SpriteTile

Version 3.0

Thank you for purchasing SpriteTile! SpriteTile comes as a pair of DLLs (recommended) or source code. You can have either DLLs or source in your project, but not both; you should remove the DLLs before importing the source. The Demos package contains a number of demos that illustrate various SpriteTile functionality. You should make sure the DLLs are present before importing the Demos package.

Note: you'll need to add two sorting layers to your project manually if you run the "Acorn Antics" or "Gem Hunt" demos, since SpriteTile can't do this automatically. The "Animated Tiles" demo also requires an added sorting layer. Select the "Edit -> Project Settings -> Tags and Layers" menu item to add the layers, then select the "Assets -> Set SpriteTile Sorting Layer Names" menu item to apply them to SpriteTile. See the "How layers work" section for more details.

If you're just starting out, you may want to have a look at the **SpriteTile Quick Start Guide** first, and come back to this document for more details. For a list of all SpriteTile functions that you can use when writing code, see the **SpriteTile Reference Guide**. If you have any questions you can contact sales@starscenesoftware.com.

contents

Terminology ----------2

Tile editor overview---------3

Tile editor: tiles ----------4

(file operations tileset controls set selection
tile defaults moving tiles tile window
view options)

Tile editor: level ----------12

(file operations importing Tiled maps)

Layer controls----------16

Light controls ----------19

Selection controls ------22

(order fill)

Tools ----------26

Map window ----------28

(multi-selection overlapping tiles lines
bookmarks cell properties view options)

Tile editor: groups ----------34

(file operations group type standard groups
random groups terrain groups)

How layers work----------39

How colliders work---------40

Coding basics ----------42

(setting the camera Int2 struct TileInfo struct
loading a level creating a level in code
tile animation camera rotation memory usage)

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:

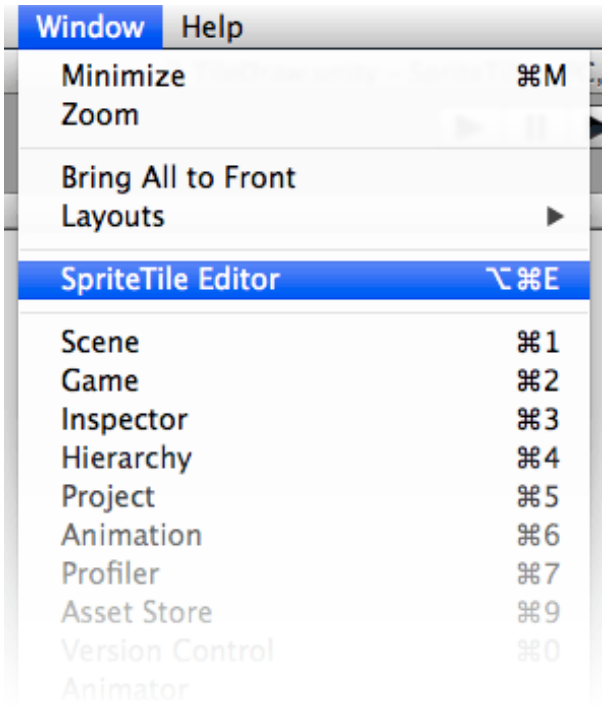
1) Remove the DLLs from the Plugins folder and the SpriteTile/Editor folder.

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



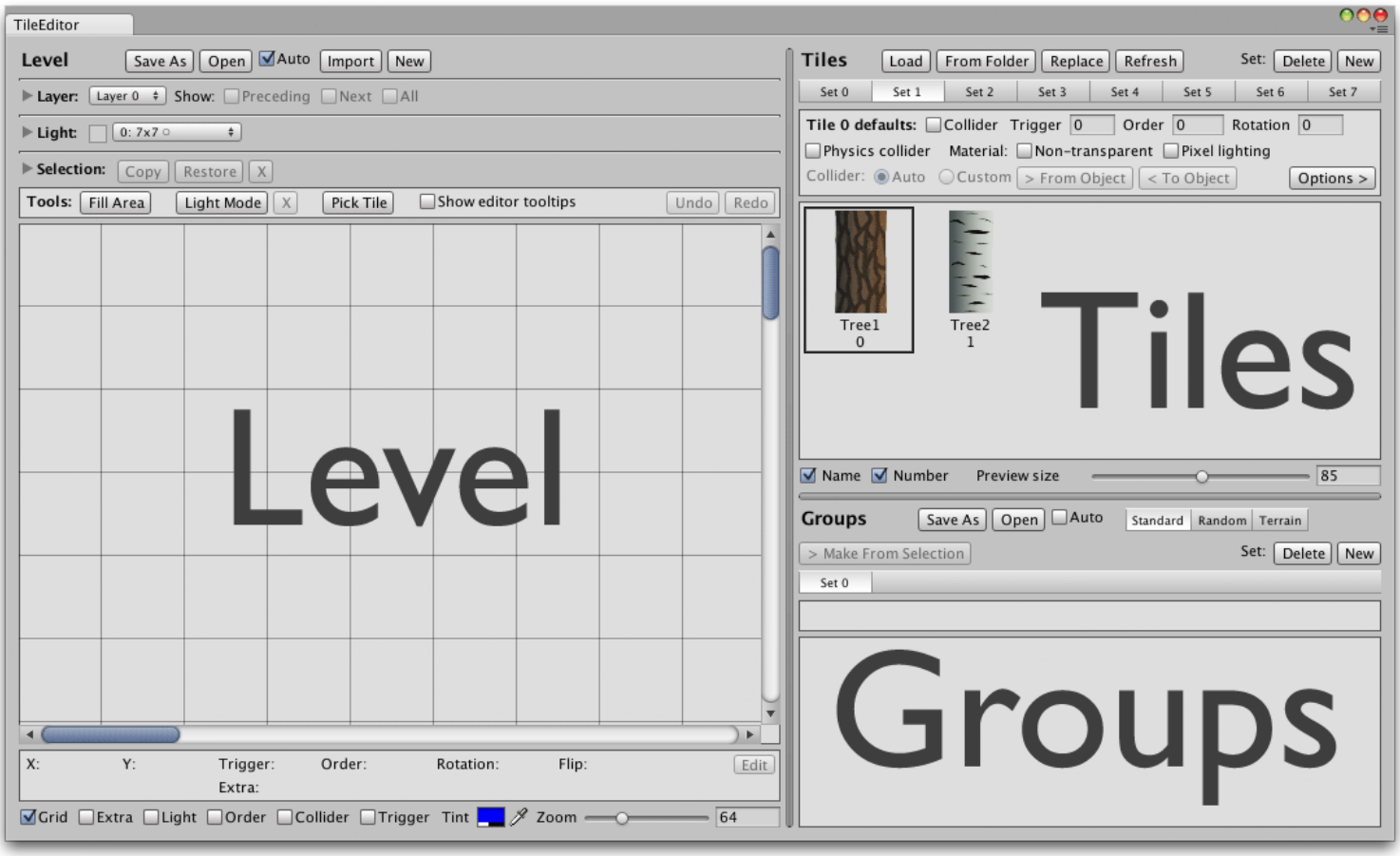
The TileEditor window is where you can create and edit SpriteTile levels. You can access it by selecting “SpriteTile Editor” in the Unity Window menu, or by pressing Alt-Cmd-E (on Mac) or Alt-Ctrl-E (on Windows).

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 may see that there are two draggable dividers: one between the Level section and the Tiles/Groups section, and one between the Tiles section and the Groups section. These can be dragged to give more space to whichever section you like. Note that the TileEditor window must be resized to be wider than its default width before the vertical divider can be dragged, since there is a minimum size for each section.

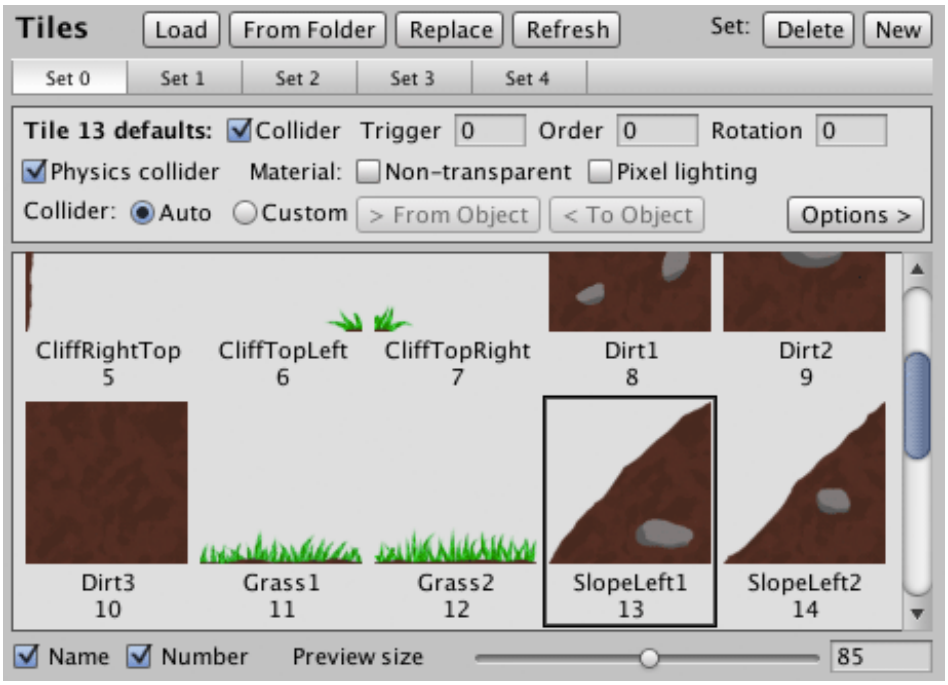
You need at least one tile before you can draw anything in the Level section, so we’ll start by taking a look at the Tiles section.



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.

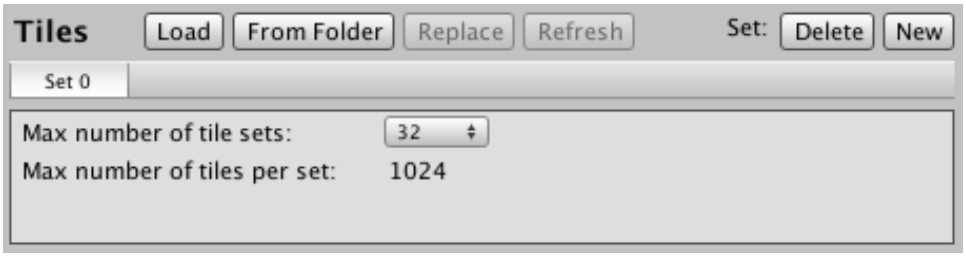
Note: the TileManager file should not be renamed or moved! It's required for SpriteTile to function.

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



setting up tile sets

When you start the TileEditor for the first time, you'll be able to choose the maximum number of tile sets for your project. The fewer tile sets, the more tiles you can use in each set. The total number of possible tiles always adds up to 32,768.



The default of 32 sets is recommended, since that gives you plenty of sets to make organizing tiles easy. You can use fewer sets if you really need to have large numbers of tiles in each set.

Once you start adding tiles or sets, the max number of sets can't be changed, and this configuration panel will no longer be available. But if necessary, you can delete all tiles, and it will show up again. Then you can make changes and re-load your tiles.

file operations

Load: use this to select 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 (that is, a sprite atlas), 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 the maximum number of tiles allowed in a set, or if they would cause the tile count for the current set to exceed the maximum, they won't be loaded.

If you loaded a texture atlas, and have since added new tiles to the atlas, you can use the Load button to re-load it. Only the new tiles will be added. If you re-load a texture or atlas that already exists, and no new tiles have been added, you'll be asked if you want to update the tiles or cancel.

From Folder: this button loads all the sprite textures in a folder into the current set at once. If you're using a single sprite per texture (such as if you have Unity Pro and are using the auto-atlas functionality), 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 black 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:

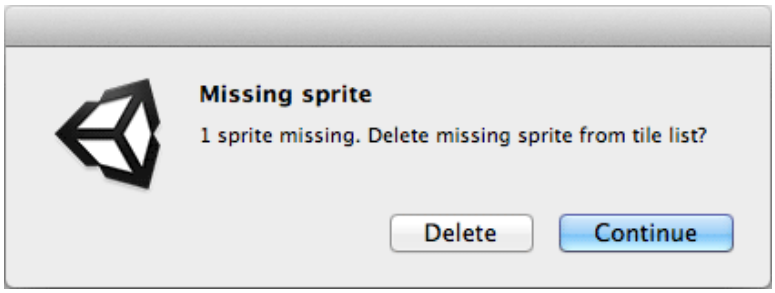
What sort of things can I change?

- 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)

Note that if you want to delete a sprite, it's better if you delete the sprite from the TileEditor first, then from your project. However, if you delete the sprite from your project first, opening the TileEditor will display a dialog that asks you if you want to delete it from the tile list. Clicking "Delete" will automatically delete the tile. Clicking "Continue" will leave the tile in the list, though it won't be usable, and you can delete it manually later.



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, and any groups that may have used the deleted tiles will also be removed.

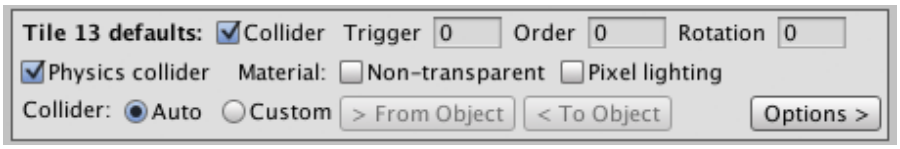
Careful with this! Remember that tiles are shared between all your levels. If you delete a tile set that's used by levels you don't currently have loaded, deleting the set may adversely affect the unloaded levels, since they will contain references to tiles that don't exist in SpriteTile anymore. It's still possible to load a level with missing tiles if this happens; you'll just get a message about it, and the missing tiles show up in the level as an "X".

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.

As discussed above, there's a maximum number of sets that can be used, the number of which depends on what you configured. It's OK to have empty sets that you reserve for future use. There must always be at least one set—if you click Delete when Set 0 is selected, and there are no more sets, it 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. The tile number here will change when you click on different tiles, and each tile has its own settings. The defaults shown are always for the currently-selected tile. If you have multiple tiles selected, it will say “Multi defaults” instead of the tile number, and any changes made to the defaults will be applied to all selected tiles.

The defaults for the collider, trigger, order, and rotation are used when you initially lay down tiles in the map window, but can be changed in the level later if desired on a per-cell basis. This way you can, say, have the collider default checked for a tile, so when you draw with the tile in the level, all cells containing that tile will have a collider. You can go back and remove the collider from individual cells later if you want. If you turned the collider default off, then drawing with that tile would no longer include a collider, but it wouldn’t affect tiles already placed in the level.

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.

Trigger: if you want a tile to have a trigger number associated with it, then you can set it here. So when drawing with that tile, the trigger number will be placed automatically too. (As long as it’s not 0, that is. If it’s 0, then it’s ignored so that the tile won’t affect any existing trigger numbers.) You can always change the trigger number for tiles in the level view later if desired. See Trigger in Cell Properties for more details.

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 cell basis. See **Order in layer** in Cell Properties for more details. If the default order value is 0, then drawing with the tile will not affect order values that may already exist in the map.

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.

Physics collider: this is used to mark tiles as polygon collider generators. Regardless of whether cells in your level are marked as collider cells, to use a particular tile as a physics collider, this checkbox must be on. Turning on **physics collider** will also turn on **collider** automatically, if it isn’t already, since cells need to be marked as collider cells in order for physics colliders to be activated. 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 the **SetTileMaterial** function.

Pixel lighting: along the same lines, the default sprite shader is unlit, so if you want tiles to react to realtime lights in Unity, 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.

The next two options are only available if **Physics collider** is checked:

Collider: Auto: when creating a polygon collider for this tile, SpriteTile will use Unity's built-in polygon collider generation, which works based on the shape of the sprite. Note that if a tile is drawn with semi-transparent pixels, the auto-collider generation may not work; for best results, pixels should be either fully transparent or fully opaque. You can check this by dragging a sprite into the Unity scene view, and adding a Polygon Collider 2D component. If Unity adds a default pentagon shape instead of the expected shape, then no collider will be generated in SpriteTile. You can fix this by making semi-transparent pixels more transparent, or by using a custom collider (described next).

Custom: if the auto-generated collider isn't sufficient for any reason, you can use a custom collider instead. Follow these steps:

1) In the Unity editor, create a GameObject with a PolygonCollider2D or BoxCollider2D component.

2) Edit the collider as desired. See the PolygonCollider2D page in the Unity documentation for details about editing polygon colliders. If using a box collider, you can edit the size and center.

3) With this GameObject selected, switch back to the SpriteTile editor.

4) Make sure "custom collider" is active, and click the "**> From Object**" button. The collider you created is then copied to this tile, and you'll get a message saying the operation was successful.

Note that the collider on the GameObject is not linked to the SpriteTile collider. So you can delete the GameObject in the Unity editor, and the tile will retain the custom collider.

If the From Selected button remains grayed-out, that means you don't have an object selected in Unity. Switch back to Unity, select the object, then switch back to the TileEditor window.

You can use the "**< To Object**" button to edit existing custom tile colliders. Follow these steps:

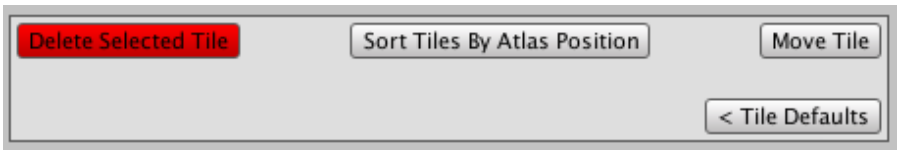
1) In the Unity editor, create a GameObject.

2) With this object selected, switch back to the SpriteTile editor.

3) Click the "**< To Object**" button. The custom tile collider is copied to the object as a PolygonCollider2D component, where it can be edited as usual.

After editing, the collider can then be copied back to the tile by clicking the From Object button. Note that auto-generated colliders can also be copied to objects for editing, by changing the collider from Auto to Custom and clicking the "**< To Object**" button.

Options: changes the tile defaults panel to the options panel, which has buttons for deleting, sorting, and moving tiles. These are detailed below.



tile options

The options panel replaces the tile defaults panel when clicking the “Options” button. Clicking on the “Tile Defaults” button will go back to the tile defaults panel.

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.

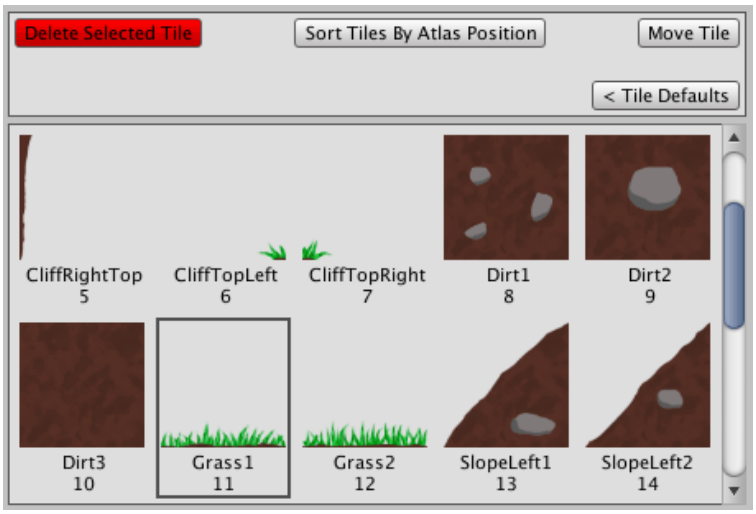
If you have multiple tiles selected, the button will read “Delete Selected Tiles”, and clicking it will remove all the tiles.

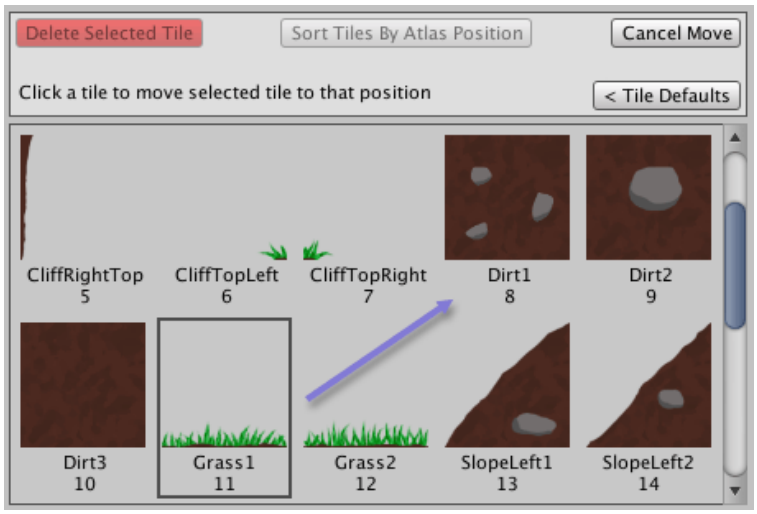
Sort Tiles By Atlas Position: this button will only work if all tiles in the current set are part of an atlas. If any of them are single sprites, you’ll get a dialog informing you of this. Clicking this button will rearrange all the tiles in the set, so that they are sorted in the TileEditor the same way they are in the sprite atlas, top to bottom, left to right. Normally, an imported atlas will be sorted like this anyway, but it’s possible that the tiles can be rearranged in some circumstances. Also, you can use this to reset the tiles to the default atlas positions if you’ve manually moved them yourself (see below), and have changed your mind.

Move Tile: you can use this to move the currently selected tile around in the tile list. If you have multiple tiles selected, the button will read “Move Tiles”, and all selected tiles will be moved simultaneously. See below for details.

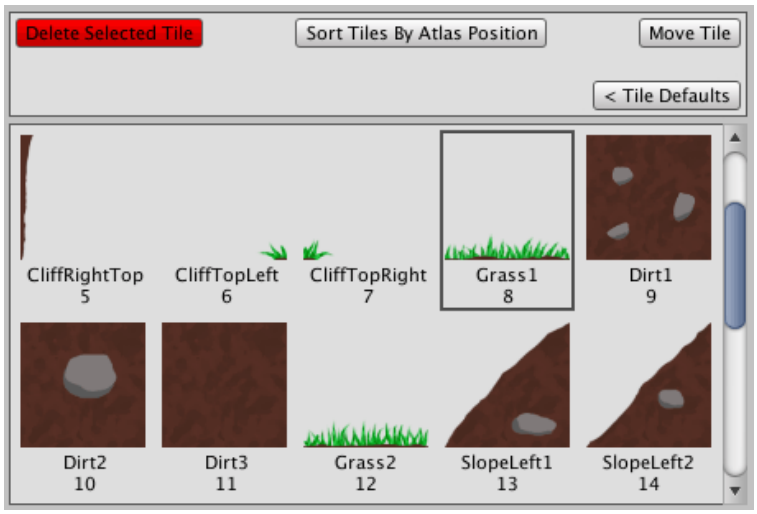
moving tiles

There are times when you might want to rearrange the order of the tiles in the list. To do this, first make sure you're in the options panel, select the tile or tiles that you want to move, then click the "Move Tile" 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 Tile".



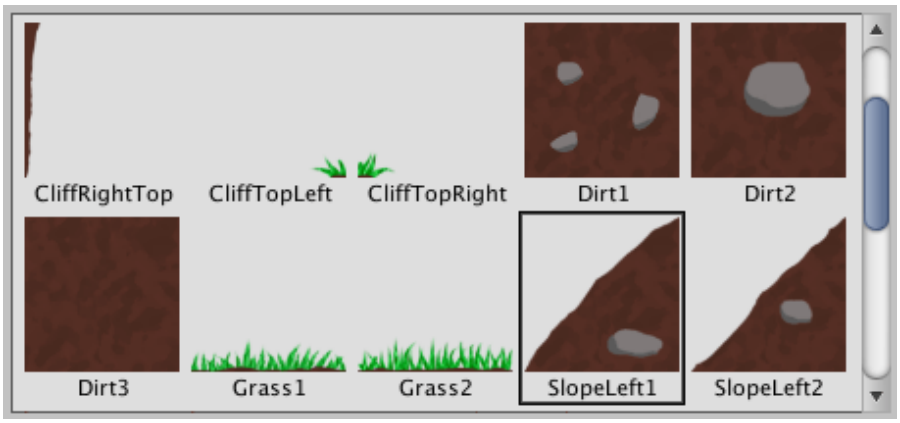


We then click on tile #8, so tile #11 becomes #8 in the list and the following tiles get pushed down. If we repeat the process by clicking on tile #12, Move, and then tile #9, the grass tiles would be all in a row.



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, but that's often the reason for moving tiles in the first place. 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.

When you're moving a tile, the "Move Tile" button is replaced by a "Cancel Move" button. Click this if you decide not to move the tile after all. Clicking "Tile Defaults" will also cancel a move action before swapping the panel back to the tile defaults.



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 black box around it. The selected tile is the one that's used for operations in the map 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, changes to any of the defaults will be applied to all selected tiles. This is also useful for adding tiles to Random groups.

Another way to multi-select is to draw a box around the desired tiles. Use the right mouse button to click and drag in the tile window (Command-click will also work on Macs), and any tiles that the box touches will be selected. They will also be copied to the copy buffer (see Selection Controls below), so you can draw with them as a group in the level, or add them to a standard group using the "make from selection" button.

Also, if you alt-click on a tile, it will be added to the selection. Alt-clicking an already-selected tile will remove it from the selection. You can combine right-button drag-selecting with alt-clicking if desired.

view options

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

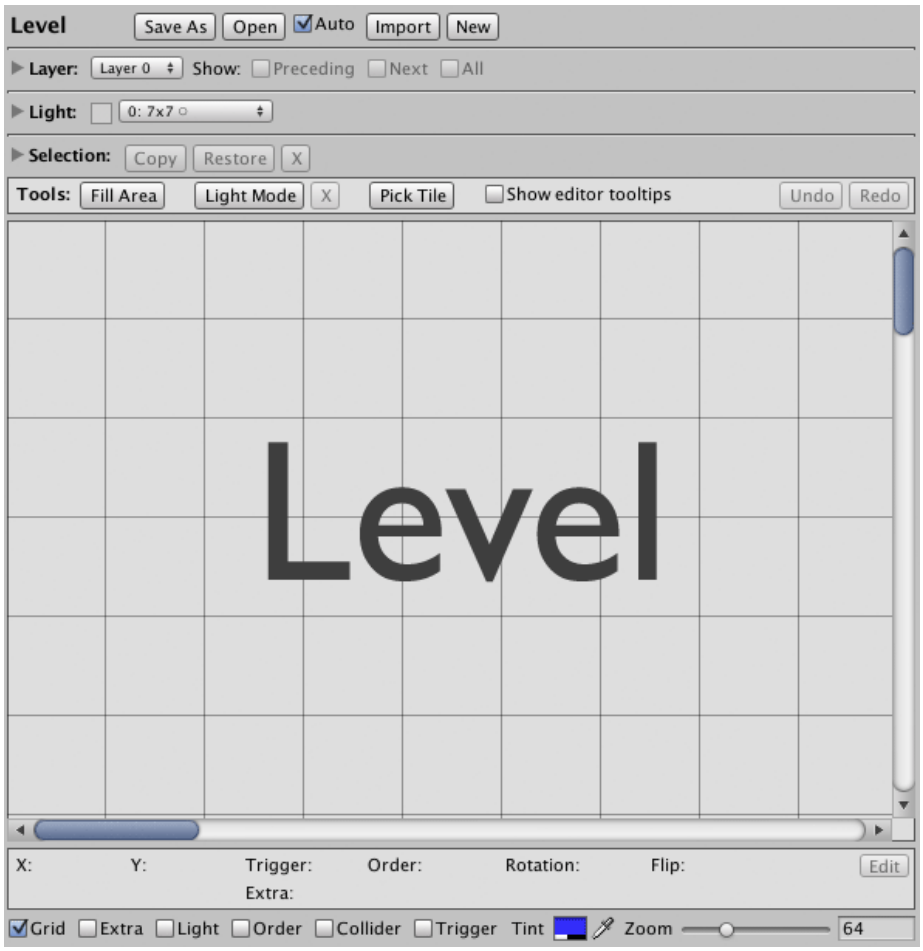
Number: this checkbox toggles the tile number display under the tile images. The tile number is frequently useful for coding, so you can refer to the appropriate tile with functions like SetTile and so on.

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. •

The **Level** section is where you will spend most of your time building levels. You can draw with the currently-selected tile into the map 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. If you've made unsaved changes to the current level, a dialog will ask if you want to save the changes before loading a new level.

Note that if a loaded level contains tiles that have been deleted, a dialog will list the missing tiles, and the missing tiles themselves will be displayed in the level as a box with an X in it, so you can easily see where they were.

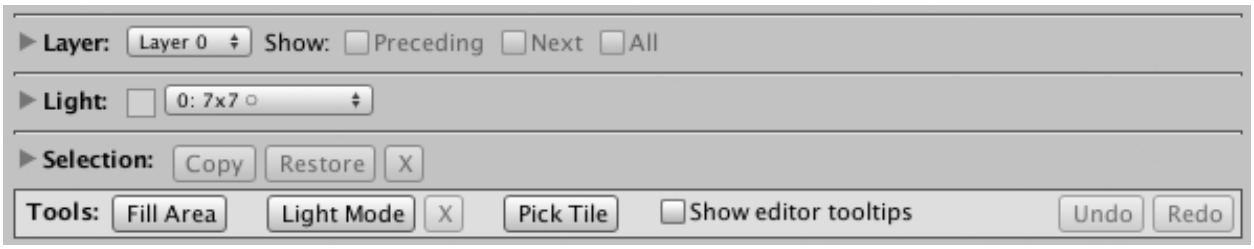
Auto: if this is checked, the last saved or opened level will be automatically loaded every time you open the TileEditor window.

Import: imports a Tiled .tmx file. See Importing a Tiled Map below for more details.

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.

controls

Below the file operation controls are several sections, each of which can be collapsed independently. When collapsed, the section will display only a few of the most commonly-used controls, so you can have them open only if you need the more comprehensive controls. The sections are Layer, Light, and Selection, with a Tools box below, so later we'll look at them in order.

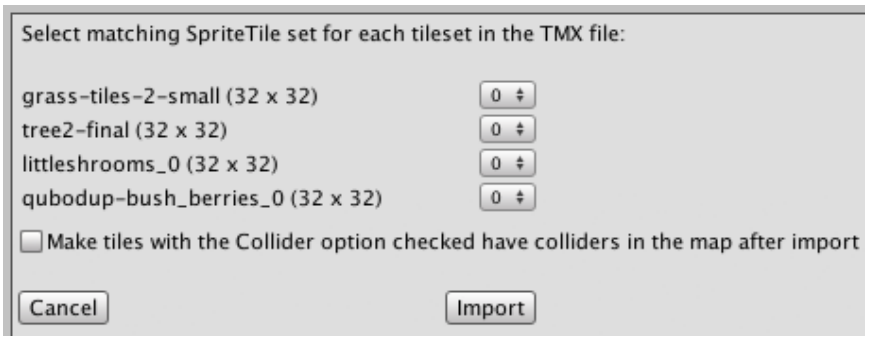


importing a tiled map

If you have a map that was made with the Tiled map editor (.tmx), you can import it into SpriteTile, as long as it's an orthogonal map (isometric and staggered are not supported) and saved as uncompressed XML. If the map was saved in another format, you can go into the Tiled preferences and set "Store tile layer data" to XML, then save the map again.

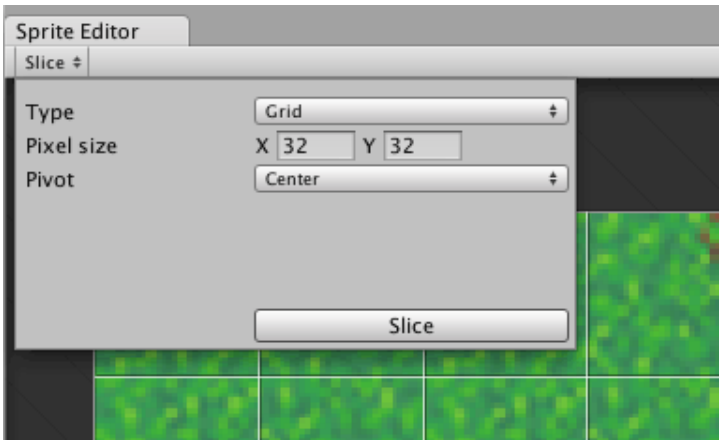
Since Tiled and SpriteTile store tiles in a fundamentally different manner, there's a little manual work involved. SpriteTile needs the tiles to be Unity sprites, so you will first need to set these up. Basically, every tileset used in the Tiled file should be made into a SpriteTile set.

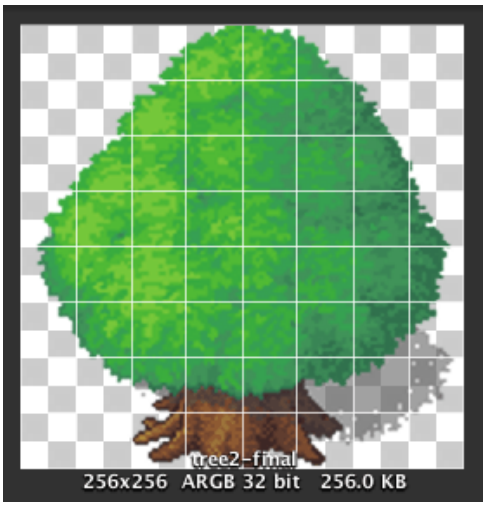
To see how this is done, let's take a look at this sample Tiled map. Click on the **Import** button in the TileEditor, find and select the "example.tmx" file, and the level view will switch to this:



So we need four sets.

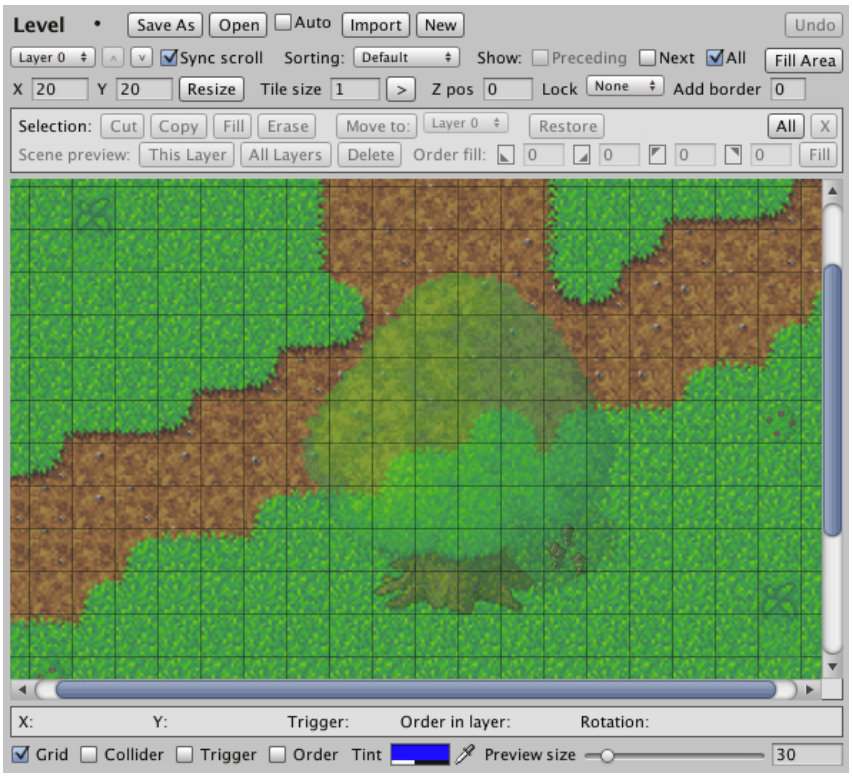
- Find the listed image files and drag them into Unity.
- In the texture inspector, make sure the **Texture Type** for each one is set to **Sprite**.
- The **Sprite Mode** can be left as **Single** for individual images (such as the "littleshrooms" and "qubodup-bush_berries" files), but needs to be changed to **Multiple** for tilesets (such as the "grass-tiles-2-small" and "tree2-final" files).
- Next to the file name, the tile pixel size is listed in parentheses. Tilesets need to be sliced using this tile size—click on the **Sprite Editor** button, click on **Slice** in the upper-left corner of the sprite editor, and change the pixel size to whatever is shown in the list. In this case, they are all 32 x 32 tiles. Then click the **Slice** button, and **Apply**.





Note that Unity can sometimes be too smart for its own good when creating tilesets from images with transparency. Any sprite that's completely empty is skipped, which will cause errors when attempting to import the Tiled map, since the numbers won't match. For example, in this case the tree2-final tileset has a few sprites in the corners that don't get created.

Fortunately, it's easy to trick Unity into doing what we want. First load the file into Photoshop or a similar app, and export it without transparency, replacing the original (but don't close the image in Photoshop yet!). This way the image can be sliced properly in Unity, since none of the sprites will be skipped. After slicing, go back to Photoshop and re-save the image, but with transparency, replacing the non-transparent version. Presto, the image has transparency but the number of sprites is still correct.



Now we can create the sets in the TileEditor. If necessary, click on the **New** set button in the Tiles section to make a new set. If you don't have any sets loaded yet, then you can use set 0 and don't need to make a new set at this time. In either case, now click on the **Load** button. Find the images you just converted and load the first one. (If the image file was sliced, SpriteTile will load all the tiles automatically.) Now click on the **New** button to make a new set, then load the next tileset, and so on until each tileset is loaded into its own set.

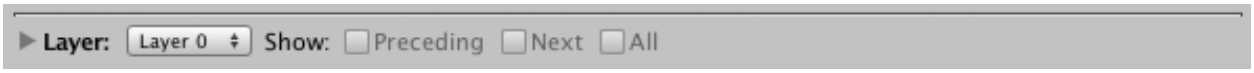
Finally, go back to the **Level** section where the list of tilesets is still waiting patiently for us to finish this stuff. Click on the drop-down buttons to select a matching SpriteTile set for each Tiled tileset. For example, if you imported these four tilesets as set 0, set 1, set 2, and set 3, then select 0, 1, 2, and 3 respectively.

When you click **Import**, the Tiled map is loaded. If there are multiple layers (this example has three), then they are loaded into separate layers in the TileEditor. You should save this as a SpriteTile file now, since only the TileEditor can import .tmx files—when you use Tile.LoadLevel, it can only load SpriteTile levels. Here the TileEditor shows the final result, with layer 0 as the primary layer and the other layers overlaid on top:

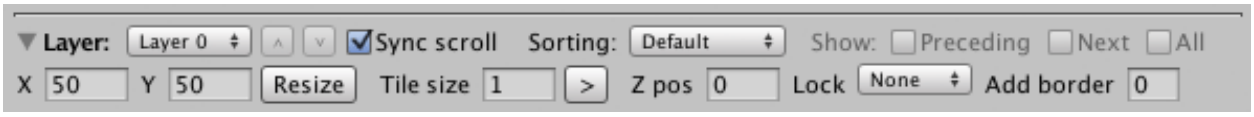
Colliders

In some cases you may want tiles, which have been set to have colliders in the tile defaults, to automatically be set as collider cells when importing a TMX level. That's what the "Make tiles with the Collider option checked have colliders in the map after import" option on the import screen is for—just check the box, and it will do exactly that. This way you don't need to manually set colliders yourself after importing.

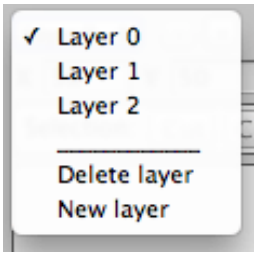
layer controls



When open, the layer section displays all the controls. When closed, only the layer drop-down menu and layer visibility controls are displayed.



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.

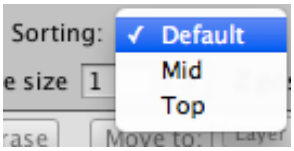


Delete layer: this will delete the currently-active layer.

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 scroll: 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.



Sorting drop-down menu: By default, the first sorting layer (as created in the Unity Tags And Layers project setting) is assigned to the first layer in your level, the second sorting layer is assigned to the second layer, etc. But you can use this drop-down menu to manually assign an arbitrary sorting layer to the current layer. See How Layers Work for more details.

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

Show: Next: 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.

Show: All: if this is checked, then all layers that have the same dimensions as the current layer are shown, at 50% opacity.

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.

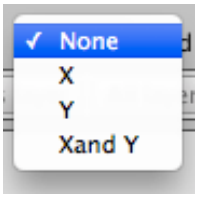
You can get the total size of a level in units by multiplying the X or Y values by the tile size, so a 50 x 25 level with a tile size of 2.0 would be 100 x 50 units.

> 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.

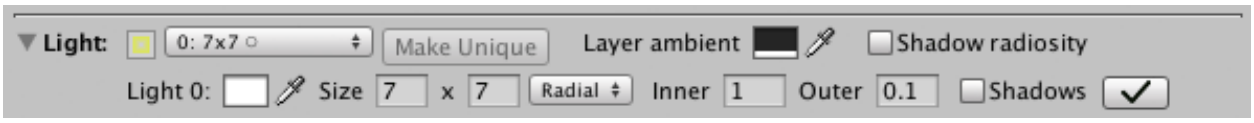
Add border: If you have oversized tiles that don't fit inside cells, you may find that they "pop in" when scrolling rather than smoothly becoming visible. To fix this, you can add more cells that extend beyond the border of the screen. Keep in mind that every number above 0 decreases efficiency a bit, 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.

This is also useful when rotating the camera (see Camera Rotation). In this case the exact amount can depend on how much you're rotating the camera, so you may need to experiment a little to find the best value.

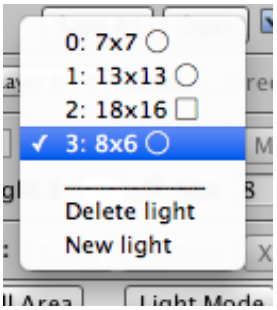
Note that the Unity GUI has some limitations when it comes to displaying rotated and scaled tiles inside a scrollview, and the add border value has no effect in the editor anyway, so what you may see in the TileEditor when scrolling is **not** representative of what you'll actually see in your game.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

light controls

When open, the light section displays all the controls. When closed, only the light selection indicator and drop-down menu are displayed.



Tile-based lights in SpriteTile are created by using light styles. You can have an unlimited number of styles. You can also change an existing style, and all lights using that style are changed automatically. This makes it easy to adjust many lights at once—for example, let's say you're making a dungeon level, and have placed a bunch of lights used for torches. Later you decide to lower the ambient light level, and now the torches seem too bright. But all you have to do is change the style used for the torches, and they're all adjusted without having to change each one. Note the light styles are saved with the level, so each level has its own set of styles.



To change between different styles, or to add and delete lights, use the light pop-up menu. The lights are listed by number (which you can refer to when using functions such as SetLight in code), followed by the tile dimensions of the light, and a symbol that indicates whether the light is a radial or box light.

The selection indicator to the left of the pop-up menu is used to show whether a light is currently selected or not. If you have the Light Mode tool active (see Tools) and have a light currently selected, the selection indicator lights up, otherwise it's grayed out. This helps remind you that you have a light selected, and might want to de-select it before changing the light style. (Otherwise the selected light will change to the newly selected style.)

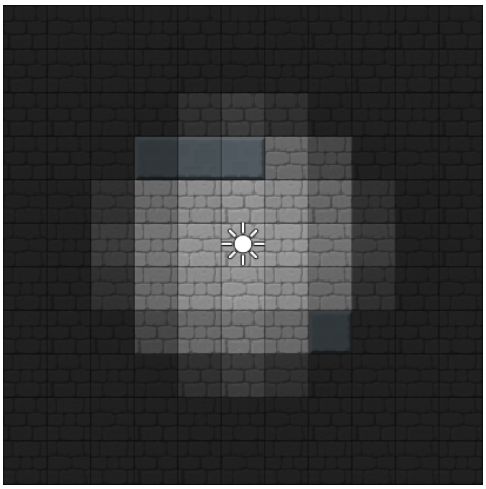
The Make Unique button is grayed out unless you have selected a light using the Light Mode tool. You can use this to automatically create a new light style, based on the existing style that you have selected. This can be useful if you want to change a single light without affecting others. For example, using our torch example, let's say you made the color a nice yellowish red. But you've selected one torch and want to make it bright blue magical torch instead. So click the Make Unique button, and now you have a copy of the torch style, where you can change the color (and other properties) without affecting the rest of the torches. (Note that you can use the new light style to make other lights, of course, so it would no longer actually be unique in that case.)

The layer ambient color is applied to the current layer. By default it's 50% gray. Note that if you make it white, no lights will be visible, since lights are additive and it's not possible to be brighter than white. If the ambient color is set to black, then tiles will be completely black, except when inside a light's radius. You can see the ambient color of the level by using the Light overlay option (see View Options), which is automatically turned on when using the Light Mode tool. You can use transparency in the ambient color, which will work as long as the tile materials you're using support transparency; the default sprite shader does.

The shadow radiosity checkbox indicates whether any lights in the level that cast shadows will use a radiosity effect or not. See below for illustrations of this effect.

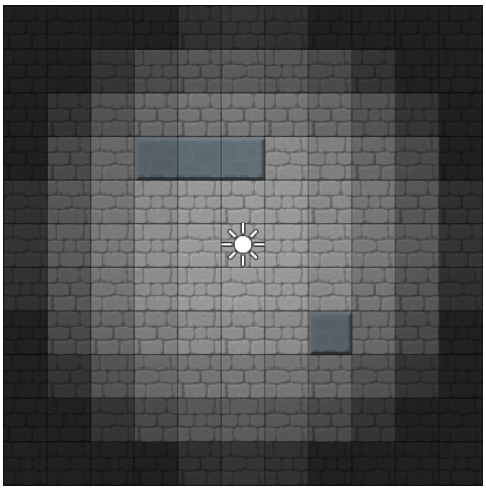
The next line of controls reflects the values of the currently-selected light style. So changing the light style using the pop-up menu will change the values here.

To start with, the color for lights can be changed in the same way as the light ambient color. As with the ambient color, transparency will work as long as the shaders you're using support it. Transparency in lights won't work unless the ambient color also uses transparency, since transparency, like colors, is additive.

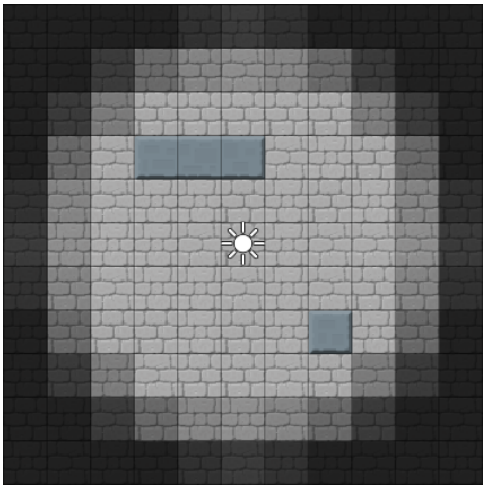


The size of the light can be anything (up to the size of the layer), as long as it's at least 1 x 1. The width and height don't have to be the same. Note that while radial lights can use even values, they appear the same as the next-smallest odd value. That is, a 12x12 radial light will look the same as an 11x11 light, and a 16x10 light would look like a 15x9 light. Box lights can use any value, even or odd.

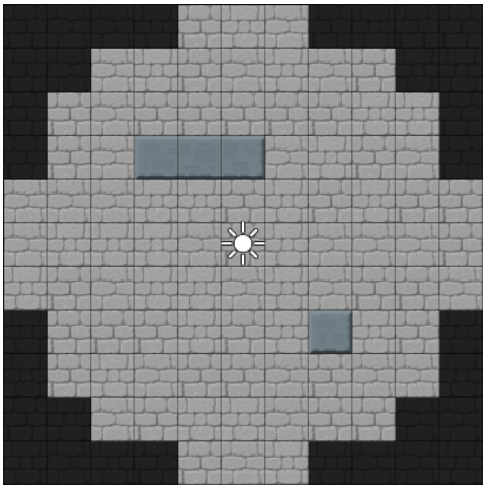
Here we have a 7x7 light on the left (so the radius is 3 tiles, plus one for the center tile), and it's changed to 11x11 on the right:



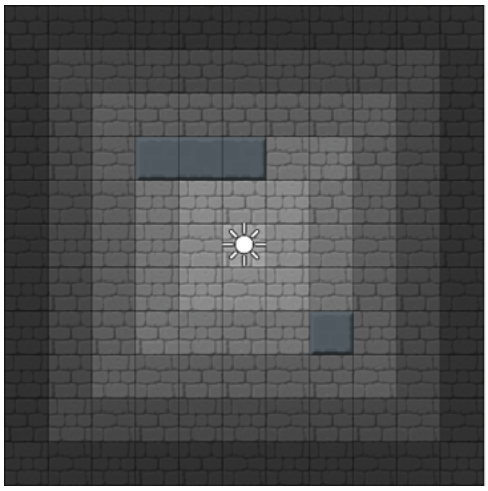
The inner and outer values refer to the intensity of the light at the center and outer edge. The intensity value ranges from 0.0 (no brightness) to 1.0 (full brightness). Since lights are additive, an ambient color of, say, 25% gray means a light with a 0.75 inner value would be 100% bright at the center. The lights pictured above have inner and outer values of 0.75 and 0.1 respectively.

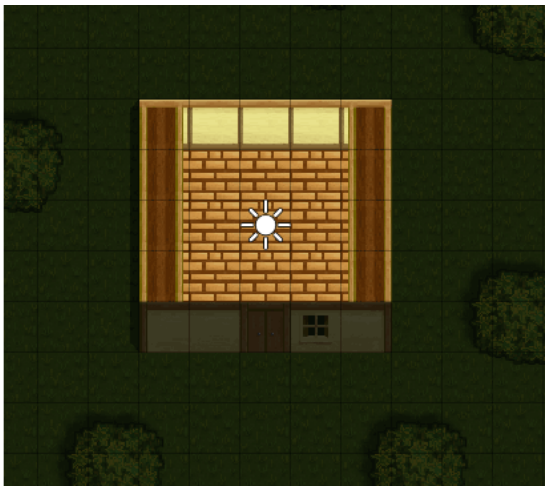


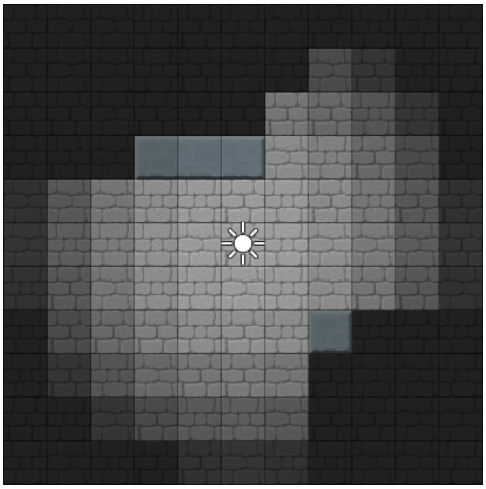
The intensity values aren't limited to 1.0, and can be set higher for certain effects (but not less than 0.0). Changing the light above to have an inner value of 1.5 results in the image on the left; essentially this has a steeper fall-off. Changing both to the same value makes a solid-colored light, as seen on the right. (The inner and outer values can even be inverted, for a negative-like effect.)



Lights can be either radial or box style, as selected by the style pop-up menu (next to the size). Box lights can be used effectively as area lights. For example, a box light with equal inner and outer intensity values can be used to light up interiors in night scenes.

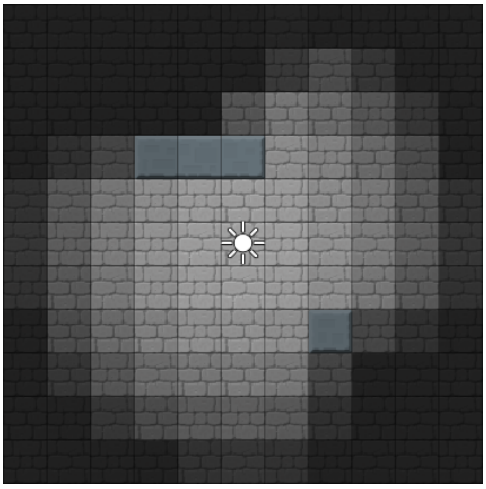






Lights can cast tile-based shadows by selecting the Shadows checkbox. This causes any cells that are marked as collider cells (see How Colliders Work) to block lights. On the left, the light has been set to cast shadows, and the dark cyan tiles have been marked as colliders, so they block any shadow-casting lights.

This is where the shadow radiosity effect comes in. If it's enabled, a simple radiosity effect at the shadow edges is enabled, as shown on the right. This often makes shadow-casting lights look better.



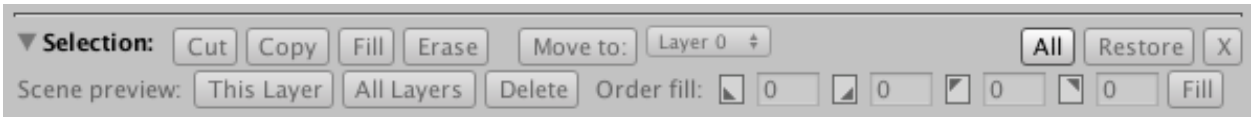
When done setting light values, click the accept button (✓) to apply them to the current light style.

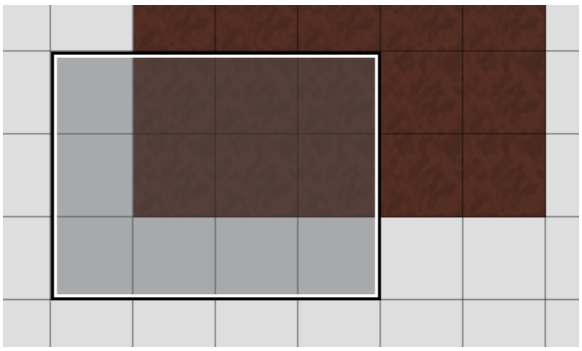
light performance

There can be an unlimited number of lights in a level without affecting performance measurably, even if they all cast shadows. This is achieved by essentially baking the lights into tile colors. However, lights can still be moved dynamically in code. (See the SpriteTile Reference Guide for details of light functions such as SetLight, DeleteLight, MoveLight, etc.) The baking is very fast, and moving a light only needs to re-bake that particular light, plus the relevant sections of any other lights it might be overlapping. Depending on various factors, mostly CPU speed, light size, and shadow-casting, dozens or even hundreds of lights can be moved per frame. Moving lights is fastest when the size is relatively small and they don't cast shadows. If they do cast shadows, shadow radiosity slows the speed down a little.

selection controls

When open, the selection section displays all the controls. When closed, it only displays the copy, restore, and deselect buttons.

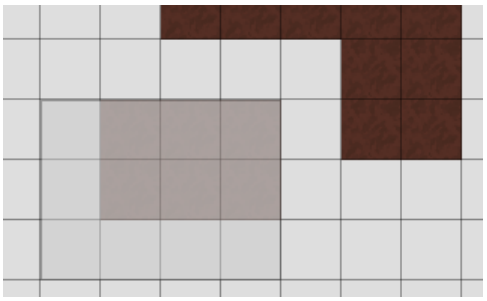




Aside from clicking on individual cells, you can also select an area. Click and drag with the right mouse button down to select an area, which is indicated by a black and white outline with a faded interior. (Mac users should make sure that right-clicking is enabled in the macOS system preferences; alternately, Command-clicking can be used instead.) In the illustration to the left, the selected area is shown as a 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 in the selection control strip (on the far right).

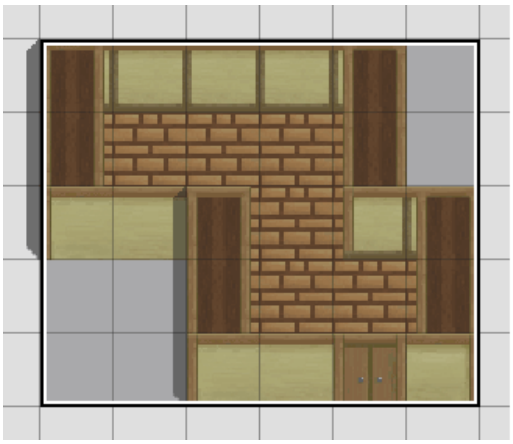
If there's no selection, the buttons are grayed out, aside from the "All" button.

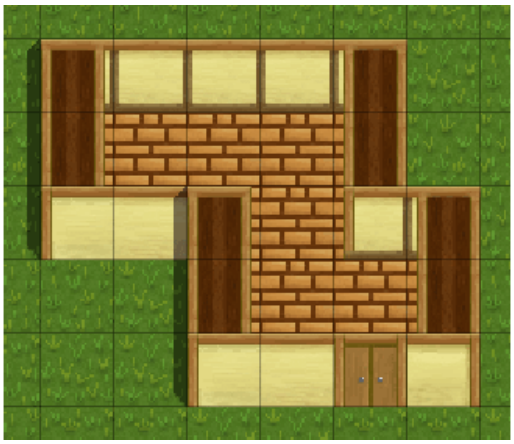


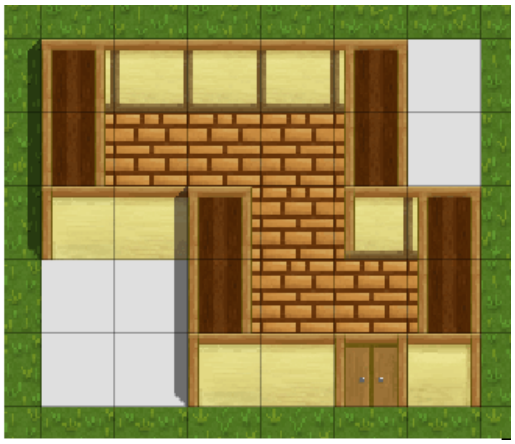
Cut: removes all tiles within the selection box and puts them into the copy buffer. The copy buffer appears as a faded-out representation of the tiles in the selection, which you can move around freely, then paste into the level by left-clicking. You can delete with the copy buffer by middle-clicking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

Note that when pasting a group, empty tiles are normally not included, so they won't overwrite any existing tiles. By holding down **Shift**, however, you can force empty tiles to be included when pasting. In the illustration below, a group with empty tiles is selected, and first pasted normally onto some grass tiles, then pasted again while holding Shift.







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.

Move To: moves the tiles in the selection to the layer indicated by the drop-down selection menu next to this button. This is similar to cutting the selection, then switching to another layer and pasting it, but is faster and easier. It only works with layers that have the same dimensions—if you try to move a selection to an unsuitable layer, you'll get a dialog box informing you of this. If you click "Move To" when the indicated layer is the same as the current layer, nothing will happen. (Similarly, the button is inactive if you only have one layer in your level, since there would be nowhere to move to.) If you hold down **Shift** while clicking "Move To", empty tiles are included when moving, just like holding down Shift while pasting a group.

One use for "Move To" is to make it easy to merge layers—say you have a layer with a background, and another layer with a foreground. You don't actually need them to be separate layers in the end, but it's convenient to work like this...when you work on the foreground layer, this allows you to add and delete tiles without disturbing the background. (You can also turn on "Show preceding" to see the background while working on the foreground.) When done, select the entire foreground layer and use "Move To" to move it to the background layer. You can then delete the foreground layer, and are left with a single combined layer.

Note that using "Move To" requires two undo operations to completely undo the action: one for the source layer and one for the destination layer.

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

All: selects all cells in the current layer. You can also use **Alt-A** to do the same thing.

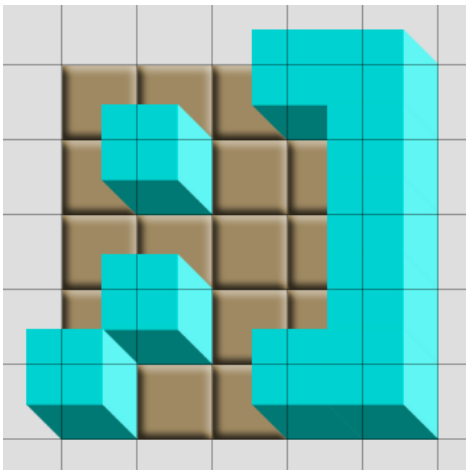
X: this removes the selection box (but doesn't delete tiles). It will also deactivate an active copy buffer. You can also do this by pressing the **Escape** key or right-clicking on one spot in the level.

Scene preview: a SpriteTile level can contain far more tiles than the Unity editor would be able to display at once without crashing, which is why SpriteTile normally operates in play mode and uses its own editor window. But you can use these buttons if you need to see what a level looks like in the Unity scene view.

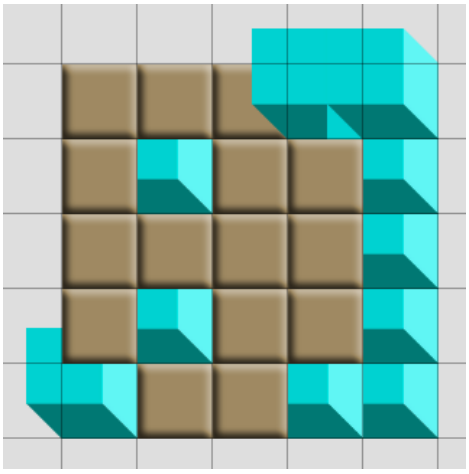
> **This Layer:** shows the current layer in the editor, using the contents of the selection box. (Use the "All" button or Alt-A to select the entire layer.) In order to prevent the Unity editor from getting bogged down by too many objects, the preview is limited to 10,000 tiles. This can be any configuration, 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.

Order fill: allows you to set a range of order-in-layer values for the selection, as defined by the lower-left, lower-right, upper-left, and upper-right corners. See below for more details.

order fill

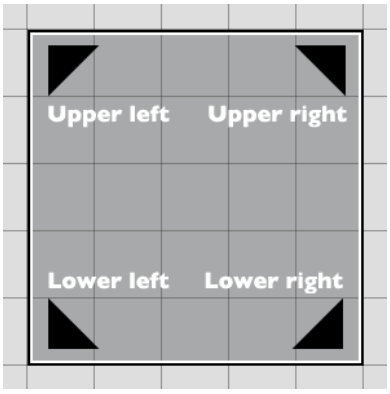


If you have isometric tiles or similar effects, it's necessary to use the order-in-layer value for tiles to ensure they draw in the proper order. While it's easy enough to change the order value for each tile, it would be even easier to fill an entire area with a range of order values, depending on the effect you want to achieve. For example, let's take a look at these overlapping isometric tiles:

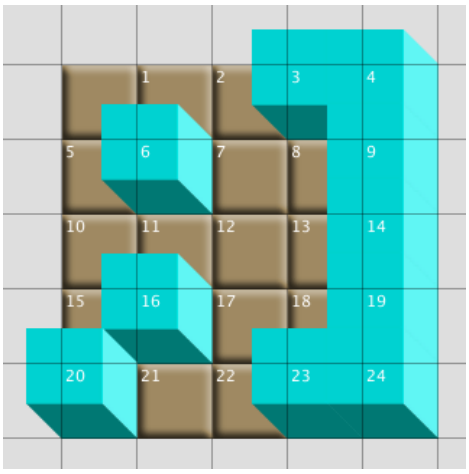


Drawing them in the TileEditor without order values may appear correct, but since overlapping tiles with the same order values have an undefined drawing order, what you get at runtime is likely to be something like this:

To fix this, you can manually change the order values, but that can be tedious with large levels, and sometimes mistakes aren't obvious even with the order overlay active (see View Options). It would be more convenient if we could define the order values for the entire level, and not have to worry about it.



That's where the order fill button comes in handy. To set it up, we need to define the values in the four corners of the selection, and then the **Fill** button will calculate the proper order value for each cell. The corners of the selection are lower left, lower right, upper left, and upper right. Note that the order values can range from a minimum of -32768 to a maximum of 32767, which is a limit imposed by Unity.



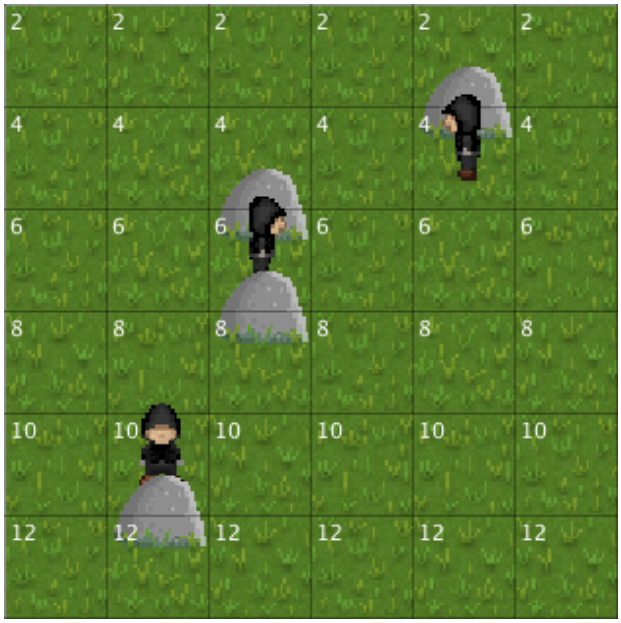
What values you should use depends on what you're doing exactly. In this particular case, we want the bottom row of tiles to draw on top of the next row, and so on in a descending order as we go upwards. Likewise, the rightmost column should draw on top of the next column to the left, and so on in a descending order as we go leftwards. So, for this 5x5 grid, we need 25 values, starting at 24 in the bottom-right and going to 0 in the upper-left. Therefore the values for the corners would be 20 (lower left), 24 (lower right), 0 (upper left), and 4 (upper right). First select the desired area, then enter the values, and finally click the Fill button. When you click the button, the view overlay mode switches to Order, if it isn't active already, so you can see the results immediately.

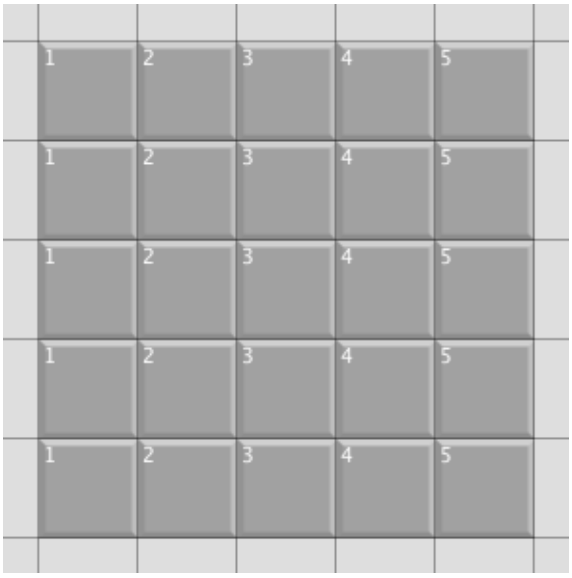
Once you have the order values set up, you can turn the order overlay off if desired. You can then create the level without having to manually set the order value for each tile, and the order values for all the cells will always be correct. (Just make sure the default order value for all tiles is set to 0, so they don't overwrite the existing order values in the level. See Tile Defaults.)

Let's take a look at another example. This time we'll have a typical top-down RPG perspective that's somewhat angled. We'll have a field of grass with some rocks in it, and the rocks are made with a double-height tile that overlaps the tile above it. We also have a character that can walk around in this field. So we have two things we want to do: make sure the rocks always draw on top of the grass, and make sure the character is drawn over and under the rocks as appropriate.

The simplest thing to do is to have each row increase its order value by two. In this 6x6 example, the four corner values will be 12, 12, 2, and 2.

The character will be programmed so that as he goes down the rows, his order value also increases by two, but while we're using even values for the level, we'll use odd values for the character. On the top row, his value will be 3, increasing to 5 for the next row, and so on down to 13 for the last row. In this example, the top-most character has an order value of 5, so he draws on top of the rock, which has an order value of 4 (which, in turn, draws on top of the grass tile above, which has a value of 2). The middle character's value is 7, so he draws on top of the rock above (6) but behind the rock below (8). And the bottom-most character's value is 11, so he draws behind the rock (12).



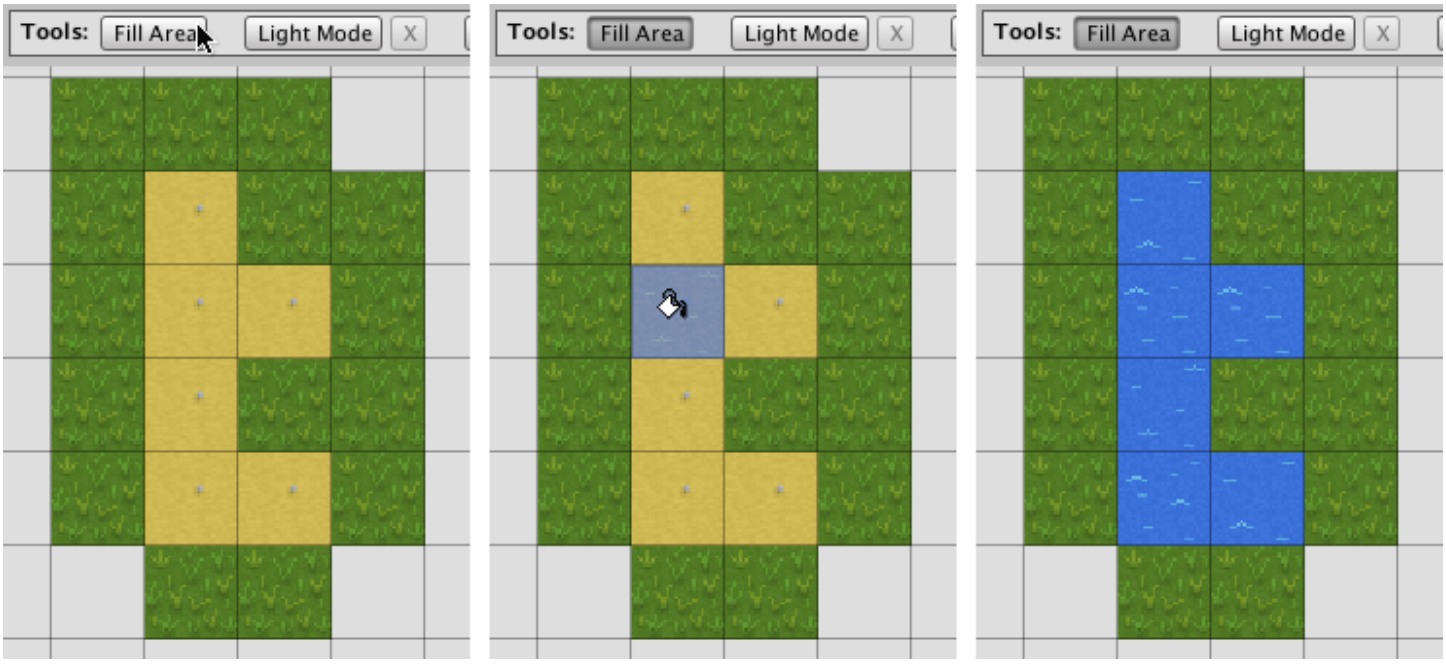


So with the ability to define the order value for each corner of a selection, you can automatically set the order values for all cells, depending on what you need for any situation. For one last example, the values here have been set to 1, 5, 1, and 5, which makes the order start at 1 for the left column and increase to 5 for the right column.

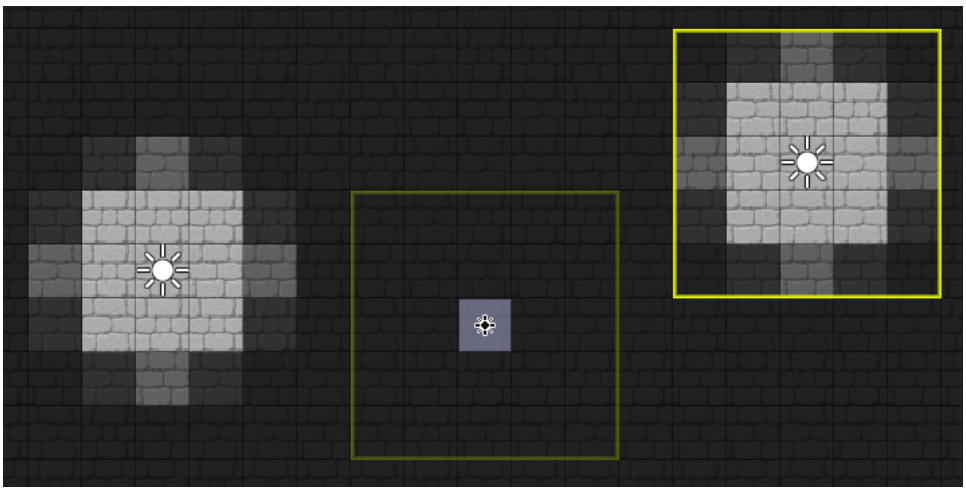
tools

These are tools that allow you to perform various actions in the map area.

Fill Area: clicking this button will toggle Fill Area mode, which flood-fills an area with the currently selected tile. When Fill Area mode is on, the cursor will change to a paint bucket icon in the map view, and clicking on a cell will fill all contiguous cells of the same tile with the new tile. If you have a random group active (see Random Groups), then each fill tile is randomly chosen from the tiles in the group. Click the Fill Area button again to return to normal tile drawing. You can also use the keyboard shortcut **Shift-F** to toggle Fill Area mode.



Light Mode: when selected, this activates light mode, which allows placing, moving, and deleting lights (see Light Controls). Selecting Light Mode automatically turns on the Light overlay mode, which shows lights and tile colors, and the cursor changes to a light icon. As long as Light Mode is active, left-clicking will place a light, and dragging a placed light with the mouse will move it to a new location. (Dragging a light onto an existing light will remove the original light.) Middle-clicking or control-clicking a light will delete it.



If a light is selected, it's shown with a yellow outline. When moving the light cursor around the map, a faint yellow outline shows the size of the current light style, so you can get an idea what a light will look like when placed. Note that regular tiles can't be drawn in light mode.

X: when in light mode, this deselects a light that's been clicked on. You might use this if, for example, you want to change to a different light style without affecting the selected light. It's grayed out if a light hasn't been selected. You can also deselect lights by using the **Esc** key.

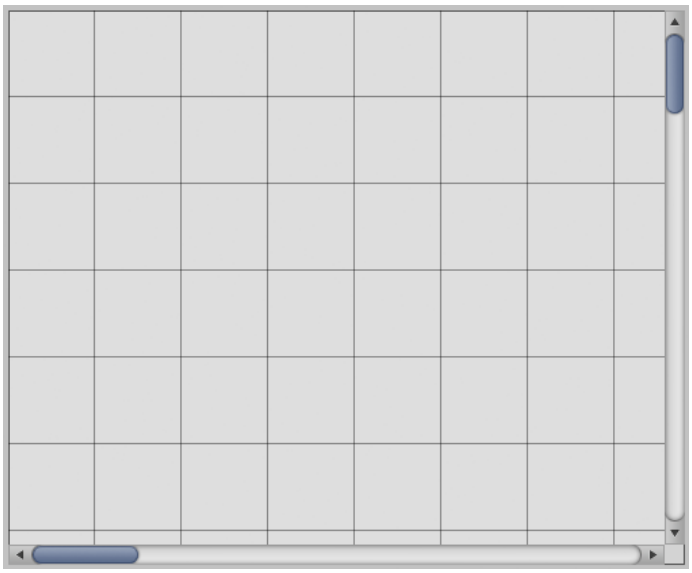
Pick Tile: use this to pick any tile in the map window, which then becomes the current selected tile. This can be a convenient way of quickly selecting tiles that already exist in the map, so you don't have to scroll through the list of tiles. You can also use the **P** key to pick tiles instead of using the button. Note that using the P key immediately picks the tile under the cursor rather than toggling tile picking mode.

Show editor tooltips: when this is active, tooltips for most controls will be shown when the mouse pointer is hovered over them. This can include usage hints, and keyboard shortcuts for controls that have them. (See also the TileEditor Keyboard Shortcuts document, which is a handy reference for all shortcuts in one place.)

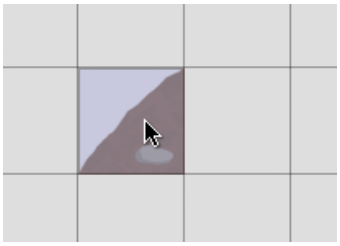
Undo: undoes the last action you did when performing actions in the map window. There's no hard limit to the number of undo steps, though once the undo buffer reaches approximately 2GB, the oldest steps will be removed. Certain actions will reset the undo buffer, such as creating a new level or deleting 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.

Redo: the opposite of undo, but otherwise follows the same rules about automatic layer switching when appropriate. The button is grayed out if no redos are available. After undoing some steps, performing actions other than redo will remove any undo steps that might have existed after the current one. In other words, the undo history is linear. You can also use **Alt-Shift-Z** instead of clicking the button.



map window



This is where you actually create your level. When you move the mouse pointer over the map window, the cell under the pointer will be highlighted with a faded-out version of the currently-selected tile:

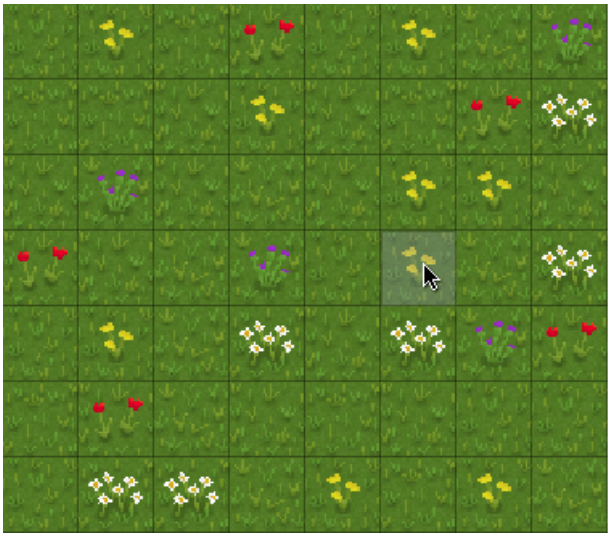
Clicking on the highlighted cell will put the currently-selected tile in that cell. You can also click-and-drag to place multiple tiles.

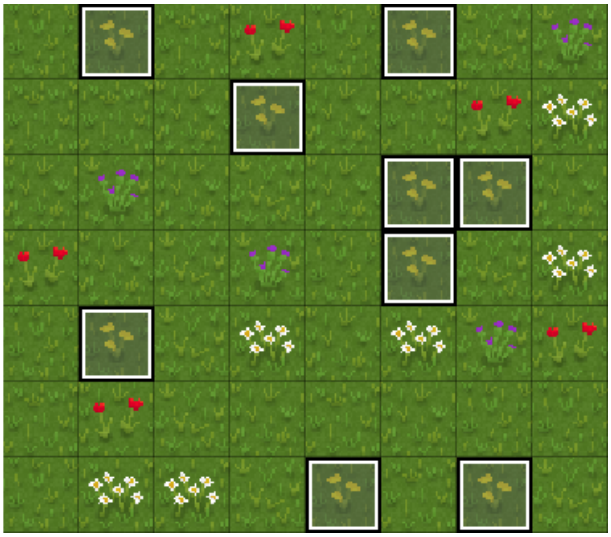
Erase tiles by middle-clicking or control+left-clicking (this works with click-and-drag as well). You can also use the **Delete** key.

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. Zoom by using alt+scroll wheel, or by using the level preview size slider.

multi-selection

Aside from dragging out a selection box (see Selection Controls), another way to select multiple tiles is by double-clicking a cell in the Level view. This causes all tiles of that type in the current layer to be selected. In this example, double-clicking on the yellow flower selects all yellow flower tiles in the layer (whether currently visible or not):

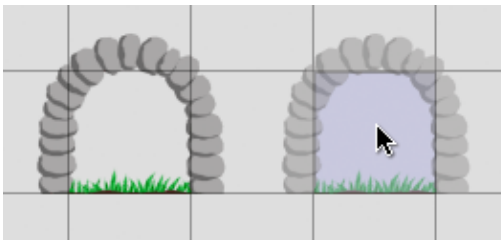




Note that this sort of multi-selection can't do all the things that drag-selecting does. Cut, copy, editor preview, and making groups aren't available. But the selection can use fill, delete, and move to. Also, alt-clicking will enable you to set trigger, order-in-layer, and rotation values for all the selected tiles at once. (See Cell Properties for details about alt-clicking tiles.) Keyboard shortcuts for cycling order etc. also work.

overlapping tiles and anchor points

If you have a tile selected that doesn't fit exactly into a cell, then the preview will extend beyond the bounds of the cell, with the cell itself highlighted in light blue. 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. You should, however, make sure to set the order-in-layer numbers appropriately; tiles drawn on top should have higher numbers. If overlapping tiles have the same order number, drawing order is undefined. That is to say, they may sometimes look right in the editor, but will likely not look right at runtime. You can turn on the order overlay (see View Options) to make this process easier.

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.

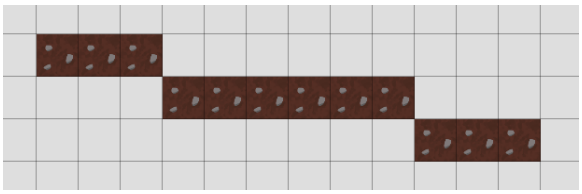
For Single sprites, go to the texture import settings for the sprite in Unity and change the Pivot from Center to something else, most likely Custom. This gate tile, for example, is twice the size of a normal tile and has the custom pivot set to X = 0.5, Y = 0.25, so that it lines up correctly. When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

drawing lines

If you want to quickly draw lines of tiles, you can click at the beginning of the line, then hold down **Shift** and click where you want the end to be. A line of tiles will be automatically drawn between the two points. Also, you can also hold down Shift while continuously drawing, which will prevent gaps between tiles that can happen when moving the mouse quickly.



bookmarks

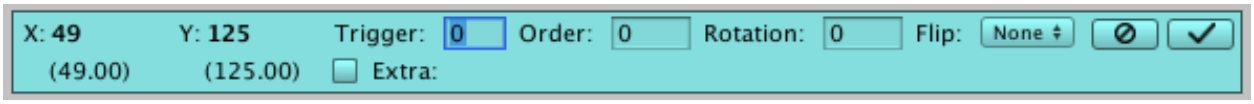
In large levels, you may find it useful to set bookmarks, to make it easy to quickly move the view around to specific locations. You can do this by using function keys: the shift key plus a function key will set a bookmark to the current view location and zoom level. After you've done that, the function key will recall that location. Using alt plus shift and a function key will delete that bookmark.

So, for example, pressing Shift + F1 will set a bookmark for the current view location in the level, and F1 will move the view to that bookmark. Alt + Shift + F1 would delete the bookmark. The bookmarks are saved along with the level, so you can continue to use them later, after re-loading the level.

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. You can change the trigger, order in layer, rotation, flip, and some extra properties by alt-clicking on a particular cell, in which case the cell properties are highlighted and the numbers become editable. If you have multiple tiles selected, then the Edit button can be used, though alt-clicking will also work.

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



X and Y: the cell coordinates. The world coordinates are displayed in parentheses below 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.

When dragging out a selection box with the right mouse button, the X and Y coords are replaced with W and H, which means width and height, so you can see the dimensions of the selection box.

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: 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 ⅕ (0.2) of a degree. So 15.1 would become 15.0, and 15.25 would become 15.2.

Flip: a map cell can be flipped on the X axis, the Y axis, both (XY), or none.

⚠ Note that when SpriteTile is running in play mode, flipping on the X and Y axes actually flips the sprite by rotating 180° on the Y and X axes, rather than scaling by -1, since scaling breaks sprite batching and rotating doesn't. This has no effect on normal usage, but if you use a custom material with the Tile.SetTileMaterial function, make sure the material uses a shader that doesn't do backside culling. If the shader does backside culling, flipped sprites will be invisible.

Extra: if this checkbox is selected, then several additional properties can be edited. These are: number (a float), label (a string), and GO (a GameObject prefab). The number and label are useful for cases where the standard trigger property is not enough, and can be used with the Tile.GetProperty function in code.

The GameObject must be a prefab that exists in the project, and any tiles that contain a GameObject extra property will cause the associated prefab to be instantiated when the level is loaded. This way, non-tile objects can be incorporated into levels easily, without having to manually position them in the scene view. If the TileEditor is docked, prefabs can be dragged onto the GameObject slot; otherwise you can click the small target icon and select prefabs from a list of the ones that exist in your project. To remove a GameObject, select the slot and hit the delete key, or select “None” from the prefab list. See also the Extra overlay feature.

multi-tile editing

Note that if you have multiple cells selected, either by drawing a selection box or by double-clicking to select all tiles of that type, the info here will be applied to all the cells in the selection when you’re done. You can use this to do things like set trigger areas, change the order-in-layer numbers for multiple cells, etc., without having to edit every cell individually. Additionally, any cells in the multiple selection which have different values from the others will have the relevant property displayed as a dash, just like multi-editing in the Unity editor.

When this happens, applying properties will leave the dashed properties alone. So, for example, if you had a number of tiles with various order-in-layer values but you want to apply the same trigger value to them all, multi-editing will allow you to do this easily without overwriting the existing order-in-layer values.

Trigger, order, and extra properties for all tiles may be viewed as a map overlays by using the appropriate overlay modes (see View Options).

shortcuts

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.

As a shortcut for editing **Flip**, you can use the **X** key to flip the cell the mouse pointer is currently over on the X axis, and the **Y** key to flip on the Y axis.

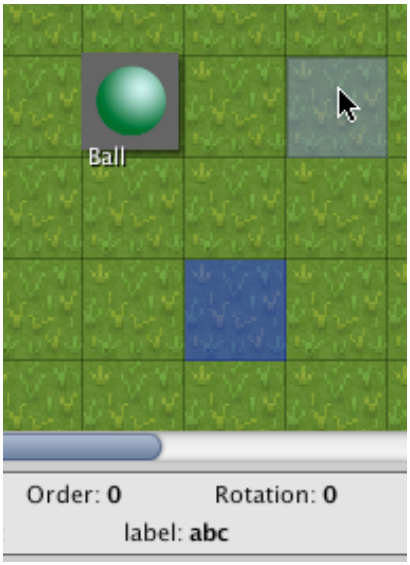
view options

These control how the level is shown in the editor. They're for informational purposes 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.

Light: when active, tile colors are shown, which makes lights and the layer ambient color visible. The light overlay is automatically enabled if Light Mode is activated (see Tools).

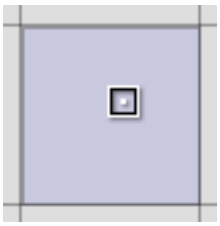
Extra: any cells which have an extra property (see Cell Properties) are shown when this overlay is active (you can also use the **E** key to toggle it). Cells with numbers or labels are shown with a tinted overlay (blue by default; see below for tint information). Cells with GameObject properties are shown with a thumbnail image of the associated prefab. The image on the right shows three cells with an extra property. The mouse is hovering over a cell with a label (in this case, "abc"), which can be seen in the property information box. One cell has a GameObject property with a prefab called Ball, where the prefab name is shown below the thumbnail. Note that the prefab is not shown actual size in the level.



It's possible to add a GameObject property to a level, then delete the prefab from the project. In this case, the GameObject will be shown with a "missing prefab" icon. This won't cause any errors, but it's generally best to remove the GameObject property or replace it with an existing object.

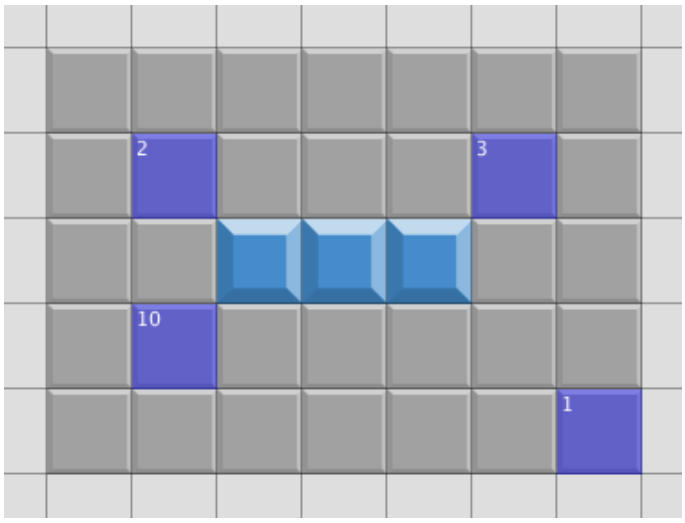
Order: this shows any non-zero order-in-layer numbers on cells, which is especially useful for making sure overlapping tiles have order values set correctly. You can continue drawing tiles normally while the Orders overlay is active. You can also use the **O** key to toggle this.

Collider: this shows any cells that are an active collider cell by using a tinted overlay. 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 used. You can also press the **C** key to toggle the colliders overlay on and off.



Note that when the Collider 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 or control-click to delete them. The mouse cursor changes to a special collider cursor (a small black and white square) to make it clear when you're drawing collider cells. 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 Collider overlay on, drawing operations only affect the collider cells, and leave the tiles themselves alone.

Trigger: similar to Collider, 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. Shown on the right, we have four triggers, labeled 2, 3, 10, and 1:



Note that some overlays are mutually exclusive. That is, turning one on will automatically turn others off. For example, of the order, collider, and trigger overlays, only one can be active at a time. Also, the collider and trigger overlays will disable the light overlay and vice versa.

Tint: if you'd rather have some other color instead of blue, change it here to whatever you like. You can also change the transparency (by default it's 40% transparent). The tint color is saved along with the level, so different levels can have different tint colors.

Zoom: changes the size of tiles in the map window. Holding down the **Alt** key while using the mouse scroll wheel will also work; when zooming in like this, the view will zoom toward the mouse pointer. •

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.

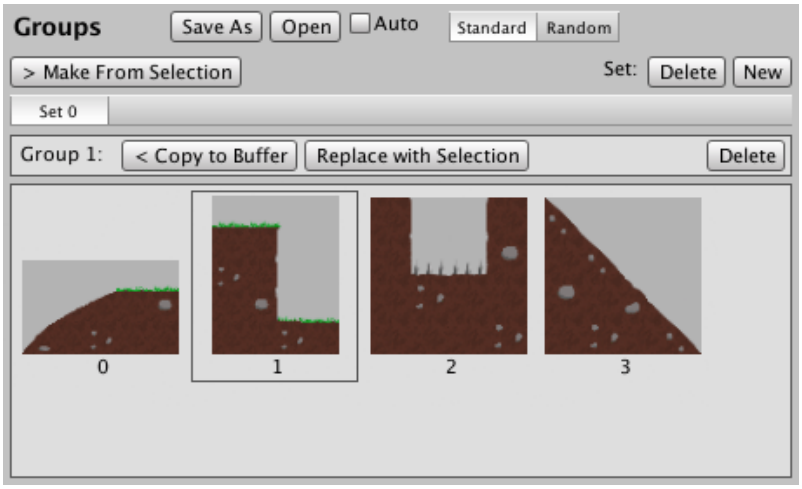
Auto: if checked, the most recently saved or opened group file is automatically loaded every time the TileEditor window is opened.

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.

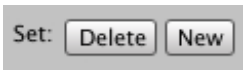
Standard

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 map 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.

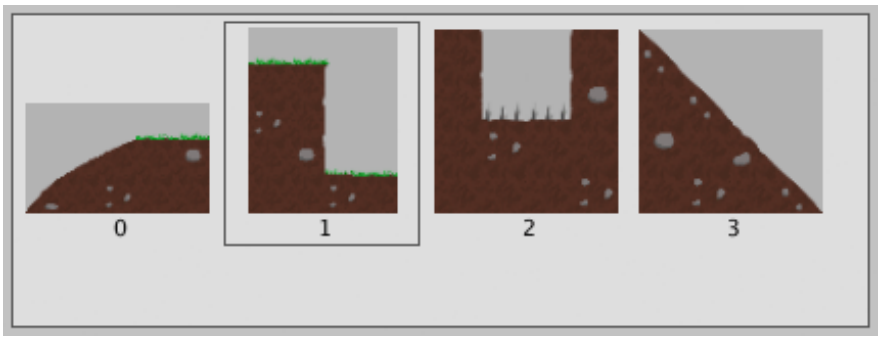
group controls

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 map 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.

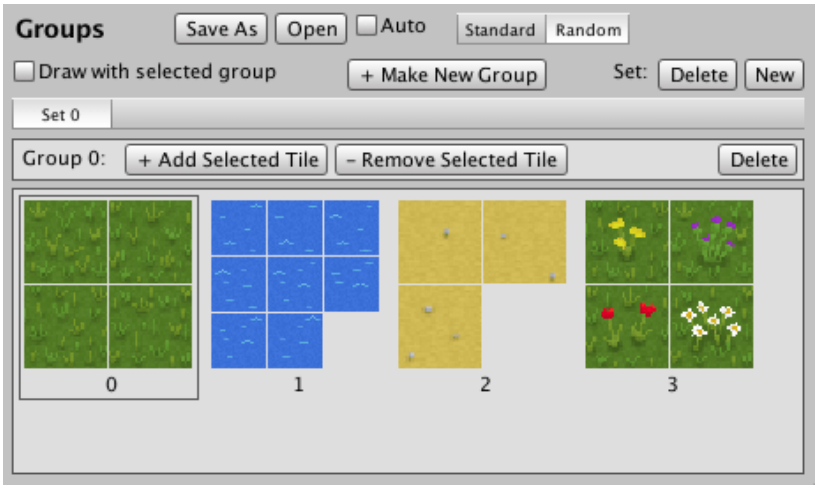


group window

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.

If you want to use Standard groups at runtime, you can use the Tile.LoadGroups and Tile.CopyGroupToPosition functions (see the SpriteTile Reference Guide). •

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. As with Standard groups, you can have multiple sets, which you can manage with the Set buttons.

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, and if you use the Fill Area button, the filled tiles will be randomly selected too. Remember to uncheck this when you want to draw with tiles normally! When random drawing is active, the preview tile under the mouse cursor will be the first tile in the random group, rather the than currently-selected tile, to help remind you.

make new group

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

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.

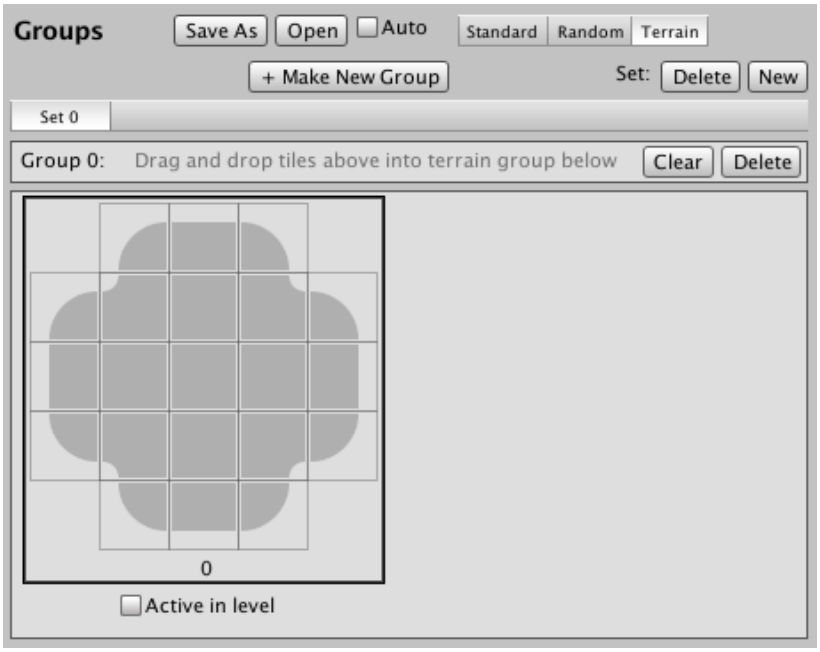
If you want to use Random groups at runtime, you can use the Tile.LoadGroups and Tile.UseRandomGroup functions (see the SpriteTile Reference Guide). •

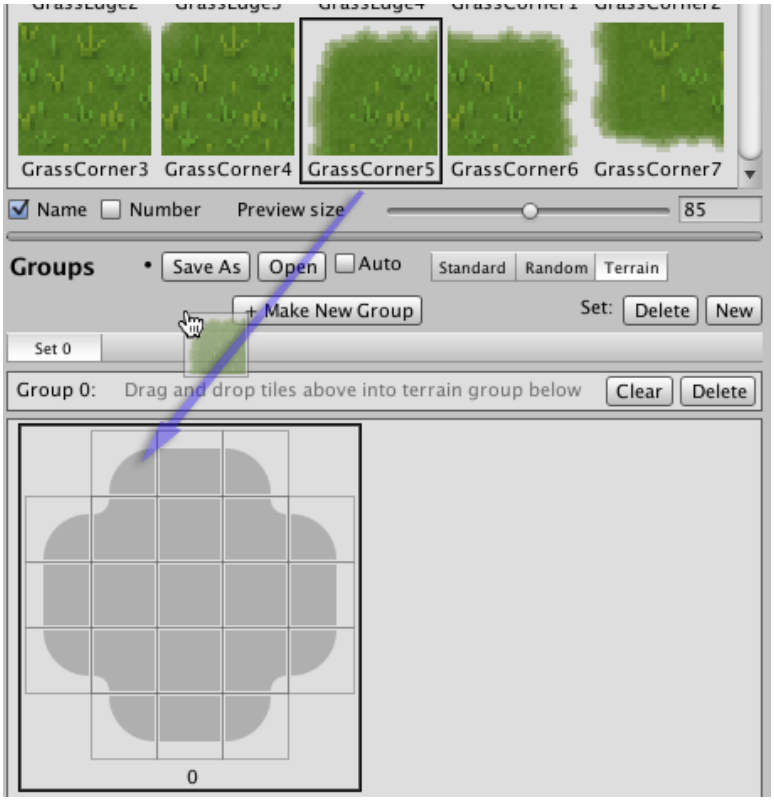
Terrain

You can use **Terrain** groups to quickly draw shapes made of related groups of tiles, such as paths, cliffs, shores, walls, and so on. As with Standard groups, you can have multiple sets, which you can manage with the Set buttons.

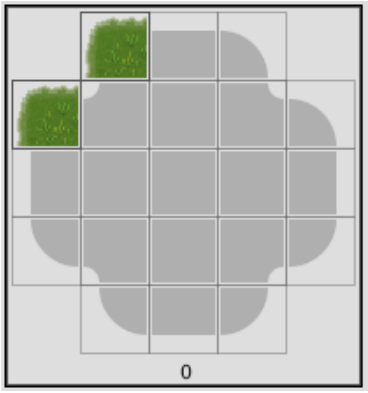
make new group

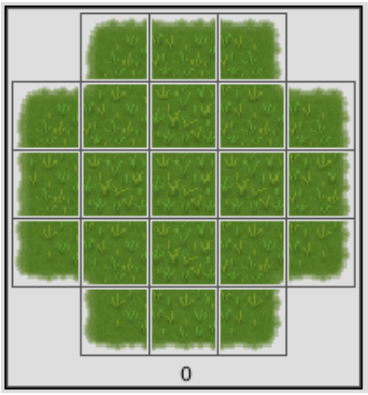
Make New Group: clicking this will create a new Terrain group. It will initially show a grid of the terrain shapes, which you can fill by dragging tiles from the Tiles section into the appropriate slots in the Terrain group:





Keep dragging tiles over until the group is filled. There are 13 tile types in a Terrain group: four outer corners, four inner corners, four edges, and one center. (It will actually still work to some extent if some tiles are left blank, but ideally they should all be used.) Any cells in the group that use the same tile will be automatically filled as soon as one is completed, so even though there are 21 cells, you only need to drag 13 tiles.





group controls

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

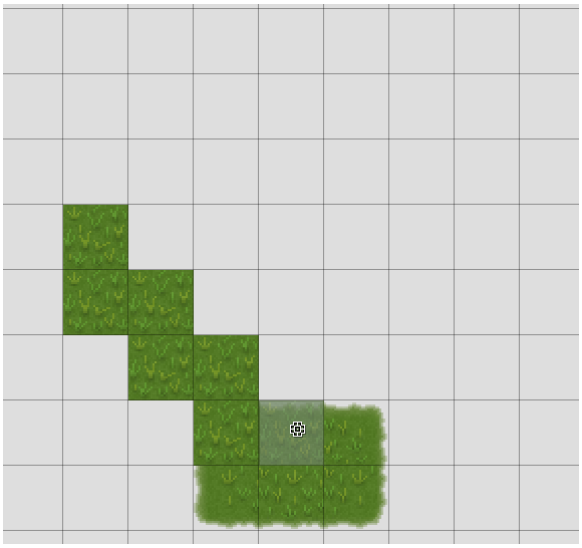
Clear: if you want to erase all the tiles in the selected group and start over, click this.

Delete: this removes the selected group from the set altogether. For both buttons, if the group contains any tiles, you'll get a dialog asking you to confirm the operation.

drawing with terrain groups

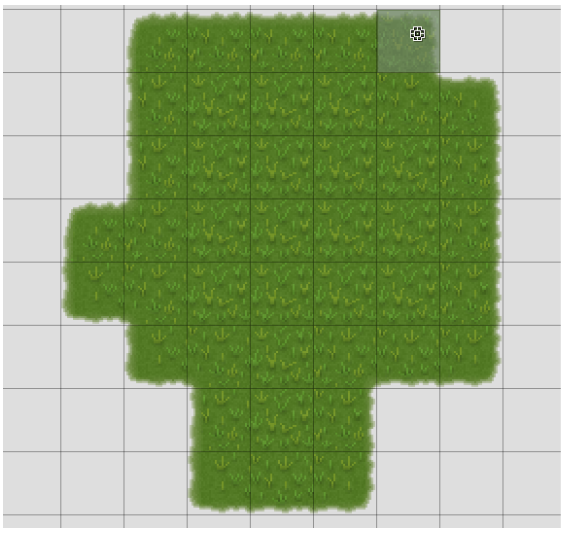
When you have a Terrain group ready and want to use it in the level, click the "Active in level" checkbox below the group. You can have any number of Terrain groups active at once. Now, if you have a tile selected in the Tiles section that belongs to an active Terrain group, you will draw with that group. (You can also double-click the terrain group to automatically select the middle tile of the group and turn "Active in level" on if it isn't already.) The mouse cursor will change to a terrain cursor to make it clear you're drawing with an active terrain group.

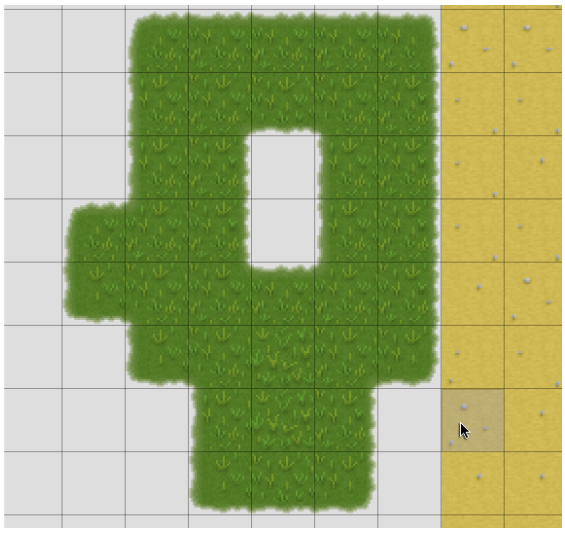
Once you start drawing, the tile in the highlighted cell (and all surrounding tiles) will automatically change to the appropriate tile, as soon as there are enough contiguous tiles to make a determination. This generally requires at least two tiles next to each other, horizontally and vertically.



You can also delete Terrain group tiles, or draw over them with other tiles not in the group, and as long as the Terrain group is active, the appropriate tiles will be adjusted. If you deactivate the Terrain group, then you can go back to drawing one tile at a time as usual, in case you want to have full manual control.

If you want to use Terrain groups at runtime, you can use the Tile.LoadGroups and Tile.UseTerrainGroup functions (see the SpriteTile Reference Guide). •



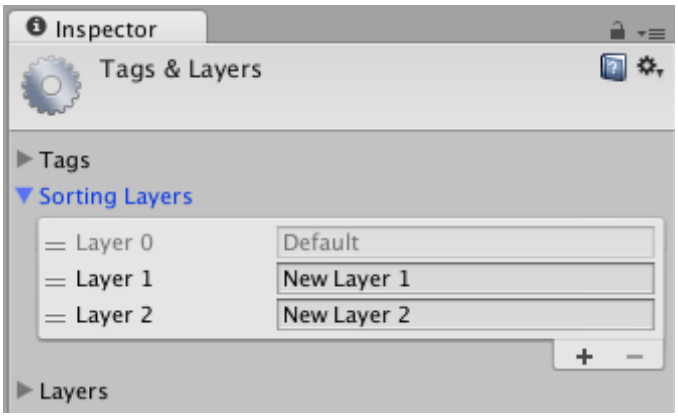


Each level can have an unlimited number of layers. You can add layers by using the layer drop-down menu (see Layer Controls).

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 really matters for perspective cameras. Orthographic cameras ignore it.

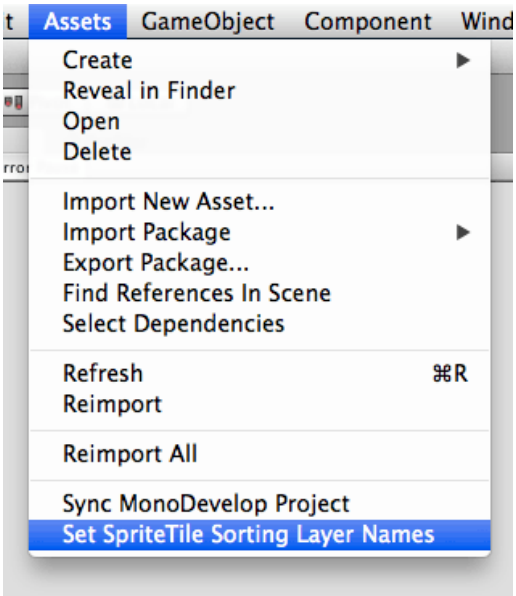
One common usage of layers would be for parallax scrolling using a perspective camera, where each layer is at a different z distance from the camera. (See the Acorn Antics demo game for an example of this.) But layers can be used for other purposes as well, such as a top-down RPG where layers are used for building roofs and other effects. (See the Gem Hunt demo scene.)

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**). Unfortunately there's no way to add layers automatically through scripting, so this is something you'll have to take care of yourself; click the + button to add layers. So if you have three layers, for example, you should have at least three entries in the Sorting Layers list, as shown on the right. You can name the layers anything you like. Attempting to load a level that contains more layers than you have sorting layers in your project will result in an error.

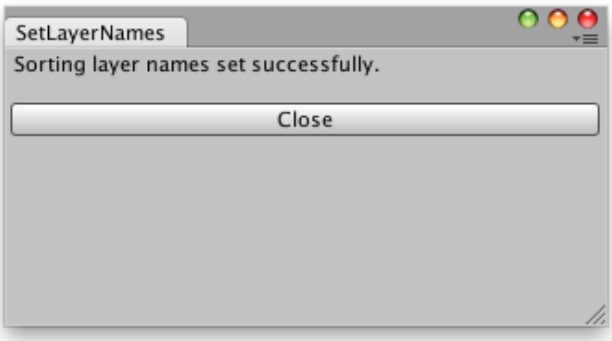


Note that the layers should be ideally be in numerical order, as shown above. It's possible to rearrange the sorting layers by dragging them in the Tags & Layers inspector, but this may cause undesired results. However, you can manually assign different sorting layers to your SpriteTile layers in the TileEditor (see Layer Controls).

Also note that all sorting layers must have unique names.



When using more than one layer, there's one final step: setting the layer names. SpriteTile needs to be able to read the sorting layer names, but this can't be done at runtime, so it has to be done in the editor. Fortunately this is really simple: just select the "**Assets -> Set SpriteTile Sorting Layer Names**" menu item. That's it, so you can close the resulting window which tells you that the action was successful. You'll need to select this menu item any time you add or rename layers in the Tags & Layers inspector. This can be done at any time—if the TileEditor window is open, it will automatically pick up the changes for use in the Sorting pop-up menu. •



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.

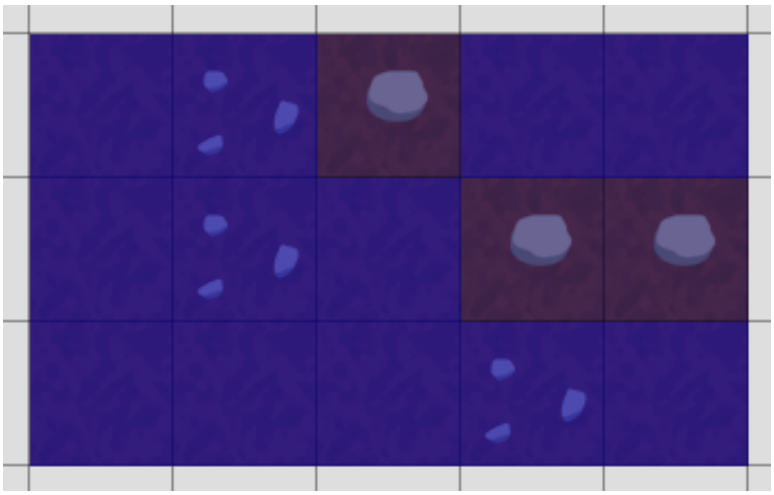
For example, say you're making a top-down RPG, and you want wall tiles to be considered as blocking cells, so players can't go through them. When the player moves, you can read collider information from the appropriate cell in your level, and prevent movement if the player tries to enter a wall tile. 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:



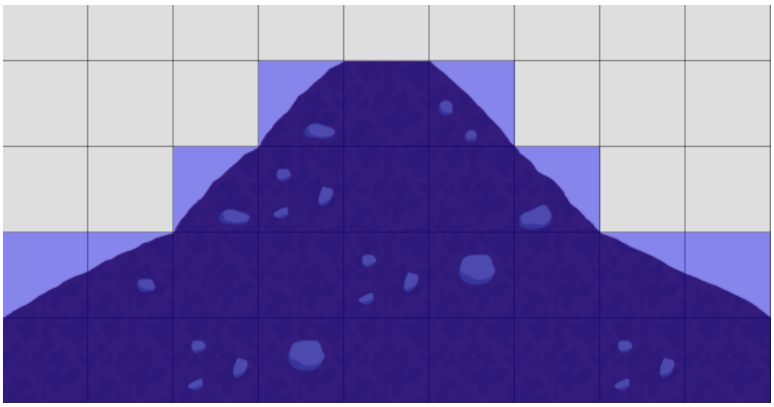
You can see here that there are three tiles being used, specifically the Dirt1, Dirt2, and Dirt3 tiles from the Acorn Antics demo game. The collider overlay has been turned on. The Dirt1 and Dirt3 tiles have the "Use physics collider" checkbox **on**. The Dirt2 tile has that checkbox **off**. All of 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.

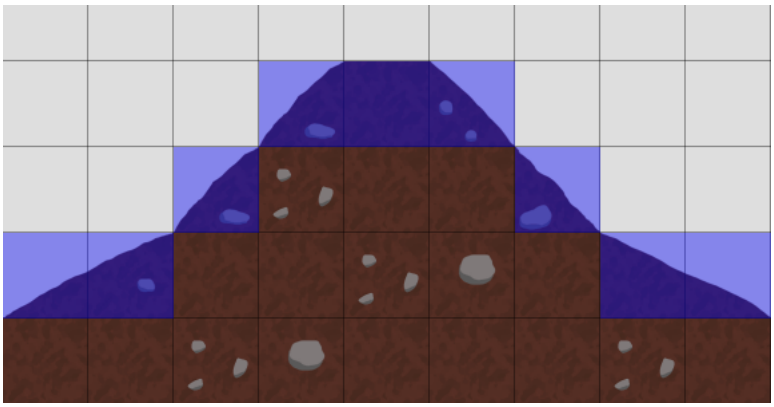
Note that since polygon colliders are purely 2D and are not affected by depth, any active colliders in multiple layers all behave as if they're on the same layer.

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. •

• coding basics

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. For a complete list of all SpriteTile functions, see the **SpriteTile Reference Guide** document. All functions are in the Tile class. Unless otherwise noted, all code examples 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 reference guide. You can also use GetLevelBytes to 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);
     }
```

You can call SetCamera again with a different camera if needed. (Such as if you load a new Unity scene and the old camera was destroyed.)

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 if you have large levels. Also, changing the screen resolution at runtime, or changing the screen 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.

Int2 includes Int2.zero and Int2.one, which equal Int2(0, 0) and Int2(1, 1) respectively. You can also use Int2.up, Int2.down, Int2.left, and Int2.right, which equal Int2(0, 1), Int2(0, -1), Int2(-1, 0), and Int2(1, 0) respectively. Additionally, upLeft, upRight, downLeft, and downRight can be useful in certain cases. These are Int2(-1, 1), Int2(1, 1), Int2(-1, -1) and Int2(1, -1).

```
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();
}
```

```
     // C#
     public Vector2 coord;
     private Int2 _coord;

     void Start () {
          _coord = new Int2(coord);
          Vector2 v2 = _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);
```

TileInfo is actually similar to 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);
```

You can use TileInfo.empty to refer to empty tiles, which is the equivalent of TileInfo(0, -1).

```
var mapPosition = new Int2(25, 40);
var thisTile = Tile.GetTile (mapPosition);
if (thisTile == TileInfo.empty) {
    Debug.Log ("Nothing at position " + mapPosition);
}
```

TileInfo variables can be added, subtracted, and compared, but not multiplied or divided.

```
var t1 = new TileInfo(1, 4);
var t2 = new TileInfo(1, 5);
var t3 = new TileInfo(2, 9);
if (t1 + t2 == t3) {
    Debug.Log ("Equal");
}
```

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 entry in the SpriteTile Reference Guide.

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);
     }
}
```

See the SpriteTile Reference Guide for more details about the parameters used with Tile.NewLevel.

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 a cell that was set to TileInfo(1, 10), it will always return TileInfo(1, 10) regardless of whether it's animated or not.

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 have an animation sequence where you're using all the frames in the level, such as a water animation where each tile is randomly selected. It's 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);
```

Note that creating or loading a new level does not stop AnimateTile or AnimateTileRange. Once these functions are set up, they will continue until you manually stop them.

Another way to do tile animation is with AnimateTilePosition. Unlike AnimateTile and AnimateTileRange, AnimateTilePosition only animates a single tile at the specified position. It's still cosmetic only, so the actual tile value will remain the same regardless of animation. Aside from specifying a position, it works much like AnimateTile. Let's animate a tile at position (35, 42):

```
Tile.AnimateTilePosition (new Int2(35, 42), new TileInfo(4, 16), 8, 0.25f);
```

This makes the tile start animating with set 4, tile 16, and it will show 8 frames of animation, including 4, 16. So it will cycle from 4, 16 up to and including 4, 23. The framerate is specified at 4fps (0.25 seconds per frame). You can also specify AnimType.Loop, Reverse, or PingPong.

Since you're specifying a position, it makes sense that you should be able to specify a layer, in those cases where you have more than one. Just include that after the position, so here we tell SpriteTile to use position (35, 42) on layer 3:

```
Tile.AnimateTilePosition (new Int2(35, 42), 3, new TileInfo(4, 16), 8, 0.25f);
```

Like AnimateTile, you can specify a sequence of frames in an array, so let's use the "frames" array from the previous page:

```
Tile.AnimateTilePosition (new Int2(35, 42), frames, 0.25f);
```

Unlike AnimateTile, animating a position stops the animation if a new level is loaded. You can stop it manually, of course:

```
Tile.StopAnimatingTilePosition (new Int2(35, 42));        // Use default layer 0
Tile.StopAnimatingTilePosition (new Int2(35, 42), 3);     // Use layer 3
```

If there's no animated tile at the specified position, you'll get an error message.

camera rotation

At times you may want to use camera rotation, such as emulating the “Mode 7” effect in old SNES games by tilting the camera on the X axis. Or maybe you have a character that stands still while the level appears to rotate around him on the Z axis. Due to the way SpriteTile works, setting the camera rotation directly isn’t possible, so you can use the CameraRotationX, CameraRotationY, and CameraRotationZ functions instead.

These functions must be called after Tile.SetCamera has been called, and after a level has been loaded or created in code. Simply specify the degrees you want the camera to be rotated around a particular axis; the screenshot above was created by using this code in the Start function:

```
Tile.SetCamera();
Tile.LoadLevel (level);
Tile.CameraRotationX (-20.0f);
```

Note that rotation around the X and Y axes requires the use of a perspective camera. If the camera is orthographic, an error message will be printed. Also, the rotation is clamped between -90.0° and 90.0°.

Rotation around the Z axis is possible with both perspective and orthographic cameras. It’s not clamped, so any number can be used:

```
Tile.CameraRotationZ (180.0f);
```

Note that it’s not possible to combine rotations on different axes—if you use CameraRotationX, for example, then any rotation that may have been applied with CameraRotationY or CameraRotationZ will be removed.

One important thing that needs to be done when rotating the camera is to increase the “add border” property for levels. This can be done in the TileEditor for levels that you’re loading; otherwise levels created in code should use the addBorder property when using Tile.NewLevel. If you don’t increase the “add border” property, then some tiles at the edges will appear to pop in when the camera moves. The exact number you use for addBorder depends on how much you rotate the camera, so you may need to experiment a little in order to get a number that’s not too big but which still prevents the pop-in effect.

memory usage

For basic usage, SpriteTile uses 7 bytes per tile. For example, a 1000x1000 tile level would use a little under 6.7MB.

If lights or color functions (SetColor, GetColor) are used, another 4 bytes per tile are used by default. If UseTrueColor (false) is used, then memory usage drops to an extra 2 byes per tile instead. This means each tile takes a total of either 9 or 11 bytes.

If optional tile properties are used, each tile that has a property takes at least 24 bytes, with each character in the string property using an additional 2 bytes (since UTF16 is the encoding used for strings). Tiles that don't have any optional properties don't use any extra memory.

If lights are used, each tile that has a light uses 24 additional bytes, or more if the light uses shadows. For shadows, memory usage is 1 byte per tile of the light size, where the size is computed using the largest dimension. e.g., a 21x15 light would use 21x21 for the size, for a total of 441 bytes. Tiles that don't have lights don't use any extra memory.

If physics colliders are used, the additional memory usage depends on how many tiles have physics colliders, and the complexity of the colliders.

There's also some overhead for displaying tiles as sprites on-screen. This depends on the number of layers and how many tiles are displayed based on tile size and camera size. This overhead is fixed and doesn't change regardless of how large the level is.