2D Experimental Preview Release 1 Reference Guide

Contents

9-Slice

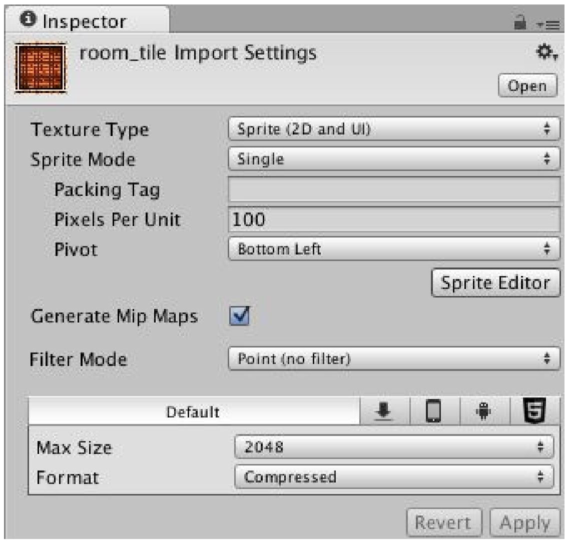
Overview

9-Slice is a 2D technique which allows you to reuse an image with variable dimension without preparing multiple assets. This requires user to define areas on the image that can be stretched or repeated and areas that needs to be in constant size. With these areas defined, when an image dimension changes, it will stretch/repeat the defined areas making the image usable in multiple dimension

9-Slice Sprite

To have a 9-Slice Sprite, first you will need to define the borders of the Sprite via the Sprite Editor Window.

1. Bring up the Sprite Editor Window by clicking on the “Sprite Editor” button from the asset’s inspector



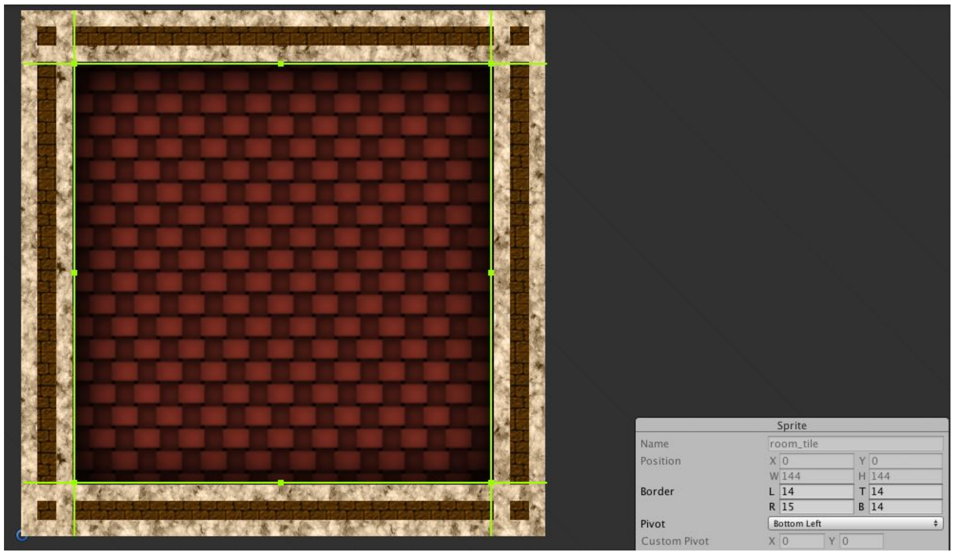
Inspector
room_tile Import Settings
Open

Texture Type: Sprite (2D and UI)
Sprite Mode: Single
Packing Tag:
Pixels Per Unit: 100
Pivot: Bottom Left
Sprite Editor
Generate Mip Maps: ✓
Filter Mode: Point (no filter)

Default
Max Size: 2048
Format: Compressed

Revert Apply

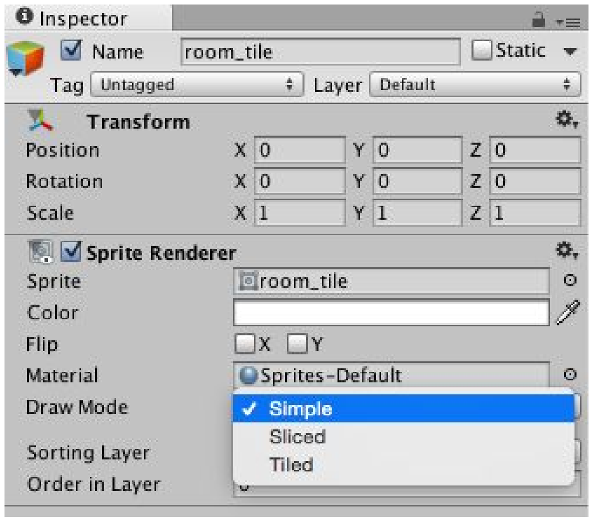
2. In the Sprite Editor Window, you can either define the borders of the Sprite via the property window or by dragging the green dots in the Sprite



Sprite

Name: room_tile
Position: X 0, Y 0, W 144, H 144
Border: L 14, T 14, R 15, B 14
Pivot: Bottom Left
Custom Pivot: X 0, Y 0

3. Click Apply on the Sprite Editor Window.
4. Drag the Sprite into the Scene view
5. In the SpriteRenderer's Inspector, change the 'Draw Mode' property to either 'Sliced' or 'Tiled'; depending on the desired behaviour.



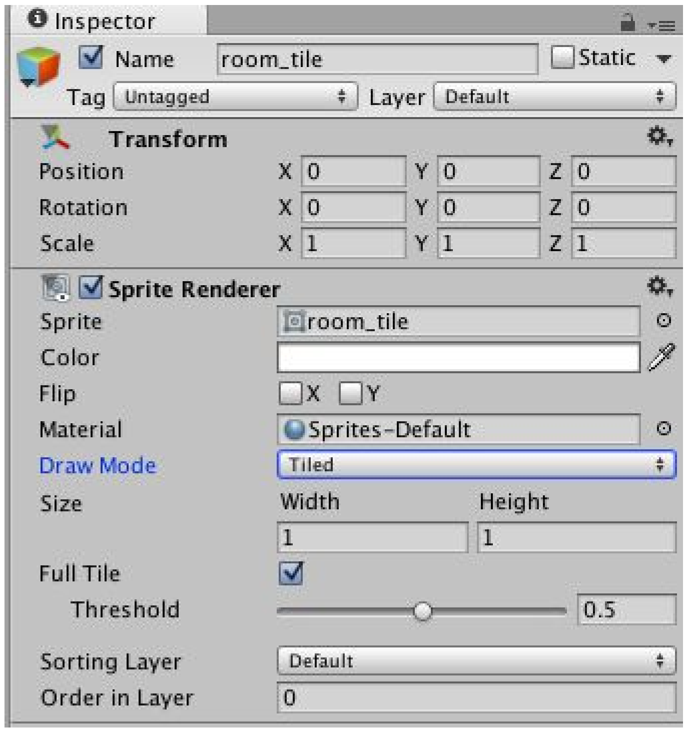
Inspector

Name: room_tile ☐ Static
Tag: Untagged Layer: Default

Transform
Position: X 0, Y 0, Z 0
Rotation: X 0, Y 0, Z 0
Scale: X 1, Y 1, Z 1

Sprite Renderer
Sprite: room_tile
Color:
Flip: ☐ X ☐ Y
Material: Sprites-Default
Draw Mode: [✓ Simple] Sliced Tiled
Sorting Layer
Order in Layer

6. In either Sliced or Tiled mode, there will be extra properties that you can adjust. To change the dimension of the Sprite, you can either adjust it via the Width/Height property or via the Rect Tool.



Inspector

☑ Name: room_tile ☐ Static

Tag: Untagged Layer: Default

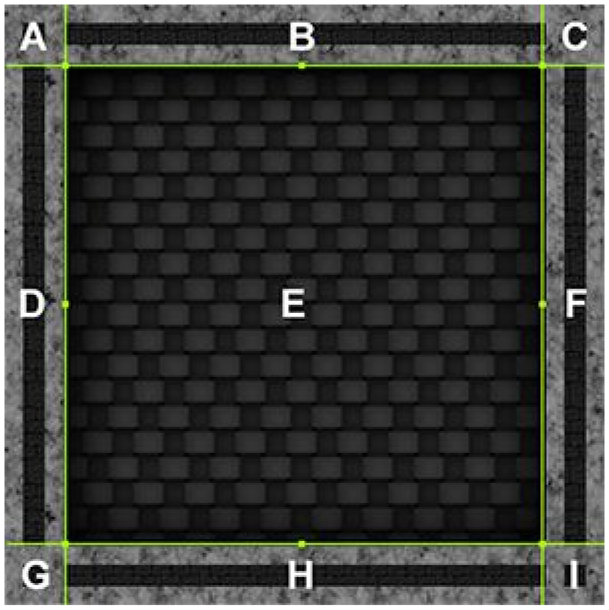
Transform

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

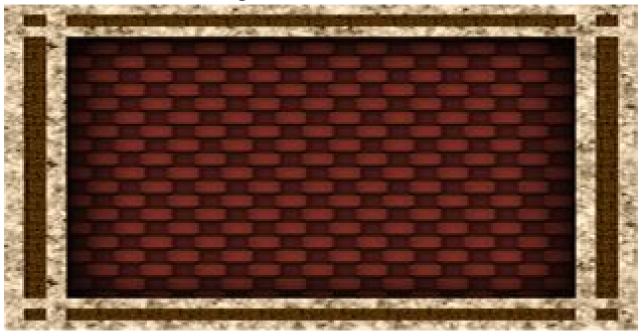
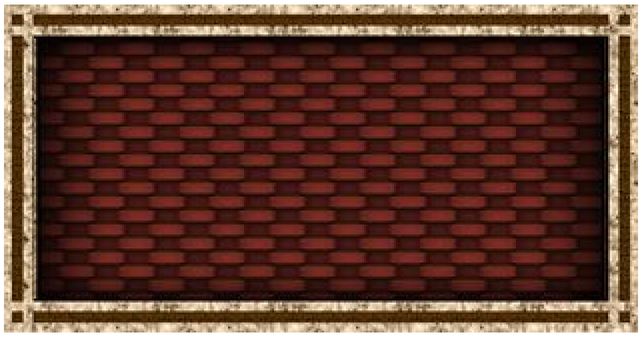
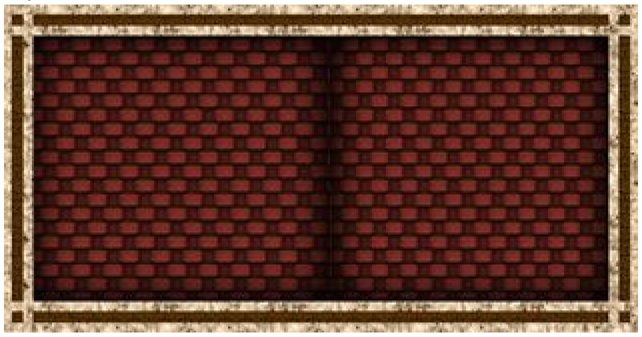
☑ **Sprite Renderer**

Sprite	room_tile
Color	
Flip	☐ X ☐ Y
Material	Sprites-Default
Draw Mode	Tiled
Size	Width 1, Height 1
Full Tile	☑
Threshold	0.5
Sorting Layer	Default
Order in Layer	0

Draw Mode



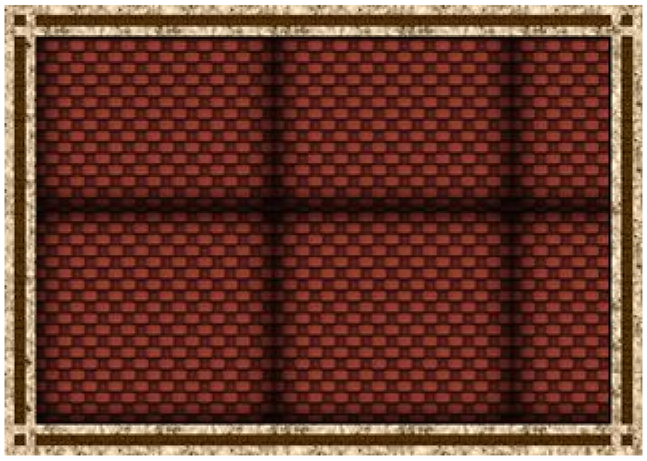
A 9-Slice image is where an image is divided into 9 portion.
The A, C, G and I sections' size will not change no matter what the dimension of the image is.
The B and H section will be repeated horizontally.
The D and F section will be repeated vertically.
The E section will be stretched or repeated both horizontally or vertically.

Simple	This is the standard SpriteRenderer behaviour where the image will be scaled when the dimension is changed. 
Sliced	In Sliced mode, the B, D, E, F, H section will be stretched to fit the dimension 
Tiled	In Tiled Mode, the B, D, E, F, H section will be repeated to fit the dimension 

Full Tile and Threshold

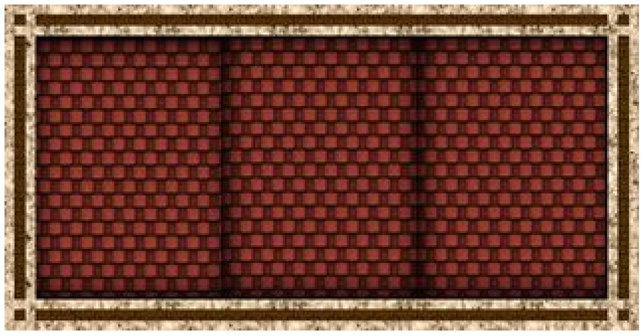
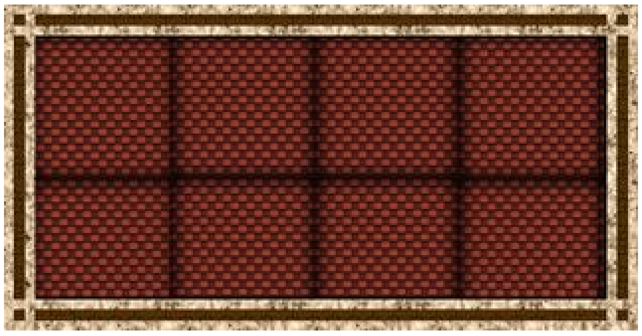
The Full Tile and Threshold properties are to control when the repeatable sections should repeat based on the dimension.

When Full Tile is turned off, the repeated sections will repeat as the dimension changed



When Full Tile is turned on, the repeated sections will only repeat when the dimension reaches the Threshold specified. The Threshold value is between 0 - 1. When Threshold value is 1, the image will repeat when the image is stretched twice the original size.

The following images shows the differences in repeating for an image with the same dimension but different Threshold value

Threshold = 0.5	Threshold = 0.1
	

Limitations

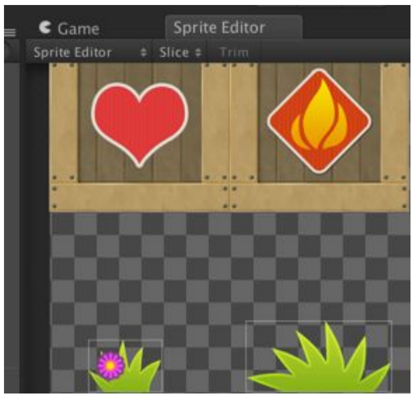
2D Box Collider will automatically reset to the size of the mage when the dimension is changed in Sliced or Tiled Draw Mode.
The other 2D Colliders do not automatically update when in Sliced or Tiled Draw Mode
The previous 2D Colliders behaviours still exist when in Simple Draw Mode.

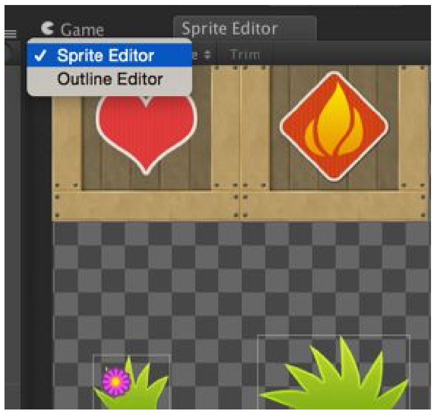
Sprite Outline Editor

Overview

Sprite Outline Editor allows you to edit how the tight mesh for a Sprite should be generated. This feature can be access via the current Sprite Editor WIndow

Sprite Outline Editor



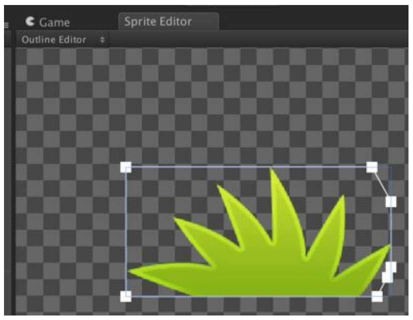


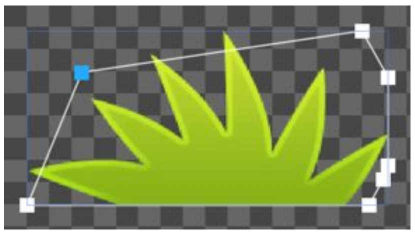
On the top left hand corner of Sprite Editor Window, there is now a new popup menu that allows you to switch between different modes of Sprite Editor Window.

In Sprite Editor mode, it allows you to manipulate/change sprites properties like the previous Sprite Editor Window.

To edit the outline of a sprite, you will need to change it to the Outline Editor mode.
To have a 9-Slice Sprite, first you will need to define the borders of the Sprite via the Sprite Editor Window.

When in Outline Editor mode, the selected sprite will show the control points and the outline of the sprite.



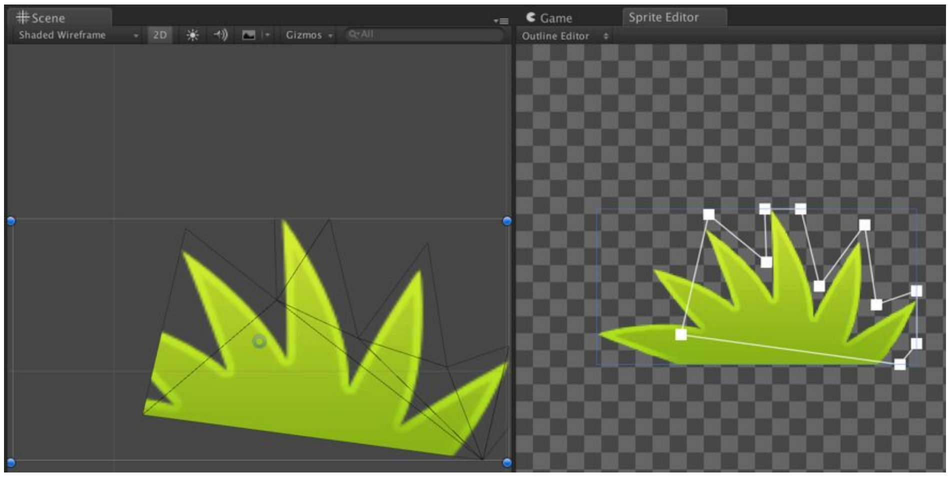


You can click and drag a point to change it position of the point

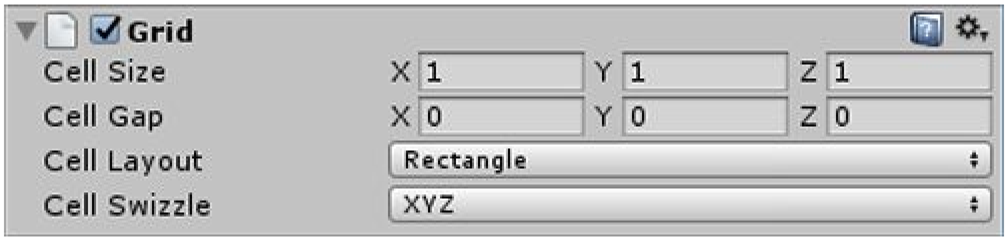
When the mouse is on an outline, a transparent control point will appear. By dragging the transparent control point will create a new control point.

To have multiple outlines, drag in an empty space within the sprite. This will create a new outline with 4 control points.

Once you are done, remember to apply the changes.



Grid


Grid

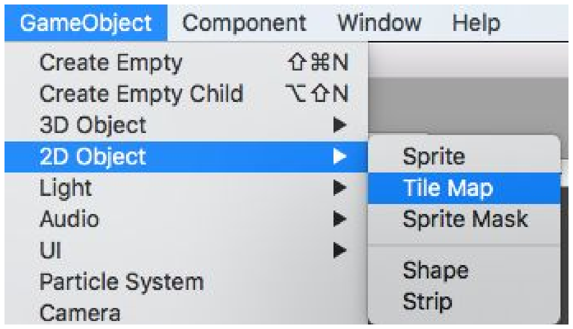
Cell Size	X 1	Y 1	Z 1
Cell Gap	X 0	Y 0	Z 0
Cell Layout	Rectangle		
Cell Swizzle	XYZ		

- Allows users to specify a layout for organisation of object placement
- Supports 3D coordinates
- Provides space conversions from grid cell, local and world
- Supported layouts:
    - Rectangular
    - Isometric
    - Hexagonal
    - Custom (Not implemented currently)
- Adjust cell sizes, gaps and swizzles between coordinate axes

Tile Map

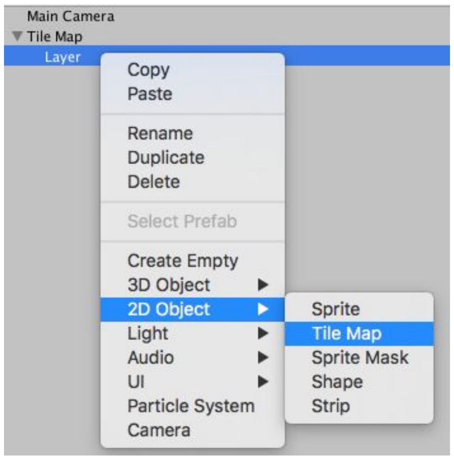
How to create a Tile Map

1. From the GameObject Menu, move to 2D Object and select Tile Map



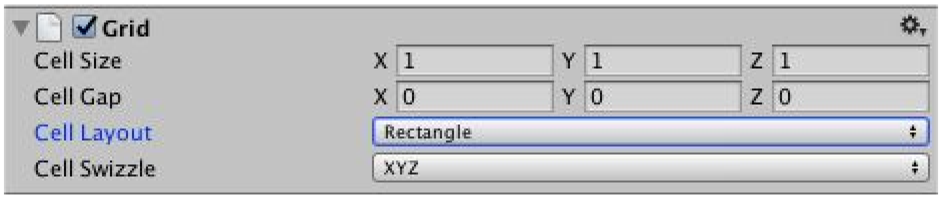
2. This will create a new GameObject with a child GameObject in the scene. The GameObject is called Tile Map and contains a Grid which determines the layout of any child Tile Maps. The child GameObject is called Layer and contains a Tile Map and Tile Map Renderer. The Layer GameObject is where you paint tiles on.

3. To create a new layer in the Tile Map, select the Tile Map GameObject or the Layer GameObject, and click on GameObject/2D Object/Tile Map in the menu or right-click on the selected GameObject and click on 2D Object/Tile Map.



4. A new GameObject called Layer (*) will be added into the hierarchy of the selected GameObject. You can paint tiles on this new GameObject as well.

Adjusting Grid for Tile Map



Grid

Cell Size X 1 Y 1 Z 1

Cell Gap X 0 Y 0 Z 0

Cell Layout Rectangle

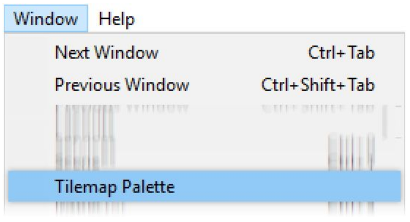
Cell Swizzle XYZ

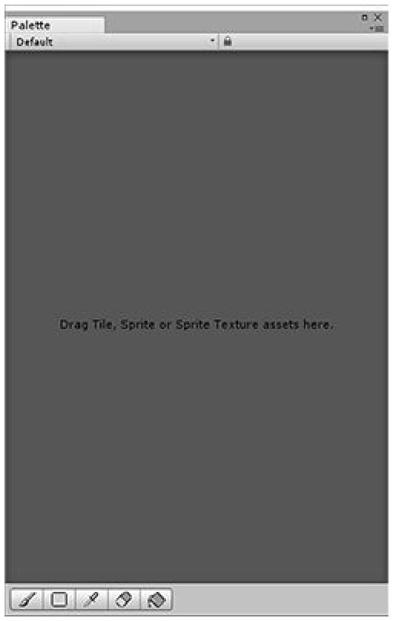
1. Select the Tile Map GameObject with the Grid component.
2. Adjust the Grid component in the inspector.
    a. Cell Size: Size of each cell in the Grid
    b. Cell Gap: Size of the Gap between each cell in the Grid
    c. Cell Layout: Layout of the Grid
        i. Rectangle: Standard rectangular Tile Map layout
        ii. Hexagon: Hexagonal Tile Map layout (Grid visualisation does not work yet, but tiles will be laid out correctly)
        iii. Isometric: Isometric Tile Map layout
        iv. Custom: Work in progress
    d. Cell Swizzle: Swizzles the cell positions to other axii. For Example. In XZY mode, the Y and Z coordinates will be swapped, so an input Y coordinate will map to Z instead and vice versa.
3. Changes in the Grid will affect all child Layer GameObjects with the Tile Map and Tile Map Renderer components.

Tile Map Tiles

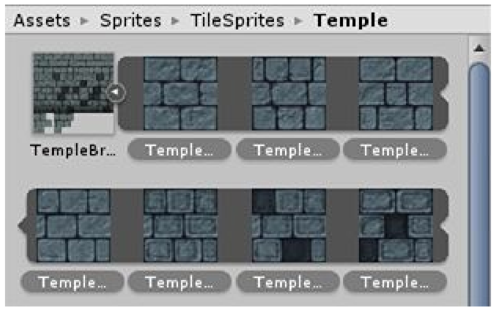
How to create a Tile from a Sprite

1. Open the Tile Map Palette from Window/Tilemap Palette

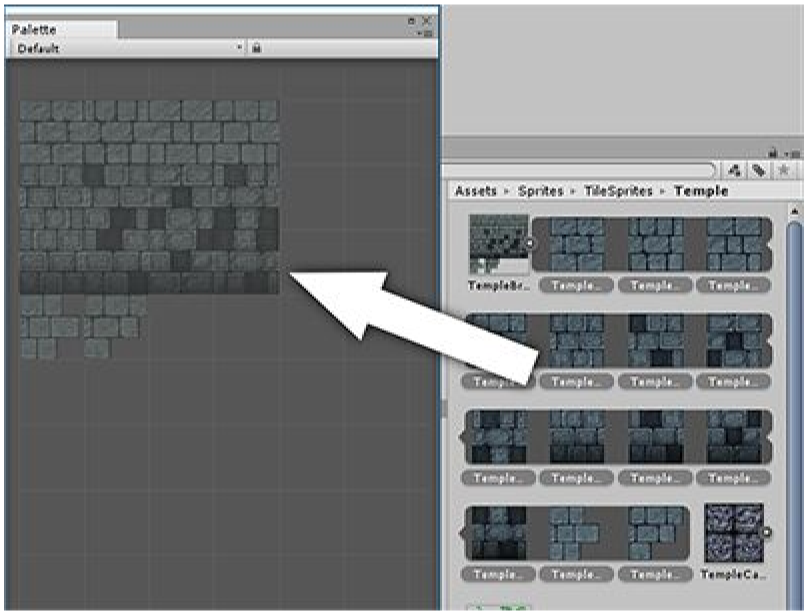




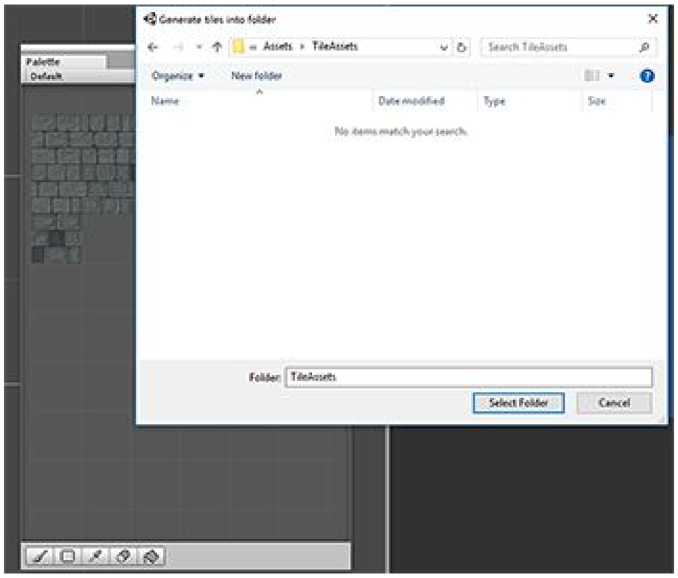
2. Select the Sprite/s to be used from the Asset window.



3. Drag and drop the Sprite/s onto the Tile Map Palette.



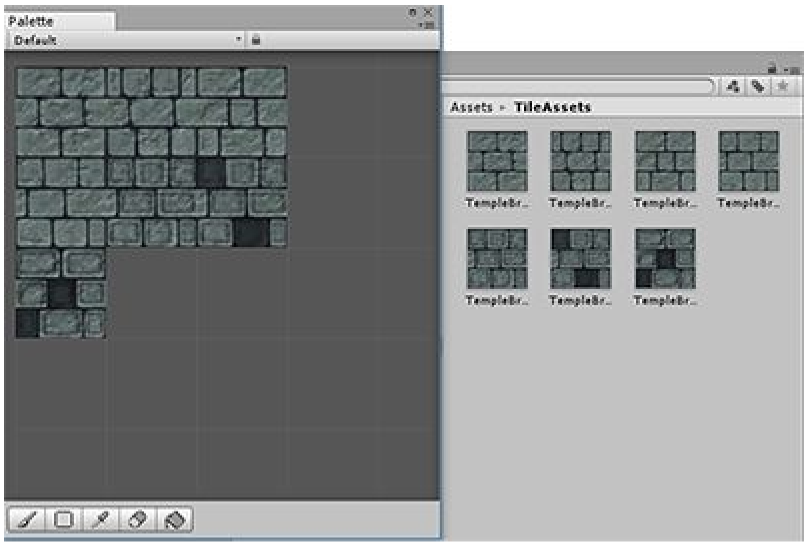
4. A dialog box will appear, allowing you to choose where to store the new Tile assets.



No items match your search.

Folder: TileAssets

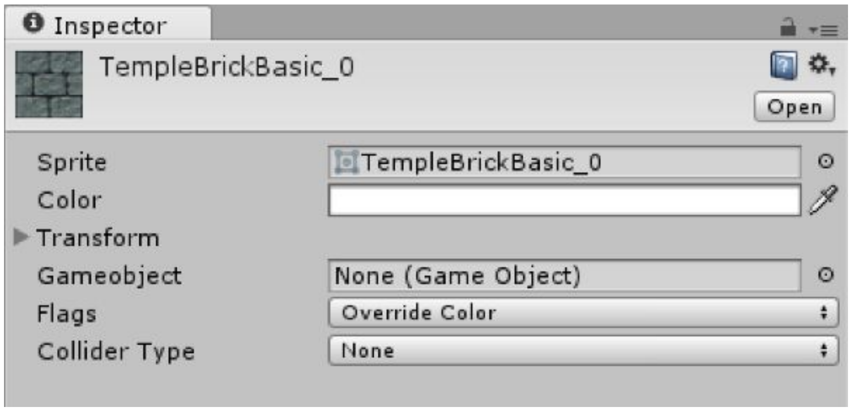
5. New Tile assets will be generated in the selected folder and the tiles will be placed on the Tile Map Palette.



6. Sprite can also be added individually to the Tile Map Palette
7. **Remember to save your project to ensure your Tile Map Palette configuration is saved!**

Tile Asset

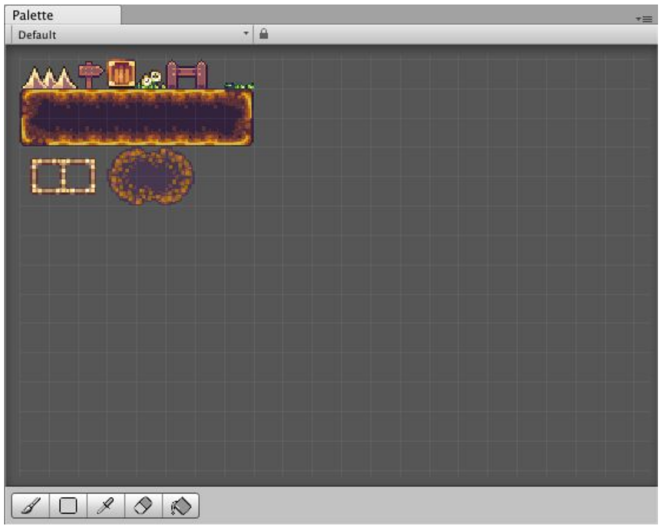
Typically tiles are actual sprites that are arranged on a tile map. In this implementation, we use an intermediary asset that references the sprite instead. This enables us to extend the tile itself in many ways, creating a robust and flexible system for tiles and tile maps.



Sprite: TempleBrickBasic_0
Color:
Gameobject: None (Game Object)
Flags: Override Color
Collider Type: None

Properties
Sprite - The sprite that is used by this tile asset
Color - Used to tint the material
GameObject - Attach a GameObject to the tile
Flags - Multiple flag options
Collider Type - Currently not supported

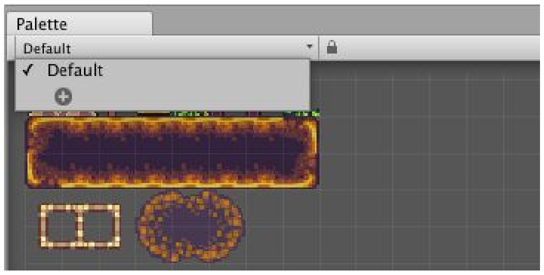
Tile Map Palette



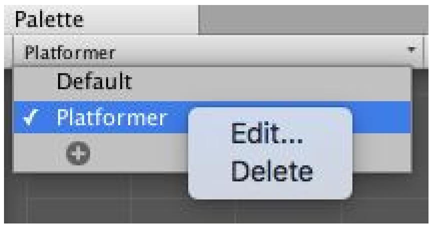
- Alt + Left Mouse Button to pan
- Middle Mouse Button to pan
- Mouse Wheel to zoom in or out
- Left-click to select a tile
- Left-click drag to select multiple tiles

Managing Tile Map Palettes

1. To create new Tile Map Palettes, select the drop-down menu of the Tile Map Palette and click on the + button.



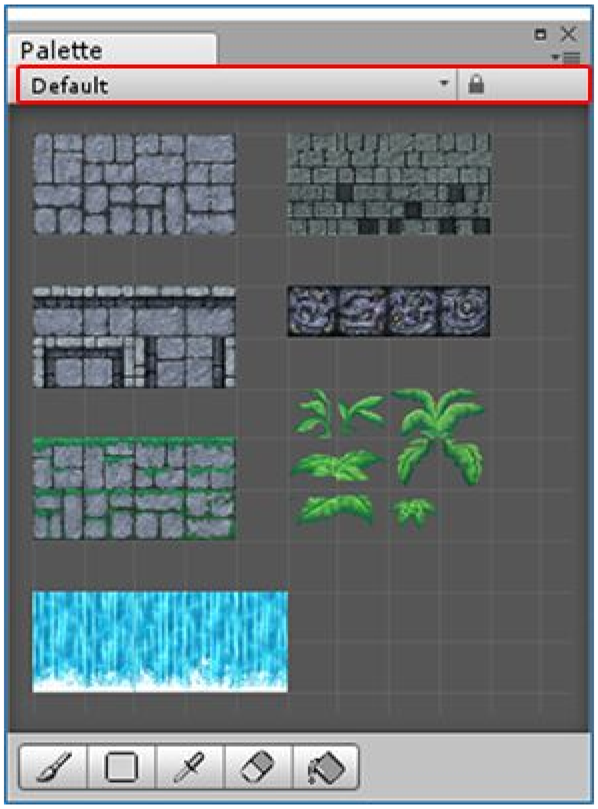
2. To rename a Tile Map Palette, select the Tile Map Palette to rename in the drop-down menu and right-click to open a menu. Click on Edit… in the menu to rename the Tile Map Palette.



3. To delete a Tile Map Palette, select the Tile Map Palette to delete in the drop-down menu and right-click to open a menu. Click on Delete in the menu to delete the Tile Map Palette.
4. Note: The Default Tile Map Palette cannot be renamed or deleted in this way.
5. Remember to save your project to save the changes!

Editing Tile Map Palette

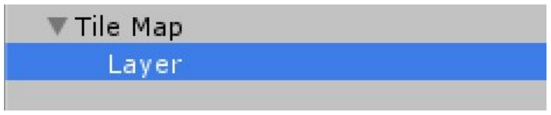
1. Select the desired palette from the drop-down menu.
2. Click on the lock at the side of the drop-down menu.



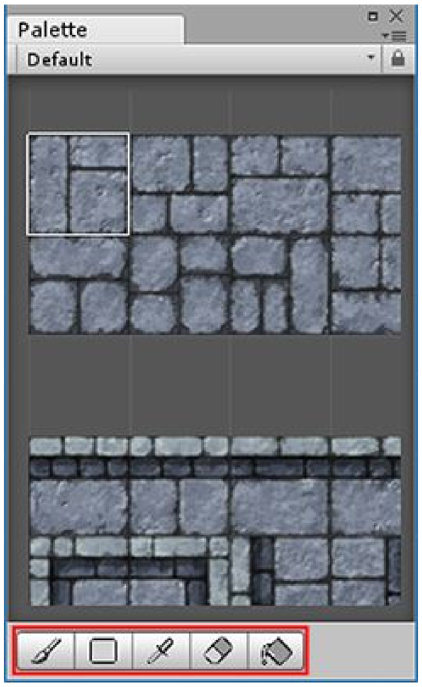
3. Adjust the Palette
   - Click to paint or erase tiles on the Palette, depending on the tool selected.
   - Ctrl-Click to select tiles on the Palette. Multiple tiles can be selected at once.
4. When done, click on the open lock at the side of the drop-down menu to lock the edited Palette.
5. **Remember to save your project to save the Palette!**

Painting Tile Maps

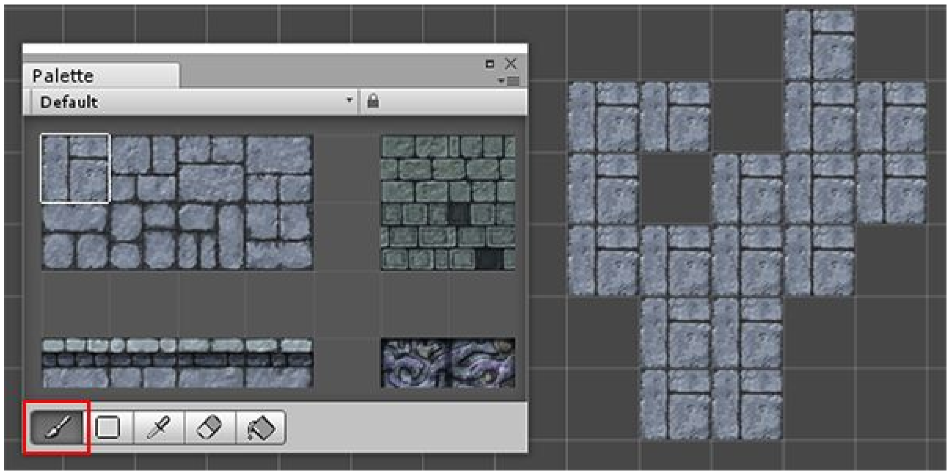
1. The Tile Map needs to be selected to enable the painting tools.



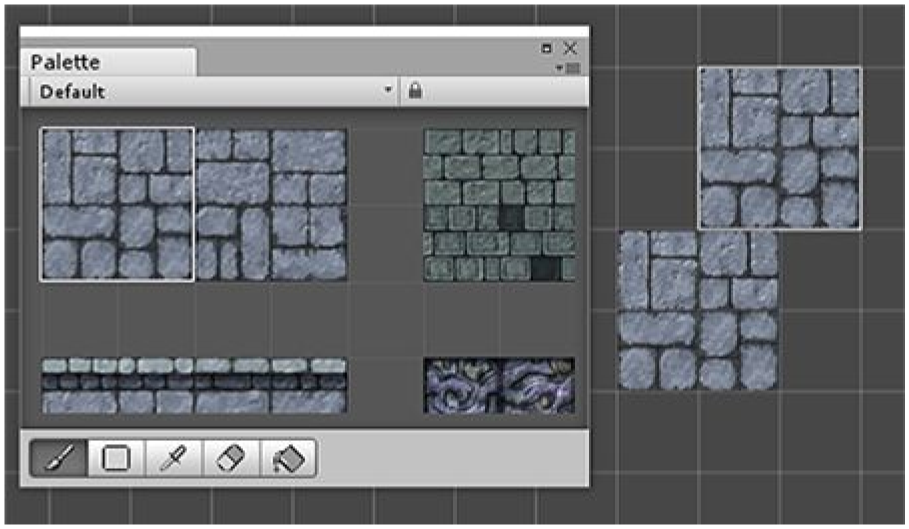
2. The tile painting tools are found on the Tile Map Palette



3. Click on the Paint Tool icon , select a tile from the Tile Map Palette and Left-click on the Tile Map in the Scene View to start laying out tiles.

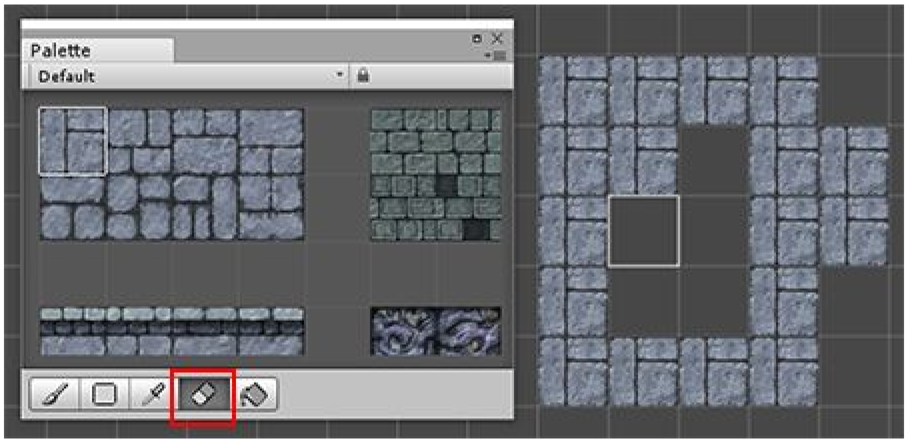


4. A selection of tiles can be painted with the Paint Too. Left-click and drag in the Tile Map Palette to make a selection.

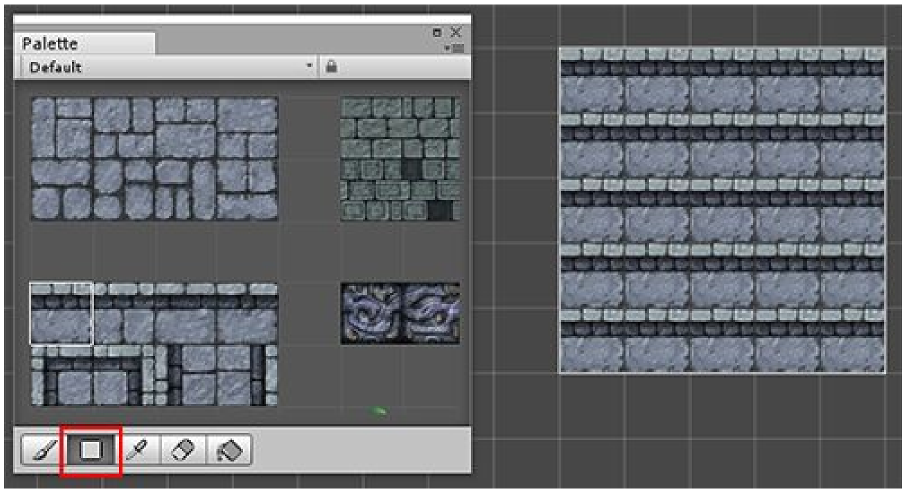


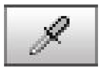
I

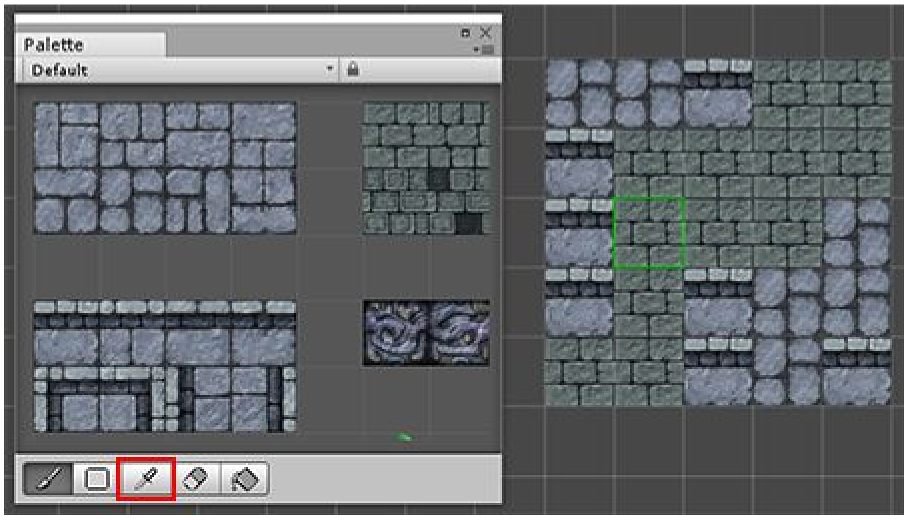
5. Holding Shift while using the Paint Tool will toggle the Erase Tool

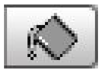


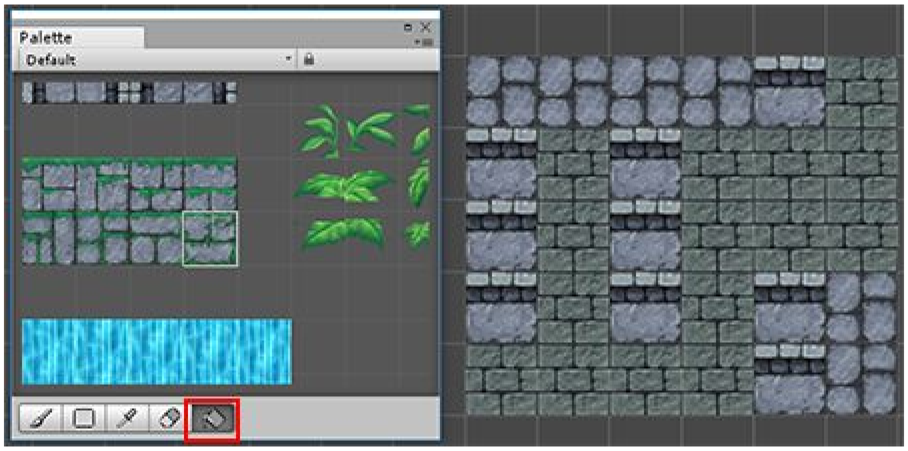
6. The Rectangle tool draws a rectangular shape on the Tile Map and fills it with the selected tiles.

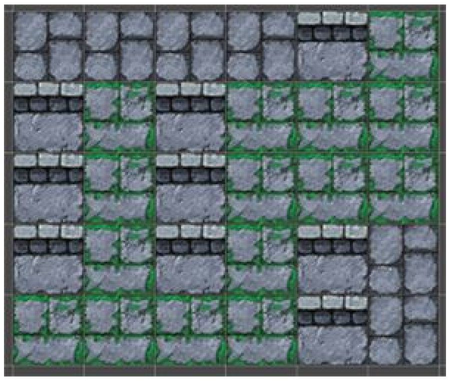


7. The Picker Tool  is used to pick tiles from the Tile Map to paint with. Left-click and drag to select multiple tiles. Holding Control key while in Paint Tool mode will toggle the Picker Tool.



8. The Fill Tool is used to fill a selection of tiles with the selected tile.





Tile Map Shortcut Keys

Shift + Left-Click = Erase Tiles
Ctrl + Left-Click = Select Tiles
Alt + Left-Click = Pan Scene View

Following keys only work if focus is on the Scene View. So if you just selected a tile from the palette, Right-Click in the Scene View to get focus first.
Y = Flip selected tile Up or Down
X = Flip selected tile Left or Right
. (Period) = Rotate Clockwise
, (Comma) = Rotate Anti-Clockwise

How to create a Programmable Tile

1. Create a new class inheriting from BaseTile (or any useful sub-classes of BaseTile like Tile).
2. Override any required methods for your new Tile class. The following are the usual methods you would override:
    a. RefreshTile determines which tiles in the vicinity will be updated as this Tile is added to the Tile Map.
    b. GetTileData determines what the Tile will look like on the TileMap.
3. Create instances of your new class using ScriptableObject.CreateInstance<(Your Tile Class>(). You may convert this new instance to an asset in the Editor in order to use it repeatedly by calling AssetDatabase.CreateAsset().
4. You can also make a custom editor for your tile. This works the same way as custom editors for scriptable objects.
5. Remember to save your project to ensure that your new Tile assets are saved!

BaseTile

All tiles to be added to the Tile Map must inherit from BaseTile. BaseTile provides a fixed set of APIs to the TileMap to communicate its rendering properties. For most cases of the APIs, the location of the tile and the instance of the tile map the tile is placed on is passed in as arguments of the API. You may use this to determine any required attributes for setting the tile information.

```
public void RefreshTile(Vector3Int location, ITileMap tileMap)
```

RefreshTile determines which tiles in the vicinity will be updated as this Tile is added to the Tile Map. By default, the BaseTile will call `tileMap.RefreshTile(location)` to refresh the tile at the current location. Override this to determine which tiles need to be refreshed due to the placement of the new tile.

Example: There is a straight road and you place RoadTile next to it. The straight road isn't valid anymore. It needs a T-section instead. Unity doesn't automatically know what needs to be refreshed, so RoadTile needs to trigger the refresh onto itself, but also onto the neighboring road.

```
public bool GetTileData(Vector3Int location, ITileMap tileMap, ref TileData tileData)
```
GetTileData determines what the Tile will look like on the TileMap. See TileData below for more details.

```
public bool GetTileAnimationData(Vector3Int location, ITileMap tileMap, ref TileAnimationData
tileAnimationData)
```
GetTileAnimationData determines if the Tile is animated. Return true if there is an animation for the tile, other returns false if there is not.

```
public bool StartUp(Vector3Int location, ITileMap tileMap, GameObject go)
```
StartUp is called for each tile when the TileMap updates for the first time. You can run any start up logic for tiles on the TileMap if necessary. The argument go is the instanced version of the object passed in as gameobject when GetTileData was called. You may update go as necessary as well.

```
public Sprite GetPreview()
```
GetPreview returns the Sprite that will be used as a preview on the Tile Map Palette or in the Asset window. This can be used as identification for your tile.

Tile

The Tile class is a simple class that allows a sprite to be rendered on the TileMap. Tile inherits from BaseTile. The following is a description of the methods that are overridden to have the Tile's behaviour.

```
public Sprite sprite;
public Color color = Color.white;
public Matrix4x4 transform = Matrix4x4.identity;
public GameObject gameobject = null;
public TileFlags flags = TileFlags.OverrideColor;
```
These are the default properties of a Tile. If the tile was created by dragging and dropping a Sprite onto the Tile Map Palette, the tile would have the sprite property set as the sprite that was dropped in. You may adjust the properties of the tile instance to get the tile required.

```
public void RefreshTile(Vector3Int location, ITileMap tileMap)
```
This is not overridden from BaseTile. By default, it will only refresh the Tile at that location.

```
public override bool GetTileData(Vector3Int location, ITileMap tileMap, ref TileData
tileData)
{
	tileData.sprite = this.sprite;
	tileData.color = this.color;
	tileData.transform = this.transform;
	tileData.gameobject = this.gameobject;
```

```
        tileData.flags = (int)this.flags;
        return true;
}
```
This fills in the required information for TileMap to render the Tile by copying the properties of the Tile instance into tileData.

```
public bool GetTileAnimationData(Vector3Int location, ITileMap tileMap, ref TileAnimationData
tileAnimationData)
```
This is not overridden from BaseTile. By default, the Tile class does not run any Tile animation and returns false.

```
public bool StartUp(Vector3Int location, ITileMap tileMap, GameObject go)
```
This is not overridden from BaseTile. By default, the Tile class does not have any special start up functionality. If `tileData.gameobject` is set, the Tile Map will still instantiate it on start up and place it at the location of the tile.

```
public override Sprite GetPreview()
{
        return sprite;
}
```
This returns the sprite as the preview and will be used as the icon representing the tile in the Assets folder.

Other Useful Classes:

TileFlags

```
None = 0
```
No flags are set for the tile. This is the default for most tiles.

```
OverrideColor = 1 << 0
```
Set this flag if the Tile script controls the color of the Tile. If this is set, the Tile will control the color as it is placed onto the tile map. You will not be able to change the tile’s color through painting or using scripts. If this is not set, you will be able to change the tile’s color through painting or using scripts.

```
OverrideTransform = 1 << 1
```
Set this flag if the Tile script controls the transform of the Tile. If this is set, the Tile will control the transform as it is placed onto the tile map. You will not be able to rotate or change the tile’s transform through painting or using scripts. If this is not set, you will be able to change the tile’s transform through painting or using scripts.

```
OverrideSpawnGameObjectRuntimeOnly = 1 << 2
```
Set this flag if the Tile script should spawn its game object only when your project is running and not in Editor mode.

```
OverrideFlags = OverrideColor | OverrideTransform | OverrideSpawnGameObjectRuntimeOnly
```
This is a combination of all the override flags used by BaseTile.

ITileMap

ITileMap is the interface where Tile can retrieve data from the Tile Map when the Tile Map tries to retrieve data from the Tile.

```
Vector2Int origin { get; }
```
This returns the origin point of the tile map.

```
Vector2Int size { get; }
```
This returns the size of the tile map.

```
TileMap GetTileMap();
```
This returns the actual Tile Map instance.

```
Sprite GetSprite(Vector3Int location);
```
This returns the sprite used by the tile in the tile map at the given location.

```
Color GetColor(Vector3Int location);
```
This returns the color used by the tile in the tile map at the given location.

```
Matrix4x4 GetTransformMatrix(Vector3Int location);
```
This returns the transform matrix used by the tile in the tile map at the given location.

```
int GetTileFlags(Vector3Int location);
```
This returns the tile flags used by the tile in the tile map at the given location.

```
BaseTile GetTile(Vector3Int location);
```
This returns the instance of a tile in the tile map at the given location. If there is no tile there, it returns null.

```
void RefreshTile(Vector3Int location);
```
This requests a refresh of the tile in the tile map at the given location.

TileData

```
public Sprite sprite
```
This is the sprite that will be rendered for the Tile.

```
public Color color
```
This is the color that will tint the sprite used for the Tile.

```
public Matrix4x4 transform
```
This is the transform matrix used to determine the final location of the Tile. Modify this to add rotations or scaling to the tile.

`public GameObject gameobject`

This is the game object that will be instanced when the Tile is added to the Tile Map.

`public int flags`

These are the flags which controls the Tile’s behaviour. See TileFlags below for more details.

TileAnimationData

`public Sprite[] animatedSprites`

This is an array of sprites for the Tile animation. The Tile will be animated by these sprites in sequential order.

`public float animationSpeed`

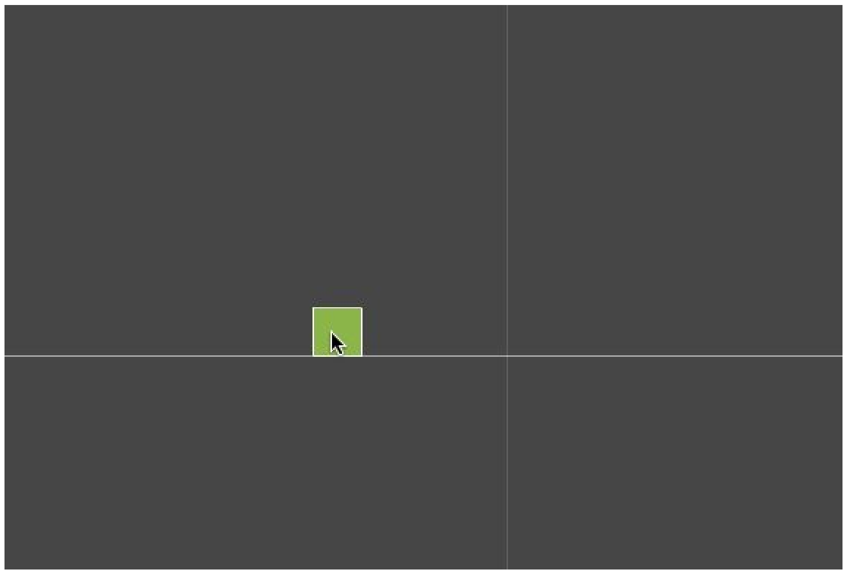
This is the speed where the Tile animation is run. This is combined with the Tile Map’s animation speed for the actual speed.

`public float animationTimeOffset`

This allows you to start the animation at a different time frame.

An Example Tile:

RoadTile



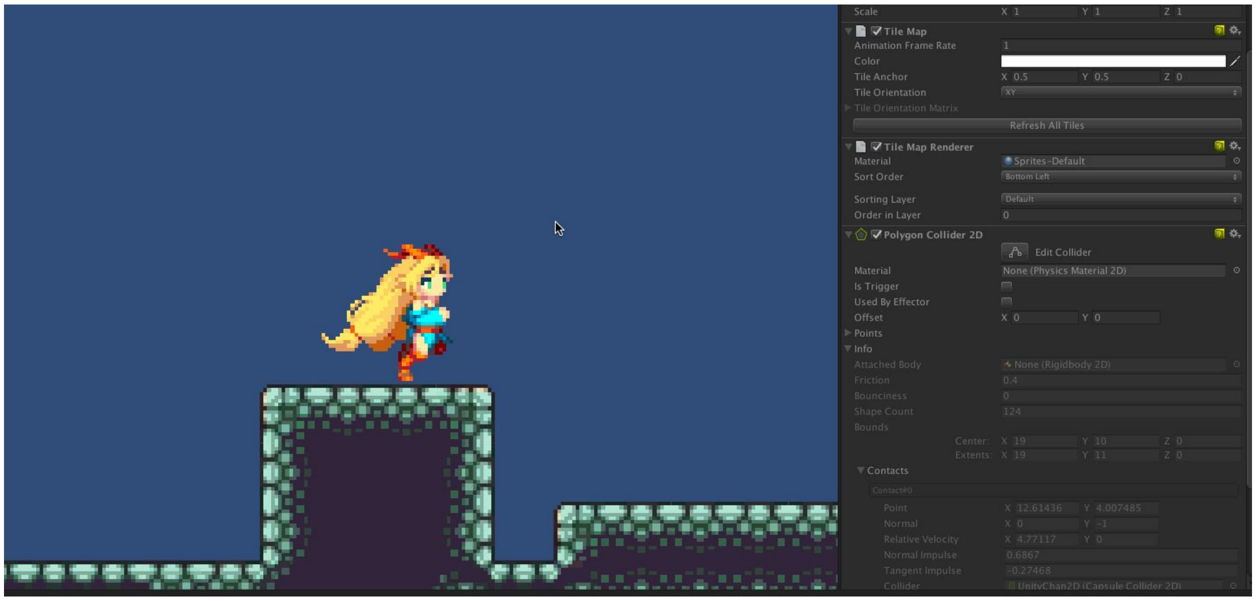
RoadTile allows users to easily layout linear segments onto the TileMap, such as roads or pipes, with a minimal set of sprites. The following is a script used to create the tile.

```
using UnityEngine;
using System.Collections;

#if UNITY_EDITOR
using UnityEditor;
```

```
#endif

public class RoadTile : Tile
{
	public Sprite[] m_Sprites;
	public Sprite m_Preview;

	// This refreshes itself and other RoadTiles that are orthogonally and diagonally adjacent
	public override void RefreshTile(Vector3Int location, ITileMap tileMap)
	{
		for (int yd = -1; yd <= 1; yd++)
			for (int xd = -1; xd <= 1; xd++)
			{
				Vector3Int position = new Vector3Int(location.x + xd, location.y + yd, location.z);
				if (HasRoadTile(tileMap, position))
					tileMap.RefreshTile(position);
			}
	}

	// This determines which sprite is used based on the RoadTiles that are adjacent to it and rotates it to fit the other tiles.
	// As the rotation is determined by the RoadTile, the TileFlags.OverrideTransform is set for the tile.
	public override bool GetTileData(Vector3Int location, ITileMap tileMap, ref TileData tileData)
	{
		int mask = HasRoadTile(tileMap, location + new Vector3Int(0, 1, 0)) ? 1 : 0;
		mask += HasRoadTile(tileMap, location + new Vector3Int(1, 0, 0)) ? 2 : 0;
		mask += HasRoadTile(tileMap, location + new Vector3Int(0, -1, 0)) ? 4 : 0;
		mask += HasRoadTile(tileMap, location + new Vector3Int(-1, 0, 0)) ? 8 : 0;

		int index = GetIndex((byte)mask);

		if (index >= 0 && index < m_Sprites.Length)
		{
			tileData.sprite = m_Sprites[index];
			tileData.color = Color.white;
			tileData.transform.SetTRS(Vector3.zero, GetRotation((byte) mask), Vector3.one);
			tileData.flags = (int) TileFlags.OverrideTransform;
			return true;
		}

		Debug.LogWarning("Not enough sprites in RoadTile instance");
		return false;
	}

	// This returns the preview sprite as the preview for the Tile
	public override Sprite GetPreview()
	{
		return m_Preview;
	}

	// This determines if the Tile at the position is the same RoadTile.
	private bool HasRoadTile(ITileMap tileMap, Vector3Int position)
	{
		return tileMap.GetTile(position) == this;
	}

	// The following determines which sprite to use based on the number of adjacent RoadTiles
	private int GetIndex(byte mask)
	{
		switch (mask)
		{
				case 0: return 0;
				case 3:
				case 6:
				case 9:
				case 12: return 1;
				case 1:
				case 2:
				case 4:
				case 5:
				case 10:
				case 8: return 2;
				case 7:
				case 11:
				case 13:
				case 14: return 3;
				case 15: return 4;
			}
			return -1;
		}

		// The following determines which rotation to use based on the positions of adjacent RoadTiles
		private Quaternion GetRotation(byte mask)
		{
			switch (mask)
			{
				case 9:
				case 10:
				case 7:
				case 2:
				case 8:
					return Quaternion.Euler(0f, 0f, -90f);
				case 3:
				case 14:
					return Quaternion.Euler(0f, 0f, -180f);
				case 6:
				case 13:
					return Quaternion.Euler(0f, 0f, -270f);
			}
			return Quaternion.Euler(0f, 0f, 0f);
		}

	#if UNITY_EDITOR
		// The following is a helper that adds a menu item to create a RoadTile asset
		[MenuItem("Assets/Create/RoadTile")]
		public static void CreateRoadTile()
		{
			string path = EditorUtility.SaveFilePanelInProject("Save Road Tile", "New Road Tile", "asset", "Save Road Tile",
	"Assets");

			if (path == "")
				return;

			AssetDatabase.CreateAsset(ScriptableObject.CreateInstance<RoadTile>(), path);
		}
	#endif
	}
```

Physics 2D and Tile Maps

- TileMapCollider2D is still under works and is coming soon!
- As a placeholder, adding a PolygonCollider2D component to the TileMap GameObject will generate a collider for the TileMap. This is temporary and will not be in the final production.



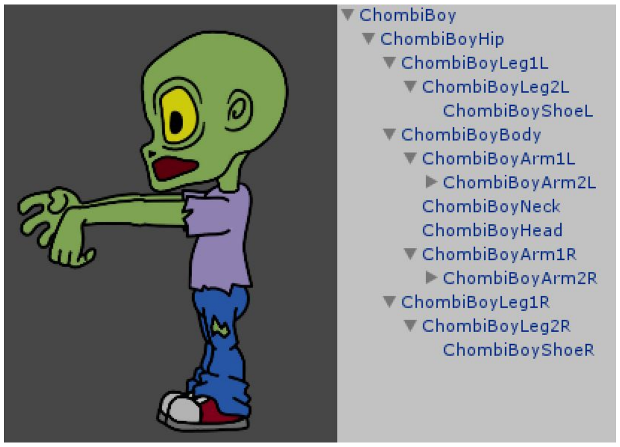
```
Scale                    X 1        Y 1        Z 1
Tile Map
Animation Frame Rate     1
Color
Tile Anchor              X 0.5      Y 0.5      Z 0
Tile Orientation         XY
Tile Orientation Matrix
Refresh All Tiles
Tile Map Renderer
Material                 Sprites-Default
Sort Order               Bottom Left
Sorting Layer            Default
Order in Layer           0
Polygon Collider 2D
                         Edit Collider
Material                 None (Physics Material 2D)
Is Trigger
Used By Effector
Offset                   X 0        Y 0
Points
Info
Attached Body            None (Rigidbody 2D)
Friction                 0.4
Bounciness               0
Shape Count              124
Bounds
              Center:    X 19       Y 10       Z 0
              Extents:   X 19       Y 11       Z 0
  Contacts
    Contact#0
    Point                X 12.61436   Y 4.007485
    Normal               X 0          Y -1
    Relative Velocity    X 4.77117    Y 0
    Normal Impulse       0.6867
    Tangent Impulse      -0.27468
    Collider             UnityChan2D (Capsule Collider 2D)
```

Sorting Group

The Sorting Group is a component that alters the order in which Renderers are rendered. Conventionally renderers in Unity will be sorted by several criteria such as Order in Layer or distance from camera. However, it was not possible to ensure that a group of renderers that share a common root be sorted together. Well, until now…

Having renderers that belong to a common root render together is extremely useful. A common use case in 2D games are with complex characters.

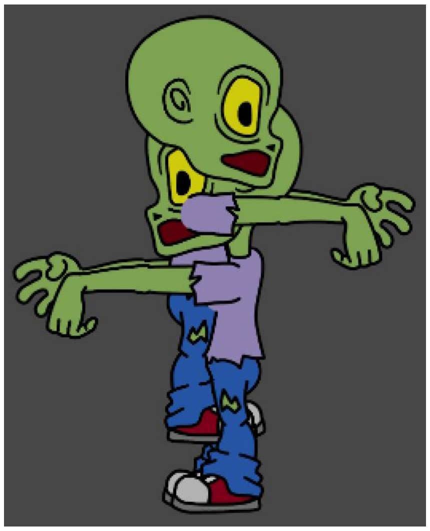
Here we have a 2D Character with multiple renderers in a hierarchy. Character is in a single Sorting Layer and uses multiple Order in Layers.



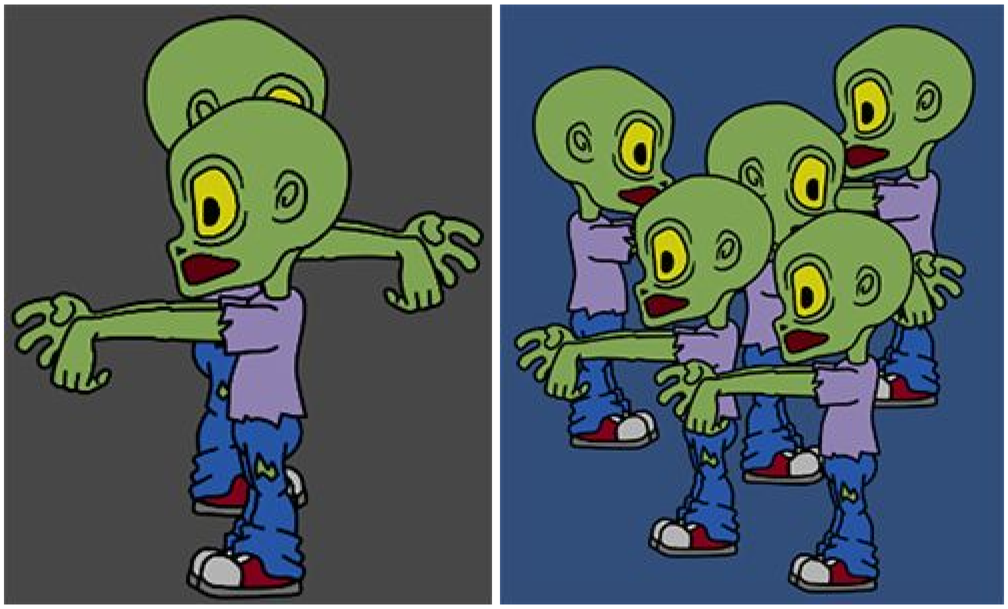
```
▼ ChombiBoy
  ▼ ChombiBoyHip
    ▼ ChombiBoyLeg1L
      ▼ ChombiBoyLeg2L
          ChombiBoyShoeL
    ▼ ChombiBoyBody
      ▼ ChombiBoyArm1L
        ▶ ChombiBoyArm2L
        ChombiBoyNeck
        ChombiBoyHead
      ▼ ChombiBoyArm1R
        ▶ ChombiBoyArm2R
    ▼ ChombiBoyLeg1R
      ▼ ChombiBoyLeg2R
          ChombiBoyShoeR
```

It is then saved into a prefab and subsequently cloned multiple times during game play. Before the advent of Sorting Group, these complex 2D character will face great trouble when they overlap on screen. You will likely find their body parts ‘interleaved’ with each other.

Here’s how it looks like when 2 complex characters overlap without Sorting Group.

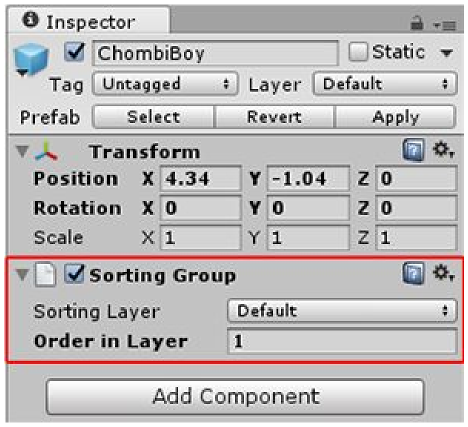


However, the desired outcome is to have all the renderers of a character render together and then followed by the next character. With a Sorting Group component added to the root of the character, their body parts no longer get mixed up.



Setting Up Sorting Group

To use the Sorting Group component, simply add it to the object's root. The Sorting Group itself does not have any visual representation and it also does not have any dependencies on other renderers. It could even be added to an empty GameObject. Once added to an object, any renderers that's on the GameObject and it's descendants will be rendered together.



Inspector
ChombiBoy   Static
Tag Untagged   Layer Default
Prefab Select Revert Apply

Transform
Position X 4.34 Y -1.04 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Sorting Group
Sorting Layer Default
Order in Layer 1

Add Component

How are Sorting Groups sorted?

They are sorted like every other renderer, using Sorting Layer and Order in Layer.

How are children of Sorting Group sorted?

The descendants of the Sorting Group are also sorted according to the standard sorting criterions such as Sorting Layer and Order in Layer. However, there is one major difference. During the sort, the descendants of Sorting Group will only be sorted against other descendants of the most immediate Sorting Group. In other words, the Sorting Group creates a local sorting space only for its descendants. This allows each of the renderers inside the group be sorted using the same simplistic Sorting Layer and Order in Layer but localized to the containing Sorting Group.

Nested Sorting Group

Sorting Group can be nested. A nested Sorting Group will be sorted just like a regular renderer against other renderers in the same group. Do note however, branches in the hierarchy that does not have any Sorting Group will be flattened.

One Last Thing

Any renderers that are descendant of a Sorting Group will be considered as ‘alpha’ objects and thus rendered after all the opaque objects.

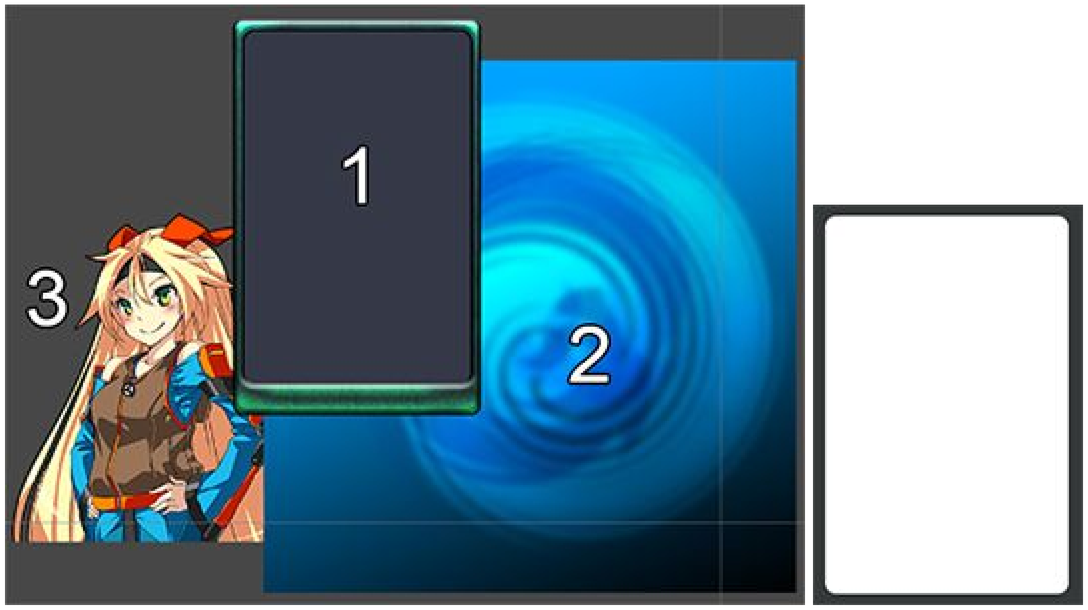
Sprite Masks

Scoped Masks

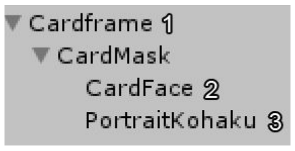
- Scoped Masks are masks that work in a GameObjects hierarchy.

Scoped Mask Example

1. In this example we are going to create a card from using 3 different sprites. A card sprite (1), a background for the card(2), a portrait sprite(3) and a sprite to be used as a mask. We need to mask out the parts of background and character that do not fit into the card. To do this, let's setup the the card game object hierarchy.

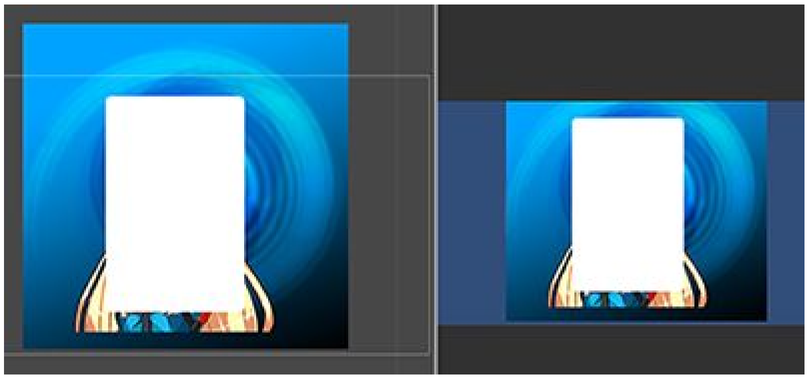


2. The Cardframe is the parent GameObject. Make the CardMask a child of it and the CardFace and Portrait should be children of the CardMask.

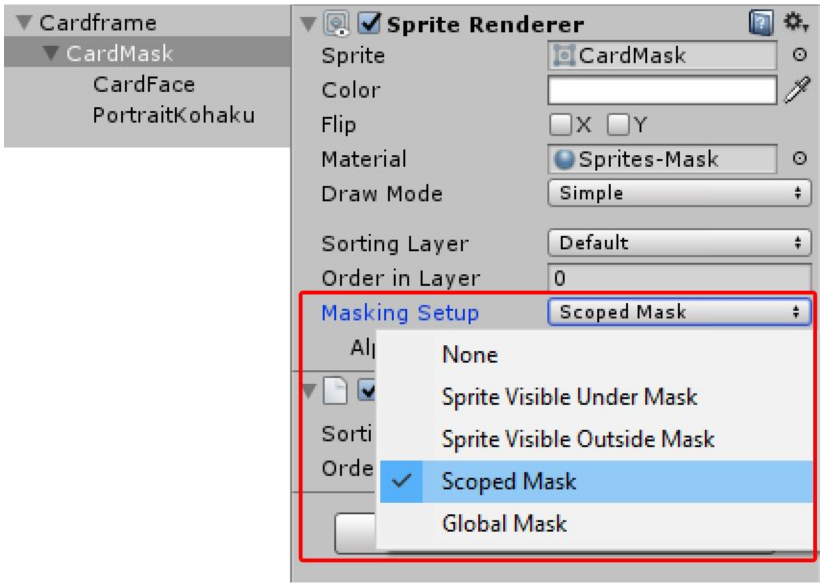


```
▼ Cardframe ①
  ▼ CardMask
      CardFace ②
      PortraitKohaku ③
```

3. Arrange the CardMask,CardFace and Character to fit the Cardframe. All the sprites are on the same Sorting Layer and their Order in Layer is 0.



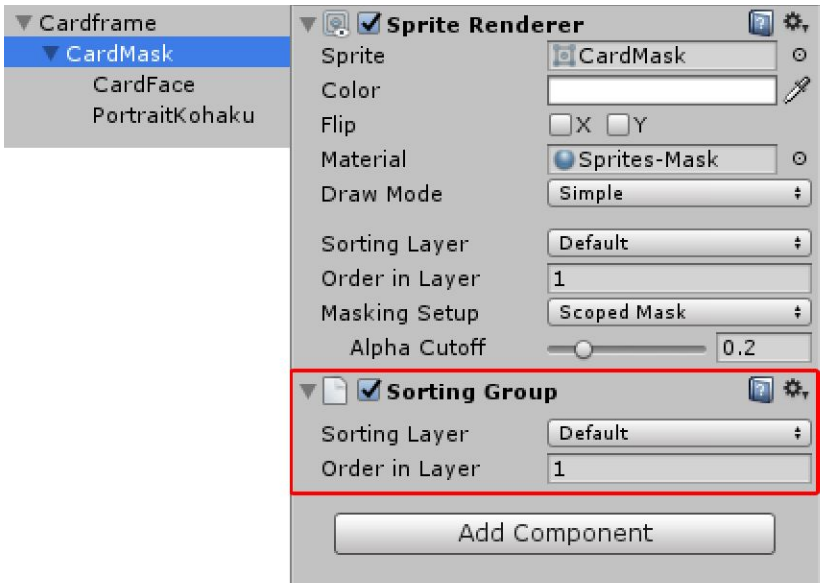
4. In the CardMask Sprite Renderer Component, set the Masking Setup to Scoped Mask.



Cardframe
CardMask
CardFace
PortraitKohaku

Sprite Renderer
Sprite: CardMask
Color
Flip: X Y
Material: Sprites-Mask
Draw Mode: Simple
Sorting Layer: Default
Order in Layer: 0
Masking Setup: Scoped Mask

5. When the Masking Setup is set to Scoped Mask, a Sorting Group Component is added. Set the Order in Layer to a higher value than the Cardframe, in this example the value is 1 so that the mask is above the Cardframe.

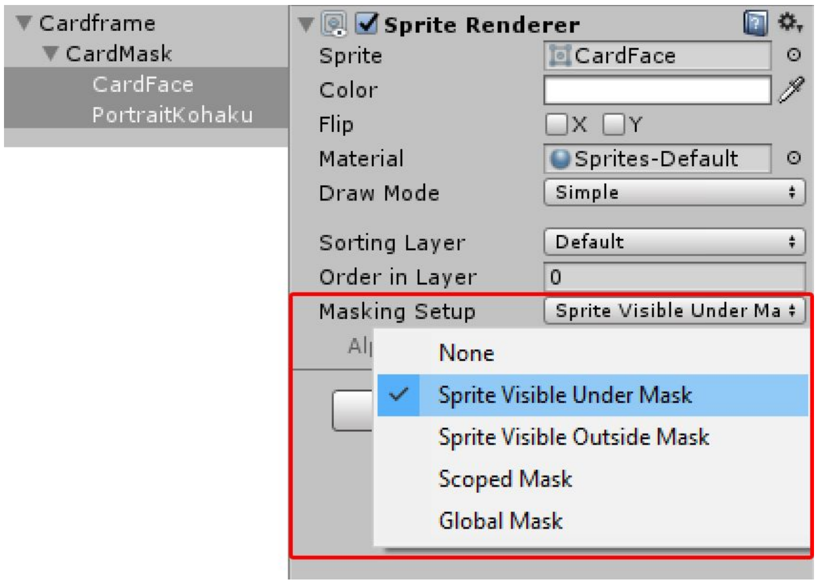


Cardframe
CardMask
CardFace
PortraitKohaku

Sprite Renderer
Sprite: CardMask
Color
Flip: X Y
Material: Sprites-Mask
Draw Mode: Simple
Sorting Layer: Default
Order in Layer: 1
Masking Setup: Scoped Mask
Alpha Cutoff: 0.2

Sorting Group
Sorting Layer: Default
Order in Layer: 1

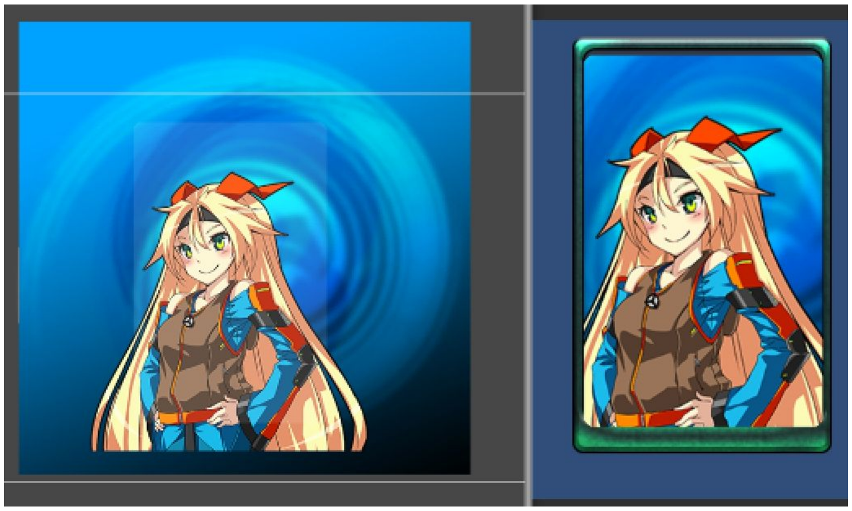
6. Set the CardFace and Potrait Masking Setup to Sprite Visible Under Mask.



Cardframe
CardMask
CardFace
PortraitKohaku

Sprite Renderer
Sprite: CardFace
Color
Flip: X Y
Material: Sprites-Default
Draw Mode: Simple
Sorting Layer: Default
Order in Layer: 0
Masking Setup: Sprite Visible Under Ma

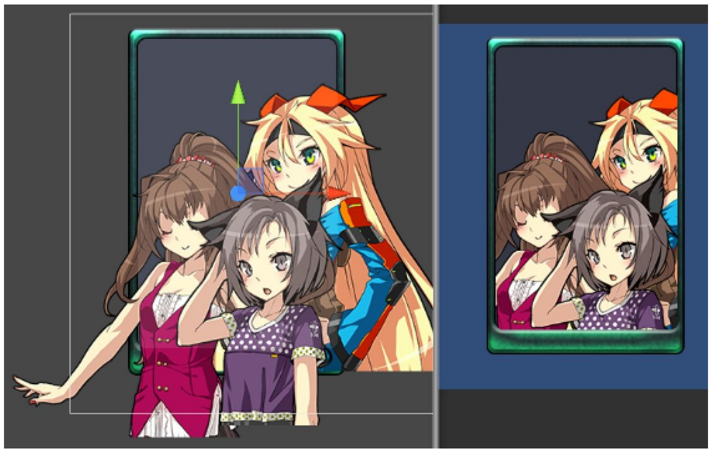
7. The result should be like the image above. The masking effect is only visible in the Game View.



Global Masks

- All sprites in the scene are affected by Global Masks.
- Global Masks are masks that affect sprites that are below itself, either in Sorting Layers or Order In Layer.

In the *GlobalMaskExample* image a Global Mask is applied to all the sprites in the scene that have their Masking Setup assigned. Each sprite has their Order in Layer set accordingly and the Global Mask has its Order in Layer set to higher or above all the sprites.



GlobalMaskExample

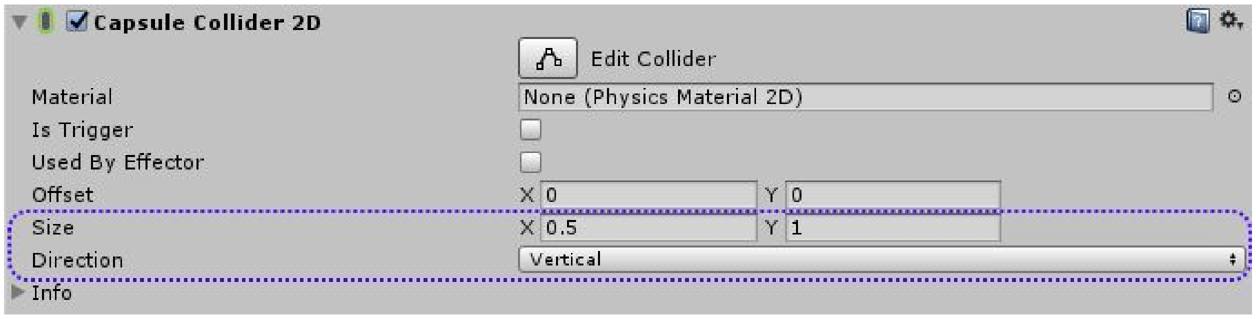
CapsuleCollider2D Component

The capsule collider is a 2D physics primitive that can be elongated in either a vertical or horizontal direction. The capsule shape has no vertex corners i.e. it has a continuous circumference that has the advantage that it won't get caught on other collider corners. The capsule shape is also a solid so anything positioned internal to the capsule is seen as being overlapped and will be moved away.

The capsule collider is configured with only two properties. The first is the **Size** which defines a box (the **Size** property is identical to the **Size** property on the *BoxCollider2D*). This box defines the region that the capsule collider will fill. The second property is the **Direction** which can be either *Vertical* or *Horizontal*. This direction controls the position of the semi-circular end-caps.

It should be noted that both the ***Size*** *and* ***Direction*** *properties are referring to X, Y, Vertical & Horizontal in the local-space of the collider and not in world-space.*

Typically the **Size** is set so it is larger in the current **Direction**. For example, below the capsule is in the *Vertical* direction and the *Y* is set to 1 which is larger than the *X* which is 0.5 (the collider AABB is also shown which visualizes the size)



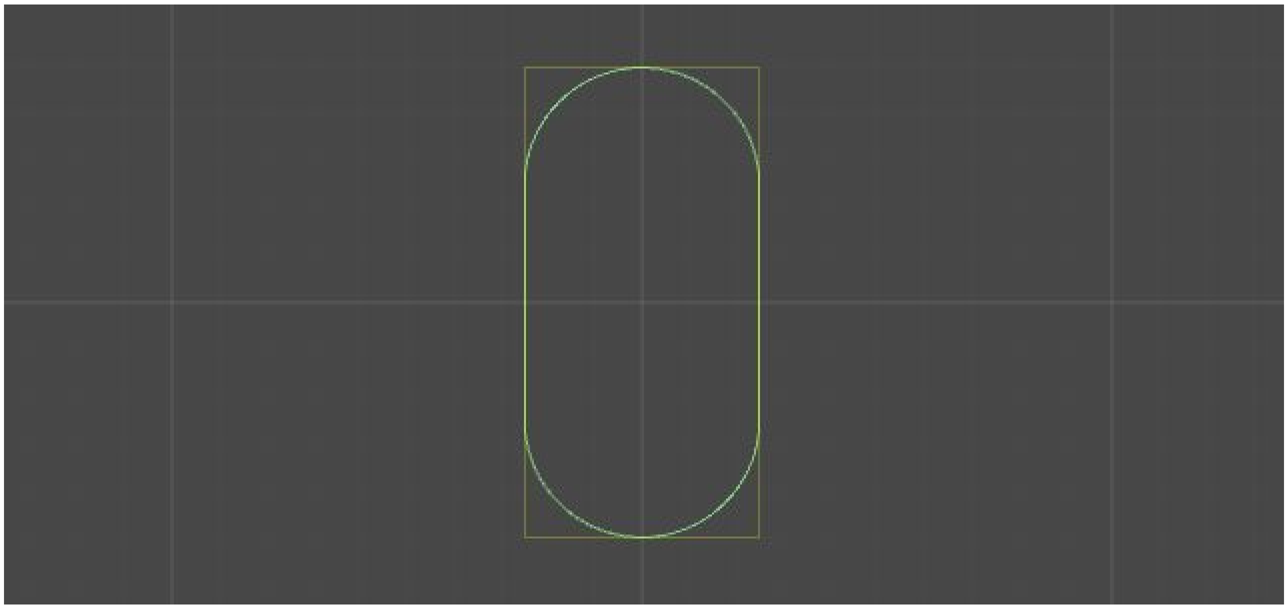
Capsule Collider 2D

Edit Collider

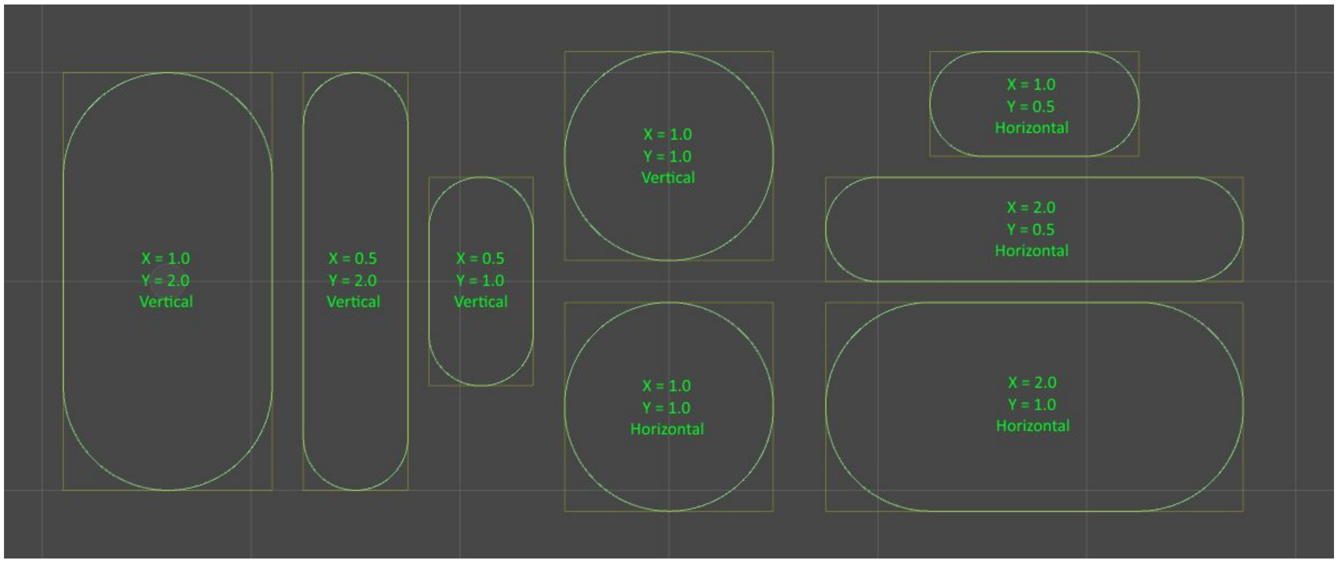
Material	None (Physics Material 2D)
Is Trigger	☐
Used By Effector	☐
Offset	X 0 Y 0
Size	X 0.5 Y 1
Direction	Vertical

Info

This produces a capsule that looks like this:



Here are other configurations that demonstrate how the capsule collider can be changed:



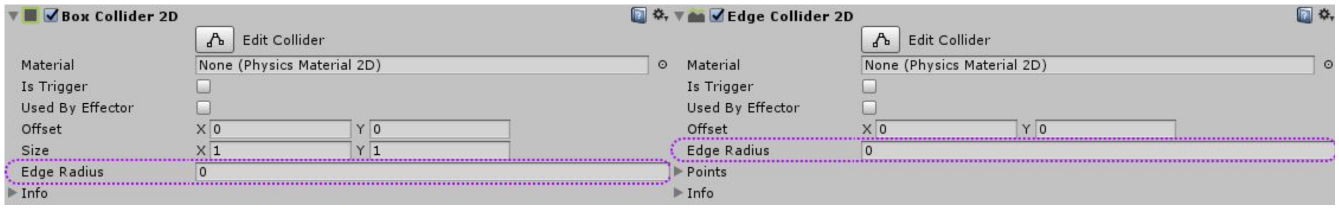
As shown above, it should be noted that when the *X* and *Y* of the **Size** property are the same then the capsule collider approximates to a circle no matter what **Direction** is selected.

There is a well known issue in physics systems (including Box2D) where moving across multiple colliders, even colliders that are perfectly aligned (numerically) causes a collider to register a collision between the two colliders that can cause the collider to slow-down or stop. Whilst the capsule collider can help reduce this problem, it isn't a solution to it. The real solution is using a single collider for a surface such as the EdgeCollider2D. Beyond that, there is a CompositeCollider2D in progress that will allow the merging of multiple colliders into a single, continuous surface that solves this issue when using a single collider for the surface isn't practical.

EdgeRadius for BoxCollider2D & EdgeCollider2D

All colliders that are vertex-based use a tiny radius that is used to ensure collision detection works correctly by providing a small 'buffer' zone where a positive collision is assumed. This feature extends that by allowing you to increase the size of that radius so that vertex on colliders become circular and edges become larger.

By default, the *EdgeRadius* is zero which produces no radius (actually it's greater than zero but it uses the default tiny gap to ensure collision detection works correctly). Here's the new property as it appears on both these components:



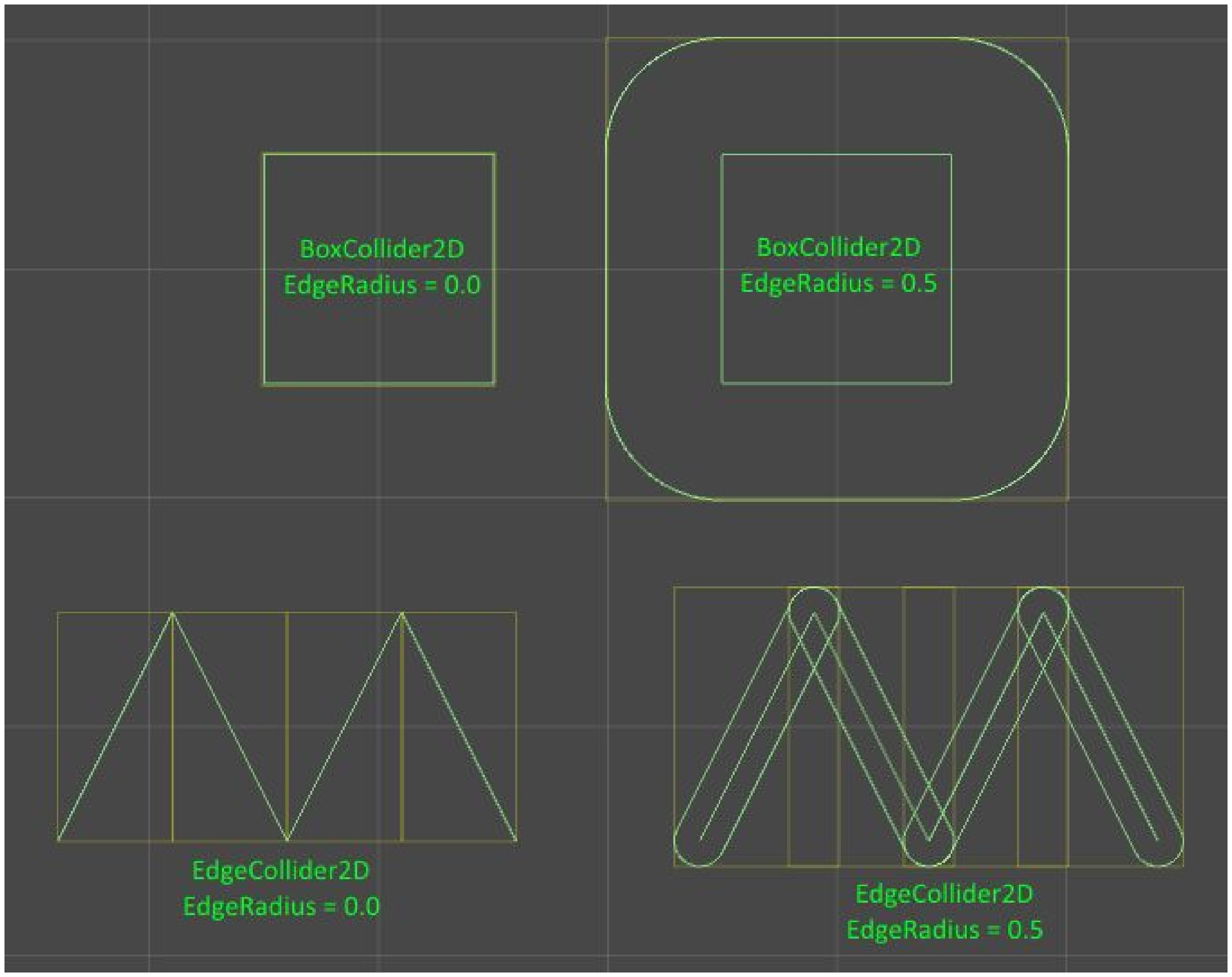
Box Collider 2D

Material	None (Physics Material 2D)
Is Trigger	☐
Used By Effector	☐
Offset	X 0 Y 0
Size	X 1 Y 1
Edge Radius	0
Info	

Edge Collider 2D

Material	None (Physics Material 2D)
Is Trigger	☐
Used By Effector	☐
Offset	X 0 Y 0
Edge Radius	0
Points	
Info	

Here's what both the *BoxCollider2D* & *EdgeCollider2D* gizmos look like when using an *EdgeRadius* greater than zero:



As can be seen, the edge radius expands by the selected radius but it should be noted that this extra radius does not affect center-of-gravity nor any auto-mass calculations that take place. It is effectively acting like a radial offset around each collider that affects collision detection distance only.

The *BoxCollider2D* gizmo (shown above) also shows the original box as specified by the **Size** property. This is the real size of the box and the *EdgeRadius* simply extends that. If you are calculating the actual size of the *BoxCollider2D* you can retrieve the collider AABB or to calculate it yourself you would simply add the *EdgeRadius* * 2 to both the width and height of the box (**Size** property).

The *EdgeRadius* feature also has zero performance impact because the 2D physics system is already using a tiny radius for its collision detection and the size of the radius has no performance impact; this feature simply provides access so that it can be changed.

NOTE: In the future, *EdgeRadius* will be available on the *PolygonCollider2D* component as well.

Rigidbody2D Body-Type

When moving a GameObject you typically set properties on the Transform component to achieve this. The Transform component defines how a GameObject (and its children) are positioned, rotated and scaled within the scene. When it is changed, it updates other components which may update where they render or where colliders are positioned etc. When you have a Collider2D component, it will be positioned, rotated and scaled according to the Transform component however, the physics system is able to move colliders and make them interact with each other so a method is required for the physics system to communicate this movement of colliders back to the Transform components.

This movement and connection with colliders is what a Rigidbody2D component is for. When adding a Rigidbody2D, you are effectively entering into an agreement with the physics system that requests that it update the Transform component to a position/rotation defined by the Rigidbody2D component. In other words, the agreement is that the Rigidbody2D is the authority on the position/rotation of the GameObject. Additionally, the agreement states that you won't override the Rigidbody2D by modifying the Transform component yourself although that is possible simply because Unity exposes all properties on all components but it can have undesired consequences.

An important aspect of a Rigidbody2D is that any Collider2D added to the same GameObject or child GameObject are implicitly attached to that Rigidbody2D. When they are attached to the Rigidbody2D, they move with it. Indeed, Collider2D should never be moved directly using the Transform or any collider offset but rather the Rigidbody2D should be moved instead. This offers the best performance and ensures correct collision detection. Also, any colliders attached to the same Rigidbody2D won't collide with each other. This allows creating a set of colliders that act effectively as a single compound collider all moving and rotating in sync with the Rigidbody2D.

When designing a scene, you are free to use a default Rigidbody2D and start attaching colliders. These colliders work out-of-the-box allowing any colliders attached to different Rigidbody2D to collide with each other.

In the past, what wasn't obvious, was that a Rigidbody2D has a feature called its body-type. There are three body-types, each of which defines a common and fixed behaviour. These body-types are:

- Dynamic
- Kinematic
- Static

Previously, the only body-type that was explicitly selectable was the **Kinematic** body-type. This was achieved by setting the *IsKinematic* property to true. When the *IsKinematic* property was false, the body-type was set to **Dynamic** (the default for the Rigidbody2D component). Finally, the **Static** body-type wasn't available explicitly either; when adding a Collider2D component *without* a Rigidbody2D component, the collider was implicitly attached to a hidden static Rigidbody2D that was permanently positioned at the world origin with zero rotation.

So in summary, a Rigidbody2D has one of three body-types. Any attached Collider2D to that Rigidbody2D inherits the body-type. It is therefore acceptable to say that a Rigidbody2D is **Dynamic**, **Kinematic** or **Static** but also acceptable to say that a Collider2D is also **Dynamic**, **Kinematic** or **Static** as defined by the Rigidbody2D component it is attached to.

So what do these three body-types mean?

They actually define two things:

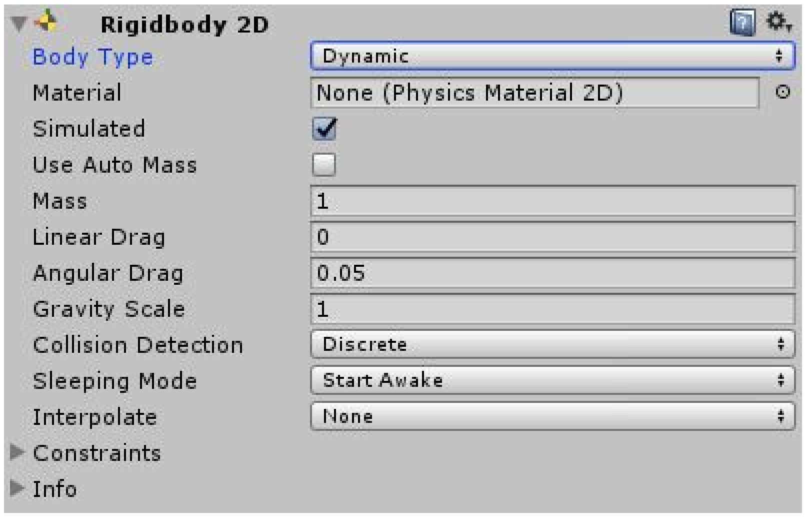
- Movement (Position & Rotation) Behaviour
- Collider Interaction

Here's a breakdown of each body-type. It's important to note that when describing two Rigidbody2D colliding with each other, it implicitly means the colliders attached to each of those bodies collide as Rigidbody2D do not collide with each other without colliders.

Dynamic

A **Dynamic** body is designed to move under simulation. It has the full set of properties available to it such as finite mass, drag (etc) and is affected by gravity and forces. A **Dynamic** body will collide with every other body-type and is the most interactive of body-types. This is the default body-type for a Rigidbody2D as it is the most common body-type for things that need to move. It's also the most performance-expensive body-type because of its dynamic nature and interactivity with everything around it. A **Dynamic** body-type should not be explicitly positioned by setting its position/rotation properties or by modifying the Transform component directly. The simulation should be left to reposition a **Dynamic** body according to its velocity which is modified via forces applied to it directly via scripts or indirectly via collisions and gravity.

All Rigidbody2D properties are available with this body-type as can be seen here:



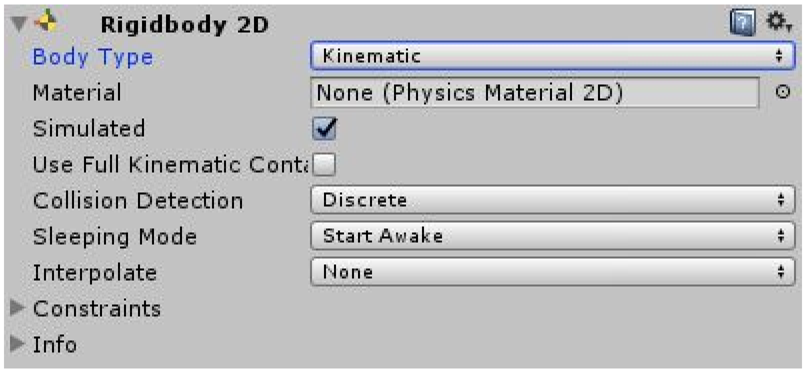
Rigidbody 2D

Body Type	Dynamic
Material	None (Physics Material 2D)
Simulated	✓
Use Auto Mass	☐
Mass	1
Linear Drag	0
Angular Drag	0.05
Gravity Scale	1
Collision Detection	Discrete
Sleeping Mode	Start Awake
Interpolate	None
Constraints	
Info	

Kinematic

A **Kinematic** body is designed to move under simulation but only under very explicit user-control. Whereas a **Dynamic** body is affected by gravity and forces, a **Kinematic** body isn't. For this reason, it is fast and less performance-expensive than a **Dynamic** body. It is designed to be repositioned explicitly via Rigidbody2D.MovePosition or Rigidbody2D.MoveRotation, typically using physics queries to detect collisions and scripts to decide where and how the body should move. Another way of describing it is similar to a **Dynamic** body but with explicit movement and collision detection control. A 2D **Kinematic** body does still move via its velocity but the velocity is not affected by forces or gravity. A **Kinematic** body does not collide with other **Kinematic** or **Static** bodies, only **Dynamic** bodies. Similar to a **Static** body (see below), it behaves as if it has infinite mass (an immoveable object) during collisions.

The mass-related properties are not available with this body-type as can be seen here:

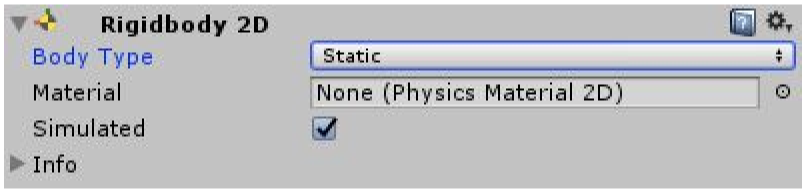

Rigidbody 2D

Body Type	Kinematic
Material	None (Physics Material 2D)
Simulated	✓
Use Full Kinematic Conta	
Collision Detection	Discrete
Sleeping Mode	Start Awake
Interpolate	None
Constraints	
Info	

Static

A **Static** body is designed to not move under simulation at all. Anything colliding with it will not cause it to move as it behaves as if it has infinite mass (same as the **Kinematic** body). A **Static** body is also the fastest body-type to use. A **Static** body only collides with a **Dynamic** body (technically, the **Dynamic** body collides with it). It should be clear that having two **Static** bodies collide is not supported and would be pointless as they are not designed to move. When adding a Collider2D without a Rigidbody2D, it is implicitly added to a hidden **Static** body. Whilst this is convenient, it should be clear that you should not move the collider. This is also why it is generally described that to have a collision, one of the Collider2D must be attached to a Rigidbody2D component. Whilst this is true, it is also not obvious why that is the case but the description above should clarify that; it is saying that two Collider2D without a Rigidbody2D are actually **Static** and won't collide but adding a Rigidbody2D to one makes it a **Dynamic** collider and **Dynamic**/**Static** colliders do interact.

Only a very limited set of properties are available on this body-type as can be seen here:


Rigidbody 2D
Body Type: Static
Material: None (Physics Material 2D)
Simulated: ✓
Info

So adding a Collider2D to a GameObject that doesn't have a Rigidbody2D component (or on any of its parent GameObject) makes the collider **Static** but also, adding a Rigidbody2D component and setting its body-type to **Static** is the same. So why have both methods? The first reason is backwards compatibility; it is very convenient to add a collider without a Rigidbody2D to implicitly indicate **Static**. The second reason is that there may be cases where a collider or set of colliders do need to be **Static** for most of the time but occasionally they are moved and/or need to be configured as a group. Whilst modifying static colliders should be avoided, the penalty isn't so severe that it can never be done however, moving a Collider2D without a Rigidbody2D is more expensive than using a Rigidbody2D with the body-type set to **Static**, particularly when multiple colliders are involved. If a set of **Static** colliders need to be moved, especially if they are all moved relatively the same then using a Rigidbody2D component with the body-type set to **Static** and moving the Rigidbody2D is far faster.

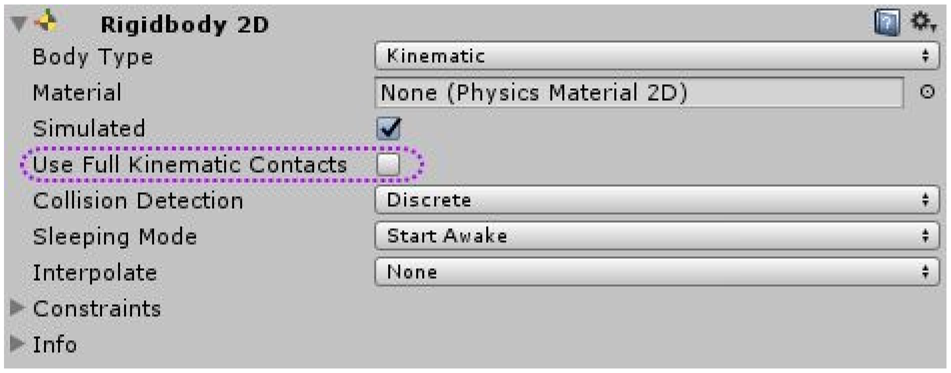
NOTE: *There is a special case where **Static/Kinematic** will interact which is when either of the Collider2D are set to be a trigger. There is also a new feature that changes what a **Kinematic** body will interact with (see **Use Full Kinematic Contacts** below).*

Changing Body Type

Changing the body-type can be a non-trivial process. When a body-type is changed, all existing contacts for the colliders attached to the Rigidbody2D need to be reevaluated during the next fixed-update as well as various mass-related internals are recalculated immediately. Depending on how many contacts and colliders are attached to the body, the performance of changing the body-type can vary.

Rigidbody2D ‘UseFullKinematicCollisions’

When selecting the Rigidbody2D body-type to **Kinematic**, you are presented with an option of ‘*Use Full Kinematic Contacts*’ as can be seen here:



Rigidbody 2D

Body Type	Kinematic
Material	None (Physics Material 2D)
Simulated	✓
Use Full Kinematic Contacts	☐
Collision Detection	Discrete
Sleeping Mode	Start Awake
Interpolate	None
Constraints	
Info	

Typically, when using a **Kinematic** body-type, it will only contact with other Rigidbody2D with a body-type of **Dynamic** (trigger colliders are an exception to this rule) and not with other **Kinematic** or **Static** bodies, therefore no *OnCollisionXXX* callbacks will occur. Whilst this is expected behaviour, it can also be inconvenient when using physics queries (raycasts etc) to detect where and how a body should move and needing to have multiple **Kinematic** bodies interacting with each other.

When the ‘*Use Full Kinematic Contacts*’ is false, the standard behaviour is used with only contacts between **Kinematic**/**Dynamic** bodies happening. When set to true, the **Kinematic** body will collide with all body types, similar to a **Dynamic** body however the difference is that the **Kinematic** body will not be moved by the physics system when contacting another body as it acts with infinite mass (an immovable object).

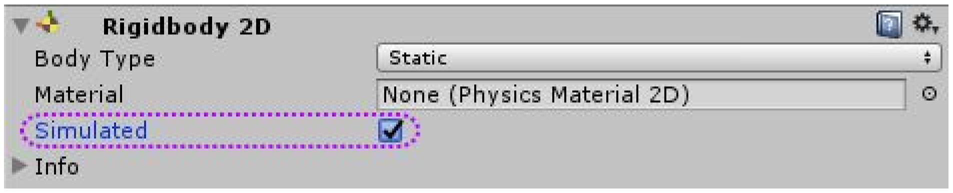
This feature allows explicit position/rotation control of the **Kinematic** body but still allows full collision callbacks. In a set-up where explicit control of all bodies are required, **Kinematic** bodies can be used in place of **Dynamic** bodies and still have full collision callback support.

Rigidbody2D ‘Simulated’

A Rigidbody2D component is used to control the position/rotation of attached colliders as well as allowing joints to use them as anchor points. When the Rigidbody2D moves, so do the attached colliders which then calculate contacts with other colliders on other Rigidbody2D. Joints also constrain Rigidbody2D positions and rotations. All of this takes simulation time.

Both collider and joint components can each be enabled or disabled. When disabled, the physics system doesn’t produce any internal objects to simulate therefore there is zero overhead when in this state. When enabled, the physics system does have internal objects to simulate. Enabling and disabling these components also has an overhead with internal physics objects having to be created and destroyed. This can become expensive when whole GameObject and/or components need to be enabled or disabled.

Each Rigidbody2D has a property ‘*Simulated*’ which controls whether the simulation should simulate the Rigidbody2D and its attached Collider2D and Joint2D. Changing this property is much faster than enabling or disabling individual Collider2D and Joint2D components.


Rigidbody 2D
Body Type: Static
Material: None (Physics Material 2D)
Simulated: ✓
Info

The following happens when set to *false*:

- The Rigidbody2D is no longer moved by the simulation (no gravity, forces etc are not applied)
- Any attached Collider2D contacts are destroyed and they no longer create new contacts
- Any attached Joint2D are no longer simulated and do not constrain any attached Rigidbody2D
- All internal physics objects for Rigidbody2D, Collider2D & Joint2D are left in memory

The following happens when set to *true*:
- The Rigidbody2D begins to move via the simulation (gravity, forces etc are applied)
- Any attached Collider2D reevaluate contacts and continue creating new contacts
- Any attached Joint2D are simulated and constrain any attached Rigidbody2D
- All internal physics objects are Rigidbody2D, Collider2D & Joint2D stay in memory

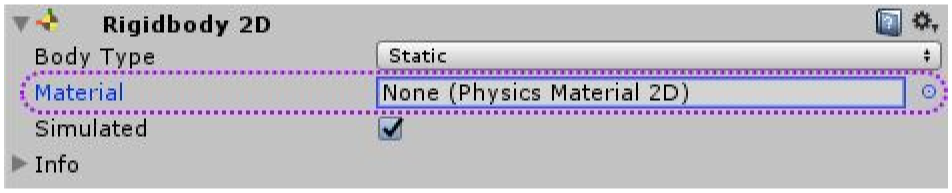
This feature can be used when you want to stop a Rigidbody2D and any attached Collider2D and Joint2D from interacting with the simulation but want to do so as fast as possible without incurring any destruction and subsequent recreation costs but start simulating again later as fast as possible.

When a Rigidbody2D is not simulated, any attached Collider2D will not be found with any physics query i.e. Raycast etc.

Rigidbody2D ‘Material’

All Collider2D have a material property that allows the assigning of *PhysicsMaterial2D*. This material is then used by that collider. If a Collider2D does not specify a material then the default global material is used instead. If no global material is specified then a fallback default is used.

In most cases, this configuration is adequate however it is also common to want to use a common/default material for all Collider2D attached to a specific Rigidbody2D. The Rigidbody2D has a ‘*material*’ property that allows just this as can be seen here:



Rigidbody 2D

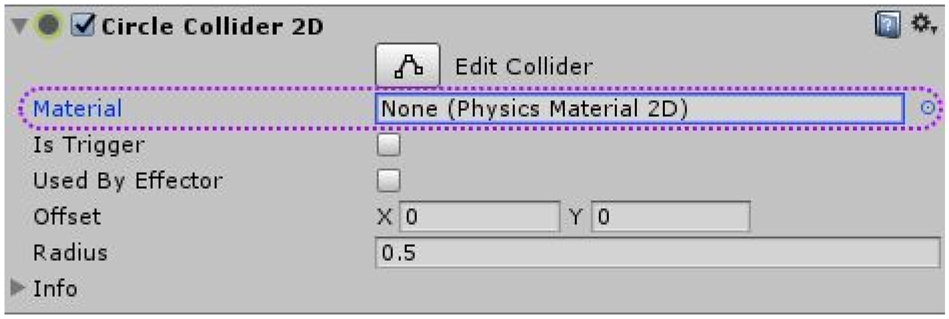
Body Type: Static

Material: None (Physics Material 2D)

Simulated: ✓

Info

When a Rigidbody2D material is set, all attached colliders will use that material unless the collider selects its own material to override that as can be seen here:



Circle Collider 2D

Edit Collider

Material: None (Physics Material 2D)

Is Trigger: ☐

Used By Effector: ☐

Offset: X 0 Y 0

Radius: 0.5

Info

A collider will therefore use the following in order of priority to determine which physics material settings to use. If any are not available then the subsequent one is used.

1. A *PhysicsMaterial2D* specified on the collider itself
2. A *PhysicsMaterial2D* specified on the attached Rigidbody2D
3. A *PhysicsMaterial2D* specified in the *Physics2D* global settings
4. The fixed defaults of *Friction = 0.4* & *Bounce = 0.0*

This feature has the advantage that colliders attached to the same Rigidbody2D set to **Static** can all use the same *PhysicsMaterial2D*.