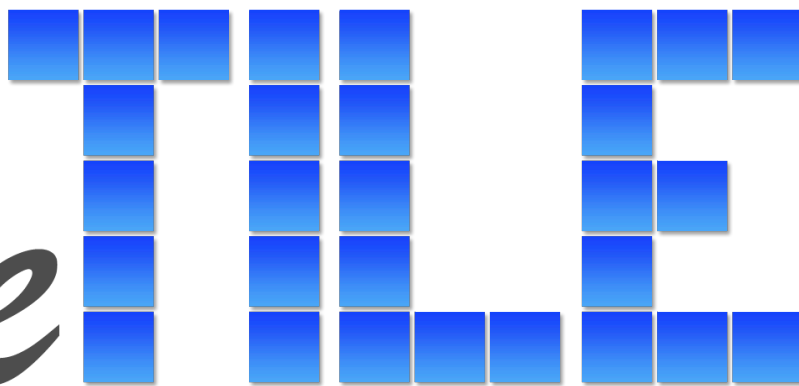


# Sprite



SpriteTile comes as a DLL, and includes an optional source code package. You can have either the DLL (recommended) or the source code in your project, but not both. The Demos package contains several demos that illustrate various SpriteTile functionality.

**Note:** you'll need to add two sorting layers to your project manually if you run the Acorn Antics demo (the Animated Tiles demo also requires an added sorting layer), since SpriteTile can't do this automatically. Select the Edit -> Project Settings -> Tags and Layers menu item to add the layers. See the "[How layers work](#)" section for more details. Have fun!

If you have any questions not answered in the documentation, suggestions for new features, or if you find any bugs, you can post a message in the [SpriteTile topic on the Unity forums](#), or email [sales@starscenesoftware.com](mailto:sales@starscenesoftware.com).

## contents

[Function index](#)-----2

[Terminology](#)-----3

[Tile editor overview](#) ----4

[Tile editor: tiles](#) -----5

([file operations](#) | [tileset controls](#) | [set selection](#) | [tile defaults](#) | [tile window](#) | [view options](#) | [moving tiles](#))

[Tile editor: level](#) -----10

([file operations](#) | [undo](#) | [layer controls](#) | [selection controls](#) | [level window](#) | [cell properties](#) | [view options](#))

[Tile editor: groups](#)-----18

([file operations](#) | [group type](#) | [standard groups](#) | [random groups](#))

[How layers work](#) -----21

[How colliders work](#)-----22

[Coding basics](#) -----24

([Setting the camera](#) | [Int2 struct](#) | [TileInfo struct](#) | [Loading a level](#) | [Creating a level in code](#) | [animation](#))

[Function reference](#) -----30

---

<a href="#">AddLayer .....</a>	<a href="#">30</a>	<a href="#">SetLayerPosition.....</a>	<a href="#">52</a>
<a href="#">AnimateTile .....</a>	<a href="#">31</a>	<a href="#">SetMapBlock .....</a>	<a href="#">52</a>
<a href="#">AnimateTileRange.....</a>	<a href="#">32</a>	<a href="#">SetOrder .....</a>	<a href="#">53</a>
<a href="#">CopyGroupToPosition .....</a>	<a href="#">33</a>	<a href="#">SetOrderBlock .....</a>	<a href="#">53</a>
<a href="#">DeleteTile .....</a>	<a href="#">33</a>	<a href="#">SetRotation .....</a>	<a href="#">54</a>
<a href="#">DeleteTileBlock.....</a>	<a href="#">34</a>	<a href="#">SetRotationBlock.....</a>	<a href="#">54</a>
<a href="#">EraseLevel .....</a>	<a href="#">34</a>	<a href="#">SetTile .....</a>	<a href="#">55</a>
<a href="#">GetCollider.....</a>	<a href="#">35</a>	<a href="#">SetTileBlock.....</a>	<a href="#">56</a>
<a href="#">GetLevelBytes .....</a>	<a href="#">35</a>	<a href="#">SetTileLayerScale .....</a>	<a href="#">57</a>
<a href="#">GetMapBlock.....</a>	<a href="#">36</a>	<a href="#">SetTileMaterial .....</a>	<a href="#">58</a>
<a href="#">GetMapPosition.....</a>	<a href="#">36</a>	<a href="#">SetTileRenderLayer .....</a>	<a href="#">58</a>
<a href="#">GetMapSize .....</a>	<a href="#">37</a>	<a href="#">SetTileScale .....</a>	<a href="#">59</a>
<a href="#">GetNumberOfLayers.....</a>	<a href="#">37</a>	<a href="#">SetTrigger .....</a>	<a href="#">59</a>
<a href="#">GetOrder .....</a>	<a href="#">38</a>	<a href="#">SetTriggerBlock .....</a>	<a href="#">59</a>
<a href="#">GetRotation .....</a>	<a href="#">39</a>	<a href="#">StopAnimatingTile .....</a>	<a href="#">60</a>
<a href="#">GetTile .....</a>	<a href="#">40</a>	<a href="#">StopAnimatingTileRange .....</a>	<a href="#">60</a>
<a href="#">GetTileSize.....</a>	<a href="#">41</a>	<a href="#">UseTileEditorDefaults .....</a>	<a href="#">60</a>
<a href="#">GetTrigger .....</a>	<a href="#">41</a>		
<a href="#">GetWorldPosition.....</a>	<a href="#">42</a>		
<a href="#">GetZPosition .....</a>	<a href="#">42</a>		
<a href="#">LoadGroups.....</a>	<a href="#">43</a>		
<a href="#">LoadLevel .....</a>	<a href="#">44</a>		
<a href="#">NewLevel .....</a>	<a href="#">45</a>		
<a href="#">ScreenToMapPosition .....</a>	<a href="#">46</a>		
<a href="#">SetBorder.....</a>	<a href="#">47</a>		
<a href="#">SetCamera.....</a>	<a href="#">48</a>		
<a href="#">SetCollider .....</a>	<a href="#">49</a>		
<a href="#">SetColliderBlock.....</a>	<a href="#">49</a>		
<a href="#">SetColliderBlockSize .....</a>	<a href="#">50</a>		
<a href="#">SetColliderLayer .....</a>	<a href="#">50</a>		
<a href="#">SetColliderMaterial .....</a>	<a href="#">51</a>		
<a href="#">SetLayerActive.....</a>	<a href="#">51</a>		
<a href="#">SetLayerColor.....</a>	<a href="#">51</a>		

SpriteTile uses terms in a specific way:

A **level** has one or more **layers**. A level is different from a scene in Unity. You can have one scene that loads many levels, and in fact the entire game can be easily done in one scene. (See the Acorn Antics demo for an example.) The layer is also referred to as a **map**.

A **layer** (or **map**) is a grid of **cells**. Each layer can be a different size. Each layer has a different sorting layer ID, which currently must be created manually in Unity using the “Tags and layers” project setting. Each layer can also be a different distance from the camera—this only has an effect if a perspective camera is used.

Each **cell** in the grid has a **tile** associated with it, and other information like rotation, order in layer, and collision. While each cell has one tile, the tile is not necessarily confined to the grid, and can overlap neighboring cells, depending on its size and orientation.

A **collider cell** is a cell that has its collider bit set to true. This isn’t related to physics in Unity. You can use the GetCollider function to query whether a particular cell is a collider cell or not, and make objects react to collider cells as if they are solid. (See the Procedural Level and Trigger Demo examples.) You would typically use this functionality when the exact shape of the tile doesn’t matter and your game doesn’t use physics, such as a top-down RPG, where all you care about is whether cells should be passable or not.

A **physics collider** is made when a specific tile is set to use physics colliders in the TileEditor. In this case, any collider cell that contains a tile which has been set to use physics colliders will generate a polygon collider in the shape of the tile. This collider uses the 2D physics system in Unity. You can mix physics colliders with the GetCollider functionality if desired.

A **sprite** is a texture that has been set to a Texture Type of Sprite, or a Texture Type of Advanced with the Sprite Mode set to Single or Multiple. Note that sprite sheets are possible (and encouraged, since they reduce draw calls). To use a sprite sheet with SpriteTile, set the Sprite Mode to Multiple, and use the [Unity sprite editor](#) to slice it up appropriately. You can then load the sprite sheet and all the sprites inside will be usable as separate tiles.

## Note about DLL and source code usage

If at some point you decide to switch from the DLL to the source code, follow these steps:

- 1) Remove the DLLs from the Plugins folder and the SpriteTile/Editor folder.
- 2) Add the source code.
- 3) On the SpriteTile/Resources/TileManager object, the TileManager reference will now be missing. Replace it with the TileManager script that you just added. Your data will not be lost when you do this.

If you want to switch from the source code to the DLL, the process is the same, except you’d remove the source code from the appropriate folders and add the DLL again.

---

# • tile editor

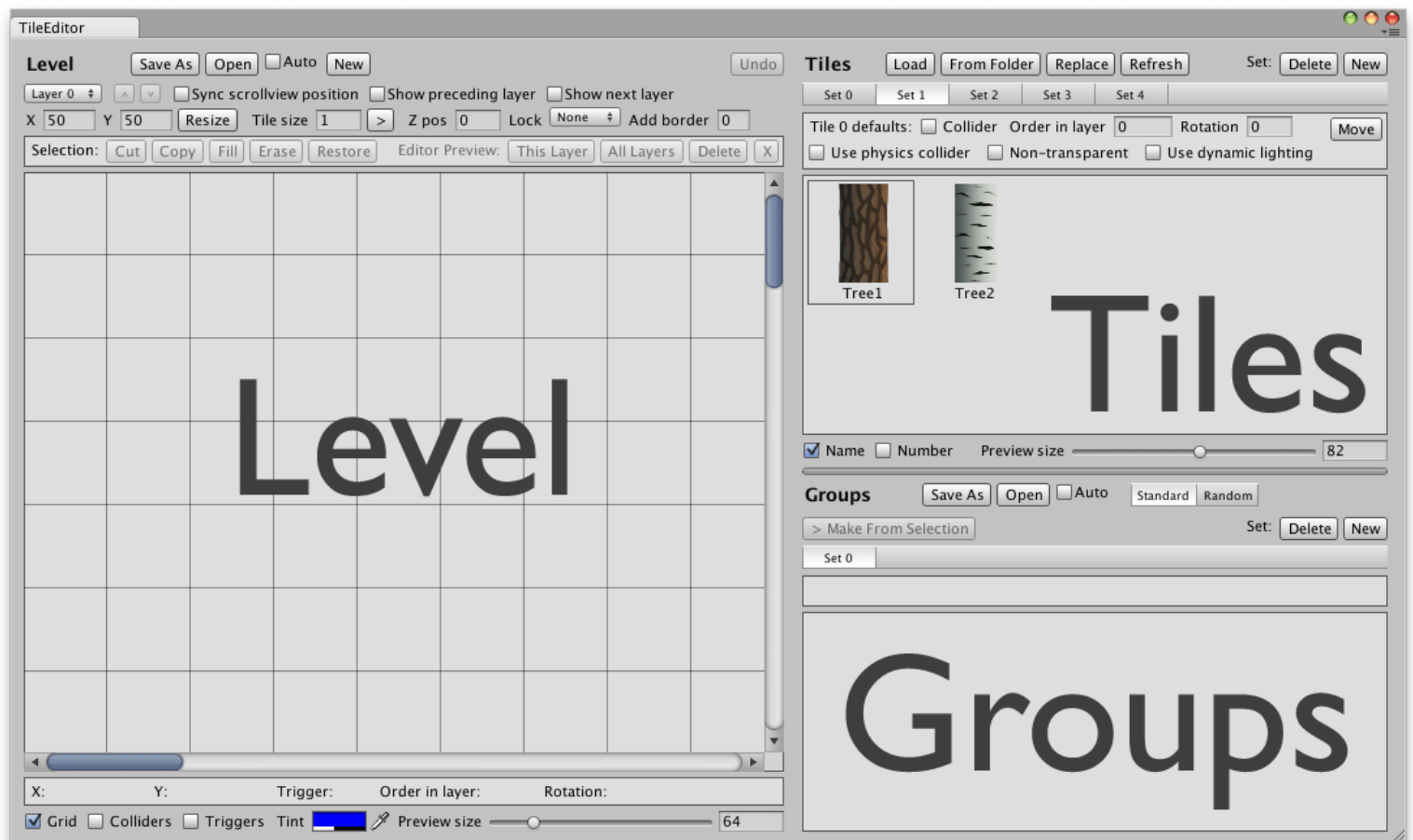
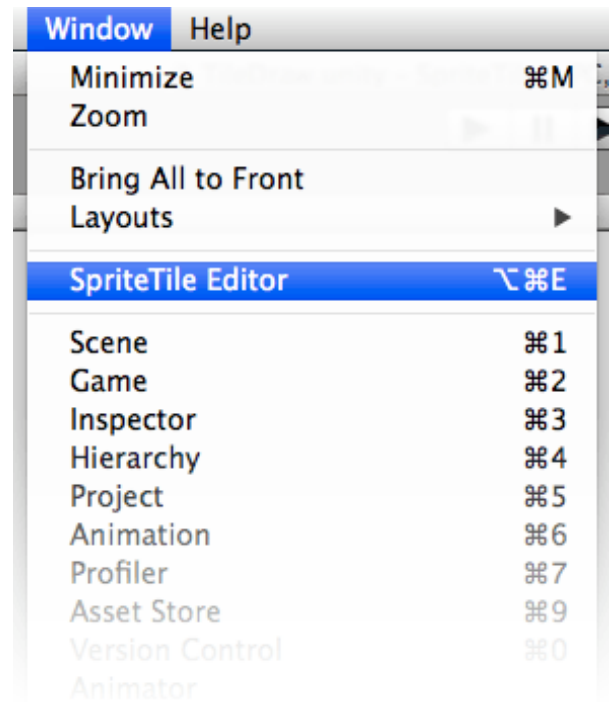
4

The TileEditor window is where you can create and edit SpriteTile levels. You can access it by selecting “SpriteTile Editor” in the Unity Window menu, or by pressing Alt-Cmd-E (on Mac) or Alt-Ctrl-E (on Windows).

The TileEditor is divided into three main sections: **Level**, **Tiles**, and **Groups**. Level is where you edit the level itself, Tiles is where you select and manage the various tiles that make up the levels, and Groups is where you select and manage your own groups of tiles that you can put together for quick access.

These sections are all independent from each other. That is, the tiles are associated with your project, and are available for all levels that you make. So if you make a new level, the tiles stay the same. Likewise, groups are separate from levels, so you can use the same groups in any level, and group files are loaded and saved separately from level files.

You need at least one tile before you can draw anything in the Level section, so we’ll start by taking a look at the Tiles section.



# • tile editor: tiles

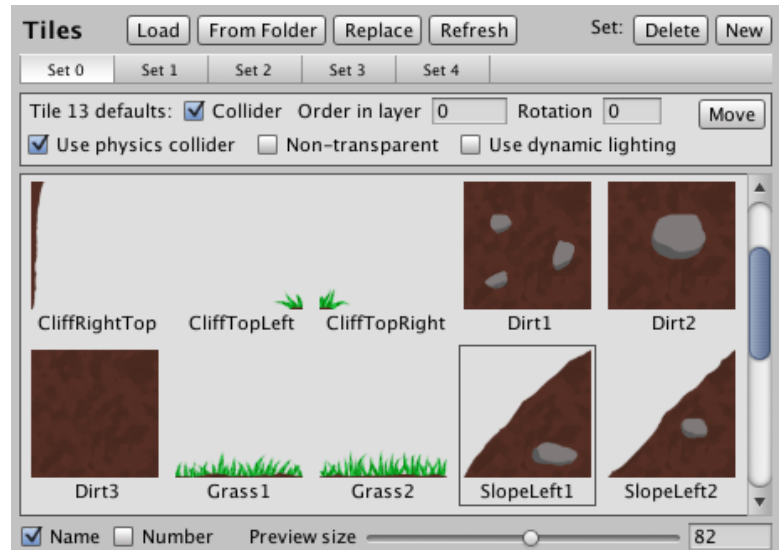
5

This is where you access all the sprites you're using as tiles for all your levels. The tile information is stored in a file called TileManager in the SpriteTile/Resources folder in your project. This file is created automatically when you open the TileEditor window for the first time.



**Note:** the TileManager file should not be renamed or moved! It's required for SpriteTile to function.

Also note that sprites should typically not be put in the Resources folder. Doing so will prevent Unity's auto-atlas function from working.



## file operations

**Load:** selects a texture from your project. The texture must be a sprite for SpriteTile to be able to use it. Textures can be set as sprites either by setting the Texture Type in the import settings to Sprite, or by setting the Texture Type to Advanced, with the Sprite Mode set to either Single or Multiple.

If a texture is a Single sprite type, then the texture will be loaded as a tile. If a texture is a Multiple sprite type, then all the sprites in the texture are loaded at once as multiple tiles. Single and Multiple sprites can be mixed freely and mostly work the same. The main difference is that the Replace button won't be usable if a Multiple sprite is selected. Note that if Multiple sprites contain more than 1024 tiles, or if they would cause the tile count for the current set to exceed 1024, they won't be loaded.

If you attempt to use Load for a Single sprite that already exists in the current set, you'll get a message to this effect and it won't be loaded. However, if you've added new tiles to a Multiple sprite, you can use Load to re-load that sprite, and the additional tiles will be added. The tiles that already exist in the current set will be ignored.

**From Folder:** loads all the sprite textures in a folder into the current set at once. This is a good way to add lots of sprites very quickly, so it makes sense to organize your sprites by folder so you can load them into sets instantly.

Note that if you've added new tiles to a folder, you can use From Folder again and only the new tiles will be added. If you use From Folder and all the tiles in that folder already exist in the current set, then you'll get a message saying so and nothing will happen.

**Replace:** loads in a new tile to replace the currently-selected tile (the one with a thin box around it). This only works for Single sprites, since it wouldn't make sense to replace part of a Multiple sprite.

**Refresh:** use this button if you've changed some property of the sprites that exist in your project and want them to be updated in the SpriteTile editor. For example, if you rename a sprite in Unity, it will keep its old name in the Tiles window until you click Refresh. Altering the sprite size or pivot also requires you to click Refresh for the changes to show up. If you've added new tiles to a Multiple sprite, refresh won't add them; use the Load button for that (see above). The following lists detail the properties that can and can't be changed:

## What sort of things can I change?

- The texture name
- The texture location—feel free to move sprites around in your project as desired
- Packing tag
- Pixels to units
- Pivot
- The texture itself (does not require clicking Refresh; SpriteTile will automatically use the new image)
- Other texture properties such as filter mode and compression (these also don't require clicking Refresh)

## What sort of things can't I change?

- Texture type (must always be Sprite, or Advanced with Sprite Mode as Single or Multiple)
- Sprite mode (if a sprite is Single it shouldn't be changed to Multiple or vice versa)
- **The sprite must not be deleted from your project if it exists in the TileEditor window!** If you want to remove it, use the Delete button in TileEditor first, before you remove the sprite from your project.

## Um, I sort of deleted a sprite from my project anyway....

Well, all is not lost. Try not to do it again though! You'll probably get a bunch of errors until you fix this problem. Here's how:

As mentioned above, SpriteTile uses a file called TileManager in the SpriteTile/Resources folder. If you select it, you'll see an item called **Sprite Data List** in the inspector. Open this, and you'll see an array of tile sets, with an array of textures inside each tile set. Keep opening sub-lists until you drill down to the actual **Sprite** entry that corresponds to the sprite you deleted. You'll see **Missing (Sprite)** listed for the deleted sprite. Drag a sprite onto this entry—any sprite will do. Now you can go back to the SpriteTile editor and delete it properly if needed.



## tileset controls

The **Delete** button for the Set control group deletes the currently-selected set. If any tiles exist in the set, you'll be asked to confirm this operation. If you continue, then any tiles that you've used from this set will be removed from the level.



Careful with this! Remember that tiles are shared between all your levels. If you delete a tile set that's used by a level you don't currently have loaded, this may make the level unloadable since it will contain references to tiles that don't exist in SpriteTile anymore.

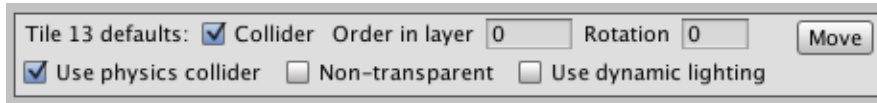
The **New** button, as you might expect, creates a new tile set. It's useful to have different sets that contain tiles that are related somehow. For example, if you have several layers in your levels, you might use Set 0 for tiles that are used in the first layer, Set 1 for tiles that are used in the second layer, and so on. Different themes are another good use for sets—an ice level could use one set, and a volcanic level could use another set. Sets are also used to refer to tiles when using SpriteTile functions in code, so it's a good idea to make sure the tile sets are organized in a way that makes sense for you.

You can have a maximum of 32 sets, and each set can have a maximum of 1024 tiles. It's OK to have empty sets that you reserve for future use. There must always be at least one set; clicking Delete when Set 0 is selected, and there are no more sets, will remove all of the tiles in that set, but Set 0 itself will remain.



## set selection buttons

The buttons on the second row are used to switch between different sets. The currently-selected set is indicated by a pressed-in look. You can also switch sets by using the [ and ] bracket keys on your keyboard, where [ will cycle down through the sets and ] will cycle up.



## tile defaults

The third row is devoted to defaults for individual tiles, plus the physics collider setting and material settings. The number here will change when you click on different tiles, and each tile can have different settings. The defaults shown are always for the currently-selected tile. These defaults are used when you initially lay down tiles in the Level window, but can be changed in the level later if desired on a per-cell basis.

**Collider:** if checked, then any cells where this tile is first drawn will be automatically marked as collider cells. Collider cells can always be toggled on and off in your level as desired, but if you have a tile that you normally want to be used as a collider cell, then it can be convenient to have this checkbox on: it saves having to draw the tile first then set the collider cell later.

**Order in layer:** if tiles overlap, you'll need to define which tiles are drawn on top and which are underneath. If you have certain tiles that you always (or usually) want drawn in a particular order, you can set the default here. Then it will always use the order you defined when being drawn in the level, though you can change it later on a per-cell basis. See [Order in layer](#) in Cell Properties for more details.

**Rotation:** since tiles can be rotated to any degree, you may want some tiles to normally have a particular rotation when you draw them in the level. Setting the rotation of a tile here will cause it to always have that rotation when drawn, though again you can change each cell later as desired.

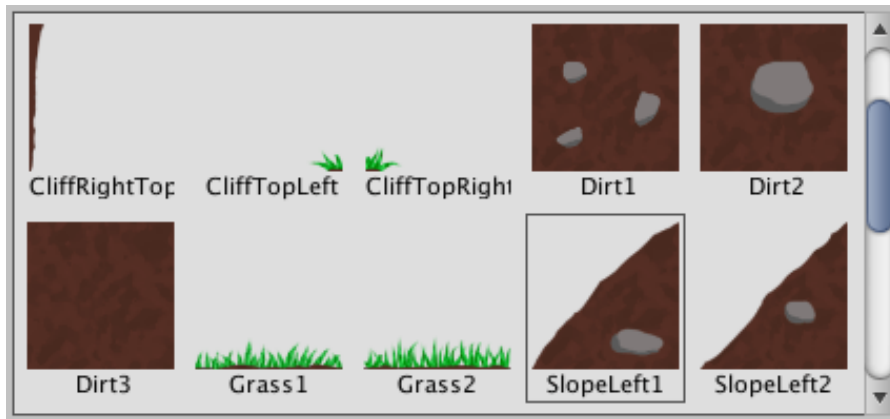
**Move:** allows you to rearrange the order of tiles in the list. See [Moving Tiles](#) below.

**Use physics collider:** this is used to mark individual tiles as polygon collider generators. Regardless of whether cells in your level are marked as collider cells, to use a tile as a physics colliders, this checkbox must be on. See [How Colliders Work](#) for more details.

**Non-transparent:** since the default sprite shader uses transparency, it may be less than optimal to have all your sprites use transparency, in those cases where the sprites don't actually have any transparent pixels. For example, with a top-down RPG, most of your ground tiles are likely to be solid squares of grass, dirt, etc. By clicking this checkbox, the sprite will use a material with a non-transparent shader (that is otherwise identical to the default sprite shader). For best results when using atlases, group your tiles by transparency, so that transparent tiles are in one atlas and non-transparent tiles are in another. Since this is a default, it can be overridden by using [SetTileMaterial](#).

**Use dynamic lighting:** along the same lines, the default sprite shader is unlit, so if you want tiles to react to realtime lighting, then you can use this checkbox. This can be mixed with Non-transparent, so there are four default materials: transparent unlit, non-transparent unlit, transparent lit, and non-transparent lit. This setting can also be overridden by using [SetTileMaterial](#). Since using lighting is slower than not using lighting, it's best to leave this off unless you specifically need it.

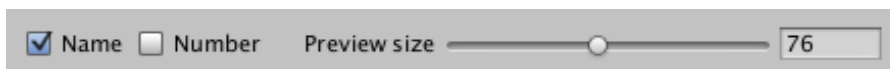




## tile window

This shows a list of all tiles in the selected set. Select a tile by clicking on it; the currently-selected tile is shown with a thin box around it. The selected tile is the one that's used for operations in the Level window such as drawing or filling areas. You can also switch between tiles by using the ; and ' keys on your keyboard, where ; cycles to the previous tile and ' cycles to the next tile.

**Multi-select:** Note that you can select multiple tiles at once by holding down **Shift** and clicking on another tile. This will cause both tiles and every tile in between to be selected. With multiple tiles selected, changing any of the defaults will be applied to all selected tiles. This is also useful for adding tiles to [Random groups](#).



## view options

**Name:** this checkbox toggles whether the sprite file name is displayed under the tile images or not.

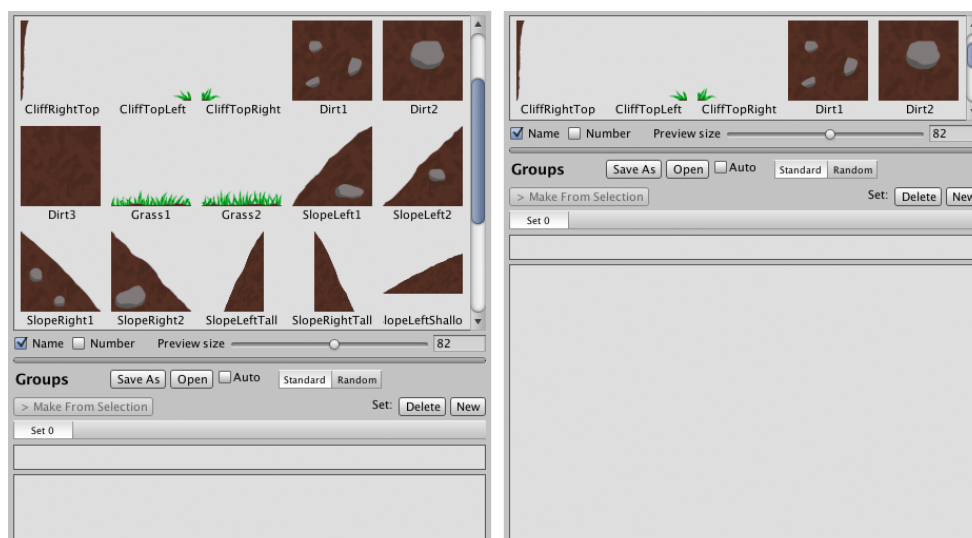
**Number:** this checkbox toggles the tile number display under the tile images. The tile number is frequently useful for coding, so you can refer to the appropriate tile with functions like SetTile and so on.

**Preview size:** you can use this slider to control the size of the tiles in the tile window. If you have many tiles in a set, it may be useful to reduce the size so you can see more tiles at once. This has no effect on tiles displayed in your game; it's purely cosmetic and affects the TileEditor window only.



## divider

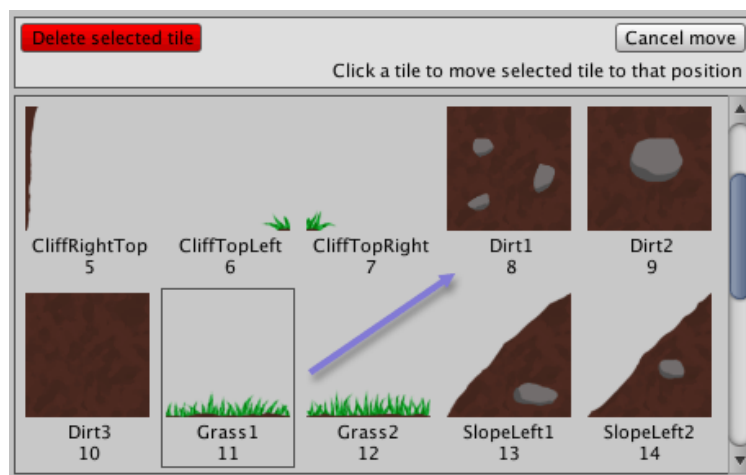
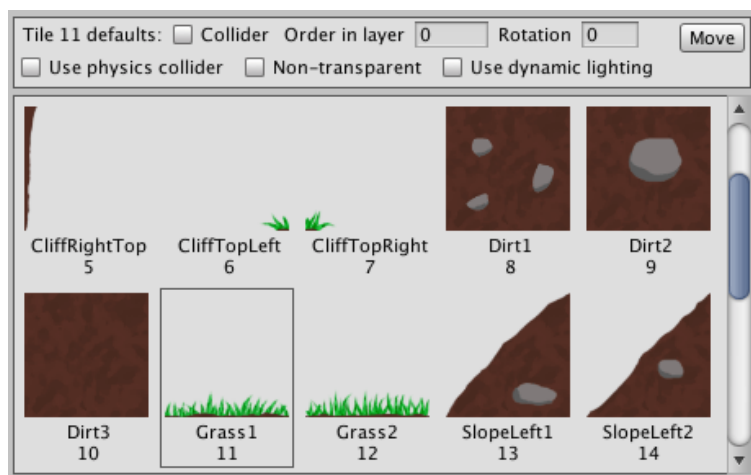
You can drag this up or down to make more room for the groups or the tiles sections, as you prefer.





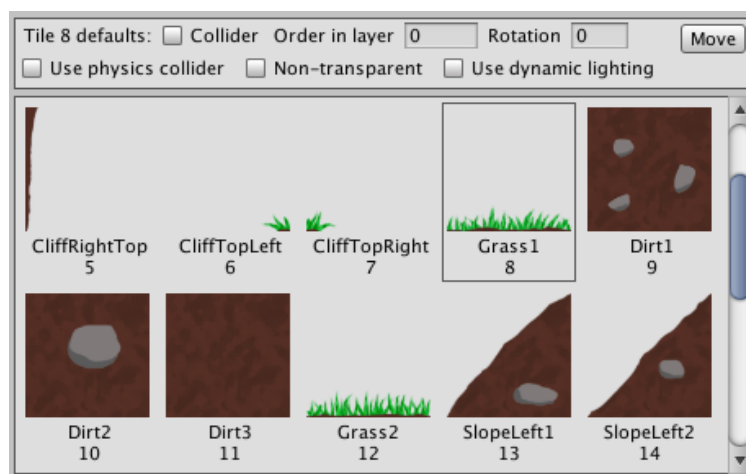
## moving tiles

There are times when you might want to rearrange the order of the tiles in the list. To do this, first select the tile you want to move, then click the **Move** button. For example, here we have some tiles that we've imported where they are initially in alphabetical order, but later on we decide that it would be better for the grass tiles to be grouped together. Perhaps we have code where it would be nice if those tiles had sequential values. In any case, we start by clicking on tile #11, then click Move. The tile defaults change to a new set of buttons (described below).



We then click on tile #8, so tile #11 becomes #8 in the list and the following tiles get pushed down. If we repeat the process by clicking on tile #12, Move, and then tile #9, the grass tiles would be all in a row.

**NOTE:** moving tiles in this list will not affect how the level looks. You won't need to re-draw anything, which also applies to any other levels or groups you might have saved, so you can rearrange tiles to your heart's content without damaging anything. You will possibly need to update code, though that's really the point of this exercise. For example, if your code was expecting Set 0, Tile 11 to be the Grass1 tile, after moving it, you'd use Set 0, Tile 8 instead.



**Delete selected tile:** this button removes the selected tile from the current set. Any instance of the tile in your level will be removed. If any groups contain the tile, they will be deleted (after confirmation). Note that deleting a tile will change the numbering of any tiles after the deleted tile.



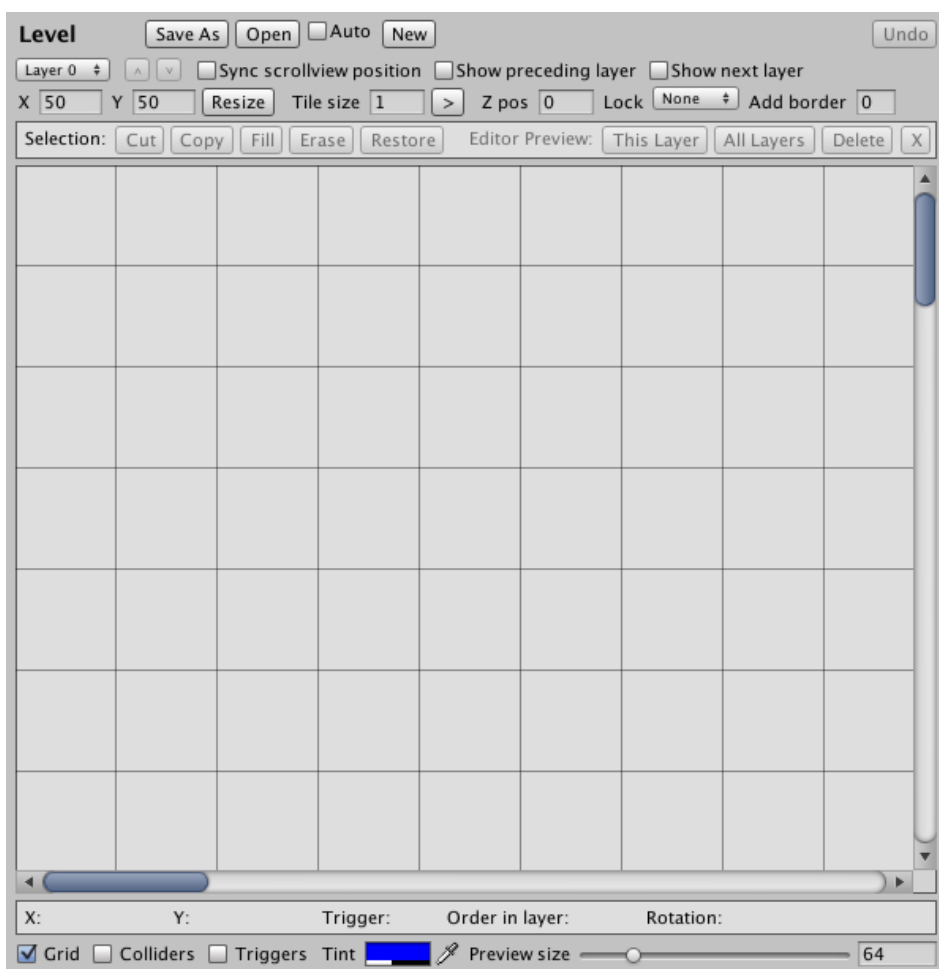
Careful with this! As mentioned above about deleting sets, deleting tiles can mess things up for any levels or groups that you don't have loaded. Only use it if you're really sure the tile is not used anywhere else. If in doubt, just leave it—having an unused tile doesn't really hurt anything.

**Cancel move:** pretty obvious...if you decide not to move a tile, click this button. Clicking the same tile again will also cancel the move.

The **Level** section is where you will spend most of your time building levels. You can draw with the currently-selected tile into the Level window, and set properties for any tile in the grid.

Note that the Level section is independent from the Tiles section, so any tiles you have in your project apply to any and all levels that you may work on. As long as you've added some sprites from your project, they will always show up in the Tiles section when you open the TileEditor window, but levels, on the other hand, must be loaded. You can have an unlimited number of levels saved in your project.

You can close the TileEditor window and re-open it without having to re-load a level, but only as long as you haven't saved any scripts or entered play mode. So it's a good idea to save levels before you close the window, since that will eliminate the possibility of changes being lost.

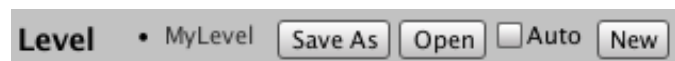


## file operations

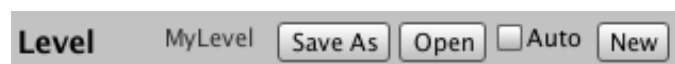
**Save As:** this will, naturally, save the level into your project, into a location of your choosing. SpriteTile levels are saved with a .bytes extension, so they can be easily loaded as TextAsset files inside Unity. They can also be loaded as external files by using System.IO.File operations such as ReadAllBytes (in which case the extension doesn't really matter). You can also press **Alt-Shift-S** to do a Save As operation.

Once a file is saved or loaded, the file name will show up to the left of this button. This tells you at a glance what level you're currently working on.

Note that files with unsaved changes have a dot next to the file name. So if you've called your level "MyLevel", it will look like this:



Once you've saved the level, it will look like this:



You can press **Alt-S** to save a changed level without having to go through the Save As dialog.

**Open:** loads a new level, which replaces the current level, if any. Make sure you save changes before loading a new level in place of an existing one, since loading can't be undone.

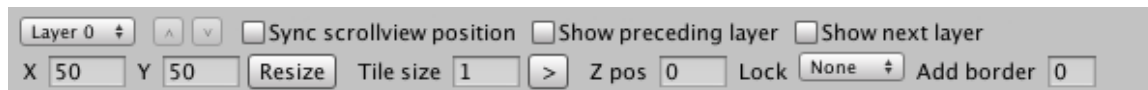
**Auto:** if this is checked, the last saved or opened level will be automatically loaded every time you open the TileEditor window.

**New:** deletes the current level so you can start from scratch. You'll get a dialog asking you to confirm this operation, since it can't be undone.

## undo

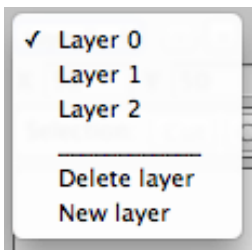
**Undo:** undoes the last action you did when drawing or changing the property of a tile in the Level window. There's no particular limit to the number of times you can undo, although certain actions will reset the undo buffer, such as creating a new level or deleting a layer. The button is grayed out if no undos are available. You can also use **Alt-Z** instead of clicking the button.

Note that if you perform an action on one layer, switch to another layer, and then perform an undo, you'll be switched back to the layer you were on when you performed that action. However, if you have "Show preceding layer" or "Show next layer" checked (see below), and the action you performed was on the shown layer, you won't be switched, since you can see the undo operation performed.




## layer controls

**Layer drop-down menu:** this allows you to switch between layers, delete the current layer, or add a new layer. See [How Layers Work](#) for more details. You can also use **alt-up arrow** and **alt-down arrow** to change the current layer.



**Delete layer:** this will delete the currently-active layer.

**New layer:** this will add a new layer to the list. Before creating a new layer, you can enter the desired X and Y dimensions of the new layer (without clicking on the Resize button) and the layer will be sized appropriately. Of course, you can also change the size later after creating the layer.

 **Up and down buttons:** if you have more than one layer, these allow you to move the current layer up or down in the list of layers (as shown in the drop-down menu), in order to rearrange the layers. For example, if you had layer 1 as the current layer and clicked the up arrow, it would become layer 0, and the old layer 0 would become layer 1. The arrows are grayed out if the operation wouldn't be possible. For example, the up arrow is non-functional if you have layer 0 selected, since there aren't any layers above it.

**Sync scrollview position:** if this is checked and you have multiple layers, and you switch between layers that have the same X and Y dimensions, then the scrollview position and level preview size will be synchronized between the layers. This is useful for when you have stacked layers and are placing tiles that should be lined up with other layers, since it saves having to manually scroll each layer to the same place. If this box is not checked, then each layer remembers its own scrollview position and preview size.

**Show preceding layer:** if this is checked, the preceding layer (if possible) will be shown underneath the current layer, at 50% opacity. It's possible for the preceding layer to be shown if there actually is one (for example, layer 0 doesn't have one) and it has the same dimensions as the current layer. If it's not possible, the control will be grayed out.

**Show next layer:** this is similar to showing the preceding layer, but for the next layer, which will be drawn on top of the current layer at 50% opacity. It's not possible to show the next layer if the current layer is last in the list, or if the next layer has different dimensions.

**X and Y boxes:** these control the width and height of the level, in terms of number of cells. 50 x 50 is the default but you can use any numbers, with 1 x 1 as the minimum. The actual size of the level in terms of units depends on the tile size (see below).

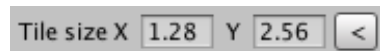
**Resize:** the X and Y values you enter won't actually take effect until you click the Resize button.

**Tile size:** how big each cell is in units. When creating a new layer, the tile size is automatically set depending on how big the first sprite in the current set is. For example, if the sprite is 200 pixels wide and uses 100 pixels to units in its import settings, then the tile size will be 2.0.

Typically you would use a tile size so that tiles are positioned right next to each other—for example, if your sprites are mostly 100 x 100 pixels and use 100 pixels to units in the import settings, then the tile size should be 1.0. However this isn't enforced—tiles can have gaps between them or overlap with no problems—and you can change the tile size to any number, as long as it's at least .01.

You can get the total size of a level in units by multiplying the X or Y values by the tile size, so a 50 x 25 level with a tile size of 2.0 would be 100 x 50 units.

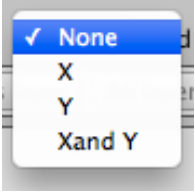
**> button:** normally the tile grid is square, but if you need non-square tiles, the ">" button expands the tile size to show the X and Y sizes separately. You can then change these independently, as shown in the example below where the Y size is twice the X size.



**< button:** this sets the tile size back to square.

**Z pos:** how far the layer is positioned from the origin on the z axis. This only has an effect if you use a perspective camera. With an orthographic camera, the value won't make any difference.

**Lock:** when you move the camera, layers will normally scroll freely. You may want to prevent some layers from scrolling, however, such as a background layer that you want to be locked in place.



**None:** the layer will scroll with no restrictions.

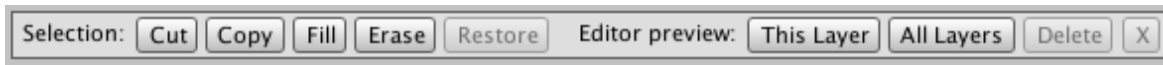
**X:** the layer is locked on the X axis. It can scroll up and down on the Y axis, but will never move horizontally.

**Y:** the layer is locked on the Y axis. It can move back and forth on the X axis, but will never move vertically.

**X and Y:** the layer is locked on both axes and will not move at all.

**Add border:** If you have oversized tiles that don't fit inside cells, you may find that they “pop in” when scrolling rather than smoothly becoming visible. To fix this, you can add more cells that extend beyond the border of the screen. Keep in mind that every number above 0 decreases efficiency a bit, so try to use an add border value that's only as big as you actually need. A good rule of thumb is that if an oversized tile has more than two cells of overlap, add 1 to the add border value. For example, if your biggest tile completely overlaps three cells, then use 1 for the add border value. If the biggest tile overlaps four cells, then use 2 for the add border value, and so on.

Note that the Unity GUI has some limitations when it comes to displaying rotated and scaled tiles inside a scrollview, and the add border value has no effect in the editor anyway, so what you may see in the TileEditor when scrolling is **not** representative of what you'll actually see in your game. You'll need to run your level “for real” in order to test the add border value.



## selection controls



Aside from clicking on individual cells, you can also select an area. Click and drag with the right mouse button down to select an area, which is indicated by a black and white outline with a faded interior. In the illustration to the left, the selected area is shown as the 4 x 3 cell outline.

You can remove the selection box by either using the Escape key, or right-clicking in a spot without dragging, or using the X button.

If there's no selection, the buttons are grayed out.

**Cut:** removes all tiles within the selection box and puts them into the copy buffer. The copy buffer appears as a faded-out representation of the tiles in the selection, which you can move around and paste into the level by left-clicking. You can delete with the copy buffer by middle-clicking or using the delete key. You can also press **Alt-X** to cut.

**Copy:** puts the selected tiles into the copy buffer without deleting them. Otherwise this behaves the same as using Cut. You can also press **Alt-C** to copy.

**Fill:** fills the selection box with the currently-selected tile. Useful for quickly creating large areas of tiles. If a [Random group](#) is selected and "Draw with selected group" is active, then each individual tile in the selection box will be randomly selected from the tiles in the group. You can also press the **F** key to fill.

**Erase:** deletes all tiles within the selection box. You can also press the **delete** key.

**Restore:** if you had a copy buffer active, then deactivated it, you can restore it with this button.

**Editor preview:** SpriteTile normally operates in play mode, but you can use these buttons if you need to see what a level looks like in the Unity scene view.

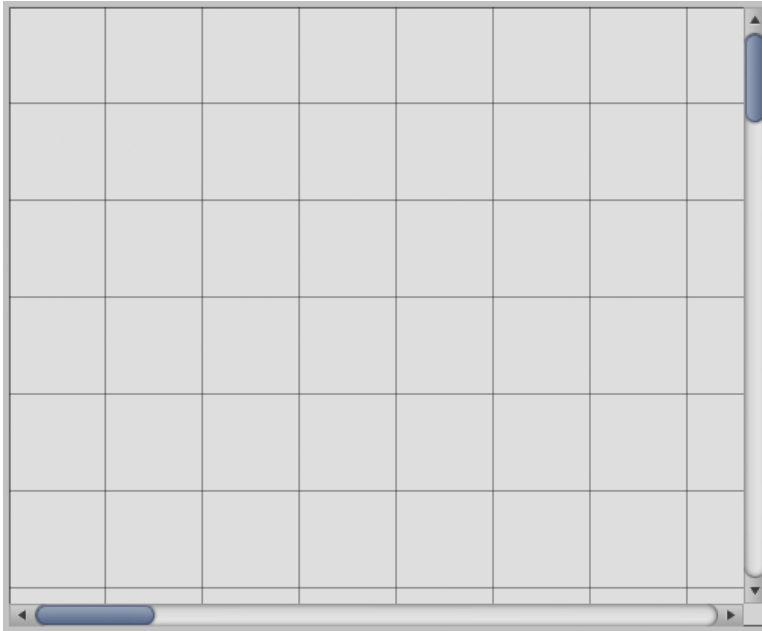
**This Layer:** shows the current layer in the editor, using the contents of the selection box. You can use **Alt-A** to select the entire layer. Note that every tile will be displayed using a separate GameObject for the preview, so to prevent the Unity editor from getting bogged down by too many objects, the preview is limited to 10,000 tiles. This can be any configuration, such as 100x100, 200x50, etc. You'll get a message if your selection is too big to be previewed.

**All Layers:** shows all layers in the editor, using the contents of the selection box. This works best if all layers have the same dimensions, but will still work even if they don't, though it may be difficult to tell ahead of time exactly what will be shown in that case.

**Delete:** removes the preview from the scene view. This will also happen when the TileEditor window is closed.

**X:** this removes the selection box (but doesn't delete tiles). It will also deactivate an active copy buffer. You can also do this by pressing the **Escape** key or right-clicking on one spot in the level.





## level window

This is where you actually create your level. When you move the mouse pointer over the level window, the selected tile will be highlighted:



Clicking on the highlighted tile will put the currently-selected tile in that cell. You can also click-and-drag to place multiple tiles.

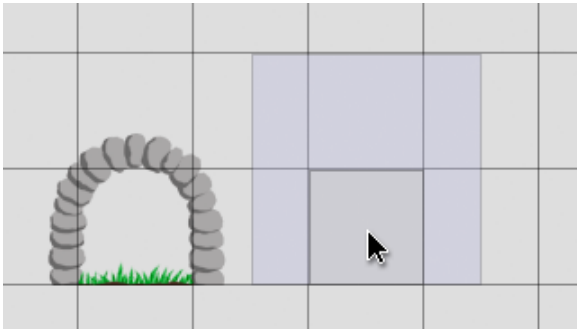
Middle-click to delete tiles (also works with click-and-drag). You can also use the **Delete** key.

If the level is too big to see all at once, you can move around by using the scroll bars or the mouse scroll wheel. You can also pan the view by holding down the **Space** bar and clicking while dragging.

As a shortcut for selecting tiles in the Tiles window, you can press the **P** key to pick whatever tile is in the highlighted cell.

## overlapping tiles and anchor points

If you have a tile selected that doesn't fit exactly into a cell, then in addition to highlighting the actual cell that the mouse pointer is over, an additional highlight in light blue shows the size of the tile. Here we have a gate tile that covers several tiles around it:



Note that oversized tiles don't prevent you from placing other tiles in the surrounding cells. If you need to change the anchor point of the tile, the method depends on whether the sprite used for the tile has its Sprite Mode as Single or Multiple.

For Single sprites, go to the texture import settings for the sprite in Unity and change the Pivot from Center to something else, most likely Custom. This gate tile, for example, is twice the size of a normal tile and has the custom pivot set to  $X = 0.5$ ,  $Y = 0.25$ , so that it lines up correctly. When done, click Apply.

For Multiple sprites, click on the sprite atlas in Unity, then click on the Sprite Editor button. You can then click on the sprites in the sprite editor and change the pivot points for each one as desired. When done, click Apply in the sprite editor window.

In either case, in order to have the changed pivots applied to the appropriate tiles in the TileEditor, click Refresh in the Tiles section.



X:	Y:	Trigger:	Order in layer:	Rotation:
----	----	----------	-----------------	-----------

## cell properties

The information here is blank unless the mouse pointer is actually over a cell. Each cell has its own info that can be changed independently from all other cells.

**X and Y:** the cell coordinates. The world coordinates are displayed in parentheses after the cell coordinates. If the tile size is 1.0, then the world and cell coordinates would be the same. You can use the cell coordinates to refer to specific cells when using the SetTile or GetTile functions; the world coordinates are for informational purposes (such as lining up objects in the scene view). SpriteTile coordinates are bottom-up, where (0, 0) is the lower-left corner.

**Trigger:** an arbitrary ID number that you can assign to cells, ranging from 0 to 255. The trigger doesn't actually do anything by itself, but you can use it with the GetTrigger function to program various events.

**Order in layer:** this is the same as Order in Layer when used with sprites in Unity. If you have oversized tiles that overlap, higher numbers draw on top of lower numbers. You can also use this to interact with "regular" sprites that you create outside SpriteTile. For example, if you have a sprite character that uses an order of 0, and the character is in the same spot as a particular tile with an order of 1, then that tile will be drawn on top of the character. The order number can be in the range from -32768 to 32767.



Note that overlapping tiles or sprites that use the same number for the order in layer are not defined as to which actually draws on top. They may or may not appear correct, and it could change later... even if it looks right in the editor, it might be wrong in a build. So double-check the order for cells that overlap and make sure it's set explicitly.

**Rotation:** this can be any number from 0.0 to 360.0, by increments of  $\frac{1}{5}$  (0.2) of a degree. So 15.1 would become 15.0, and 15.25 would become 15.2.

You can change the trigger, order in layer, and rotation by alt-clicking on a particular cell, in which case the cell properties are highlighted and the numbers become editable:

X: 7 (8.96)	Y: 44 (56.32)	Trigger: 0	Order in layer: 0	Rotation: 0	Done
-------------	---------------	------------	-------------------	-------------	------

You can enter the desired numbers, then click **Done** or hit the **Return** key. If you change your mind, press **Escape** to cancel. (You might encounter a Unity textfield bug where the number fields don't behave correctly. If so, closing the TileEditor window and re-opening it seems to fix the problem.)

Note that if you have an active selection box, the info here will apply to all the cells in the selection. You can use this to do things like set trigger areas without having to edit every cell individually.

As a shortcut for editing **Order in layer**, you can use the , and . keys to cycle the order up and down for whatever cell the mouse pointer is currently over.

As a shortcut for editing **Rotation**, you can use the , and . keys with **Alt** held down to cycle the rotation up or down by 5 degrees. You can use the , and . keys with **Shift** held down to cycle the rotation up or down by 90 degrees.



## view options

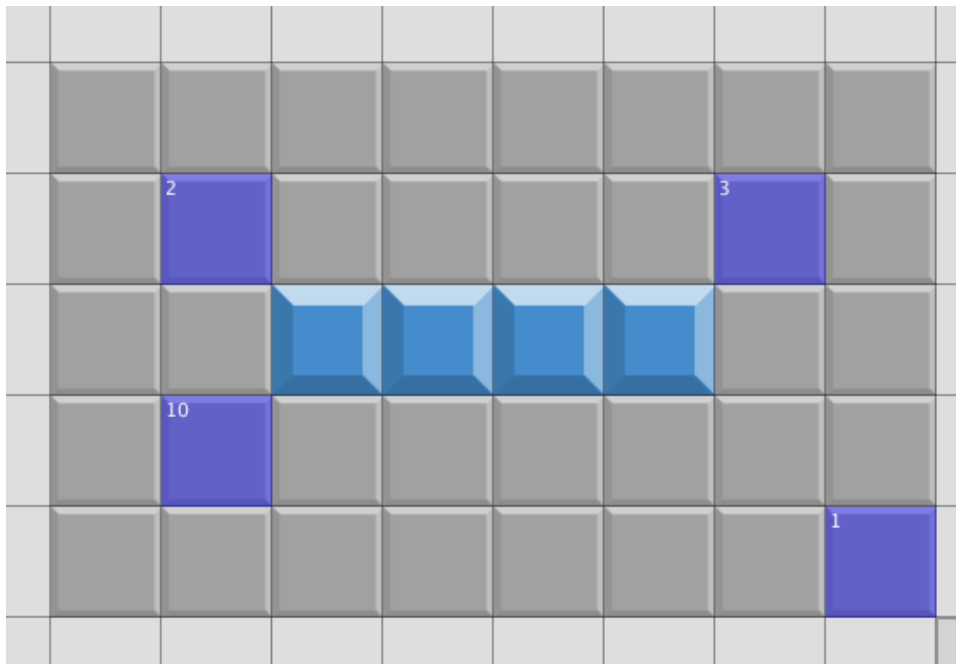
These control how the level is shown in the editor. They're cosmetic only and don't affect how your level appears in your game.

**Grid:** this allows you to toggle the grid on and off. It may be easier to place tiles with a visible grid, but if it's distracting then you can turn it off. You can also press the **G** key to toggle the grid on and off.

**Colliders:** this shows any cells that are an active collider cell by using a tinted overlay (blue by default). See [How Colliders Work](#) for more details. Note that if you're using a physics collider, the entire cell is tinted blue even if the tile is not in the shape of a square. In other words, colliders are shown on a cell-by-cell basis and aren't necessarily representative of the actual polygon collider that's automatically generated. You can also press the **O** key to toggle the overlay on and off.

Note that when the Colliders overlay is on, drawing with the mouse will draw collider cells instead of tiles. You can left-click to set collider cells and middle-click to delete them. If you have an active selection box, then you can press the **F** key to fill in the selection with collider cells, and the **Delete** key to remove them. So with the Colliders overlay on, drawing operations only affect the collider cells, and leave the tiles themselves alone.

**Triggers:** similar to Colliders, this will show a tinted overlay for any cells that have a trigger ID of greater than 0. The actual trigger number is also drawn on the tiles. You can use this to quickly see where you placed triggers. Unlike Colliders, you can continue drawing tiles normally while the Triggers overlay is active. You can also use the **T** key to toggle this. Note that either the Colliders overlay or the Triggers overlay can be active, but not both at once. Here we have four triggers, labeled 1, 2, 3, and 10:



**Tint:** if you'd rather have some other color instead of blue, change it here to whatever you like.

**Preview size:** changes the size of tiles in the level window; you can use this to zoom in or out. Holding down the **Alt** key while using the mouse scroll wheel will also work.

The **Groups** section is where you can manage groups of tiles. Groups can be saved and loaded like level files. They're independent from levels, though, so you can use the same groups with any number of levels. Like tiles, you can have different sets, with multiple groups in each set. There are two types of groups, Standard and Random, which are saved together in the same file.

## file operations

**Save As:** similar to the Save As button for levels, this saves the group sets into a file. Likewise, any unsaved changes to the groups makes a dot appear next to the Save As button.

**Open:** loads a group file. You can also use the Open button in the Level section—if you use the Open Level button, and the TileEditor only finds groups in the file, it will ask if you want to load the file as groups rather than as a level.

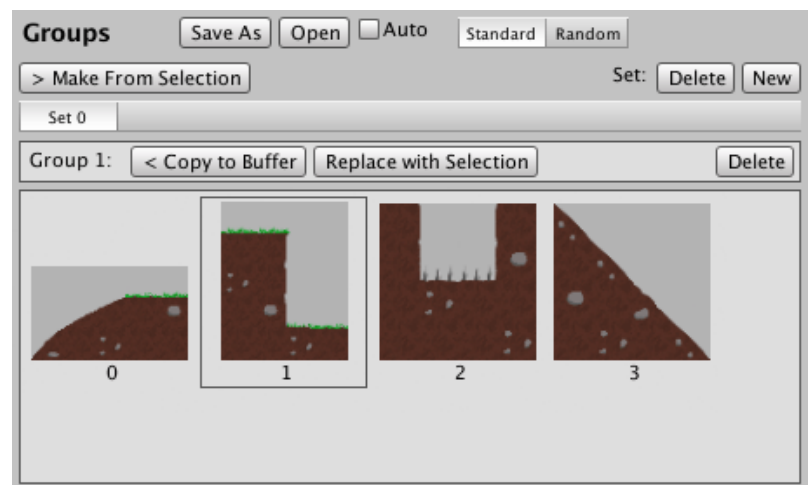
**Auto:** if checked, the most recently saved or opened group file is automatically loaded every time the TileEditor window is opened.

## group type

You can switch between Standard and Random groups with these buttons. Standard and Random groups have somewhat different options, as detailed below in the respective sections.

# Standard

With **Standard** groups, if you have part of a level that you want to re-use frequently, then you can save it to use as a building block here. The basic idea is that you can turn anything in the selection box into a group.



## make from selection

**Make From Selection:** if you make a selection box in the Level window, then you can copy the tiles in that selection into a group by clicking this button. Each time you do this, it will add a new group to the currently-selected group set.



## group set controls

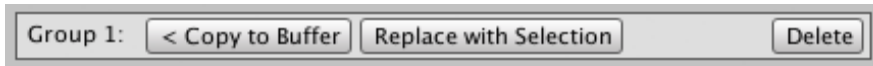
**Delete:** this deletes the currently-selected group. While this should be used with care, it's simpler than deleting tiles, since deleting groups can't affect any levels that you might have saved.

**New:** this creates a new group set. If you have a lot of groups, you may find it easier to organize them into sets for easy access.



## group set selection buttons

If you have multiple group sets, then they are represented by buttons here, so you can switch between group sets by clicking the appropriate button. You can have up to 32 different group sets.



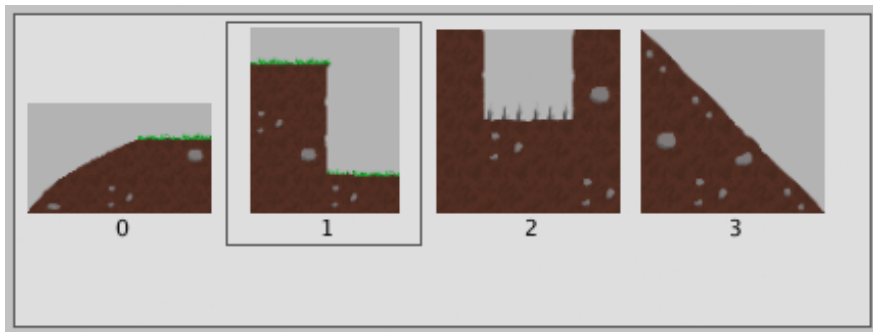
## group controls

These buttons won't appear unless you have at least one group in the set.

**Copy to Buffer:** copies the currently-selected group to the copy buffer, so it can be pasted into the Level window in the usual way by left-clicking.

**Replace with Selection:** if you want to update a group or just replace it with a different one, then clicking this will replace the currently-selected group with whatever's in the active selection box.

**Delete:** removes the currently-selected group.

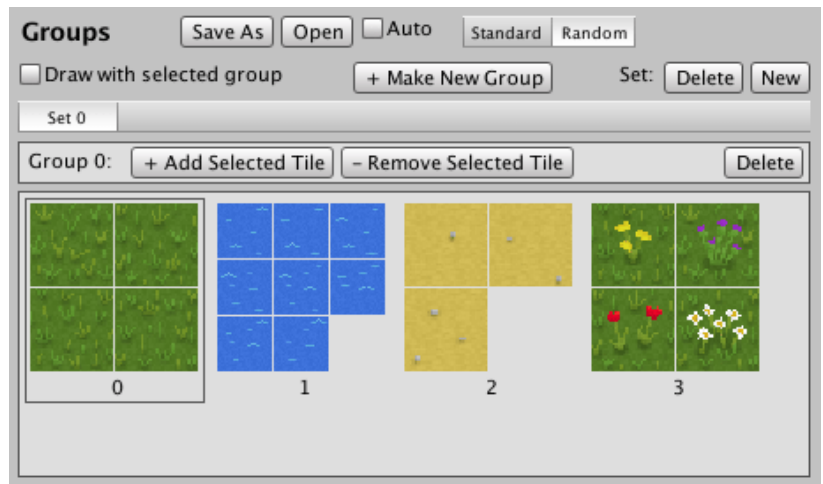


## group window

This works essentially the same way as the Tiles window, although multi-selection is not available. You can see all the groups in a set here, and the currently-selected group is indicated by a thin box around it. Double-clicking a group will select it and copy it to the copy buffer.

# Random

With **Random** groups, you can create lists of tiles where drawing with the Random group will randomly select from the list. This is particularly useful for things like having some variations of a tile (say grass, dirt, water, etc.) where you can quickly create variety by drawing with the Random group instead of having to manually place the tile variations yourself.


☒ Draw with selected group

## draw with selected group

This checkbox will be inactive unless you have selected a Random group that contains tiles. When checked, then you can draw in the Level view, and tiles will be randomly chosen from the currently-selected group, rather than using the selected tile from the Tiles view. Also, if you use the Fill button to fill a selection box, all the tiles in the selection will be randomly chosen individually. Remember to uncheck this when you want to draw with tiles normally!

+ Make New Group

## make new group

**Make New Group:** clicking this will create a new Random group. It will be empty until you add some tiles using the “Add Selected Tile” button.

Group 0: + Add Selected Tile - Remove Selected Tile Delete

## group controls

These buttons won’t appear unless you have at least one group in the set.

**Add Selected Tile:** adds the currently-selected tile from the Tile view to the currently-selected Random group. If you have multiple tiles selected, then this button will read “Add Selected Tiles” and all the selected tiles will be added at once. There is a maximum of 16 tiles which can be added to a Random group. Only one of each tile may be added.

**Remove Selected Tile:** deletes the currently-selected tile from the Random group. If you have multiple tiles selected, then this button will read “Remove Selected Tiles” and all of the tiles that exist in the group will be removed.

**Delete:** this deletes the group entirely. If the group contains any tiles, you’ll get a dialog asking you to confirm this.

Note that the group set controls and set selection buttons work the same way as with Standard groups. Double-clicking a group will select it and turn on “draw with selected” if it’s not on already.


# • how layers work

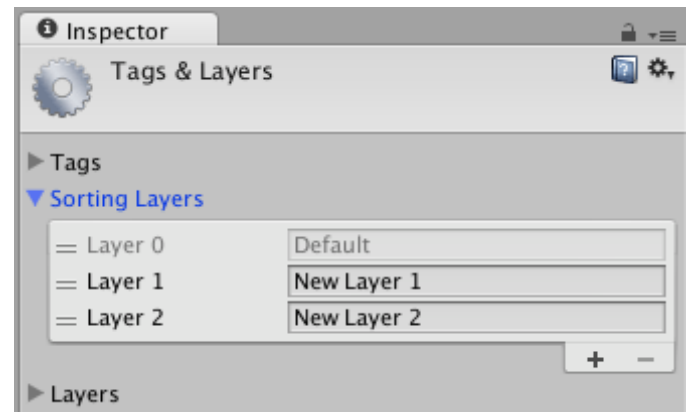
21

Each level can have an unlimited number of layers. You can add layers by using the layer drop-down menu (see [Layer Controls](#)).

Each layer has its own x/y dimensions, tile size, z position (in world space), and layer lock. Layer 0 is the bottom layer, which each additional layer rendered on top of the preceding one. You can set the z position of each layer to any arbitrary number as long as it's not below 0, though this only matters for perspective cameras. Orthographic cameras ignore it.

One common usage of layers would be for parallax scrolling using a perspective camera, where each layer is at a different z distance from the camera. (See the Acorn Antics demo game for an example of this.) But layers can be used for other purposes as well, such as a top-down RPG where you could have a layer used for the roofs of buildings, where that layer is deactivated when the player enters the buildings.

 If you use more than one layer, you must manually add additional sorting layers in Unity using the Tags and Layers manager (Edit -> Project Settings -> Tags and Layers). There's no way to add layers automatically through scripting, so this is something you'll have to take care of yourself. It doesn't matter what the layers are actually called; SpriteTile only cares about the number. So if you have three layers, you should have at least three entries in the Sorting Layers list, as shown on the right. Attempting to load a level that contains more layers than you have sorting layers in your project will result in an error.



Each cell in the level can be considered a collider cell, or not. The **collider overlay** will show which cells are collider cells in your level. You can read this collider information for each cell through scripting, by using the `GetCollider` function.

For example, say you're making a top-down RPG, and you want wall tiles to be considered as blocking cells, so players can't go through them. When the player moves, you can read collider information from the appropriate cell in your level, and prevent movement if the player tries to enter a wall tile. It would be convenient if all wall tiles were marked as collider cells automatically, so in the defaults for the wall tile, toggle the Collider checkbox on. Then all wall tiles that you draw in your level will automatically be marked as collider cells. You can remove collider cells from your level later if desired—for example, you want some walls to be illusions that the player can pass through.

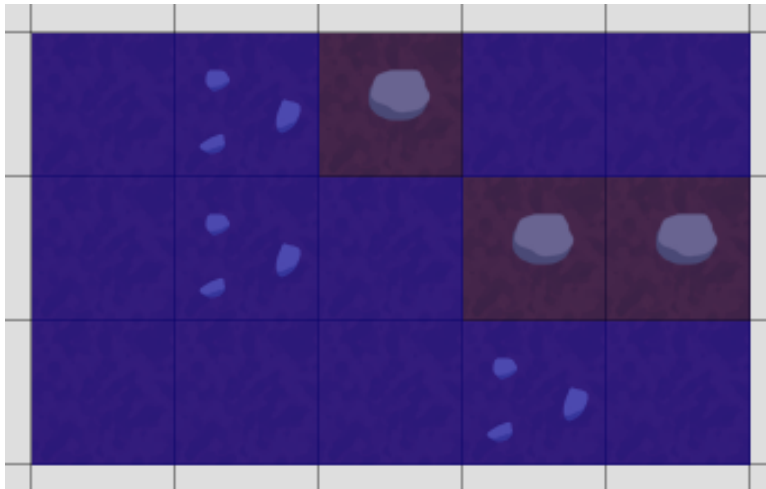
It's also possible to use physics colliders. If you're making a game where the player interacts with tiles using physics, then `SpriteTile` will use the polygon collision shapes that Unity generates automatically. If you have various tiles for the ground, for example, then sloped tiles will automatically have colliders in the shape of the slope.



In order for physics colliders to work, the “Use physics collider” checkbox must be on for the appropriate tile! If your character falls through tiles and you weren't expecting this, make sure this checkbox is active for all tiles that you want to generate polygon colliders.

If the “Use physics collider” checkbox is off, then no polygon colliders will be generated for that tile, even if cells are marked as collider cells. It doesn't necessarily hurt too much for polygon colliders to be generated if you don't need them, but it's more efficient if they aren't. So if your game doesn't use physics, make sure the “Use physics collider” checkbox is off for all tiles.

If you're using physics, it can be slightly confusing as to which tiles will generate polygon colliders or not, since a cell marked as a collider cell doesn't necessarily mean it will use a polygon collider. To help with this, `SpriteTile` has two different tints for the collider overlay:



You can see here that there are three tiles being used, specifically the Dirt1, Dirt2, and Dirt3 tiles from the Acorn Antics demo game. The collider overlay has been turned on. The Dirt1 and Dirt3 tiles have the “Use physics collider” checkbox **on**. The Dirt2 tile has that checkbox **off**. All of the cells in this group have been marked as collider cells in the level. The Dirt1 and Dirt3 tiles have a dark blue tint, and the Dirt2 tiles have a lighter blue tint. This shows that the Dirt2 cells are collider cells and will return true if the `GetCollider` function is used, but will not generate polygon colliders, whereas the Dirt1 and Dirt3 tiles will generate polygon colliders as well as returning true with `GetCollider`.

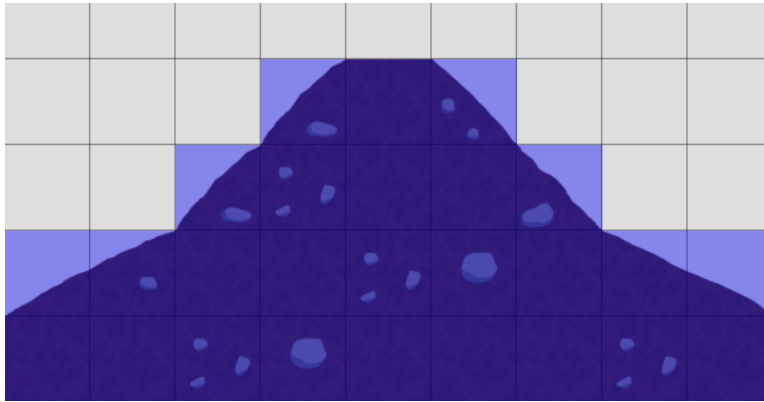
So, in short: blue tint = collider cell. Dark tint = polygon collider, light tint = no polygon collider. It's conceivable that you might use polygon colliders *and* `Tile.GetCollider` in the same game, so the different tints help to make it more obvious what's going on. Likewise, if you have non-functioning polygon colliders and aren't sure why, the light tint will instantly make it clear which tiles need to have the “Use physics collider” checkbox activated.

Note that since polygon colliders are purely 2D and are not affected by depth, any active colliders in multiple layers all behave as if they're on the same layer.

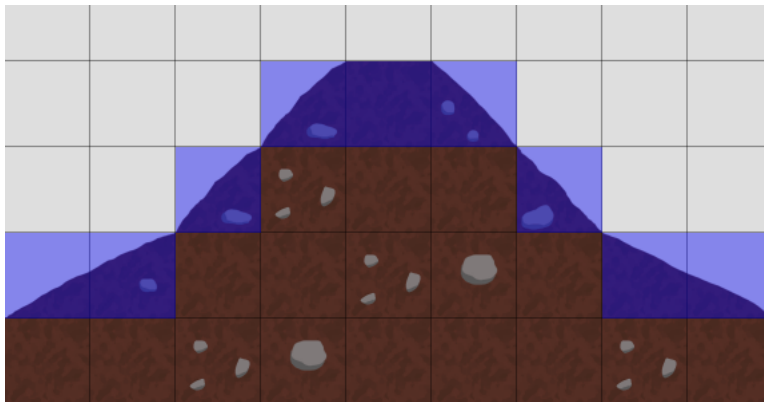


If you're using polygon colliders, you should keep in mind efficiency of usage, or more specifically, only mark cells in your level as collider cells where actually necessary. If there's no chance that a player will touch a particular cell, then don't mark it as a collider cell. Again, it doesn't necessarily hurt to have physics colliders generated unnecessarily, but levels will load a bit faster and the physics engine will probably work a bit faster if you keep collider cells to a minimum.

For example, here we have a hill in an Acorn Antics level:



The player has no way of touching the tiles inside the hill, so the active collider cells in the middle are wasted. Instead, only the outer cells should be active:



Even though we removed most of the collider cells here, it's generally most convenient if you just go ahead and use collider cells without concern for efficiency while you're testing your level. When the level is finalized, then you can go back and remove unneeded collider cells as a final step.

Note that this concern with efficiency applies to physics colliders **only**. If you're just marking the cells as collider cells to check with `Tile.GetCollider`, and the "Use physics collider" checkbox for all tiles is off, then it doesn't matter at all how many collider cells you have.

## Collider blocks

By default, SpriteTile groups polygon colliders into blocks made of 5x5 tiles. This is to prevent making a polygon collider object for every active collider cell, which wouldn't be very efficient. It can still result in a lot of objects for huge levels, so in that case you may want to increase it to a larger size, like 10x10 or more. You can use the [SetColliderBlockSize](#) function to do this. A potential downside is that setting tiles in code rebuilds the appropriate collider block, so larger blocks are slower to rebuild. So if you're setting lots of tiles in code very frequently, you might want to use a smaller block size. Also, large blocks can be less efficient for the physics engine to process, so it's a bit of a balancing act.

This section explains the fundamentals of how to use your SpriteTile levels once you've created them, and how to modify levels you've loaded or even create new levels entirely with code, as well as other features such as animation.

The basic idea behind SpriteTile is that you set up your camera, load or create levels, then move the camera as desired. SpriteTile then updates the tiles as necessary depending on the camera view.

To conveniently use SpriteTile functions, you should import the SpriteTile namespace in your scripts by using the “import” or “using” commands depending on the language you're using.

```
import SpriteTile; // Unityscript
using SpriteTile;  // C#
```

Some examples of basic SpriteTile coding are on the following pages. You can also look at the various scripts included in the Demos package to see more complex SpriteTile coding in action. Use the [Function Reference](#) section for specific details of SpriteTile function usage. All functions are in the Tile class. Unless otherwise noted, the code examples in the Function Reference work for both Unityscript and C#.

Note that any of the SpriteTile functions can be used to modify a loaded level at runtime if desired. The level will only be modified in memory, not permanently. If you want to save a level modified with code, you can use [Tile.GetLevelBytes](#) along with appropriate file IO code, as shown in the function reference. You can also use GetLevelBytes to permanently save levels created entirely with code.

## setting the camera

SpriteTile needs to know what camera you're using so it can draw tiles properly. This is done by using SetCamera, which should be called first, before using any other tile functions. Once you've called SetCamera, you can then move the camera around using Transform.Translate or Transform.position, or by using animation. The only thing to be aware of is that camera rotation is currently not supported, so the camera is forced to always have no rotation. Without SetCamera, SpriteTile won't be able to function.

You can choose to call SetCamera with no arguments, in which case any cameras tagged MainCamera are used. If you want to use a specific camera, you can pass in that camera instead. This code, for example, will use the camera component of whatever object the script is attached to:

```
Tile.SetCamera (camera);
```

This code will use a camera specified by a public variable:

```
var tileCam : Camera; // Unityscript

function Awake () {
    Tile.SetCamera (tileCam);
}
```

```
public Camera tileCam; // C#

void Awake () {
    Tile.SetCamera (tileCam);
}
```

Normally you should only call SetCamera once, but if you've loaded a new scene in Unity and the camera that you used doesn't exist anymore, then you should call SetCamera again in the new scene.

As mentioned above, once you've called SetCamera, you can move the camera around normally using its transform component. This includes the Z axis, so if you're using a perspective camera, the tiles will update properly as you zoom in and out. If you're using an orthographic camera, the same thing applies to the orthographic size. Note, however, that zooming out a long way can potentially create a huge amount of GameObjects, which in extreme cases can freeze/crash Unity. So take care not to zoom out too far. Also, changing the screen resolution at runtime, or changing the orientation (for mobile devices) is supported and will work seamlessly.

If you have multiple cameras tagged MainCamera, all of them will be used when calling SetCamera. You can also use a camera array:

```
var cameras : Camera[]; // Unityscript
public Camera[] cameras; // C#
...
Tile.SetCamera (cameras);
```

Each camera will render its own set of tiles, so you can use this for split-screen games or other situations where one camera isn't enough. You can use the viewport rects of the cameras to set up any arbitrary arrangement.

## the Int2 struct

Since SpriteTile heavily uses tilemaps (of course), it's useful to refer to x/y map coordinates in code. Unity already has a Vector2 struct, but that uses floats. It's better to use ints in this case since there's no such thing as a fraction of a map cell, so SpriteTile includes an Int2 struct. It works pretty much the same as Vector2 except it uses ints. It doesn't include all of the Vector2 functions, but you can do most of the standard operations such as addition, subtraction, ToString(), and so on. It includes Int2.zero and Int2.one, which equal Int2(0, 0) and Int2(1, 1) respectively.

```
var coord : Int2 = Int2(5, 7); // Unityscript
Int2 coord = new Int2(5, 7);   // C#

var anotherCoord = Int2.one;
coord.x = 7;
coord = anotherCoord * 2;
if (coord != anotherCoord) {
    Debug.Log (coord);
}
```

You can also create an Int2 by passing in a Vector2. This can be useful since Unity doesn't show custom structs in the inspector, so you can make a public Vector2 and use that to create an Int2 later. You can also use ToVector2() to create a Vector2 from an Int2.

```
// Unityscript
public var coord : Vector2;
private var _coord : Int2;

function Start () {
    _coord = new Int2(coord);
    var v2 : Vector2 = _coord.ToVector2();
}
```

## the TileInfo struct

Some functions return TileInfo. This is a struct which contains set and tile information, as ints.

```
var thisTile : TileInfo = Tile.GetTile (transform.position); // Unityscript
TileInfo thisTile = Tile.GetTile (transform.position); // C#
Debug.Log ("The set is: " + thisTile.set + " and the tile is: " + thisTile.tile);
```

It's actually much like an Int2, but in this case using "set" and "tile" for the properties makes code more understandable and explanatory than "x" and "y" would. You can use TileInfo variables with SetTile and SetTileBlock rather than passing in the set and tile separately as ints, though both methods work:

```
var thisTile = new TileInfo(3, 1);
Tile.SetTile (new Int2(10, 25), thisTile);
Tile.SetTile (new Int2(10, 25), 3, 1);
```

## loading a level

1. Create a level using the TileEditor.
2. Call `Tile.SetCamera` in your script.
3. Call `Tile.LoadLevel` with the desired level. Aside from loading an internal `TextAsset` file, the level can be loaded from an external file (even from the web). This is shown in the [LoadLevel](#) function reference.

Example:

```
// Unityscript
import SpriteTile;

var myLevel : TextAsset;

function Awake () {
    Tile.SetCamera();
    Tile.LoadLevel (myLevel);
}
```

```
// C#
using UnityEngine;
using SpriteTile;

public class MyScript : MonoBehaviour {
    public TextAsset myLevel;

    void Awake () {
        Tile.SetCamera();
        Tile.LoadLevel (myLevel);
    }
}
```

## creating a level in code

1. Call `Tile.SetCamera`.
2. Call `Tile.NewLevel` with the desired level parameters.
3. Use functions like `SetTile`, `SetCollider`, etc. The functions can be used anywhere in your code, as long as a level has been loaded with `LoadLevel` or created with `NewLevel`.

Example:

```
// Unityscript
import SpriteTile;

function Awake () {
    Tile.SetCamera();
    Tile.NewLevel (Int2(50, 30), 0, 1.28, 0.0, LayerLock.None);
    Tile.SetTileBlock (Int2.zero, Int2(15, 15), 0, 10);
}
```

```
// C#
using UnityEngine;
using SpriteTile;

public class MyScript : MonoBehaviour {
    void Awake () {
        Tile.SetCamera();
        Tile.NewLevel (new Int2(50, 30), 0, 1.28f, 0.0f, LayerLock.None);
        Tile.SetTileBlock (Int2.zero, new Int2(15, 15), 0, 10);
    }
}
```

## tile animation

1. Create the tiles you want to be animated. It's easiest if they are numbered sequentially in the TileEditor.
2. (Optional) If the tiles aren't numbered sequentially, create a `TileInfo` array containing the tiles in the animation sequence.
3. Call `Tile.AnimateTile` or `Tile.AnimateTileRange` with the tile or tiles to be animated, a range of tiles or a `TileInfo` array, the framerate, and optionally an `AnimType`.

Example (all code on this page is assumed to be inside a function, after `SetCamera` has been called and a level has been loaded or created):

```
Tile.AnimateTile (new TileInfo(1, 10), 4, 10.0f);
```

This example animates tile #10 in set 1. The range is 4, which means that the animation sequence includes tile #10 and the next 3 tiles in the set. In other words, tiles 10, 11, 12, and 13 in set 1. The tile is animated at 10 frames per second. Once `AnimateTile` is called, all instances of `TileInfo(1, 10)` in the level are animated. Note that this animation **is cosmetic only**, and does not actually modify the tiles in the level. So if you use `Tile.GetTile` on an animated cell that uses `TileInfo(1, 10)`, it will always return `TileInfo(1, 10)` regardless.

By default, animations use `AnimType.Loop`. This means that tiles are animated sequentially, then start over again after reaching the last frame. Other options are `AnimType.Reverse` and `AnimType.PingPong`. `Reverse`, as you would expect, plays the animation frames in reverse order. `PingPong` cycles through the frames, reaches the end, goes back in reverse order to the first frame, then repeats the process. For example:

```
Tile.AnimateTile (new TileInfo(1, 10), 4, 10.0f, AnimType.Reverse);
```

It may be that the tiles you want to animate aren't numbered sequentially, or they are in different sets. In this case you can create a `TileInfo` array containing all the frames in the animation sequence, and use this array instead of a range.

```
var frames = [TileInfo(1, 4), TileInfo(1, 7), TileInfo(1, 9)]; // Unityscript  
TileInfo[] frames = {new TileInfo(1, 4), new TileInfo(1, 7), new TileInfo(1, 9)}; // C#  
Tile.AnimateTile (new TileInfo(1, 10), frames, 10.0f); // Unityscript and C#
```

In some cases you may be using all the frames in your animation sequence in a level, and want to animate them all. It would be possible to create `TileInfo` arrays for all the tiles to be animated and call `AnimateTile` for each tile, but it's much simpler to use `AnimateTileRange`. Other than animating all the tiles in the range, it works the same as `AnimateTile`. For example, let's say we want to animate tiles #10-13 in set 1, and each of those tiles should cycle through that range independently:

```
Tile.AnimateTileRange (new TileInfo(1, 10), 4, 10.0f);
```

If you want a tile to stop animating, use `StopAnimatingTile` with the appropriate `TileInfo`, and the animation will be stopped immediately:

```
Tile.StopAnimatingTile (new TileInfo(1, 10));
```

You can use `StopAnimatingTileRange` to stop a range of tile animations. In addition to stopping tiles animated with `AnimateTileRange`, note that any tiles in the range that are not actually animating are ignored, so you could also use this to stop all animations in a set at once. For example, if there were 50 tiles in set 1, this would stop all animations that had been started from all `AnimateTile` calls:

```
Tile.StopAnimatingTileRange (new TileInfo(1, 0), 50);
```



## AddLayer

```
static function AddLayer (mapSize : Int2,  
                        addBorder : int,  
                        tileSize : float (or Vector2),  
                        zPosition : float,  
                        layerLock : LayerLock) : void
```

Adds a new layer to the level. Since the layer is last in the list, it will be rendered on top of preceding layers. The project must have enough layers in the Tags and Layers manager to accommodate the new layer (see [“How Layers Work”](#)).

mapSize: the dimensions, in tiles, of the level. Must be at least 1x1.

addBorder: the number of additional rows/columns that are added around the screen border, in case oversized tiles are used. See [Add Border](#) in the Tile Editor: Level section. Must be 0 or greater.

tileSize: the dimension, in units, of each cell in the level. Must be at least .001. For a non-square tile grid, use a Vector2 instead of a float.

zPosition: the distance from the origin along the Z axis. Only has an effect with perspective cameras. Must be at least 0.0.

layerLock: whether the layer should be prevented from moving on the X or Y axes. Uses the LayerLock enum:

LayerLock.None: the layer is not locked.

LayerLock.X: the layer is locked on the X axis, but can move on the Y axis.

LayerLock.Y: the layer is locked on the Y axis, but can move on the X axis.

LayerLock.XandY: the layer is locked on both the X axis and the Y axis.

```
// Creates a 20x20 layer with no added border, a tile size of 1.0,  
// located at z = 0.0, with no layer lock  
Tile.AddLayer (new Int2(20, 20), 0, 1.0f, 0.0f, LayerLock.None);  
// Creates a 50x25 layer with a 1 tile added border, a non-square tile size  
// of (1.0, 2.0), located at z = 5.0, locked on the Y axis  
Tile.AddLayer (new Int2(50, 25), 1, new Vector2(1.0f, 2.0f), 5.0f, LayerLock.Y);
```

```
static function AddLayer (levelData : LevelData) : void
```

As above, but the layer parameters are contained in a variable with a LevelData type. The LevelData properties are mapSize (Int2), addBorder (int), tileSize (Vector2), zPosition (float), and layerLock (LayerLock). If you want a square grid, just use the same value for the X and Y tileSize.

```
// Creates a 50x25 layer with a 1 tile added border, a non-square tile size  
// of (1.0, 2.0), located at z = 5.0, locked on the Y axis  
var myLevelData = new LevelData(new Int2(50, 25), 1, new Vector2(1.0f, 2.0f),  
                                5.0f, LayerLock.Y);  
Tile.AddLayer (myLevelData);
```

## AnimateTile

```
static function AnimateTile (tileInfo : TileInfo,  
                             range : int;  
                             frameRate : float;  
                             animType : AnimType = AnimType.Loop) : void
```

Animates all instances of tileInfo in all layers in the current level.

The range is the number of tiles, including the tile specified by tileInfo, that will be included in the animation sequence. The range added to tileInfo must not exceed the number of tiles in the current set, and must be at least 2.

The frameRate is how fast the animation sequence will be played back, in terms of frames per second. It must be at least 0.0 (although using 0.0 will of course not result in any actual animation).

The animType is AnimType.Loop by default, which plays the animation from the first frame to the last, then starts over. AnimType.Reverse will play the animation from the last frame to the first and then start over, and AnimType.PingPong will play the animation from the first frame to the last, back to the first, and then repeat the cycle.

Animation can be stopped with [StopAnimatingTile](#).

```
// Animates tile #15 in set 2, using tiles 15-20, at 5fps, with AnimType.Loop  
Tile.AnimateTile (new TileInfo(2, 15), 6, 5.0f);  
// Animates tile #10 in set 1, using tiles 10-19, at 15fps, in reverse  
Tile.AnimateTile (new TileInfo(1, 10), 10, 15.0f, AnimType.Reverse);
```

```
static function AnimateTile (tileInfo : TileInfo,  
                             tileInfoArray : TileInfo[];  
                             frameRate : float;  
                             animType : AnimType = AnimType.Loop) : void
```

As above, but instead of specifying a range, the tile animation sequence is supplied as an array of TileInfo. The tiles can be from any set, in any order.

```
// Animates tile #15 in set 2, using the supplied TileInfo array, at 5fps  
// Unityscript  
var tiles = [TileInfo(1, 4), TileInfo(1, 5), TileInfo(2, 8)];  
Tile.AnimateTile (TileInfo(2, 15), tiles, 5.0);  
// C#  
TileInfo[] tiles = {new TileInfo(1, 4), new TileInfo(1, 5), new TileInfo(2, 8)};  
Tile.AnimateTile (new TileInfo(2, 15), tiles, 5.0f);
```

## AnimateTileRange

```
static function AnimateTileRange (tileInfo : TileInfo,  
                                range : int;  
                                frameRate : float;  
                                animType : AnimType = AnimType.Loop) : void
```

Similar to [AnimateTile](#), but all tiles in the range are animated independently. All tiles use the range as specified by `tileInfo + range`, looping as necessary. For example, if `tileInfo` is `TileInfo(1, 10)` and the range is 3, then `TileInfo(1, 10)` will animate using tiles 10, 11, and 12. `TileInfo(1, 11)` will animate using tiles 11, 12, and 10. `TileInfo(1, 12)` will animate using tiles 12, 10, and 11.

The range is the number of tiles, including the tile specified by `tileInfo`, that will be included in the animation sequence. The range added to `tileInfo` must not exceed the number of tiles in the current set, and must be at least 2.

The `frameRate` is how fast the animation sequence will be played back, in terms of frames per second. It must be at least 0.0 (although using 0.0 will of course not result in any actual animation).

The `animType` is `AnimType.Loop` by default, which plays the animation from the first frame to the last, then starts over. `AnimType.Reverse` will play the animation from the last frame to the first and then start over, and `AnimType.PingPong` will play the animation from the first frame to the last, back to the first, and then repeat the cycle.

Animation can be stopped with [StopAnimatingTileRange](#).

```
// Animates tiles #5-9 in set 2, at 8fps, with AnimType.Loop  
Tile.AnimateTileRange (new TileInfo(2, 5), 5, 8.0f);  
// Animates tiles #10-19 in set 1, at 15fps, in reverse  
Tile.AnimateTile (new TileInfo(1, 10), 10, 15.0f, AnimType.Reverse);
```

```
static function AnimateTileRange (tileInfo : TileInfo,  
                                range : int;  
                                tileInfoArray : TileInfo[];  
                                frameRate : float;  
                                animType : AnimType = AnimType.Loop) : void
```

As above, but all tiles in the range will use a tile animation sequence that's supplied as an array of `TileInfo`. The `TileInfo` array doesn't need to be the same size as the range. The tiles can be from any set, in any order.

```
// Animates tiles #10-15 in set 2, using the supplied TileInfo array, at 5fps  
// Unityscript  
var tiles = [TileInfo(1, 4), TileInfo(1, 5), TileInfo(2, 8)];  
Tile.AnimateTile (TileInfo(2, 10), 6, tiles, 5.0f);  
// C#  
TileInfo[] tiles = {new TileInfo(1, 4), new TileInfo(1, 5), new TileInfo(2, 8)};  
Tile.AnimateTile (new TileInfo(2, 10), 6, tiles, 5.0f);
```

## CopyGroupToPosition

```
static function CopyGroupToPosition (position : Int2,  
                                     offset : Vector2 = Vector2.zero;  
                                     layer : int = 0;  
                                     groupSet : int,  
                                     groupNumber : int) : void
```

Copies a specified group to a specified position, with an optional offset. Groups must be loaded using [LoadGroups](#) first.

The position is a location within a layer. It must be within bounds of the layer. However, if the group would extend beyond the layer bounds at that position, it will be clipped appropriately without errors.

The optional offset can be used to set the “pivot point” or center of the group. That is, the group will be placed at the specified position, and then moved by the offset. The offset must be zero or negative; positive offsets will generate an error. For example, a 5X5 cell group with an offset of (-2, -2) will be placed where the position is the middle tile of the group.

If the layer is omitted, then CopyGroupToPosition will work on layer 0 by default.

The groupSet is the set number, as shown in the TileEditor. The groupNumber is the group number within that set, as shown in the TileEditor.

```
// Copies group 5 in set 4 to (10, 15) on layer 1, with an offset of (-2, -2)  
Tile.CopyGroupToPosition (new Int2(10, 15), new Int2(-2, -2), 1, 4, 5);  
// Copies group 2 in set 1 to (5, 5) on layer 0, with no offset  
Tile.CopyGroupToPosition (new Int2(5, 5), 1, 2);
```

## DeleteTile

```
static function DeleteTile (position : Int2,  
                            layer : int = 0,  
                            removeCollider : boolean = false) : void
```

Removes the tile in the cell at the coordinates specified by the position. Other properties of the cell are not affected. The position can't be lower than (0, 0), which is the lower-left corner of the map, and must be within bounds of the map.

If the layer is omitted, then DeleteTile will work on layer 0 by default.

If removeCollider is omitted, then the collider state of the cell is left alone. If it's set to true, then the collider is removed.

```
// Removes the tile in the cell at coords (10, 25) in layer 0  
Tile.DeleteTile (new Int2(10, 25));  
// Same thing, but uses layer 1  
Tile.DeleteTile (new Int2(10, 25), 1);  
// Removes the tile and collider in the cell at coords (10, 25) in layer 0  
Tile.DeleteTile (new Int2(10, 25), true);  
// Same thing, but uses layer 1  
Tile.DeleteTile (new Int2(10, 25), 1, true);
```

## DeleteTileBlock

```
static function DeleteTileBlock (position1 : Int2,  
                                position2 : Int2,  
                                layer : int = 0,  
                                removeCollider : boolean = false) : void
```

Like [DeleteTile](#), except it works on a block of cells—defined from position1 at one corner of the block up to and including position2 of the opposite corner—of the map in layer 0. Both positions are clamped to the size of the map if necessary, and can be in any order; that is, position1 doesn't have to be less than position2.

If the layer is omitted, then DeleteTileBlock will work on layer 0 by default.

If removeCollider is omitted, then the collider state of all the cells in the block is left alone. If it's set to true, then the collider state of the block is removed.

```
// Removes the tiles in a block of cells in layer 0, defined by (10, 18)  
// at one corner and (30, 25) at the other  
Tile.DeleteTileBlock (new Int2(10, 18), new Int2(30, 25));  
// Same thing, but uses layer 1  
Tile.DeleteTileBlock (new Int2(10, 18), new Int2(30, 25), 1);  
// Removes the tiles and colliders in a block of cells in layer 0  
Tile.DeleteTileBlock (new Int2(10, 18), new Int2(30, 25), true);  
// Same thing, but uses layer 1  
Tile.DeleteTileBlock (new Int2(10, 18), new Int2(30, 25), 1, true);
```

## EraseLevel

```
static function EraseLevel () : void
```

Deletes all tiles on all layers, removes all colliders and triggers, and sets all rotation and order-in-layer values to 0. A level must be loaded or created before this function can be used. The level itself is not deleted, and keeps the same number of layers and so on, but all tiles are empty.

## GetCollider

```
static function GetCollider (position : Int2,  
                             layer : int = 0) : boolean
```

Returns the collider state of the specified map cell.

If the layer is omitted, then GetCollider will work on layer 0 by default.

```
// Checks if the cell at (10, 20) in layer 0 is a collider  
var collider = Tile.GetCollider (new Int2(10, 20));  
if (collider) {  
    Debug.Log ("None may pass");  
}  
// Same thing, but uses layer 1  
collider = Tile.GetCollider (new Int2(10, 20), 1);
```

```
static function GetCollider (position : Vector2,  
                             layer : int = 0) : boolean
```

As above, but the position is in world space rather than map coordinates. Since Unity automatically converts Vector3 to Vector2, the position can be a Vector3 as well, such as a transform's position. In this case the Z is ignored.

```
// Checks if the cell at the current world position of this transform  
// is a collider, using layer 0  
var collider = Tile.GetCollider (transform.position);  
if (collider) {  
    Debug.Log ("None may pass");  
}  
// Same thing, but uses layer 1  
collider = Tile.GetCollider (transform.position, 1);
```

## GetLevelBytes

```
static function GetLevelBytes () : byte[]
```

Returns an array of bytes that contains the current level. This array can then be saved or uploaded using various functions in Unity.

```
var levelBytes = Tile.GetLevelBytes();  
System.IO.File.WriteAllBytes (Application.dataPath + "/MyFile.bytes", levelBytes);
```

## GetMapBlock

```
static function GetMapBlock (position1 : Int2,  
                             position2 : Int2,  
                             layer : int = 0) : MapData
```

Returns a MapData object that contains all the map data in a block defined by position1 at one corner up to and including position2 at the opposite corner. Both positions must use valid coordinates inside the map, but can be in any order; that is, position1 doesn't have to be less than position2. The map data includes tile, rotation, order in layer, collider, trigger, and material (if any per-tile materials have been set up). GetMapBlock would typically be used in combination with [SetMapBlock](#).

If the layer is omitted, then GetMapBlock will work on layer 0 by default.

```
// Copies a block from (5, 10) to (15, 20) in layer 0,  
// and pastes it to location (50, 60) in layer 1  
var mapData = Tile.GetMapBlock (new Int2(5, 10), new Int2(15, 20));  
Tile.SetMapBlock (new Int2(50, 60), 1, mapData);
```

## GetMapPosition

```
static function GetMapPosition (position : Vector2,  
                                 layer : int = 0) : Int2
```

Returns the map coordinates from the supplied world coordinates. A Vector3 can be used instead of Vector2, in which case the Z is ignored.

If the layer is omitted, then GetMapPosition will work on layer 0 by default.

```
// Prints "Lower left corner" if the map position is Int2(0, 0)  
var pos = Tile.GetMapPosition (transform.position);  
if (pos == Int2.zero) {  
    Debug.Log ("Lower left corner");  
}  
// Same thing, but uses layer 1  
pos = Tile.GetMapPosition (transform.position, 1);
```



## GetMapSize

```
static function GetMapSize (layer : int = 0) : Int2
```

Returns the map size as an Int2, where x is the number of cells on the X axis and y is the number of cells on the Y axis.

If the layer is omitted, then GetMapSize will work on layer 0 by default.

```
// Checks if layer 0 is over 500 cells wide
var mapSize = Tile.GetMapSize();
if (mapSize.x > 500) {
    Debug.Log ("That's a wide layer!");
}
// Same thing, but uses layer 1
mapSize = Tile.GetMapSize (1);
```

## GetNumberOfLayers

```
static function GetNumberOfLayers () : int
```

Returns the number of layers in the current level.

```
// Gets the number of layers in the level
var layerCount = Tile.GetNumberOfLayers();
Debug.Log ("This level has " + layerCount + " layers");
```

## GetOrder

```
static function GetOrder (position : Int2,  
                           layer : int = 0) : int
```

Returns the order-in-layer number for the specified map cell. The order number will be an int between -32768 and 32767.

If the layer is omitted, then GetOrder will work on layer 0 by default.

```
// Increases the order-in-layer for the cell at (10, 20) in layer 0 by 1  
var order = Tile.GetOrder (new Int2(10, 20));  
order++;  
Tile.SetOrder (new Int2(10, 20), order);  
// Same thing, but uses layer 1  
order = Tile.GetOrder (new Int2(10, 20), 1);
```

```
static function GetOrder (position : Vector2) : int
```

As above, but the position is in world space rather than map coordinates. A Vector3 can be used instead of Vector2, in which case the Z is ignored.

```
// Increases the order-in-layer at the current world position of this transform  
// by 1, using layer 0  
var order = Tile.GetOrder (transform.position);  
order++;  
Tile.SetOrder (transform.position, order);  
// Same thing, but uses layer 1  
order = Tile.GetOrder (transform.position, 1);
```

## GetRotation

```
static function GetRotation (position : Int2,  
                             layer : int = 0) : float
```

Returns the rotation for the specified map cell.

If the layer is omitted, then GetRotation will work on layer 0 by default.

```
// Rotates the cell at (10, 20) in layer 0 by 90 degrees  
var rotation = Tile.GetRotation (new Int2(10, 20));  
rotation += 90.0f;  
Tile.SetRotation (new Int2(10, 20), rotation);  
// Same thing, but uses layer 1  
rotation = Tile.GetRotation (new Int2(10, 20), 1);
```

```
static function GetRotation (position : Vector2,  
                             layer : int = 0) : float
```

As above, but the position is in world space rather than map coordinates. A Vector3 can be used instead of Vector2, in which case the Z is ignored.

```
// Rotates the cell at the current world position of this transform  
// by 90 degrees, using layer 0  
var rotation = Tile.GetRotation (transform.position);  
rotation += 90.0f;  
Tile.SetRotation (transform.position, rotation);  
// Same thing, but uses layer 1  
rotation = Tile.GetRotation (transform.position, 1);
```

## GetTile

```
static function GetTile (position : Int2,  
                        layer : int = 0) : TileInfo
```

Returns the set and tile numbers for the specified map cell in a TileInfo struct.

If the layer is omitted, then GetTile will work on layer 0 by default.

```
// Sees if the cell at (10, 20) in layer 0 contains tile 3 in set 0  
var thisTile = Tile.GetTile (new Int2(10, 20));  
if (thisTile.set == 0 && thisTile.tile == 3) {  
    Debug.Log ("Found tile 3");  
}  
// Same thing, but uses layer 1  
thisTile = Tile.GetTile (new Int2(10, 20), 1);
```

```
static function GetTile (position : Vector2,  
                        layer : int = 0) : TileInfo
```

As above, but the position is in world space rather than map coordinates. A Vector3 can be used instead of Vector2, in which case the Z is ignored.

```
// Sees if the cell in layer 0 at the current world position of this transform  
// contains tile 3 in set 0  
var thisTile = Tile.GetTile (transform.position);  
if (thisTile.set == 0 && thisTile.tile == 3) {  
    Debug.Log ("Found tile 3");  
}  
// Same thing, but uses layer 1  
thisTile = Tile.GetTile (transform.position, 1);
```

## GetTileSize

```
static function GetTileSize (layer : int = 0) : Vector2
```

Returns the size of the tiles for a specified layer, in world units. If the layer is omitted, then GetTileSize will work on layer 0 by default. If a square tile grid has been used, the X and Y will be the same.

```
// Prints the equivalent in world units of 50 cells along the x axis in layer 0
var tileSize = Tile.GetTileSize();
Debug.Log ("50 cells is " + tileSize.x*50 + " units");
// Same thing, but uses layer 1
tileSize = Tile.GetTileSize (1);
```

## GetTrigger

```
static function GetTrigger (position : Int2,
                             layer : int = 0) : int
```

Returns the trigger of the specified map cell. The trigger is an int between 0 and 255.

If the layer is omitted, then GetTrigger will work on layer 0 by default.

```
// If the cell at (10, 20) in layer 0 has a trigger of 1, set the cell to tile 10
var trigger = Tile.GetTrigger (new Int2(10, 20));
if (trigger == 1) {
    Tile.SetTile (new Int2(10, 20), 0, 10);
}
// Same thing, but uses layer 1
trigger = Tile.GetTrigger (new Int2(10, 20), 1);
```

```
static function GetTrigger (position : Vector2,
                             layer : int = 0) : int
```

As above, but the position is in world space rather than map coordinates. A Vector3 can be used instead of Vector2, in which case the Z is ignored.

```
// If the cell at the current world position of this transform in layer 0
// has a trigger of 1, set the cell to set 0, tile 10
var trigger = Tile.GetTrigger (transform.position);
if (trigger == 1) {
    Tile.SetTile (transform.position, 0, 10);
}
// Same thing, but uses layer 1
trigger = Tile.GetTrigger (transform.position, 1);
```

## GetWorldPosition

```
static function GetWorldPosition (position : Int2,  
                                   layer : int = 0) : Vector3
```

Returns the world position in units that corresponds to the specified cell.

If the layer is omitted, then GetWorldPosition will work on layer 0 by default.

```
// Moves the transform to the world position for (10, 20) in layer 0  
var worldPos = Tile.GetWorldPosition (new Int2(10, 20));  
transform.position = worldPos;  
// Same thing, but uses layer 1  
var worldPos = Tile.GetWorldPosition (new Int2(10, 20), 1);
```

## GetZPosition

```
static function GetZPosition (layer : int = 0) : float
```

Returns the distance from the origin along the Z axis.

If the layer is omitted, then GetZPosition will work on layer 0 by default.

```
// Gets the z position of layer 0  
var zPos = Tile.GetZPosition();  
transform.position = new Vector3(5, 10, zPos);  
// Same thing, but uses layer 1  
var zPos = Tile.GetZPosition (1);
```

## LoadGroups

```
static function LoadGroups (level : TextAsset) : void
```

Loads a SpriteTile group file from a TextAsset file. If groups have been loaded previously, they will be replaced by the new groups. Once loaded, groups can then be used with the [CopyGroupToPosition](#) function.

```
var myGroups : TextAsset; // Unityscript
```

```
function Start () {  
    Tile.LoadGroups (myLevel);  
}
```

```
public TextAsset myGroups; // C#  
  
void Start () {  
    Tile.LoadGroups (myGroups);  
}
```

```
static function LoadGroups (bytes : byte[]) : void
```

Loads a SpriteTile group file from a byte array. The byte array can be obtained from external files, downloaded from the WWW, or some other method.

```
var pathToGroupFile : String; // Unityscript
```

```
function Start () {  
    var bytes = System.IO.File.ReadAllBytes (pathToLevelFile);  
    Tile.LoadGroups (bytes);  
}
```

```
public string pathToGroupsFile; //C#  
  
void Start () {  
    var bytes = System.IO.File.ReadAllBytes (pathToLevelFile);  
    Tile.LoadGroups (bytes);  
}
```



## LoadLevel

```
static function LoadLevel (level : TextAsset) : void
```

Loads a SpriteTile level from a TextAsset file. If a level already exists, it will be replaced by the new level. Loading a level will reset the position of any layers, so if [SetLayerPosition](#) had been used, it will need to be called again. Also, any materials set up with [SetTileMaterial](#) will need to be set up again.

```
var myLevel : TextAsset; // Unityscript
```

```
function Awake () {  
    Tile.SetCamera();  
    Tile.LoadLevel (myLevel);  
}
```

```
public TextAsset myLevel; // C#  
  
void Awake () {  
    Tile.SetCamera();  
    Tile.LoadLevel (myLevel);  
}
```

```
static function LoadLevel (bytes : byte[]) : void
```

Loads a SpriteTile level from a byte array. The byte array can be obtained from external files, downloaded from the WWW, or some other method.

```
var pathToLevelFile : String; // Unityscript
```

```
function Awake () {  
    Tile.SetCamera();  
    var bytes = System.IO.File.ReadAllBytes (pathToLevelFile);  
    Tile.LoadLevel (bytes);  
}
```

```
public string pathToLevelFile; //C#  
  
void Awake () {  
    Tile.SetCamera();  
    var bytes = System.IO.File.ReadAllBytes (pathToLevelFile);  
    Tile.LoadLevel (bytes);  
}
```

## NewLevel

```
static function NewLevel (mapSize : Int2,
                        addBorder : int,
                        tileSize : float (or Vector2),
                        zPosition : float,
                        layerLock : LayerLock) : void
```

Creates a new level with one layer, using the specified parameters. If a level already exists, it's erased. Any layers set with [SetLayerPosition](#) will be reset, and any materials set with [SetTileMaterial](#) will also be reset.

mapSize: the dimensions, in tiles, of the level. Must be at least 1x1.

addBorder: the number of additional rows/columns that are added around the screen border, in case oversized tiles are used. See [Add Border](#) in the Tile Editor: Level section. Must be 0 or greater.

tileSize: the dimension, in units, of each cell in the level. Must be at least .001. For a non-square tile grid, use a Vector2 instead of a float.

zPosition: the distance from the origin along the Z axis. Only has an effect with perspective cameras. Must be at least 0.0.

layerLock: whether the layer should be prevented from moving on the X or Y axes. Uses the LayerLock enum:

LayerLock.None: the layer is not locked.

LayerLock.X: the layer is locked on the X axis, but can move on the Y axis.

LayerLock.Y: the layer is locked on the Y axis, but can move on the X axis.

LayerLock.XandY: the layer is locked on both the X axis and the Y axis.

```
// Makes a new level with one layer of 100x50 tiles, no added border tiles,
// a tile size of 2.0, positioned at 5.0 on the z axis, and locked on the X axis
Tile.NewLevel (new Int2(100, 50), 0, 2.0f, 5.0f, LayerLock.X);
Tile.SetCamera();
```

```
static function NewLevel (levelData : LevelData[]) : void
```

Creates a new level with multiple layers, using an array of the LevelData class. Each entry in the array contains properties for one layer, with layer 0 being the topmost layer. The LevelData properties are mapSize (Int2), addBorder (int), tileSize (Vector2), zPosition (float), and layerLock (LayerLock). If you want a square grid, just use the same value for the X and Y tileSize.

```
// Creates a new level that has two layers
// Layer 0 is 100x100 with an added border of 1, tile size 1.0, at z position 0.0
// Layer 1 is 60x20 with no added border, tile size 2.0 X 4.0, at z position 10.0
var levelData = new LevelData[2];
levelData[0] = new LevelData(new Int2(100, 100), 1, new Vector2(1.0f, 1.0f), 0.0f,
LayerLock.None);
levelData[1] = new LevelData(new Int2(60, 20), 0, new Vector2(2.0f, 4.0f), 10.0f,
LayerLock.Y);
Tile.NewLevel (levelData);
Tile.SetCamera();
```

## ScreenToMapPosition

```
static function ScreenToMapPosition (screenPos : Vector2,  
                                     layer : int = 0,  
                                     out mapPos : Int2  
                                     camNumber : int = 0) : bool
```

Converts screen coordinates (such as supplied by `Input.mousePosition`) to a map position. The `mapPos` variable must be declared before calling the function. The function returns true if the screen position is inside the map, or false if outside. This way, attempting to refer to out-of-bounds coordinates for the map can be avoided.

If the layer is omitted, then `ScreenToMapPosition` will work on layer 0 by default.

If the `camNumber` is omitted, then camera 0 is used by default. If only one camera was used with [SetCamera](#), then this should always be 0. If multiple cameras were used, then the `camNumber` refers to the respective entry in the `Camera[]` array.

```
// Unityscript  
// Prints the tile coordinates that the mouse cursor is over, for layer 0  
var mapPos : Int2;  
if (Tile.ScreenToMapPosition (Input.mousePosition, mapPos)) {  
    Debug.Log ("Mouse is over tile coords " + mapPos);  
}
```

```
// C#  
// Prints the tile coordinates that the mouse cursor is over, for layer 0  
Int2 mapPos;  
if (Tile.ScreenToMapPosition (Input.mousePosition, out mapPos)) {  
    Debug.Log ("Mouse is over tile coords " + mapPos);  
}
```

## SetBorder

```
static function SetBorder (layer : int = 0,  
                           set : int,  
                           tile : int,  
                           setCollider : bool) : void
```

Creates a 1-tile border, using the specified set and tile numbers, around the perimeter of the map. The collider cells are set as well if setCollider is true. This is particularly useful for character controllers that use [GetCollider](#): if there's always a border around the map, then you can avoid having to check for out-of-bounds movement.

If the layer is omitted, then SetBorder will work on layer 0 by default.

```
// Makes a border around the edge of the map in layer 0, using set 3, tile 1,  
// and sets the collider cells for the border  
Tile.SetBorder (3, 1, true);  
// Same thing, but uses layer 2 and sets the collider cells of the border to false  
Tile.SetBorder (2, 3, 1, false);
```

```
static function SetBorder (layer : int = 0,  
                           tileInfo : TileInfo,  
                           setCollider : bool) : void
```

As above, but uses a TileInfo struct for the set and tile info.

```
// Makes a border around the edge of the map in layer 0, using set 3, tile 1,  
// and sets the collider cells for the border  
var tInfo = new TileInfo(3, 1);  
Tile.SetBorder (tInfo, true);  
// Same thing, but uses layer 2 and sets the collider cells of the border to false  
var tInfo = new TileInfo(3, 1);  
Tile.SetBorder (2, tInfo, false);
```

## SetCamera

```
static function SetCamera () : void
```

Initializes a camera or cameras for use with SpriteTile. This should be called first, before using any other SpriteTile functions such as LoadLevel or NewLevel. It should only be called once, unless a new scene has been loaded in Unity and the original camera destroyed. Without arguments, any camera tagged “MainCamera” is used. If there are multiple cameras tagged “MainCamera”, all are used.

If other scripts depend on SpriteTile being set up, make sure they run after SetCamera is called. For example, multiple Start functions have no defined order unless the script execution order is set explicitly in Unity, so using Awake is typically a good idea.

```
static function SetCamera (camera : Camera) : void
```

As above, but uses a specific camera instead of the MainCamera tag.

```
static function SetCamera (cameras : Camera[]) : void
```

As above, but uses a specified array of cameras instead of the MainCamera tag.

```
var myLevel : TextAsset; // Unityscript
var myCam : Camera;
var myCameras : Camera[];

function Awake () {
    // Sets the camera to any camera or cameras tagged MainCamera
    Tile.SetCamera();
    // This would use myCam instead:
    // Tile.SetCamera (myCam);
    // This would use an array of cameras:
    // Tile.SetCamera (myCameras);
    Tile.LoadLevel (myLevel);
}
```

```
public TextAsset myLevel; // C#
public Camera myCam;
public Camera[] myCameras;

void Awake () {
    // Sets the camera to any camera or cameras tagged MainCamera
    Tile.SetCamera();
    // This would use myCam instead:
    // Tile.SetCamera (myCam);
    // This would use an array of cameras:
    // Tile.SetCamera (myCameras);
    Tile.LoadLevel (myLevel);
}
```

## SetCollider

```
static function SetCollider (position : Int2,  
                             layer : int = 0,  
                             active : bool) : void
```

Sets the collider to either active (true) or inactive (false), of the cell at the coordinates, specified by the position, of the map. Other properties of the cell are not affected. The position can't be lower than (0, 0) and must be within bounds of the map. If the appropriate tile in the TileEditor has the "Use physics collider" setting checked, then a polygon collider is created from the sprite shape of this tile.

If the layer is omitted, then SetCollider will work on layer 0 by default.

```
// Set the cell in layer 0 at coords (10, 25) to an active collider  
Tile.SetCollider (new Int2(10, 25), true);  
// Same thing, but uses layer 1  
Tile.SetCollider (new Int2(10, 25), 1, true);
```

## SetColliderBlock

```
static function SetColliderBlock (position1 : Int2,  
                                   position2 : Int2,  
                                   layer : int = 0,  
                                   active : bool) : void
```

Like [SetCollider](#), except it works on a block of cells in the map, defined from position1 at one corner of the block up to and including position2 of the opposite corner. Both positions are clamped to the size of the map if necessary, and can be in any order; that is, position1 doesn't have to be less than position2.

If the layer is omitted, then SetColliderBlock will work on layer 0 by default.

```
// Set the cells in layer 0, using a block defined by (10, 18) at  
// one corner and (30, 25) at the other, to active collider cells  
Tile.SetColliderBlock (new Int2(10, 18), new Int2(30, 25), true);  
// Same thing, but uses layer 1  
Tile.SetColliderBlock (new Int2(10, 18), new Int2(30, 25), 1, true);
```

## SetColliderBlockSize

```
static function SetColliderBlockSize (size : int) : void
```

Sets the size of collider blocks used for polygon colliders. (See [Collider Blocks](#) in the How Colliders Work section.) This must be at least 1, with no particular upper limit. SetColliderBlockSize must be called before any level setup is done. If no polygon colliders are used, this function has no effect.

```
function Start () { // Unityscript
    // Sets the collider block size to 10x10
    Tile.SetColliderBlockSize (10);
    Tile.LoadLevel (myLevel);
    Tile.SetCamera();
}
```

```
void Start () { // C#
    // Sets the collider block size to 10x10
    Tile.SetColliderBlockSize (10);
    Tile.LoadLevel (myLevel);
    Tile.SetCamera();
}
```

## SetColliderLayer

```
static function SetColliderLayer (layer : int) : void
```

Sets the GameObject layer of all physics colliders used by SpriteTile to the specified layer. The layer must be at least 0 and not greater than 31.

```
// Sets the physics colliders to Unity's IgnoreRaycast layer, which is layer 2
Tile.SetColliderLayer (2);
```



## SetColliderMaterial

```
static function SetColliderMaterial (material : PhysicsMaterial2D) : void
```

Sets the PhysicsMaterial2D used for polygon colliders in the level. If no material is set, the default Unity PhysicsMaterial2D is used.

```
var colliderMaterial : PhysicsMaterial2D; // Unityscript
var myLevel : TextAsset;

function Start () {
    Tile.LoadLevel (myLevel);
    Tile.SetCamera();
    Tile.SetColliderMaterial (colliderMaterial);
}
```

```
public PhysicsMaterial2D colliderMaterial; // C#
public TextAsset myLevel;

void Start () {
    Tile.LoadLevel (myLevel);
    Tile.SetCamera();
    Tile.SetColliderMaterial (colliderMaterial);
}
```

## SetLayerActive

```
static function SetLayerActive (layer : int,
                                active : boolean) : void
```

Activates or deactivates the rendering for a given layer. Polygon colliders, if any, are not affected.

```
// Turns layer 1 off
Tile.SetLayerActive (1, false);
```

## SetLayerColor

```
static function SetLayerColor (layer : int = 0,
                                color : Color) : void
```

Sets the specified layer (0 by default) to the specified color. All tiles in the layer will be tinted by the color. This function uses the Color property in the SpriteRenderer component. Transparency will work as long as the tiles don't use a non-transparent shader.

```
// Makes layer 0 be tinted red
Tile.SetLayerColor (Color.red);
// Makes layer 1 be tinted a transparent purplish color
Tile.SetLayerColor (1, new Color(.7f, .2f, .6f, .5f));
```

## SetLayerPosition

```
static function SetLayerPosition (layer : int,  
                                   position : Vector2) : void
```

Sets the specified layer to a particular position in world space. This is primarily useful for layers that use a LayerLock other than LayerLock.None, such as fixed background layers, so you can position them as desired.

A good way to figure out what exact position to use is to first run a level without using SetLayerPosition, and while the level is running, move the SpriteTileLayerX object (where X corresponds to the appropriate layer) until it lines up as desired. Make note of the layer's Transform.Position numbers, stop play mode in Unity, and use those numbers in SetLayerPosition.

```
// Moves layer 1's X position to -5.0 and Y position to -3.0 in world space  
Tile.SetLayerPosition (1, new Vector2(-5.0, -3.0));
```

## SetMapBlock

```
static function SetMapBlock (position : Int2,  
                              layer : int = 0,  
                              mapData : MapData) : void
```

Sets a block of MapData to a specified position in the map. The position is the lower-left corner of the MapData block, and the size of the block at that position must not exceed the bounds of the map.

If the layer is omitted, then SetMapBlock will use layer 0 by default.

The mapData is typically retrieved by [GetMapBlock](#).

```
// Copies a block from (5, 10) to (15, 20) in layer 0,  
// and pastes it to location (50, 60) in layer 1  
var mapData = Tile.GetMapBlock (new Int2(5, 10), new Int2(15, 20));  
Tile.SetMapBlock (new Int2(50, 60), 1, mapData);
```

## SetOrder

```
static function SetOrder (position : Int2,  
                          layer : int = 0,  
                          order : int) : void
```

Sets the order-in-layer number of the cell at the coordinates, specified by the position, of the map in layer 0. Other properties of the cell are not affected. The position can't be lower than (0, 0) and must be within bounds of the map. The order number must be between -32768 and 32767.

If the layer is omitted, then SetOrder will work on layer 0 by default.

```
// Set the order in layer of the cell in layer 0 at coords (10, 25) to -3  
Tile.SetOrder (new Int2(10, 25), -3);  
// Same thing, but uses layer 1  
Tile.SetOrder (new Int2(10, 25), 1, -3);
```

## SetOrderBlock

```
static function SetOrderBlock (position1 : Int2,  
                                position2 : Int2,  
                                layer : int = 0,  
                                order : int) : void
```

Like [SetOrder](#), except it works on a block of cells of the map, defined from position1 at one corner of the block up to and including position2 of the opposite corner. Both positions are clamped to the size of the map if necessary, and can be in any order; that is, position1 doesn't have to be less than position2.

If the layer is omitted, then SetOrderBlock will work on layer 0 by default.

```
// Set the order in layer of the cells in layer 0, using a block defined  
// by (10, 18) at one corner and (30, 25) at the other, to -3  
Tile.SetOrderBlock (new Int2(10, 25), -3);  
// Same thing, but uses layer 1  
Tile.SetOrderBlock (new Int2(10, 25), 1, -3);
```

## SetRotation

```
static function SetRotation (position : Int2,  
                             layer : int = 0,  
                             rotation : float) : void
```

Sets the rotation of the cell at the coordinates of the map, specified by the position. Other properties of the cell are not affected. The position can't be lower than (0, 0) and must be within bounds of the map. The rotation is between 0.0 and 360.0. Any numbers outside that range are repeated so they can be represented by the 0.0 to 360.0 range; that is, -45.0 would become 315.0.

If the layer is omitted, then SetRotation will work on layer 0 by default.

```
// Set the rotation of the cell in layer 0 at coords (10, 25) to 45°  
Tile.SetRotation (new Int2(10, 25), 45.0);  
// Same thing, but uses layer 1  
Tile.SetRotation (new Int2(10, 25), 1, 45.0);
```

## SetRotationBlock

```
static function SetRotationBlock (position1 : Int2,  
                                   position2 : Int2,  
                                   layer : int = 0,  
                                   rotation : float) : void
```

Like [SetRotation](#), except it works on a block of cells of the map, defined from position1 at one corner of the block up to and including position2 of the opposite corner. Both positions are clamped to the size of the map if necessary, and can be in any order; that is, position1 doesn't have to be less than position2.

If the layer is omitted, then SetRotationBlock will work on layer 0 by default.

```
// Set the rotation of the cells in layer 0, using a block defined  
// by (10, 18) at one corner and (30, 25) at the other, to 45°  
Tile.SetRotationBlock (new Int2(10, 18), new Int2(30, 25), 45.0);  
// Same thing, but uses layer 1  
Tile.SetRotationBlock (new Int2(10, 18), new Int2(30, 25), 1, 45.0);
```

## SetTile

```
static function SetTile (position : Int2,  
                        layer : int = 0,  
                        set : int,  
                        tile : int) : void
```

```
static function SetTile (position : Int2,  
                        layer : int = 0,  
                        tileInfo : TileInfo) : void
```

Sets the cell at the coordinates of the map, specified by the position, to the tile specified by the set and tile numbers. Other properties of the cell are not affected. The set and tile numbers can be seen in the TileEditor window. The position can't be lower than (0, 0) and must be within bounds of the map.

If the layer is omitted, then SetTile will work on layer 0 by default.

The set and tile numbers must refer to sets and tiles that exist in the TileManager. They can either be specified separately as ints, or by using a TileInfo struct.

```
// Sets the cell in layer 0 at coords (10, 25) to set 2, tile 5  
Tile.SetTile (new Int2(10, 25), 2, 5); // Using ints  
Tile.SetTile (new Int2(10, 25), new TileInfo(2, 5)); // Using TileInfo  
// Same thing, but uses layer 1  
Tile.SetTile (new Int2(10, 25), 1, 2, 5); // Using ints  
Tile.SetTile (new Int2(10, 25), 1, new TileInfo(2, 5)); // Using TileInfo
```

```
static function SetTile (position : Int2,  
                        layer : int = 0,  
                        set : int,  
                        tile : int,  
                        setCollider : boolean) : void
```

```
static function SetTile (position : Int2,  
                        layer : int = 0,  
                        tileInfo : TileInfo,  
                        setCollider : boolean) : void
```

As above, but the collider of the cell will also be set, depending on the value of setCollider.

```
// Sets the cell in layer 0 at coords (10, 25) to set 2, tile 5,  
// and sets the corresponding collider to true  
Tile.SetTile (new Int2(10, 25), 2, 5, true); // Using ints  
Tile.SetTile (new Int2(10, 25), new TileInfo(2, 5), true); // Using TileInfo  
// Same thing, but uses layer 1  
Tile.SetTile (new Int2(10, 25), 1, 2, 5, true); // Using ints  
Tile.SetTile (new Int2(10, 25), 1, new TileInfo(2, 5), true); // Using TileInfo
```

## SetTileBlock

```
static function SetTileBlock (position1 : Int2,  
                             position2 : Int2,  
                             layer : int = 0,  
                             set : int,  
                             tile : int) : void
```

```
static function SetTileBlock (position1 : Int2,  
                             position2 : Int2,  
                             layer : int = 0,  
                             tileInfo : TileInfo) : void
```

Like [SetTile](#), except it works on a block of cells, defined from position1 at one corner of the block up to and including position2 of the opposite corner. Both positions are clamped to the size of the map if necessary, and can be in any order; that is, position1 doesn't have to be less than position2.

If the layer is omitted, then SetTileBlock will work on layer 0 by default.

The set and tile numbers must refer to sets and tiles that exist in the TileManager. They can either be specified separately as ints, or by using a TileInfo struct.

```
// Sets a block of cells in layer 0, defined by (10, 18) at one corner and  
// (30, 25) at the other, to set 2, tile 5  
Tile.SetTileBlock (new Int2(10, 18), new Int2(30, 25), 2, 5);  
var tInfo = new TileInfo(2, 5);  
Tile.SetTileBlock (new Int2(10, 18), new Int2(30, 25), tInfo);  
// Same thing, but uses layer 1  
Tile.SetTileBlock (new Int2(10, 25), new Int2(25, 18), 1, tInfo);
```

```
static function SetTileBlock (position1 : Int2,  
                             position2 : Int2,  
                             layer : int = 0,  
                             set : int,  
                             tile : int,  
                             setCollider : boolean) : void
```

```
static function SetTileBlock (position1 : Int2,  
                             position2 : Int2,  
                             layer : int = 0,  
                             tileInfo : TileInfo,  
                             setCollider : boolean) : void
```

As above, but the collider of the cells in the block will also be set, depending on the value of setCollider.

```
Tile.SetTileBlock (new Int2(10, 18), new Int2(30, 25), 2, 5, true);  
Tile.SetTileBlock (new Int2(10, 18), new Int2(30, 25), new TileInfo(2, 5), true);
```

## SetTileLayerScale

```
static function SetTileLayerScale (layer : int = 0,  
                                   scale : float) : void
```

Sets the scale of all tiles in the specified layer to the value specified by `scale`. This works the same as `SetTileScale`, except that it only affects the specified layer and overrides any default scale value supplied by `SetTileScale`. Note, however, that any new sprites created in the layer (such as by zooming out) will use the default, so `SetTileLayerScale` may need to be called again in that case.

If the layer is omitted, then `SetTileLayerScale` will use layer 0 by default.

```
// Sets the scale of all tiles in layer 0 to 1.5  
Tile.SetTileLayerScale (1.5f);  
// Sets the scale of all tiles in layer 1 to 1.25  
Tile.SetTileLayerScale (1, 1.25f);
```



## SetTileMaterial

```
static function SetTileMaterial (material : Material) : void
```

Sets the material used for all tiles in the level. If no material is set, the default Unity sprite material is used, unless overridden by the “non-transparent” or “use dynamic lighting” options in the TileEditor.

```
// Makes tile sprites use Sprites/Diffuse shader
var newMaterial = new Material(Shader.Find ("Sprites/Diffuse"));
Tile.SetTileMaterial (newMaterial);
```

```
static function SetTileMaterial (set : int,
                                  tile : int,
                                  material : Material) : void
```

```
static function SetTileMaterial (tileInfo : TileInfo,
                                  material : Material) : void
```

Sets the default material for all tiles using the specified set and tile numbers. This will override the non-transparent or dynamic lighting options in the TileEditor.

```
// Makes all tile sprites in set 3, tile 12 use the supplied material
Tile.SetTileMaterial (new TileInfo(3, 12), myMaterial);
```

```
static function SetTileMaterial (layer : int = 0,
                                  position : Int2,
                                  material : Material) : void
```

Sets the material for the tile at the specified position. If the layer is omitted, layer 0 will be used by default. This will override all other material settings from either the TileEditor or by using SetTileMaterial with a set/tile number. Note that this usage of SetTileMaterial will cause cells in the map to use 8 bytes instead of 7, and a maximum of 256 different materials are allowed.

```
// Makes the tile at (5, 10) in layer 1 use the supplied material
Tile.SetTileMaterial (1, new Int(5, 10), myMaterial);
```

## SetTileRenderLayer

```
static function SetTileRenderLayer (layer : int) : void
```

Sets the layer used for all tiles in the level. This is the GameObject layer as opposed to the sorting layer. The layer value must be between 0 and 31.

```
// Makes tile sprites use the IgnoreRaycast layer
Tile.SetTileRenderLayer (2);
```

## SetTileScale

```
static function SetTileScale (scale : float) : void
```

Sets the scale of all tiles to the value specified by `scale`. The scale affects both the X and Y axes. This can be called at any time, even before `SetCamera`. `SetTileScale` can be used for special effects, and it can also be used if occasional 1-pixel gaps between tiles are visible when the camera is moved. Specifying a value slightly greater than 1.0 in this case will typically eliminate any such gaps.

```
// Sets the scale of all tiles to 1.001
Tile.SetTileScale (1.001f);
```

## SetTrigger

```
static function SetTrigger (position : Int2,  
                             layer : int = 0,  
                             trigger : int) : void
```

Sets the trigger number of the cell at the coordinates in the map, specified by the position. Other properties of the cell are not affected. The position can't be lower than (0, 0) and must be within bounds of the map. The trigger number must be between 0 and 255.

If the layer is omitted, then `SetTrigger` will work on layer 0 by default.

```
// Set the trigger of the cell in layer 0 at coords (10, 25) to 3
Tile.SetTrigger (new Int2(10, 25), 3);
// Same thing, but uses layer 1
Tile.SetTrigger (new Int2(10, 25), 1, 3);
```

## SetTriggerBlock

```
static function SetTriggerBlock (position1 : Int2,  
                                  position2 : Int2,  
                                  layer : int = 0,  
                                  trigger : int) : void
```

Like [SetTrigger](#), except it works on a block of cells, defined from `position1` at one corner of the block up to and including `position2` of the opposite corner. Both positions are clamped to the size of the map if necessary, and can be in any order; that is, `position1` doesn't have to be less than `position2`.

If the layer is omitted, then `SetTriggerBlock` will work on layer 0 by default.

```
// Set the trigger of the cells in layer 0, using a block defined
// by (10, 18) at one corner and (30, 25) at the other, to 3
Tile.SetTriggerBlock (new Int2(10, 18), new Int2(30, 25), 3);
// Same thing, but uses layer 1
Tile.SetTriggerBlock (new Int2(10, 18), new Int2(30, 25), 1, 3);
```

## StopAnimatingTile

```
static function StopAnimatingTile (tileInfo : TileInfo) : void
```

Stops a tile from animating. If [AnimateTile](#) has been called for the specified tileInfo, then the animation will be halted immediately. If tileInfo is currently not animating, then nothing will happen.

```
// Stops tile #10 in set 1 from animating
Tile.StopAnimatingTile (new TileInfo(1, 10));
```

## StopAnimatingTileRange

```
static function StopAnimatingTileRange (tileInfo : TileInfo,
                                         range : int) : void
```

Stops a range of tiles from animating. If [AnimateTileRange](#) has been called for the specified tileInfo and range, or if [AnimateTile](#) has been called for any of the tiles in the range, then the animation for the tile or tiles will be halted immediately. If any of the tiles in the range are currently not animating, then they will be ignored.

```
// Stops tiles 20-29 in set 1 from animating
Tile.StopAnimatingTileRange (new TileInfo(1, 20), 10);
```

## UseTileEditorDefaults

```
static function UseTileEditorDefaults (useDefaults : boolean) : void
```

Normally SetTile and SetTileBlock will use 0 as the defaults for rotation and order-in-layer, or if useDefaults is false. If useDefaults is true, however, then the [SetTile](#) and [SetTileBlock](#) functions will use the per-tile defaults set in the TileEditor for the appropriate tile (see [Tile Defaults](#)). So any defaults set for the collider, order-in-layer, and rotation are also set. For example, if set 3, tile 1 had defaults of 2 for order-in-layer, 90 for rotation, and Collider was checked, then this:

```
Tile.UseTileEditorDefaults (true);
Tile.SetTile (new Int2(5, 5), 3, 1);
```

is the equivalent of this:

```
Tile.UseTileEditorDefaults (false);
Tile.SetTile (new Int2(5, 5), 3, 1, true);
Tile.SetOrder (new Int2(5, 5), 2);
Tile.SetRotation (new Int2(5, 5), 90.0f);
```