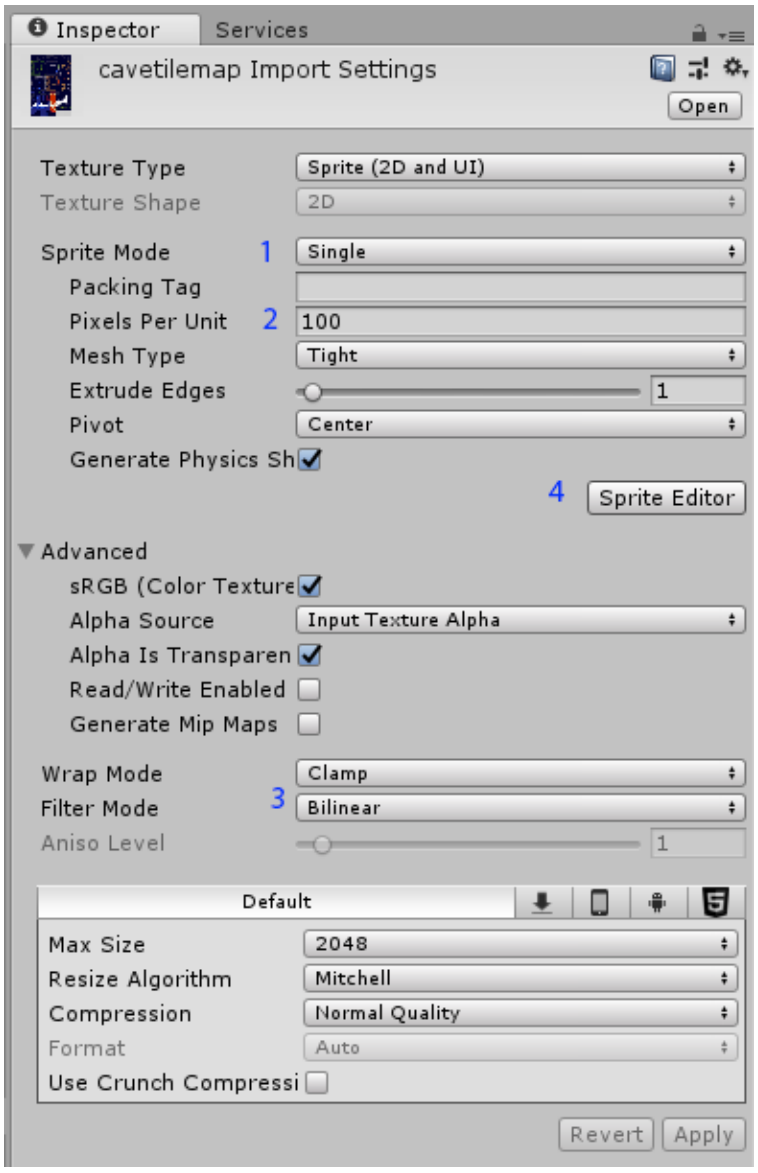
Tilemaps Lab

Tilemaps is a Unity asset that allows you to quickly create new maps and levels for games that use 2D assets, such as 2D platforming games. Tilemaps is still relatively new, so while certain assets, such as Tilemaps, are built into Unity, other assets, like Rule Tiles, need to be downloaded separately from Github.

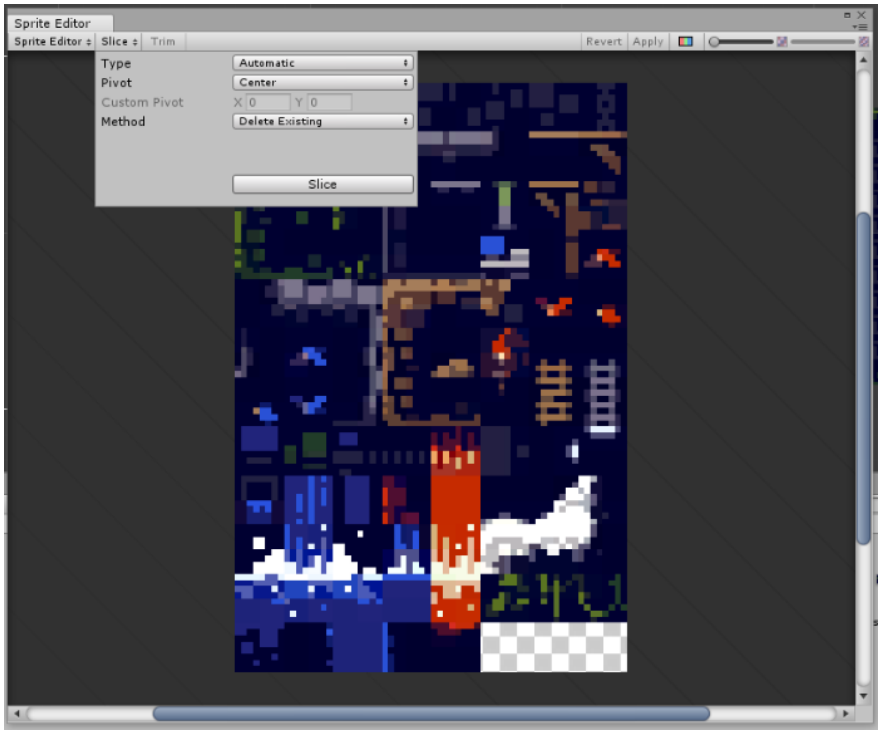
To start, since Tilemaps require art assets, we are going to import some tile maps to work with. Inside the zip folder, there should be some sprite sheets. We are going to import all of them into the Sprites folder. To use images as sprite sheets for the tile map, we have to do a few extra steps to split up the images.

For the first part of the lab, we will be using the cavetilemap.png, click on it to see this in the inspector.



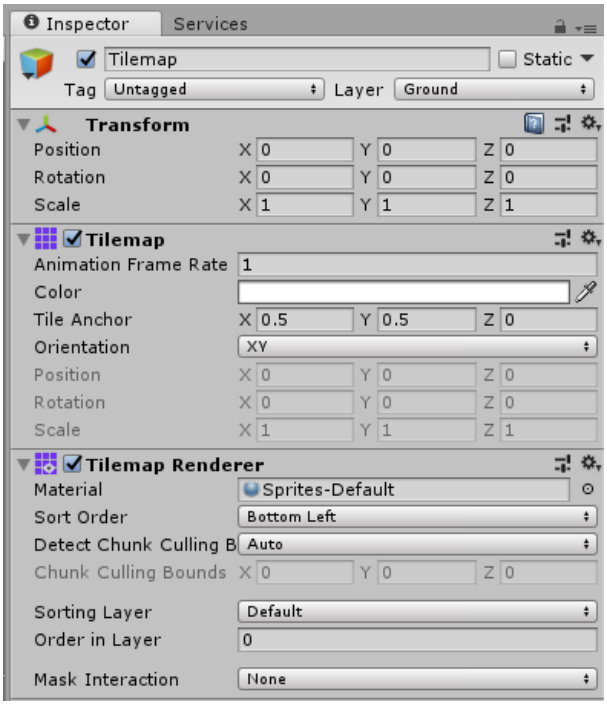
First, we need to change sprite mode(1) to Multiple so that we can split up the image into multiple tiles. Since this image is a bit small, we need to change the pixels per unit (2) to 8. To maintain the crispness of the pixel art, set Filter mode (3) to Point(no filter). Click on apply to apply the import settings to the image.

Now, to actually split up the tiles. Click on (4) to access the Sprite Editor. It will open the Sprite Editor which we can use to slice the image.



In the top left hand corner, there is the slice button that will slice spritesheets. Usually, keeping the slice setting to automatic is fine, but for this sprite sheet, we need it a bit smaller. **Set the cell size to 8x8 and slice the image.** This should slice it up into each tile.

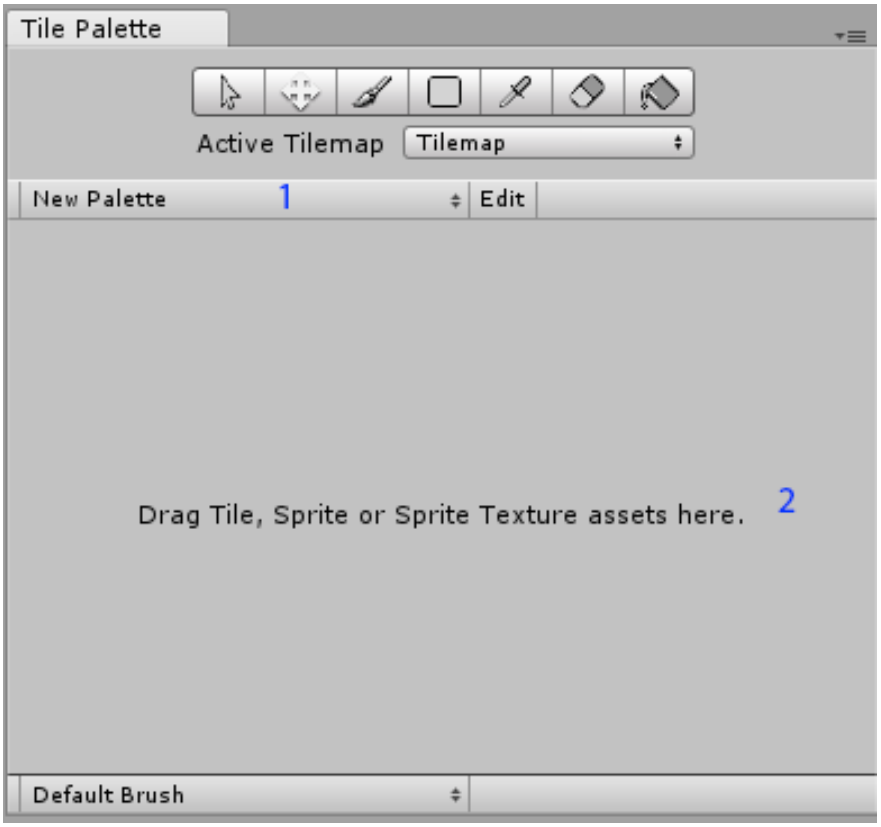
Now we can create our Tilemap. In the hierarchy, right click and create a new 2D objects. **Select the Tilemap option**. It will create a Grid that will layout the format for our tiles. Inside the Grid game object, there is a child called Tilemap. That is going to be the Tilemap that we will be working with. For now, we can rename the Tilemap to be Ground, since that will be the first thing that we will be making.



The Grid is essentially a UI grid, where it stores objects in each cell and the Tilemap is made up of 2 different componenets: Tilemap and Tilemap Renderer.

On the Tilemap component, there are variables to control how it appears. The tile anchor determines how tiles are positioned relative to the grid spaces. The orientation determines the x and y direction. The Tilemap Renderer is similar to normal Sprite Renderers.

To actually work with the Tilemap, we need a new window. Under the windows bar, open the tile palette.



Tile palettes are a collection of all Tiles. This is where all of the Tiles will be kept and largely for organizational purposes. Tiles are objects that make up the Tilemap. They hold the sprite that will be shown, as well as the positioning of that sprite. This is similar to a sprite renderer of a game object. A tile palette can be used on multiple Tilemaps and a Tilemaps can be made up of multiple tile palettes.

Create a new tile palette by pressing on (1) and name the palette Grass. There is an area where you can place new tiles and select them to put them into the scene. The tile palette has several options as to how to can paint on the tiles into the scene. On the toolbar, there are several options. From left to right:

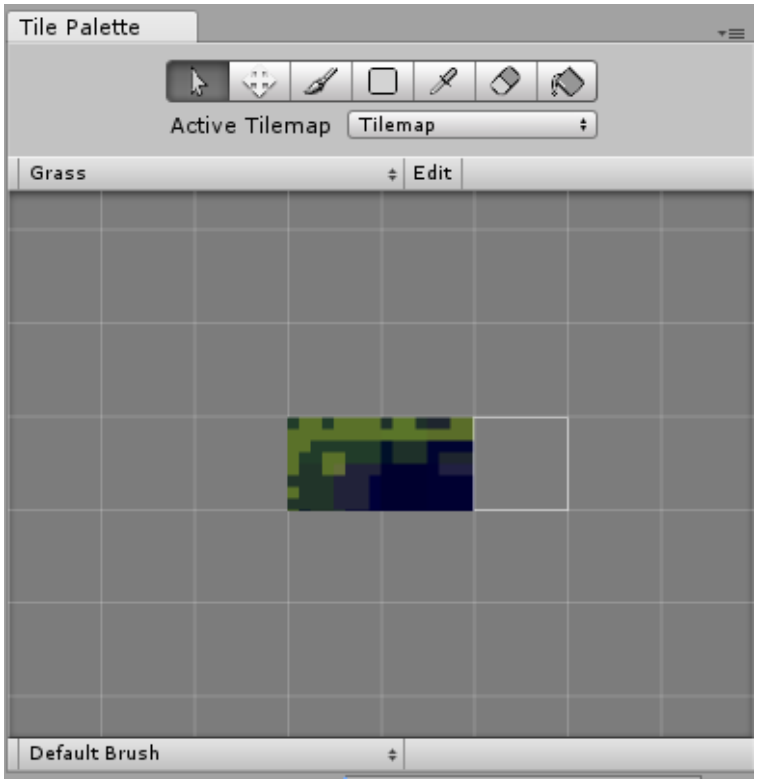
1. The mouse button: inspects the tile and allows you to edit Tile properties
2. The selection button: can select and move tiles
3. Paint brush button: the paintbrush allows you to paint onto the scene with the tiles that you want to be placed into the scene
4. Paint filled box: will fill the area within a box with the selected Tile
5. Dropper: will select the Tile of whatever is selected in the scene
6. Eraser: erases the tiles painted into the scene
7. Paint bucket (flood fill): will fill in the inside of a box that has a boundary

The image is a bit small, but using some naming conventions, hopefully it will be a little easier. There is also an image that is much larger in the zip folder. We are going to set up a grass tile palette. In the sprite sheet we just split up, we are going to use cavetilemap image 8. Drag the sprite from the project window into the blank space in the tile palette (2 in the previous picture). This will automatically create a new Tile object using our sprite. We can create this into a new folder called Tiles. **Name it ”8” so that we can**

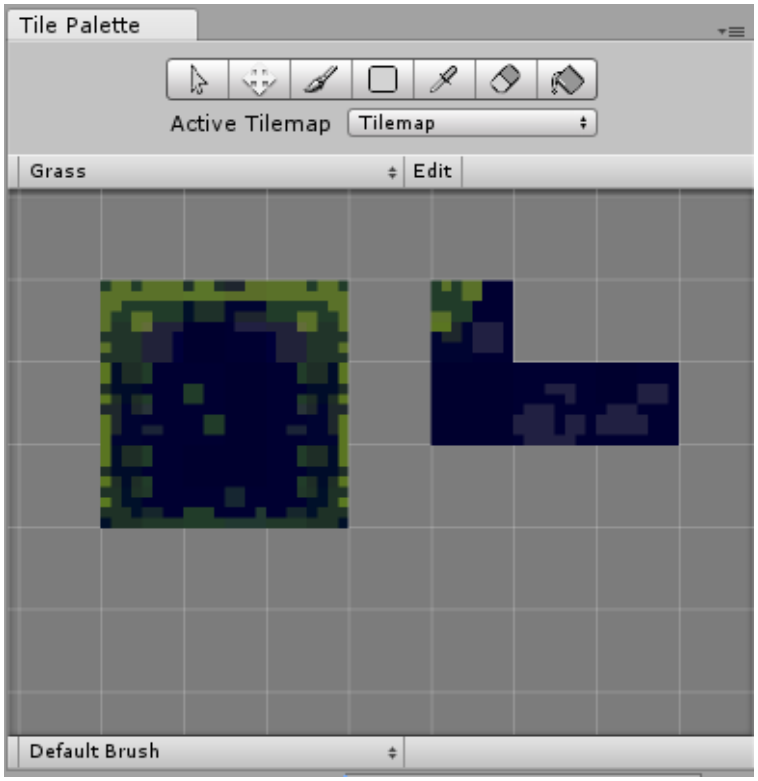
access it easily later.

Now we can see that we have our corner tile in the tile palette. If we click on the paintbrush and click and drag in the scene, we can see all of the corner piece fill the tiles we drag by. Clicking the eraser and doing the same thing will remove it. So let's put a few more tiles out so that we can complete the set of grass tiles.

The next tile that is going to be placed is cavetilemap 9. For organization, we are going to put it next to tile 8 so that the tile palette looks like this:



You can name this tile just by 9 too, just so that it is easier to get to. The next tile is going to the the opposite corner tile. If you look at the actual sprite sheet, it doesn't actually have a piece to create a square tilemap like this:

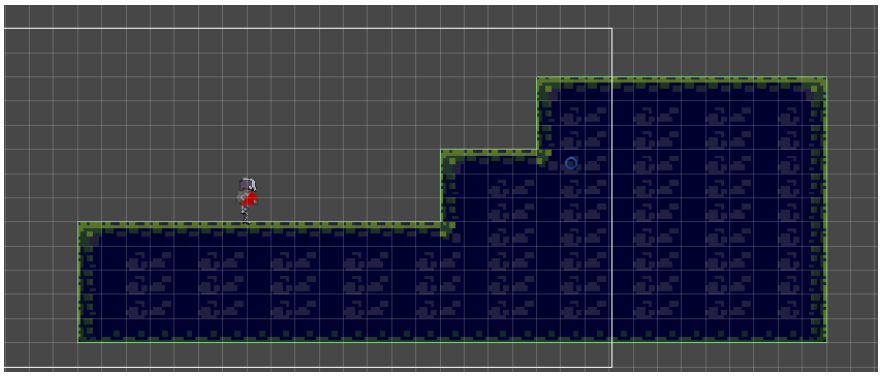


So, we are going to reuse tile 8. Now, since tile 8 has already been made, we can just drag that into place next to 9. However, this one is flipped in the wrong direction. So, if we click on the tile while the mouse button is selected and get the tile properties, we can see the position, rotation and scale. Similar to other sprite renderers, we can use this to change the image to flip it around. Under scale, **change it to -1 and the sprite will flip.** That finishes off this row quite nicely. Now, fill in the rest of the grass palette so that it looks like the picture above. The numbers for each tile is:

8 9 8’
16 0 16’
24 25 24’
where ’ means the flipped sprite

Off on the side, there are sprites that we will use to fill the insides of the ground, so those are 0 1 2 17.

Now, we can paint in the grass ground for the level. Select the paintbrush, click on 8, and place it into a box on the screen. Click 9, click and drag along a straight line. Click 8’ and place at the end of the tiles. Click on 16 and then draw a couple of squares down under the tile for 8. Do the same for 16’ on the other side. Click on 24 and make the corner. Do the bottom boundary for 25 and fill to teh end. Fill last corner with 24’. Click on the paint bucket button and select 0, then hover over the middle and fill it in with the tiles. You don’t have to match this, but this is a reference if you want something to go off of for the level:

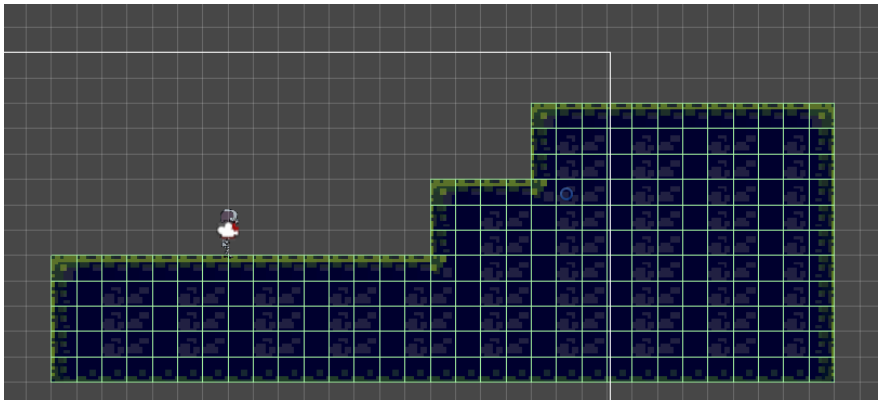


You can also use the box fill button to fill up the space so that we don't have to fill in the whole box by hand.

We can also select multiple tiles to put into the scene at once. If we select all 3 tiles: 0, 1 and 2, we can "stamp" them onto the scene. Using the paint bucket, you can fill an area with these "stamps" without having to do each one.

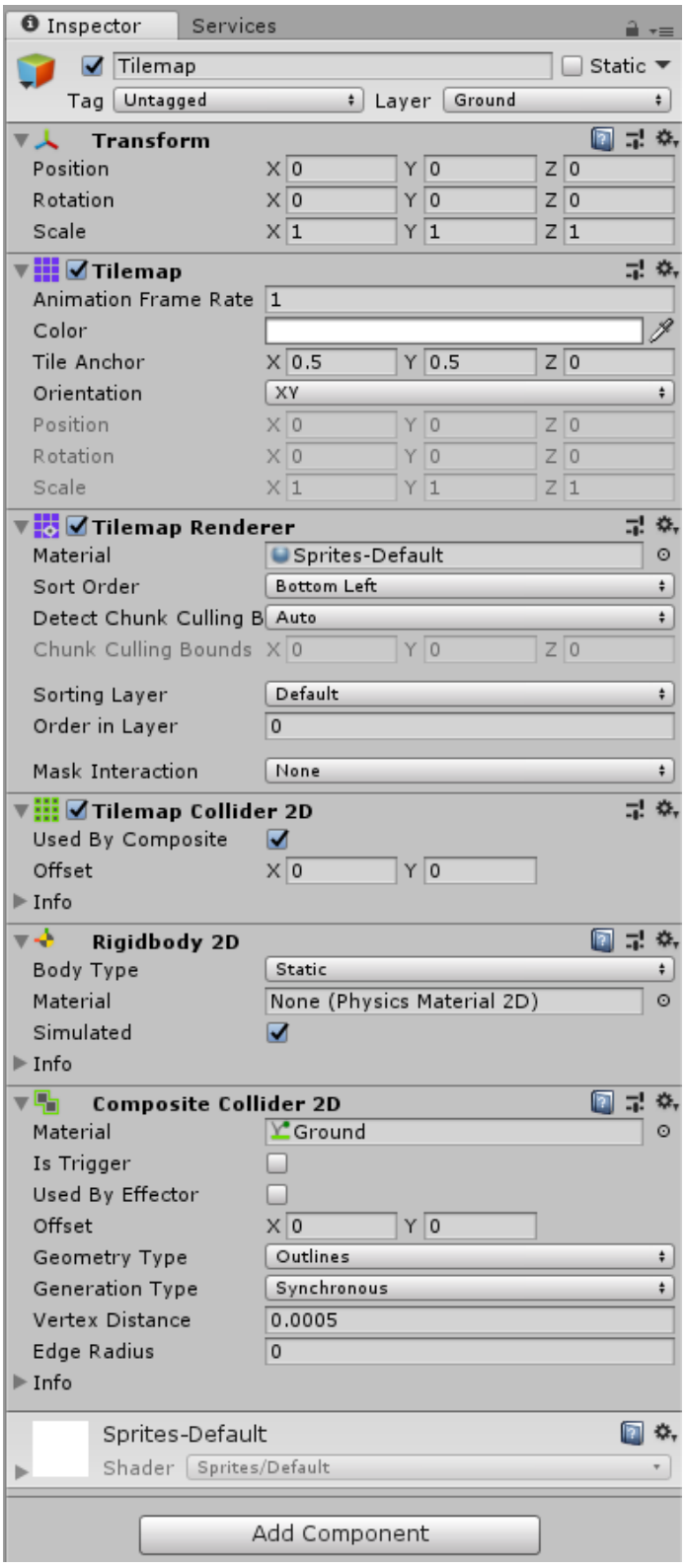
Hit play now, but the player fill fall right through the ground. That is because there aren't any colliders on the ground right now. To do that, on the Tilemap gameobject, add a Tilemap Collider 2D component.

Looking at all of the objects, each individual Tile has its own collider.

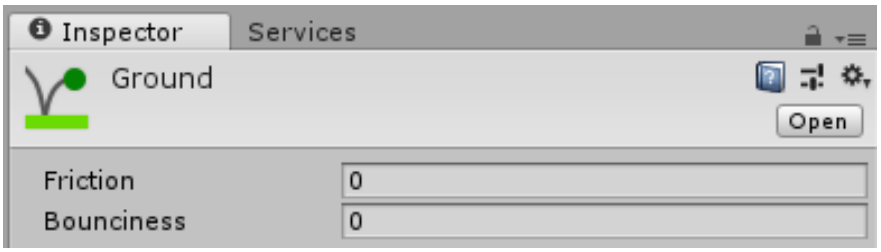


We can reduce all of this to just the outline of the large block by **also adding a Composite Collider 2D onto the Tilemap gameobject.** By adding this, the colliders will only on the outside of each large block. This also adds a RigidBody2D onto the gameobject. On the RigidBody2D, **change the body type to be static.** We don't want this to be affected by gravity.

So now, the Tilemap gameobject components should look like this:



Play it now and the player can run along and jump on the platforms! However, the player will stick to the walls if you jump into a wall. To get rid of this, we need to create a material for the CompositeCollider2D. In the project window, right click and create a new Physics Material 2D. For organization, create a new Materials folder to place this in.



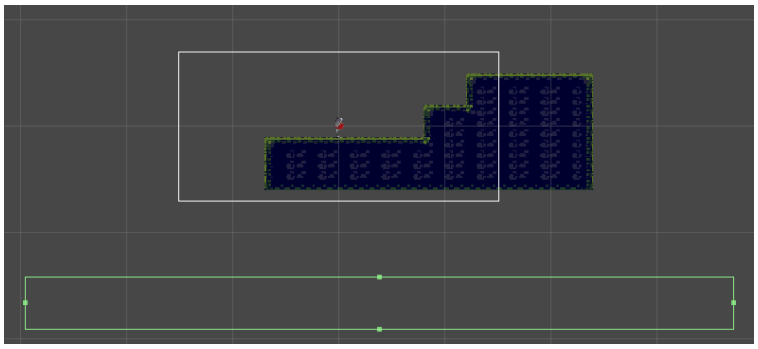
In the material, there are 2 options. Friction determines how much the player will stick to an object (like the player sticking to the wall). Bounciness determines how, well, bouncy the collider will be. To stop the player from sticking to the wall, **set the friction to 0.** Then add the material onto the composite collider 2D component on the Tilemap gameobject.

Great, now playing it feels pretty good. Unless we run off of the edge. Firstly, the player won't die and secondly, the camera will follow the player forever. To solve this, we have some assets that we can take advantage of. As you may have noticed by now, the player has a Cinemachine Virtual Camera attached to it. This allows the camera to follow the player. If you remember from the very first lab, it also allows you to set limits for how far the camera will follow.

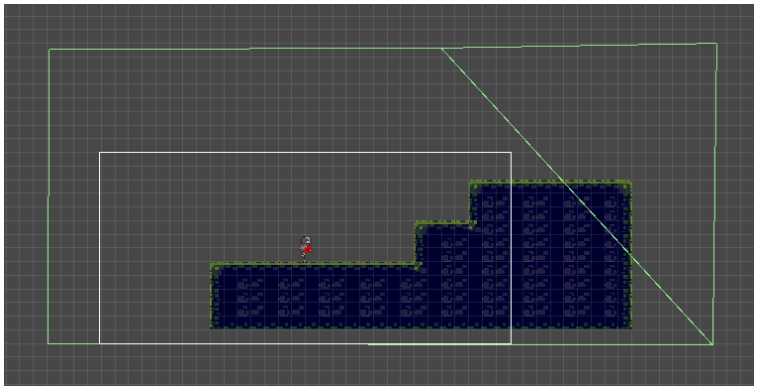
The player character was imported using some built in Unity assets that come with alot of really handy scripts and capabilities. Once that came with this 2D character was a kill zone in case it falls off into a pit. For reference, this particular asset came from Assets -> Import Packages -> 2D.

So inside the folder named "Standard Assets", you will see a file for "2D", which is where everything we want is. Go inside the prefab folder inside there and one of the prefabs is Killzone. This is a collider that will allow the player to automatically die when colliding with it.

To add it, just click and drag the prefab onto the scene. Looking at the in the inspector and clicking on Edit Collider in the Collider2D componenet will show you exactly where the collider for the kill zone is. Adjust it according to how you have build your level. Mine is here:



Now, if you play, if you fall off, the character will die and then respawn where they were originally placed. However, the camera will follow you all the way down. To make it look nicer, we can use Cinemachine 2D Confiners to set the bottom limit for the camera. The polygon collider for the confiner should look something like this:

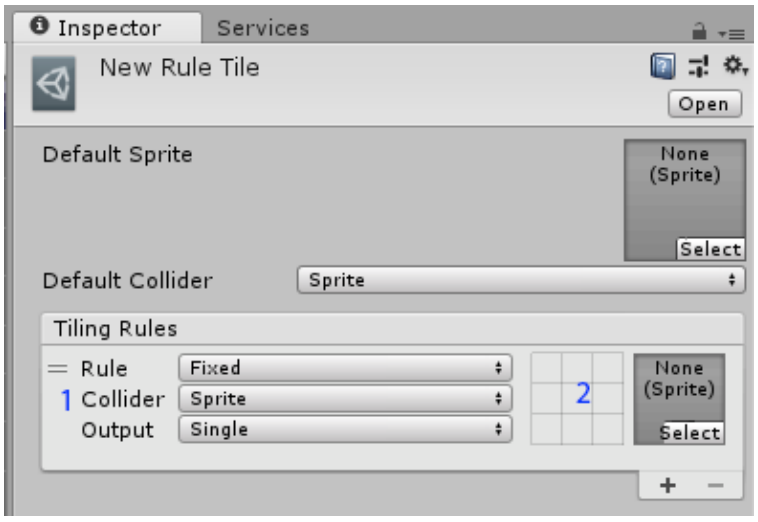


Don't forget to make the polygon collider a trigger!

Great, now it looks nice and it plays well. But doing all of the tile painting by hand is a bit tedious. Thankfully there is a way to set up tiles based on a set of rules. Rule tiles are essentially a tile that have a set of conditions so that Unity will automatically place tiles into the scene based on if there are tiles next to them.

To use this though, we are going to need to import some assets from a Github repo. For this lab, they are already imported into the project, but if you want to use them later on, they can be found here: https://github.com/Unity-Technologies/2d-extras. Inside the Tilemap folder, there should be a bunch of assets that help us create Rule Tiles. There are also alot of other assets within this package that we wont cover in the lab, but it's definitely worth taking a look at later.

So inside the Tiles folder, we are going to create a Rule Tile, and we can name it Stone because now we will be using the stone tiles. If you right click, there should be an option to create a Rule Tile. With the Stone Rule Tile selected, in the inspector, you can see that we have a default sprite and a list of conditions that will determine the sprites that will be used.



When using Rule Tiles, you don't actually need to create new tiles, so we will just use the sprites to add onto each set of rules. For the default sprite, let's use image 0 from cavetilemap. Click select for default and it will open up a new window. You can just search 0 and it should be one of the first to show up.

So now, we can start creating rules. If you click the plus, it will create a new rule for you to set. There are a few properties that we can use. (located at 1)

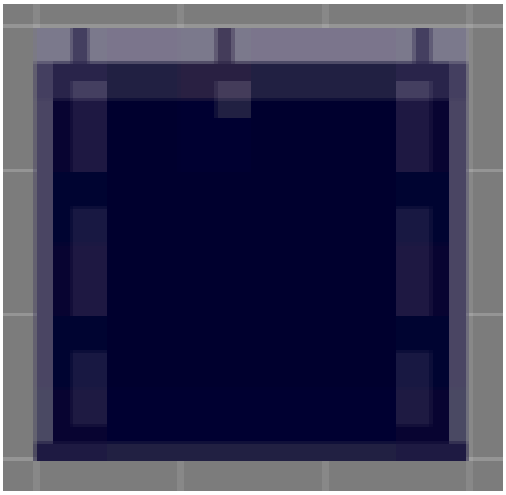
1. Rule is for how we want that particular rule appied. For example, we can make it so that a rule can be applied when also turned 90 degrees, flipped on a particular axis, or just only for that specific rule. If that seemed, confusing, I'll go over it with an actual example in a bit.
2. Collider is just the type of collider that should be attached to it. For now, we can just leave it as sprite.
3. Output is how we want to be painting that specific rule onto the scene. The options are single, which just gives you the single image, animation, which will cycle through sprites to create an animation, or random, which will randomly pick a tile based on the tile that are given. We will be going through all of the outputs in this lab

Output is how we want to be painting that specific rule onto the scene. The options are single, which just gives you the single image, animation, which will cycle through sprites to create an animation, or random, which will randomly pick a tile based on the tile that are given. We will be going through all of the outputs in this lab

In the grid (2), all boxes outside of the middle one can be set with a condition that applies to the tile spot relative to the tile. If you click it, it will set it to a different setting. When it is an X, that spot must be empty. When it have an arrow, that spot must have another tile from this Rule Tile. When it is blank, it does not matter if there is something there or not. For example, if the top left box has an X, then this specific tile will be placed only if the top left corner next to this tile does not have a tile on it.

It is important to note that tiles from other tilesets will not be counted when seeing if there is a tile present in the spot adjacent to a tile.

So, let's make our first set of rules. We want it to look like our grass tiles in the style of the tile layout, so it should look like this:

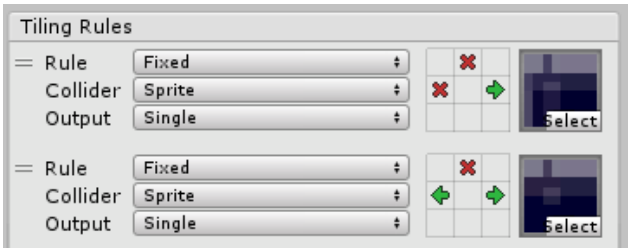


The sprite that we will need are 11 for the upper corner, 12 for the upper middle, 19 for the middle boundaries, 0 for the middle, 27 for the bottom left and 28 for the bottom middle.

For the first corner tile, select cavetilemap_11 as the sprite for the first rule. Then we can start with the conditions. Since we want this tile to show up when there is nothing on the left and top, mark those with Xs. You can change the icon in the grid by clicking on that specific part of the grid. We also want to make sure that there blocks to the right of the corner block at least. In theory, we also want to require there to be a tile underneath it, but if we have a platform of just one layer of blocks, we want the top layer to be the layer that is shown. It should look like this:

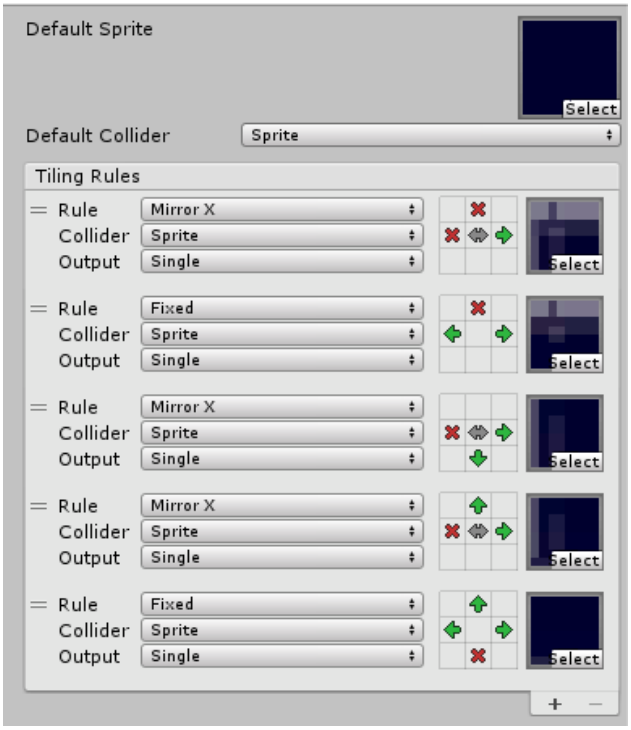
Now, let's add another rule. Click the plus button and add a new rule. A thing to note is that rules are processed from the top down. If a tile fits into a category, it will be pick that one even if it fits into a category that comes later in the list.

So for tile 12, we want to make sure that it shows up when there is a tile to the left and right of the tile. We want to make sure the top is empty. So now it should look like this:



Now for the opposite corner. Here, our work is cut out for us. Previously I mentioned the Rule section determines how the rule can be applied. The top right corner is essentially the same (in terms of sprites and the conditions for it) as the top left corner, just flipped on the x axis. So we change the Rule of that tile to so that when the conditions for the top left corner are satisfied, but for the top right corner (flipped on the x axis), it will flip the sprite along the x axis as well. So we can change that by clicking on Rule and changing it to Mirror X, or click on the center of the grid until you see the arrows pointing left and right.

Now, if you try to paint something, you can see that it will automatically determine what type of tile should be placed there. And that the top right corners are being put in place just by mirroring the rule for the top left corner. Great! Now fill in the rest of the Rules so that when you draw on the scene, you get a result that looks like this:

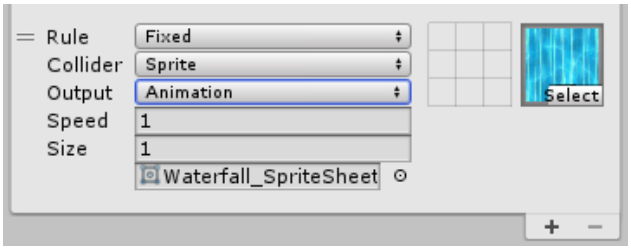


This looks great, but the middle seems a bit plain with just the blank default tile. So, let's change it up with random Rule Tiles. We can add a new tile onto this ruleset. For this one, we want to change the output to Random instead of single. This will create some new parameters: Noise, Shuffle and Size. Noise determines how random it is, shuffle is how it can change the sprites (like flipping on an axis), and size is how many random sprites you are giving it.

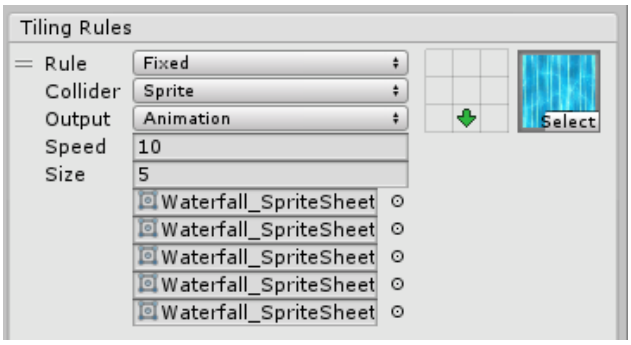
So, set the number in size to be 3. The 3 sprites we are going to use are 0, 1 and 2, so put those into the 3 slots. We can set shuffle to Mirror X so that we get a bit more variation. You might notice, that these changes are happening as you are making settings, so that is definitely very helpful to see how changes work out. Now, you can slide the slider for the noise and see what you think looks good.

Great, now we've been able to make levels easily, let's make something a bit more pretty. We can also make animation tiles using Rule Tiles. For this though, we need a new tile set that was made specifically to be a waterfall animation. If Waterfall_SpriteSheet.png hasn't been imported yet, import that. Again, we will have to change some settings for it. For this one, the pixels per unity is 128, and when you slice it, the size of the cell is 128x128.

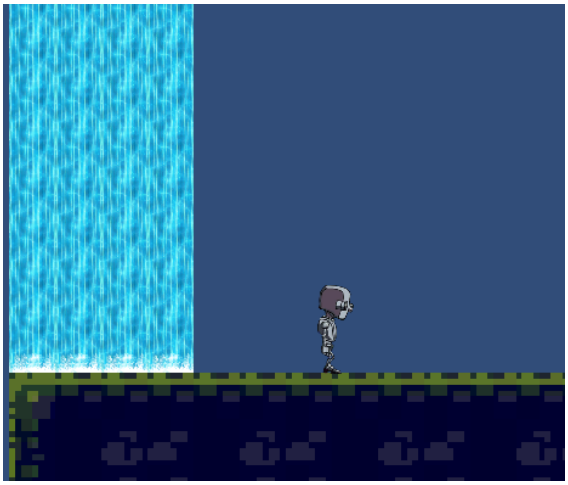
Now, let's make a new tilemap, since we probably don't want to add colliders onto this layer. So make a new Tilemap named Scenery. Create a new Palette and a new Rule tile for the waterfall. So these conditions are rather simple. We just need to check if the tile is the bottom-most tile, we'll place the tiles with the splashes there. So the first condition is making sure that there is a tile directly underneath it. Then, change Output to Animation. Then, that will create Speed and Size variables.



Speed is just how fast it cycles through and size is how many images there are in the animation. So, set Size to 5 and place all of the images in numerical order into the slots under size. So this rule should look like this:



If you place them onto the map and press play, you can see that they will cycle through the images, but a bit slow. Change the speed to 10 and it is a lot better. So we need the bottom of the waterfall as well. **Add a new condition for the bottom of the waterfall.** Now, the tiles that were originally painted are already automatically updated to reflect the new rule. Since pdf technology hasn't advanced to hold animations, I can't really show you, but this is what a still image looks like



That's the basics of Tilemaps. For programmers, there is plenty more that can be done. Rules can be scriptable so that more conditions (other than just if a tile is present or not) can be applied. Paintbrushes can even be scriptable to create more prefabs or create other properties. For people who aren't programmers, the package of assets contains plenty of other tools to use for tilemaps.

A few notes about spritesheets in general, usually sprites will require some tweaking to be able to fit it into the grids of the tilemaps, to have it look good or line up with other tiles if you have smaller details you want to add in using tilemaps. It typically is just tweaking the pixels per unit or changing the position, rotation or scale of the Tile. It will take some time and needs finicking so don't worry if something doesn't just fit right away.

Checkoff:

1. Create a level with platforms containing the palette with the bricks, grass and stones (bricks being the set of floor tiles that were not used for the lab(images like 35, 36, 37)). The level must contain Rule tiles with animations and randomness.
2. Place some animated birds, torches in the background
3. Place background tiles, such as vines, or the waterfall made in this lab, but make the tiles slightly transparent

Challenge:

1. For all: Find a new spritesheet of a different size (so tiles are not 8x8), import it and create a level. Some good websites to look through are https://itch.io/game-assets/free/tag-tilemap or https://opengameart.org/content/best-orthogonal-rectangular-tilesets-for-tilemaps
2. Programmers: create some scripted rules and create some new tilemaps using more specialized rules
3. Artists: create a sprite sheet that can be placed into a tilemap