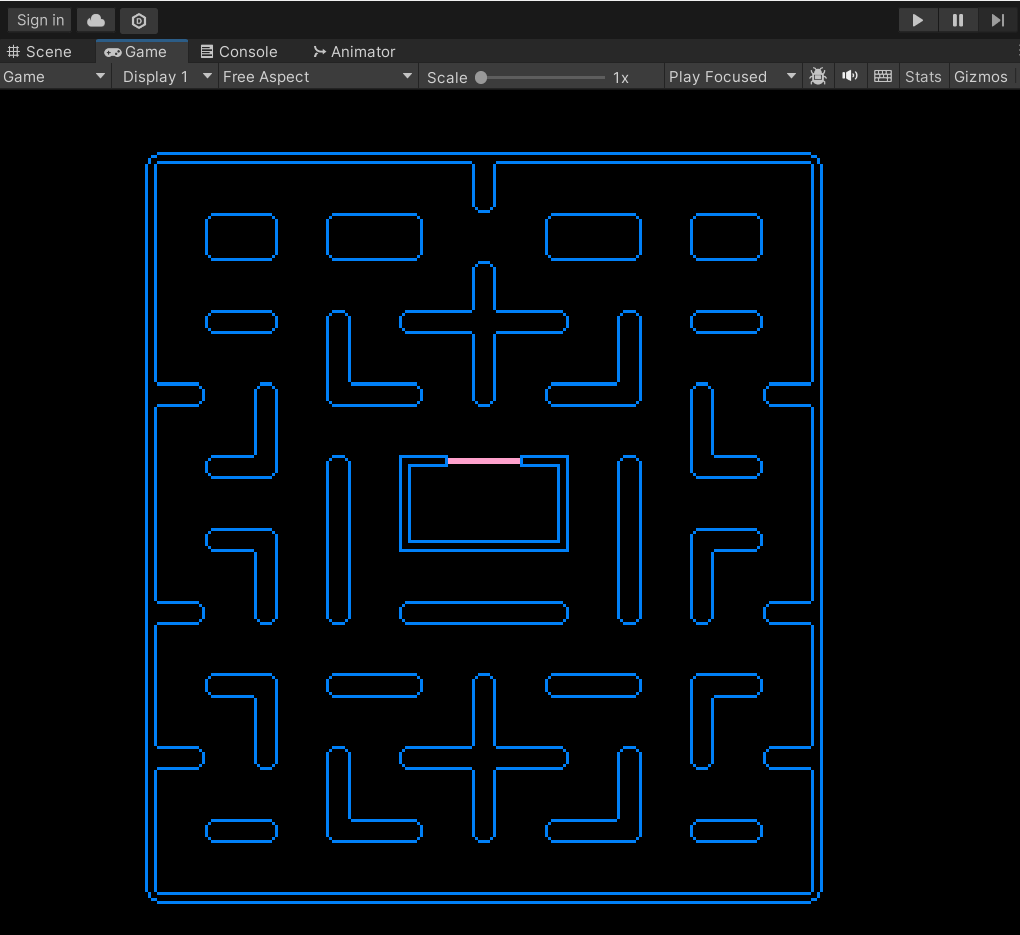


(Camera at Z = +5: Fails to capture the map in its field of view)

Back to the point…

Do the same for the map after clicking on the maze object in the hierarchy and set the X, Y and Z axes to 0.

For the more eagle-eyed observers, you’d have already found a game tab in the list of tabs as shown in the picture (if you don’t see it, go to the Window option, Panels dropdown, toggle the “3 Game” panel.

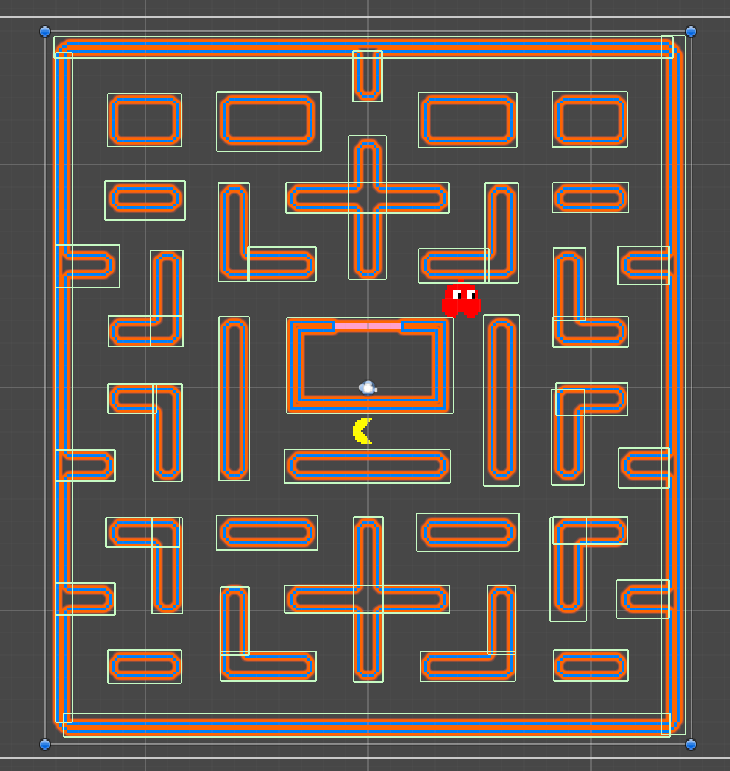


This is the actual view of the game- the players will observe this very screen when playing the game. If your background is not black yet, click on the maze object in Hierarchy, go to the Inspector and set the Background to black. If you don’t see the whole map, click on the Camera and resize it by dragging along the midpoints of the edges of the camera. These white edges denote the field of view of the camera

**Map Colliders**

After this, comes one of the most dreary, but also worthwhile steps: setting colliders to your maze. At the cost of stating the obvious, giving a map Sprite to Unity with borders and walls in the source image does not mean said borders and walls will get reflected in the game. This means even if one puts a moving Pacman object inside it, Pacman isn’t colliding with any of those “walls” since it doesn’t recognise anything that’s obstructing the way, according to Unity anyway. This means that Unity needs something that will be recognised as an obstruction... And in comes 2D Physics and 2D Colliders!

Click on the maze, go to the Inspector, click on Add Component, choose Box Collider, and resize them to fit each border and wall as shown below. (This is a dreary process, admittedly, but it works wonders if done right).

(Ignore Blinky and Pacman in the screenshot for now, just focus on how the boxes should look)

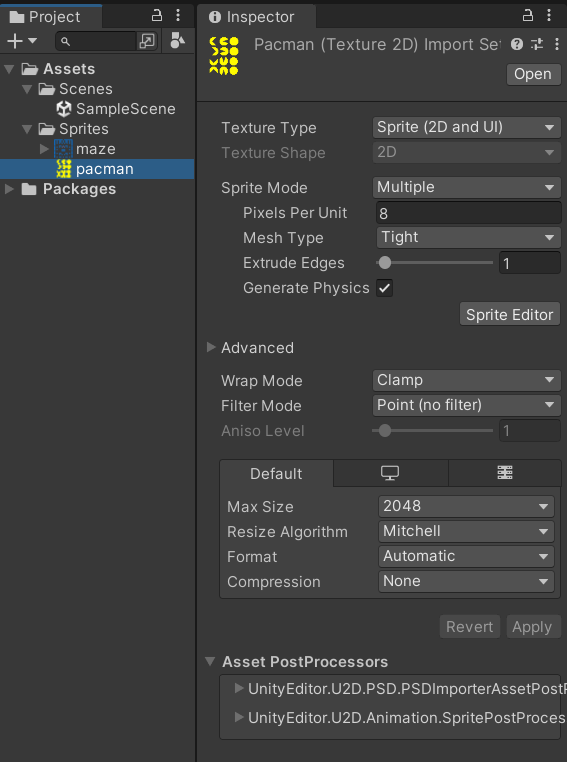
**Pacman**

Now that you’re done with the boring part (hopefully!) let’s get to the fun part: Getting ourselves a Pacman sprite.

This involves the same drill, get a Pacman sprite, import it and apply the settings as shown below.

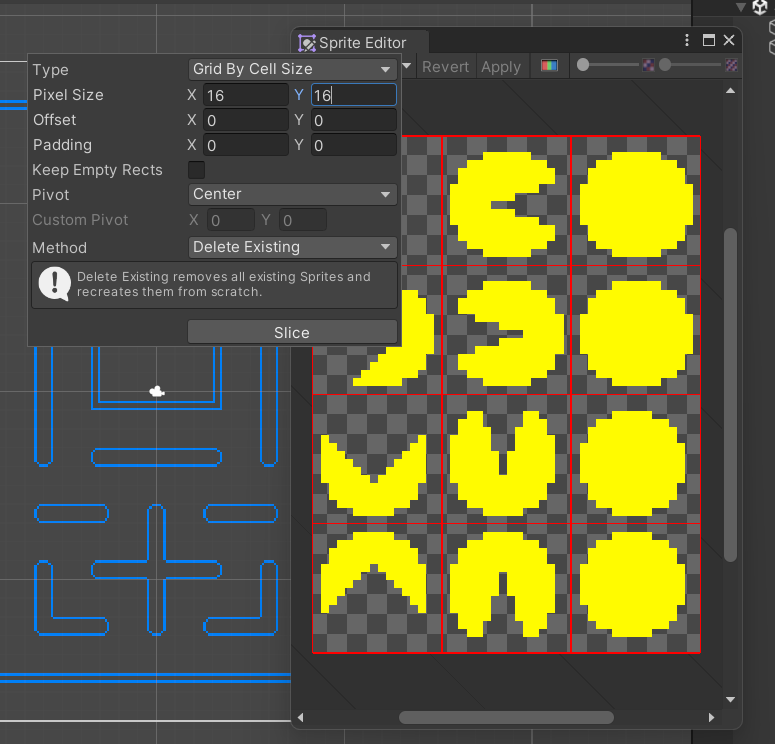


Pacman Sprite. Save this in your assets folder just like you did with the Maze.



The Sprite Mode: Multiple tells Unity that there’s more than one Pacman sprite in the imported Sprite.

We have Pacman sprites facing several directions in this image, but they need to be sliced for us to get our Pacmen facing multiple directions. (Refer to the settings screenshot given above) Click on Sprite Editor and do the following:-



This slices our various Pacman orientations neatly. Click on “Slice”, Apply and close the window. Observe: Our Pacman sprite now has multiple other Pacman sprites in the dropdown. Yep! These are the versions of the sprite you get after slicing the Pacman. Effectively, we now have to just animate the Pacman sprite. Also, observe the first 3 slices in the screenshot are to animate the Pacman when it’s facing right, the next 3 for the left animation, the 3 after that for the upward facing animation and finally, the last 3 for when the Pacman faces downwards.

**Pacman: Animation**

To go through with this, Ctrl+Click Pacman\_0, 1 and 2 and drag them anywhere on the Project window. Unity automatically recognises that these are to be animated exactly in this order when the frame changes. You’ll be prompted to save them. Create a new folder inside the Assets folder and name it PacmanAnimations (or anything else of your choice). Save the first animation as **right** (it’ll be saved as a .anim file.

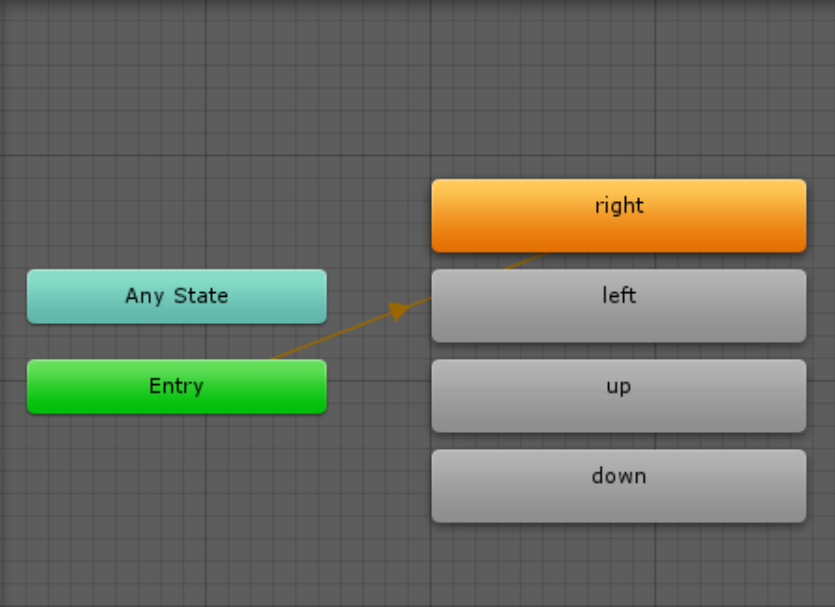
Do the same for Pacman\_3, 4, 5 (**left**.anim), Pacman\_6, 7, 8 (**up**.anim) and Pacman\_9, 10, 11(**down**.anim).

Post this, you’ll notice Pacman\_0, Pacman\_3, Pacman\_6 and Pacman\_9 in the Hierarchy tab. Only Pacman\_0 is needed for the next few steps, so do clean up by deleting Pacman\_3, 6 and 9. You’ll observe a similar pattern in PacmanAnimations within the Assets folder. Again, we only need Pacman\_0, so delete the other 3 (take care to NOT delete the .anim files).

**Mecanim**

In our hands, we have 4 animations, but Unity currently has no clue what animation to play and when. The solution to this problem has already been referenced before (hence the familiarity if you have read through the introductory part) and it goes by the name of Mecanim.

Double-click on the Pacman\_0 in the Assets/PacmanAnimations folder. Mecanim opens up. Observe: There’s an entry point predefined which shows that the animation starts with right.anim as soon as the game loads. But that’s it, there’s no other animation state there. So we create a “left”, “up” and “down” state by dragging each .anim file into the workspace.



Our Pac-Man will only have 4 movement directions (up, down, left, right). Which means that we only need the following 4 Transitions in our animation state machine:

- If DirY > 0 then go to up (like DirY=1, DirY=2, DirY=3 and so on)

- If DirY < 0 then go to down (like DirY=-1, DirY=-2, DirY=-3 and so

on)

- If DirX > 0 then go to right (like DirX=1, DirX=2, DirX=3 and so

on)

- If DirX < 0 then go to left (like DirX=-1, DirX=-2, DirX=-3 and so

on)

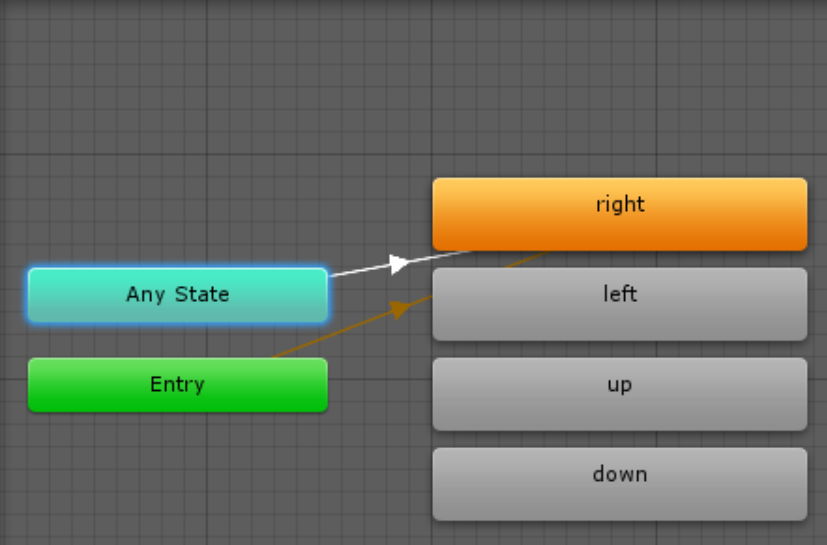
In the parameter tab of the animator window, click on the ‘+’ on the right and two parameters: DirX and DirY

We want Unity to automatically switch to different animation states based on those parameters. Transitions are used to achieve just that. For example, we could add a Transition from left to right with the Condition that DirX > 0. However it's considered best practice to have a small error tolerance because floating point comparison is not always perfect, hence why we will use DirX > 0.1 instead.

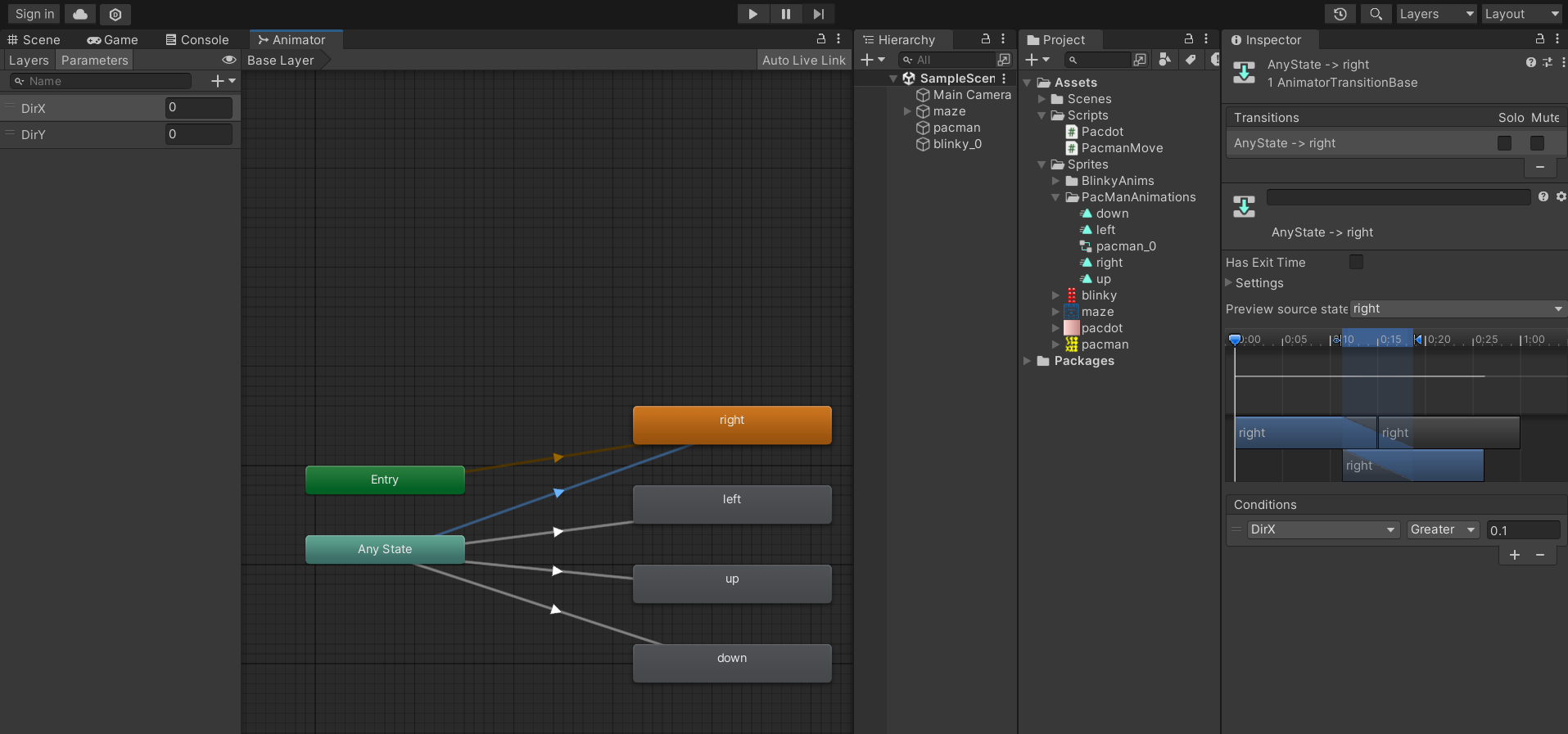
Now we would also have to use a DirX > 0.1 Transition from every other state to right. To save us from doing all this work, we can use Unity's Any State.

The Any State stands for literally any state. So if we create a Transition from Any State to right then it's the same as creating a Transition from left, up and down to right.

Let's right click the Any State and select Make Transition. Afterwards we can drag the white arrow onto the right state:

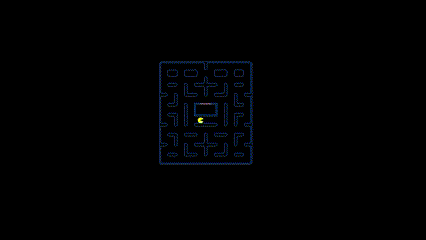


Click on the white arrow and check the inspector tab out. We need the right to play when DirX > 0.1 so set the condition as follows:-



Also, disable Transition to self in the Settings dropdown on the inspector Window for all the animation states. Also set a suitable speed for each state, say 0.5-0.8.

Click on the play button and you’ll see the animation playing. Congrats! You just animated your first Sprite.



**DIY~ Do It Yourself**

Let’s recap what we accomplished so far: We made a Map sprite, customised it, centred it, made a Pacman sprite and animated it using Mecanim. And this marks the end of Mecanim. However, Pacman needs movements as well, and this can only be done with a C# script and some code that you might use Google for. Moreover, generative AI such as ChatGPT is **not** recommended since ChatGPT requires extensive context and a Unity project cannot be summarised in less than a paragraph.

**Activity 1** **[Optional]**

Try making a script that allows Pacman to move around.

Hint: Use RayCastHit to check for obstacles (borders, walls etc. Recall that we made multiple BoxColliders around each wall) (CircleCollider has to be added to Pacman with an appropriate radius), and then play with Pacman’s current position and the destination it needs to reach with a certain speed (in decimals, try starting from int speed = 0.3f;)

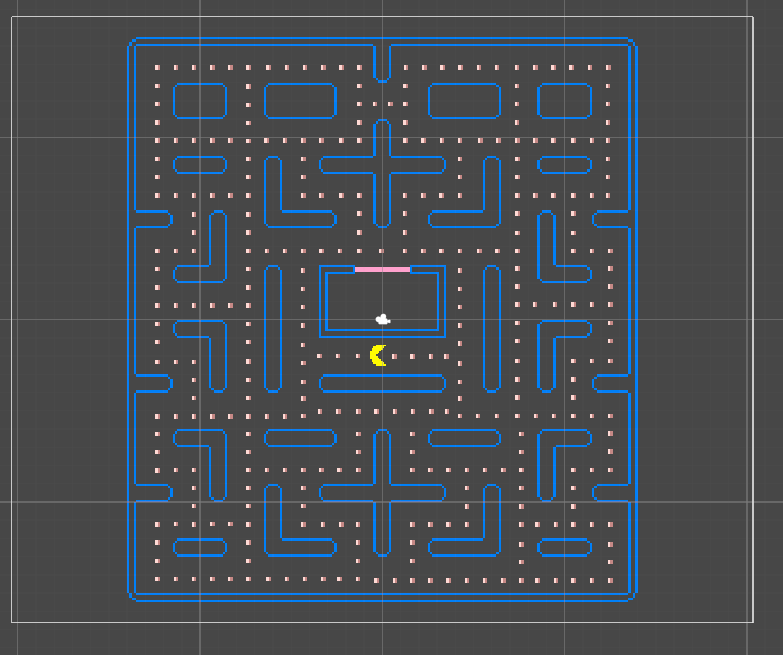
**Activity 2** (This one’s way simpler, but also a bit cumbersome)

Use [this image](https://noobtuts.com/content/unity/2d-pacman-game/pacdot.png) as a sprite. This will be the collectable dots/particles for Pacman. Import it as we did for the Maze and Pacman, then figure out how you can go about adding it to the maze in an equidistant manner

Food for thought: Can you try making these dots destructible, making it seem like pacman is eating them? Investigate OnTriggerEnter2D(). **If you are going for this, go for this first before copy pasting the collectable. This way the script (component) gets added on to each and every collectable.**

(Hint: You do not have to brute-force it completely. Exploit the symmetry of the map. Once you have a certain array of dots/collectables, see where else it’s reusable on the map. Get efficient, get creative and copy paste the same array of equidistant collectables to fit the gaps of same or more lengths on the map)

Your final result post Activity 2 should look something like this:-



Your output post activity 1 will run like this:-

