# Tiled Documentation

***Release 1.3.3***

**Thorbjørn Lindeijer**

**Apr 01, 2020**

# User Manual

**Note:** If you're not finding what you're looking for in these pages, please don't hesitate to ask questions on the Tiled Forum.

CHAPTER 1

# Introduction

## 1.1 About Tiled

**Tiled is a 2D level editor that helps you develop the content of your game. Its primary feature is to edit tile maps of various forms, but it also supports free image placement as well as powerful ways to annotate your level with extra information used by the game. Tiled focuses on general flexibility while trying to stay intuitive.**

In terms of tile maps, it supports straight rectangular tile layers, but also projected isometric, staggered isometric and staggered hexagonal layers. A tileset can be either a single image containing many tiles, or it can be a collection of individual images. In order to support certain depth faking techniques, tiles and layers can be offset by a custom distance and their rendering order can be configured.

The primary tool for editing *tile layers* is a stamp brush that allows efficient painting and copying of tile areas. It also supports drawing lines and circles. In addition, there are several selection tools and a tool that does *automatic terrain transitions*. Finally, it can apply changes based on *pattern-matching* to automate parts of your work.

Tiled also supports *object layers*, which traditionally were only for annotating your map with information but more recently they can also be used to place images. You can add rectangle, point, ellipse, polygon, polyline and tile objects. Object placement is not limited to the tile grid and objects can also be scaled or rotated. Object layers offer a lot of flexibility to add almost any information to your level that your game needs.

Other things worth mentioning are the support for adding custom map or tileset formats through plugins, the tile stamp memory, tile animation support and the tile collision editor.

## 1.2 Getting Started

### 1.2.1 Creating a New Map

When launching Tiled for the first time, we are greeted with the following window:

The first thing we'll do is to start a new map with *File -> New -> New Map...* (`Ctrl+N`). The following dialog will pop up:

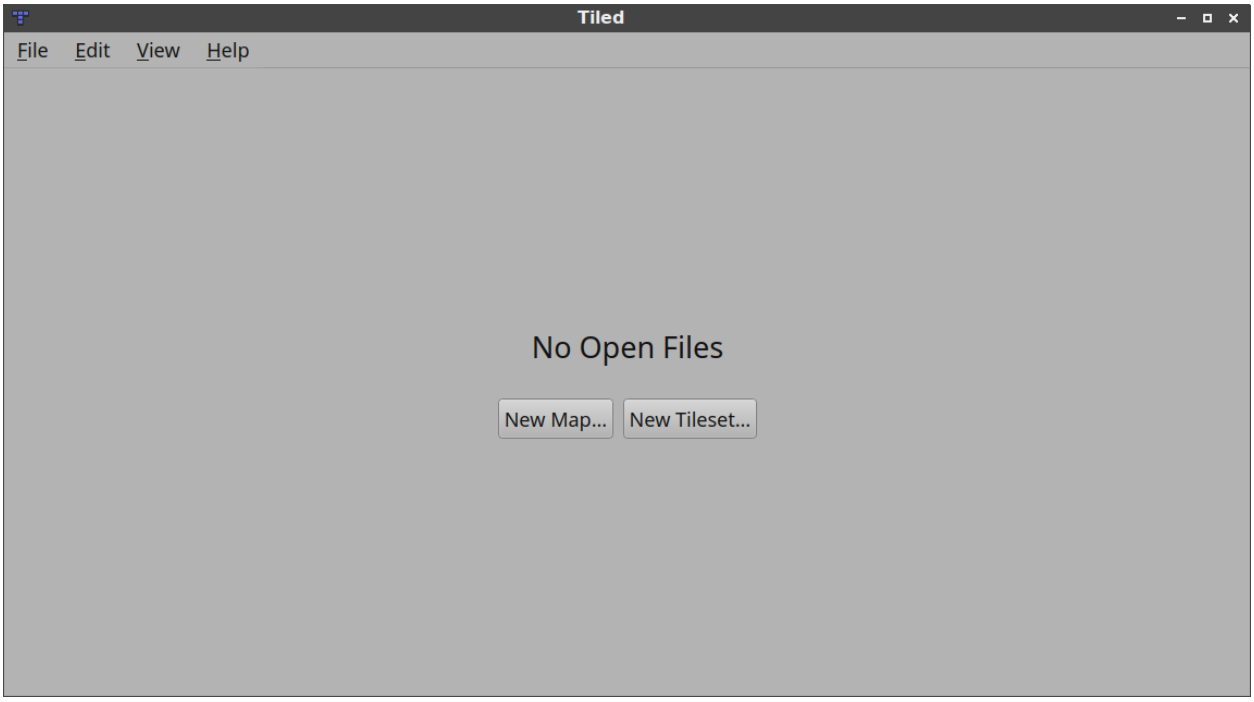



Fig. 1: Tiled Window

Here, we choose the initial map size, tile size, orientation, tile layer format, tile render order (only supported for *Orthogonal* maps) and whether the map is *infinite* or not. All of these things can be changed later as needed, so it's not important to get it all right the first time.

After saving our map, we'll see the tile grid and an initial tile layer will be added to the map. However, before we can start using any tiles we need to add a tileset. Choose *File -> New -> New Tileset...* to open the New Tileset dialog:

Click the *Browse...* button and select the `tmw_desert_spacing.png` tileset from the examples shipping with Tiled (or use one of your own if you wish). This example tileset uses a tile size of 32x32. It also has a one pixel *margin* around the tiles and a one pixel *spacing* in between the tiles (this is pretty rare actually, usually you should leave these values on 0).

---

**Note:** We leave the *Embed in map* option disabled. This is recommended, since it will allow the tileset to be used by multiple maps without setting up its parameters again. It will also be good to store the tileset in its own file if you later add tile properties, terrain definitions, collision shapes, etc., since that information is then shared between all your maps.

---

After saving the tileset, Tiled should look as follows:

Since we don't want to do anything else with the tileset for now, just switch back to the map file:

We're ready to select some tiles and start painting! But first, let's have a quick look at the *various layer types* supported by Tiled.

---

**Note:** Much of the manual still needs to be written. Fortunately, there is a very nice Tiled Map Editor Tutorial Series on GamesFromScratch.com. In addition, the support for Tiled in various *engines and frameworks* often comes with some usage information.

---

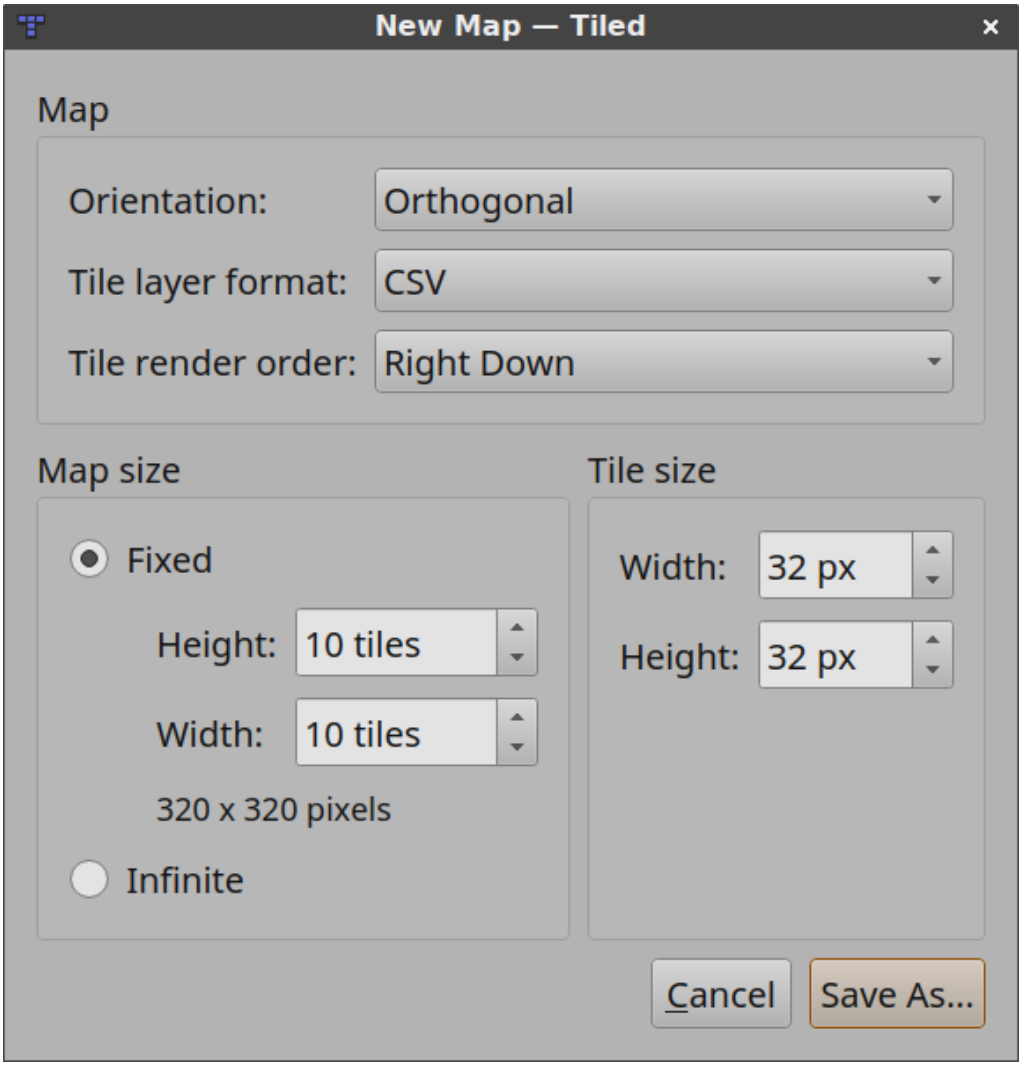

New Map — Tiled
Map
Orientation:
Orthogonal
Tile layer format:
CSV
Tile render order:
Right Down
Map size
Fixed
Height:
10 tiles
Width:
10 tiles
320 x 320 pixels
Infinite
Tile size
Width:
32 px
Height:
32 px
Cancel
Save As...

Fig. 2: New Map

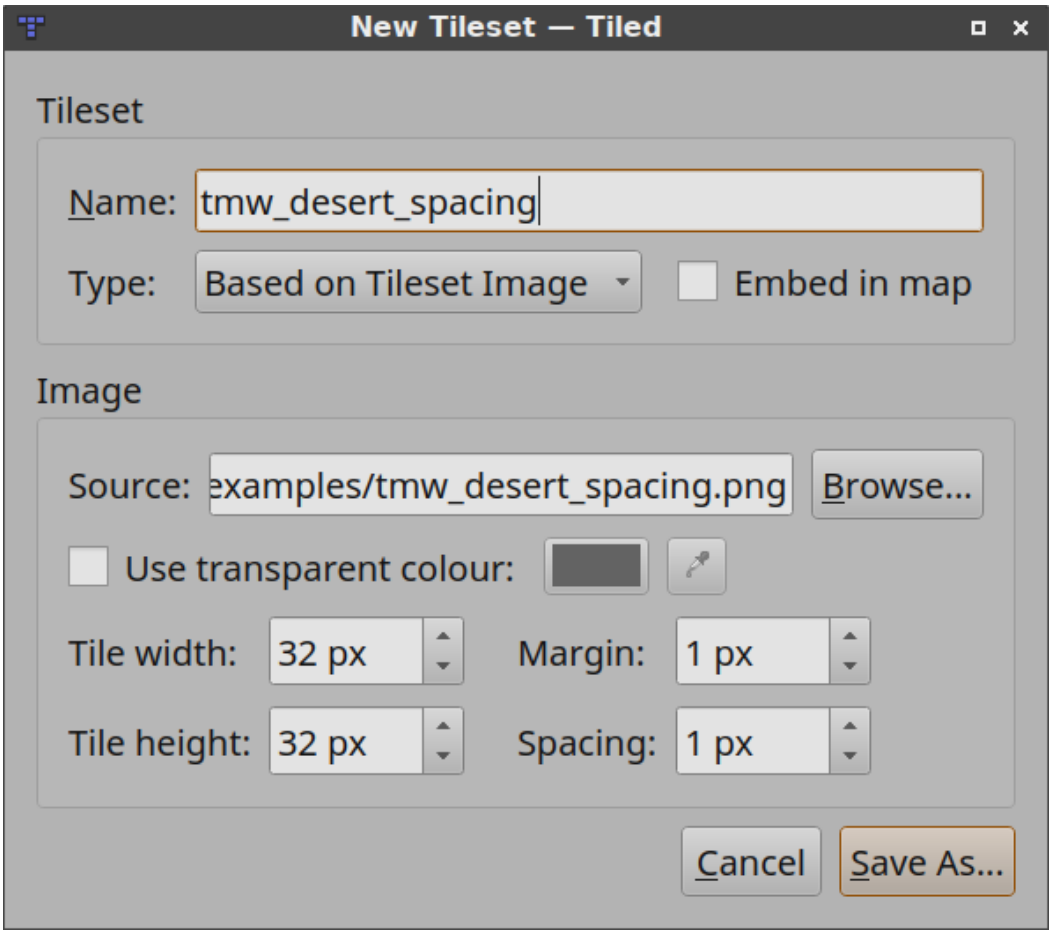



Fig. 3: New Tileset

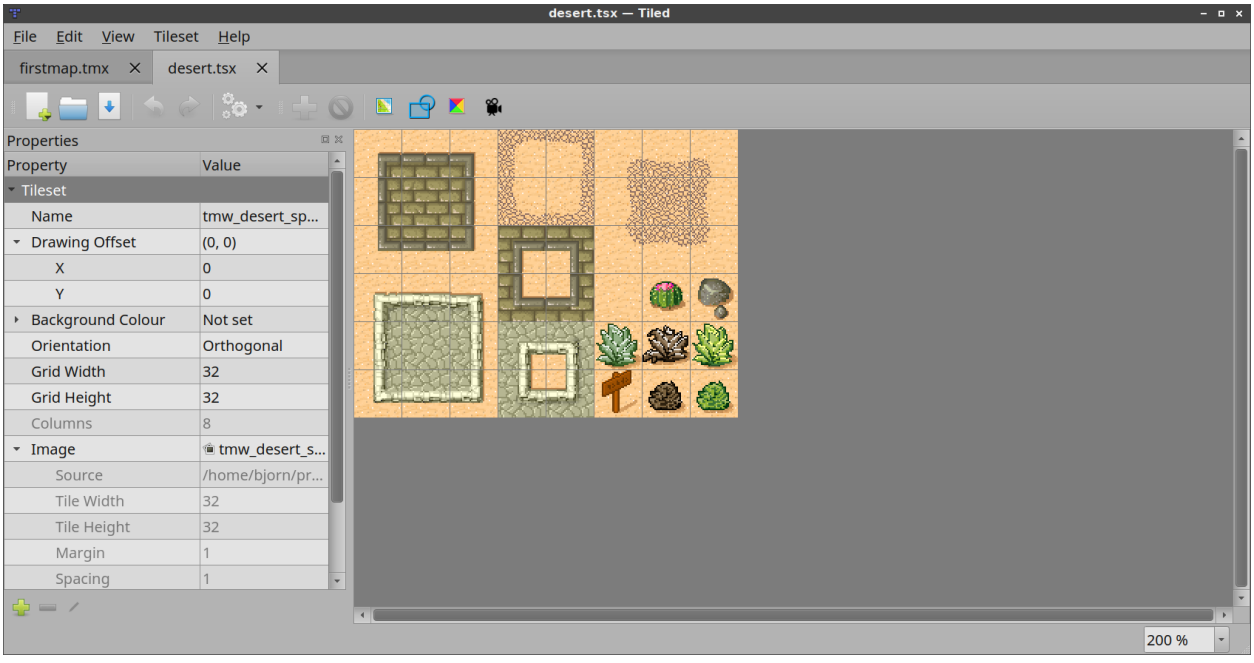



Fig. 4: Tileset Created

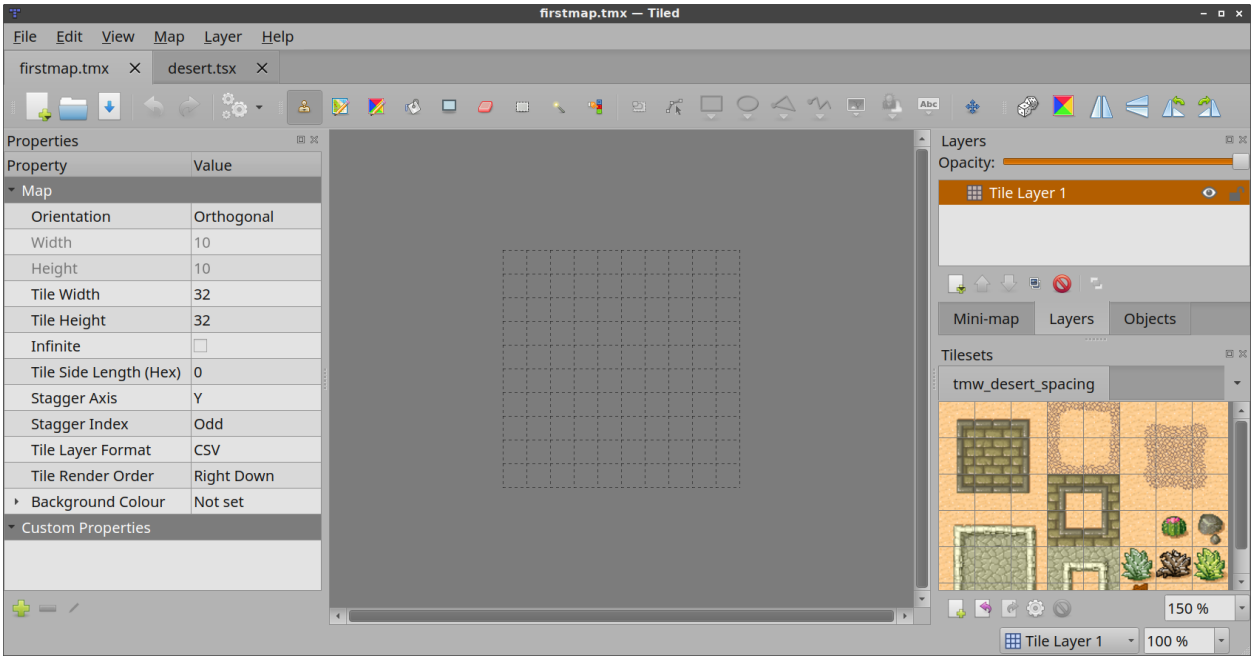



Fig. 5: Tileset Usable on the Map

CHAPTER 2

# Working with Layers

A Tiled map supports various sorts of content, and this content is organized into various different layers. The most common layers are the *Tile Layer* and the *Object Layer*. There is also an *Image Layer* for including simple foreground or background graphics. The order of the layers determines the rendering order of your content.

Layers can be hidden, made only partially visible and can be locked. Layers also have an offset, which can be used to position them independently of each other, for example to fake depth.

You use *Group Layers* to organize the layers into a hierarchy. This makes it more comfortable to work with a large amount of layers.

## 2.1 Tile Layers

Tile layers provide an efficient way of storing a large area filled with tile data. The data is a simple array of tile references and as such no additional information can be stored for each location. The only extra information stored are a few flags, that allow tile graphics to be flipped vertically, horizontally or anti-diagonally (to support rotation in 90-degree increments).

The information needed to render each tile layer is stored with the map, which specifies the position and rendering order of the tiles based on the orientation and various other properties.

Despite only being able to refer to tiles, tile layers can also be useful for defining various bits of non-graphical information in your level. Collision information can often be conveyed using a special tileset, and any kind of object that does not need custom properties and is always aligned to the grid can also be placed on a tile layer.

## 2.2 Object Layers

Object layers are useful because they can store many kinds of information that would not fit in a tile layer. Objects can be freely positioned, resized and rotated. They can also have individual custom properties. There are many kinds of objects:

- **Rectangle** - for marking custom rectangular areas

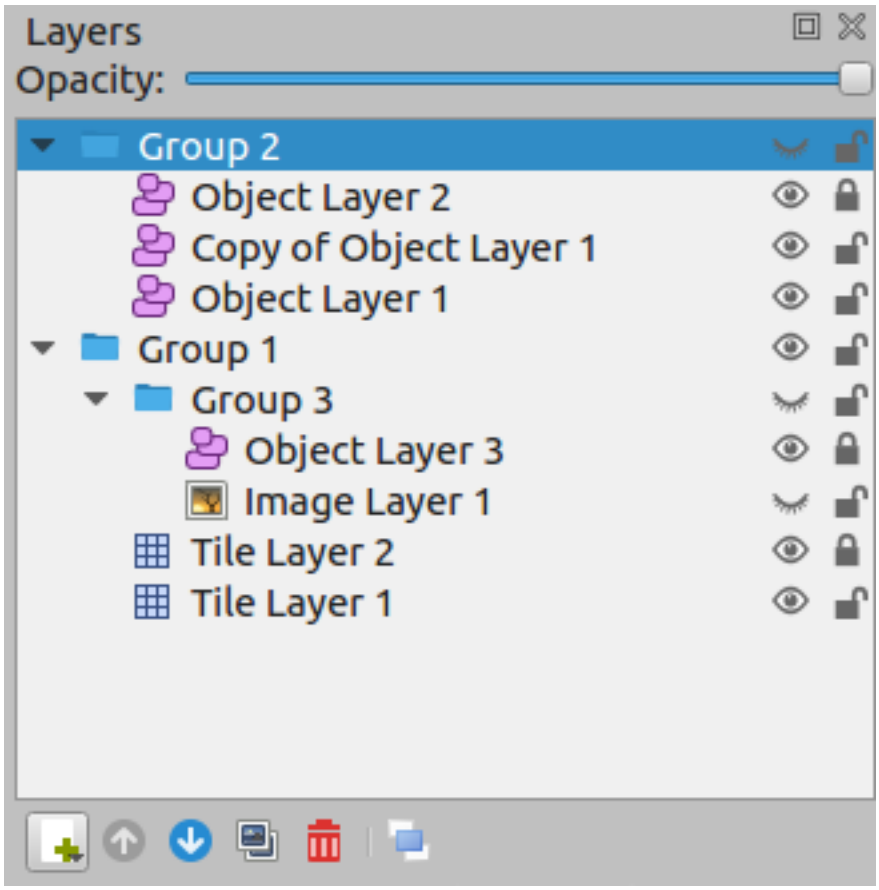



Fig. 1: The eye and lock icon toggle the visibility and locked state of a layer respectively.

- **Ellipse** - for marking custom ellipse or circular areas
- **Point** - for marking exact locations (since Tiled 1.1)
- **Polygon** - for when a rectangle or ellipse doesn't cut it (often a collision area)
- **Polyline** - can be a path to follow or a wall to collide with
- **Tile** - for freely placing, scaling and rotating your tile graphics
- **Text** - for custom text or notes (since Tiled 1.0)

All objects can be named, in which case their name will show up in a label above them (by default only for selected objects). Objects can also be given a *type*, which is useful since it can be used to customize the color of their label and the available *custom properties* for this object type. For tile objects, the type can be *inherited from their tile*.

For most map types, objects are positioned in plain pixels. The only exception to this are isometric maps (not isometric staggered). For isometric maps, it was deemed useful to store their positions in a projected coordinate space. For this, the isometric tiles are assumed to represent projected squares with both sides equal to the *tile height*. If you're using a different coordinate space for objects in your isometric game, you'll need to convert these coordinates accordingly.

The object width and height is also mostly stored in pixels. For isometric maps, all shape objects (rectangle, point, ellipse, polygon and polyline) are projected into the same coordinate space described above. This is based on the assumption that these objects are generally used to mark areas on the map.

## 2.3 Image Layers

Image layers provide a way to quickly include a single image as foreground or background of your map. They are currently not so useful, because if you instead add the image as a Tileset and place it as a *Tile Object*, you gain the ability to freely scale and rotate the image.

The only advantage of using an image layer is that it avoids selecting / dragging the image while using the Select Objects tool. However, since Tiled 1.1 this can also be achieved by locking the object layer containing the tile object you'd like to avoid interacting with.

## 2.4 Group Layers

Group layers work like folders and can be used for organizing the layers into a hierarchy. This is mainly useful when your map contains a large amount of layers.

The visibility, opacity, offset and lock of a group layer affects all child layers.

Layers can be easily dragged in and out of groups with the mouse. The Raise Layer / Lower Layer actions also allow moving layers in and out of groups.

**Future Extensions**

There are many ways in which the layers can be made more powerful:

- Ability to lock individual objects (#828).
- Moving certain map-global properties to the Tile Layer (#149). It would be useful if one map could accommodate layers of different tile sizes and maybe even of different orientation.

If you like any of these plans, please help me getting around to it faster by becoming a patron. The more support I receive the more time I can afford to spend improving Tiled!

CHAPTER 3

# Editing Tile Layers

*Tile Layers* are what makes Tiled a *tile map editor*. Although not as flexible as *Object Layers*, they provide efficient data storage and good rendering performance as well as efficient content creation. Every new map gets one by default, though feel free to delete it when you're not going to use it.

## 3.1 Stamp Brush

Shortcut: `B`

The primary tool for editing tile layers is the Stamp Brush. It can be used to paint single tiles as well as larger "stamps", which is where it gets its name from. Using the right mouse button, it can also quickly capture tile stamps from the currently active layer. A tile stamp is commonly created by selecting one or more tiles in the Tilesets view.

The Stamp Brush has some extra features:

- While holding `Shift`, click any two points to draw a line between them.
- While holding `Ctrl+Shift`, click any two points two draw a circle or ellipse centered on the first point.
- Activate the *Random Mode* using the dice button on the Tool Options toolbar to have the Stamp Brush paint with random tiles from the tile stamp. The probability of each tile depends on how often it occurred on the tile stamp, as well as the probability set on each tile in the *Tileset Editor*.
- Activate the *Wang Fill Mode* using the Wang tile button on the tool bar to have the Stamp Brush paint using the Wang methods. This makes adjacent tiles match edge and corner colors to be placed. Wang tiles are described in detail in *Using Wang Tiles*.
- In combination with the *Tile Stamps* view, it can also place randomly from a set of predefined tile stamps. This can be more useful than the *Random Mode*, which randomly places individual tiles.
- You can flip the current tile stamp horizontally/vertically by using `X` and `Y` respectively. You can also rotate left/right by using `Z` and `Shift+Z` respectively. These actions can also be triggered from the Tool Options tool bar.

## 3.2 Terrain Brush

Shortcut: `T`

The Terrain Brush allows for efficient editing with a certain type of corner-based terrain transitions. Setting it up requires associating terrain information with your tiles, which is described in detail in *Using the Terrain Tool*.

Similarly to the *Stamp Brush*, you can draw lines by holding `Shift`. When holding `Ctrl`, the size of the editing area is reduced to one corner (this currently doesn't work well in combination with drawing lines).

When holding `Alt`, the editing operations are also applied at a 180 degree rotation. This is especially useful when editing strategic maps where two sides need to have equal opportunities. The modifier works well in combination with either `Shift` for drawing lines or `Ctrl` for reducing the edited area.

## 3.3 Wang Brush

Shortcut: `G`

The Wang Brush works in a very similar way to the *Terrain Brush*, except it uses Wang sets. Key differences are:

- Wang tiles support edges as well as corners, whereas terrains only support corners. This makes Wang tiles useful for drawing paths, or fences.
- The default size is to edit one edge/corner. Holding `Ctrl` expands it to the whole tile.
- If the transition cannot be made on the immediately affected tiles, the operation is aborted.

To use the tool, a color must be selected from the Wang color view. Wang tiles and this tool are described in detail in *Using Wang Tiles*.

## 3.4 Bucket Fill Tool

Shortcut: `F`

The Bucket Fill Tool provides a quick way of filling empty areas or areas covered with the same tiles. The currently active tile stamp will be repeated in the filled area. It can also be used in combination with the *Random Mode*, or *Wang Fill Mode*.

When holding `Shift`, the tool fills the currently selected area regardless of its contents. This is useful for filling custom areas that have been selected previously using one or more *Selection Tools*.

You can also flip and rotate the current stamp as described for the *Stamp Brush*.

## 3.5 Shape Fill Tool

Shortcut: `P`

This tool provides a quick way to fill rectangles or ellipses with a certain tile or pattern. Hold `Shift` to fill an exact square or circle.

You can also flip and rotate the current stamp as described for the *Stamp Brush*.

## 3.6 Eraser

Shortcut: `E`

A simple eraser tool. Left click erases single tiles and right click can be used to quickly erase rectangular areas.

## 3.7 Selection Tools

There are various tile selection tools that all work in similar fashion:

- **Rectangular Select** allows selection of rectangular areas (shortcut: `R`)
- **Magic Wand** allows selection of connected areas filled with the same tile (shortcut: `W`)
- **Select Same Tile** allows selection of same-tiles across the entire layer (shortcut: `S`)

By default, each of these tools replaces the currently selected area. The following modifiers can be used to change this behavior:

- Holding `Shift` expands the current selection with the new area
- Holding `Ctrl` subtracts the new area from the current selection
- Holding `Ctrl` and `Shift` selects the intersection of the new area with the current selection

You can also lock into one of these modes (Add, Subtract or Intersect) by clicking on one of the tool buttons in the Tool Options toolbar.

## 3.8 Managing Tile Stamps

It can often be useful to store the current tile stamp somewhere to use it again later. The following shortcuts work for this purpose:

- `Ctrl + 1-9` - Store current tile stamp (similar to `Ctrl + C`)
- `1-9` - Recall the stamp stored at this location (similar to `Ctrl + V`)

Tile stamps can also be stored by name and extended with variations using the *Tile Stamps* view.

CHAPTER 4

# Working with Objects

Using objects you can add a great deal of information to your map for use in your game. They can replace tedious alternatives like hardcoding coordinates (like spawn points) in your source code or maintaining additional data files for storing gameplay elements. With the addition of *tile objects*, they also became useful for graphical purposes and can in some cases replace tile layers entirely, as demonstrated by the "Sticker Knight" example shipping with Tiled.

To start using objects, add an *Object Layer* to your map.

## 4.1 Placement Tools

Each type of object has its own placement tool.

A preview is shown of the object you're about to place when you hover over the map. While placing an object, you can press `Escape` or right-click to cancel placement of the object. Press `Escape` again to switch to the *Select Objects* tool.

### 4.1.1 Insert Rectangle

Shortcut: `R`

The rectangle was the first type of object supported by Tiled, which is why objects are rectangles by default in the *TMX Map Format*. They are useful for marking rectangular areas and assigning custom properties to them. They are also often used for specifying collision boxes.

Place a rectangle by clicking-and-dragging in any direction. Holding `Shift` makes it square and holding `Ctrl` snaps its size to the tile size.

If the rectangle is empty (width and height are both 0), it is rendered as a small square around its position. This is mainly to keep it visible and selectable.

### 4.1.2 Insert Point

Shortcut: `I`

Points are the simplest objects you can place on a map. They only represent a location, and cannot be resized or rotated. Simply click on the map to position a point object.

### 4.1.3 Insert Ellipse

Shortcut: `C`

Ellipses work the same way as *rectangles*, except that they are rendered as an ellipse. Useful for when your area or collision shape needs to represent a circle or ellipse.

### 4.1.4 Insert Polygon

Shortcut: `P`

Polygons are the most flexible way of defining the shape of an area. They are most commonly used for defining collision shapes.

When placing a polygon, the first click determines the location of the object as well as the location of the first point of the polygon. Subsequent clicks are used to add additional points to the polygon. Polygons needs to have at least three points. Click the first point again to finish creating the polygon. You can press `Escape` to cancel the creation of the polygon.

When you want to change a polygon after it has been placed, you need to use the *Edit Polygons* tool.

#### Polylines

Polylines are created by not closing a polygon. Right-click or press Enter while creating a polygon to finish it as a polyline.

Polylines are rendered as a line and require only two points. While they can represent collision walls, they are also often used to represent paths to be followed.

You can extend an existing polyline at either end when it is selected, by clicking on the displayed dots. It is also possible to finish the polyline by connecting it to either end of another existing polyline object. The other polyline object needs to be selected as well, since the interactive dots only show on selected polylines.

The *Edit Polygons* tool is used to edit polylines as well.

### 4.1.5 Insert Tile

Shortcut: `T`

Tiles can be inserted as objects to have full flexibility in placing, scaling and rotating the tile image on your map. Like all objects, tile objects can also have custom properties associated with them. This makes them useful for placement of recognizable interactive objects that need special information, like a chest with defined contents or an NPC with defined script.

To place a tile object, first select the tile you want to place in the Tilesets view. Then use the Left mouse button on the map to start placing the object, move to position it based on the preview and release to finish placing the object.

To change the tile used by existing tile objects, select all the objects you want to change using the *Select Objects* tool and then right-click on a tile in the Tilesets view, and choose *Replace Tile of Selected Objects*.

### 4.1.6 Insert Template

Shortcut: `V`

Can be used to quickly insert multiple instances of the template selected in the Templates view. See *Creating Template Instances*.

### 4.1.7 Insert Text

Shortcut: `X`

Text objects can be used to add arbitrary multi-line text to your maps. You can configure various font properties and the wrapping / clipping area, making them useful for both quick notes as well as text used in the game.

## 4.2 Select Objects

Shortcut: `S`

When you're not inserting new objects, you're generally using the Select Objects tool. It packs a lot of functionality, which is outlined below.

### 4.2.1 Selecting and Deselecting

You can select objects by clicking them or by dragging a rectangular lasso, selecting any object that intersect with its area. By holding `Shift` or `Ctrl` while clicking, you can add/remove single objects to/from the selection. Press `Escape` to deselect all objects.

When pressing and dragging on an object, this object is selected and moved. When this prevents you from starting a rectangular selection, you can hold `Shift` to force the selection rectangle.

By default you interact with the top-most object. When you need to select an object below another object, first select the higher object and then hold `Alt` while clicking at the same location to select lower objects. You can also hold `Alt` while opening the context menu to get a list of all objects at the clicked location, so you may directly select the desired object.

You can quickly switch to the *Edit Polygons* tool by double-clicking on the polygon or polyline you want to edit.

### 4.2.2 Moving

You can simply drag any single object, or drag already selected objects by dragging any one of them. Hold `Ctrl` to toggle snapping to the tile grid.

Hold `Alt` to force a move operation on the currently selected objects, regardless of where you click on the map. This is useful when the selected objects are small or covered by other objects.

The selected objects can also be moved with the arrow keys. By default this moves the objects pixel by pixel. Hold `Shift` while using the arrow keys to move the objects by distance of one tile.

### 4.2.3 Resizing

You can use the resize handles to resize one or more selected objects. Hold `Ctrl` to keep the aspect ratio of the object and/or `Shift` to place the resize origin in the center.

Note that you can only change width and height independently when resizing a single object. When having multiple objects selected, the aspect ratio is constant because there would be no way to make that work for rotated objects without full support for transformations.

### 4.2.4 Rotating

To rotate, click any selected object to change the resize handles into rotation handles. Before rotating, you can drag the rotation origin to another position if necessary. Hold `Shift` to rotate in 15-degree increments. Click any selected object again to go back to resize mode.

You can also rotate the selected objects in 90-degree steps by pressing `Z` or `Shift + Z`.

### 4.2.5 Changing Stacking Order

If the active *Object Layer* has its Drawing Order property set to Manual (the default is Top Down), you can control the stacking order of the selected objects within their object layer using the following keys:

- `PgUp` - Raise selected objects
- `PgDown` - Lower selected objects
- `Home` - Move selected objects to Top
- `End` - Move selected objects to Bottom

You can also find these actions in the context menu. When you have multiple Object Layers, the context menu also contains actions to move the selected objects to another layer.

### 4.2.6 Flipping Objects

You can flip the selected objects horizontally by pressing `X` or vertically by pressing `Y`. For tile objects, this also flips their images.

## 4.3 Edit Polygons

Shortcut: `E`

Polygons and polylines have their own editing needs and as such are covered by a separate tool, which allows selecting and moving around their nodes. You can select and move the nodes of multiple polygons at the same time. Click a segment to select the nodes at both ends. Press `Escape` to deselect all nodes, or to switch back to the *Select Objects* tool.

Nodes can be deleted by selecting them and choosing "Delete Nodes" from the context menu. The `Delete` key can also be used to delete the selected nodes, or the selected objects if no nodes are selected.

When you have selected multiple consecutive nodes of the same polygon, you can join them together by choosing "Join Nodes" from the context menu. You can also split the segments in between the nodes by choosing "Split Segments". Alternatively, you can simply double-click a segment to split it at that location.

You can also delete a segment when two consecutive nodes are selected in a polygon by choosing "Delete Segment" in the context menu. This will convert a polygon into a polyline, or turn one polyline object in two polyline objects.

It is possible to extend a polyline at either end, either by right-clicking those nodes and choosing "Extend Polyline", or by switching to the *Insert Polygon* tool and clicking on either end of an already selected polylines.

**Future Extensions**

Here are some ideas about improvements that could be made to the above tools:

- Some improvements could still be made to the support for editing polygons and polylines, like allowing to rotate and scale the selected nodes (#1487).
- The tools could put short usage instructions in the status bar, to help new users without requiring them to carefully read the manual (#1855).

If you like any of these plans, please help me getting around to it faster by becoming a patron. The more support I receive the more time I can afford to spend improving Tiled!

CHAPTER 5

# Editing Tilesets

To edit a tileset it needs to be opened explicitly for editing. External tilesets can be opened via the *File* menu, but in general the quickest way to edit the tileset when it is already open in the *Tilesets* view is to click the small *Edit Tileset* button in the tool bar below the tileset.

## 5.1 Two Types of Tileset

A tileset is a collection of tiles. Tiled currently supports two types of tilesets, which are chosen when creating a new tileset:

**Based on Tileset Image** This tileset defines a fixed size for all tiles and the image from which these tiles are supposed to be cut. In addition it supports a margin around the tiles and a spacing between the tiles, which allows for using tileset images that either happen to have space between or around their tiles or those that have extruded the border pixels of each tile to avoid color bleeding.

**Collection of Images** In this type of tileset each tile refers to its own image file. It is useful when the tiles aren't the same size, or when the packing of tiles into a texture is done later on.

Regardless of the type of tileset, you can associate a lot of meta- information with it and its tiles. Some of this information can be for use in your game, like *collision information* and *animations*. Other information is primarily meant for certain editing tools.

---

**Note:** A tileset can be either embedded in a map file or saved externally. Since Tiled 1.0, the default and recommended approach is to save your tilesets to their own file. This simplifies your workflow since it makes sure any meta-information is shared between all maps using the same tileset.

---

## 5.2 Tileset Properties

You can access the tileset properties by using the menu action *Tileset* > *Tileset Properties*.

**Name** The name of the tileset. Used to identify the tileset in the *Tilesets* view when editing a map.

**Drawing Offset** A drawing offset in pixels, applied when rendering any tile from the tileset (as part of tile layers or as tile objects). This is can be useful to make your tiles align to the grid.

**Background Color** A background color for the tileset, which can be set in case the default dark-gray background is not suitable for your tiles.

**Orientation** When the tileset contains isometric tiles, you can set this to *Isometric*. This value, along with the **Grid Width** and **Grid Height** properties, is taken into account by overlays rendered on top of the tiles. This helps for example when specifying *Terrain Information* or editing *Wang Sets*. It also affects the orientation used by the *Tile Collision Editor*.

**Columns** This is a read-only property for tilesets based on a tileset image, but for image collection tilesets you can control the number of columns used when displaying the tileset here.

**Image** This property only exists for tilesets based on a tileset image. Selecting the value field will show an *Edit...* button, allowing you to change the parameters relevant to cutting the tiles from the image.

Of course, as with most data types in Tiled, you can also associate *Custom Properties* with the tileset.

## 5.3 Tile Properties

**ID** The ID of the tile in the tileset (read-only)

**Type** This property refers to custom types defined in the *Object Types Editor*. See the section about *Typed Tiles* for more information.

**Width and Height** The size of the tile (read-only)

**Probability** Represents a relative probability that this tile will get chosen out of multiple options. This value is used in *Random Mode* and by the *Terrain Brush*.

**Image** Only relevant for tiles that are part of image collection tilesets, this shows the image file of the tile and allows you to change it.

## 5.4 Terrain Information

Terrain information can be added to a tileset to enable the use of the the *Terrain Brush*. See the section about *defining terrain information*.

## 5.5 Wang Sets

A tileset can contain any number of Wang sets for use with the *Wang Brush*. See *Defining Wang Tile Info* for more information.

## 5.6 Tile Collision Editor

The tile collision editor is available by clicking the *Tile Collision Editor*  button on the tool bar. This will open a view where you can create and edit shapes on the tile. You can also associate custom properties with each shape.

Usually these shapes define collision information for a certain sprite or for a tile representing level geometry, but of course you could also use them to add certain hot-spots to your sprites like for particle emitters or the source of gunshots.

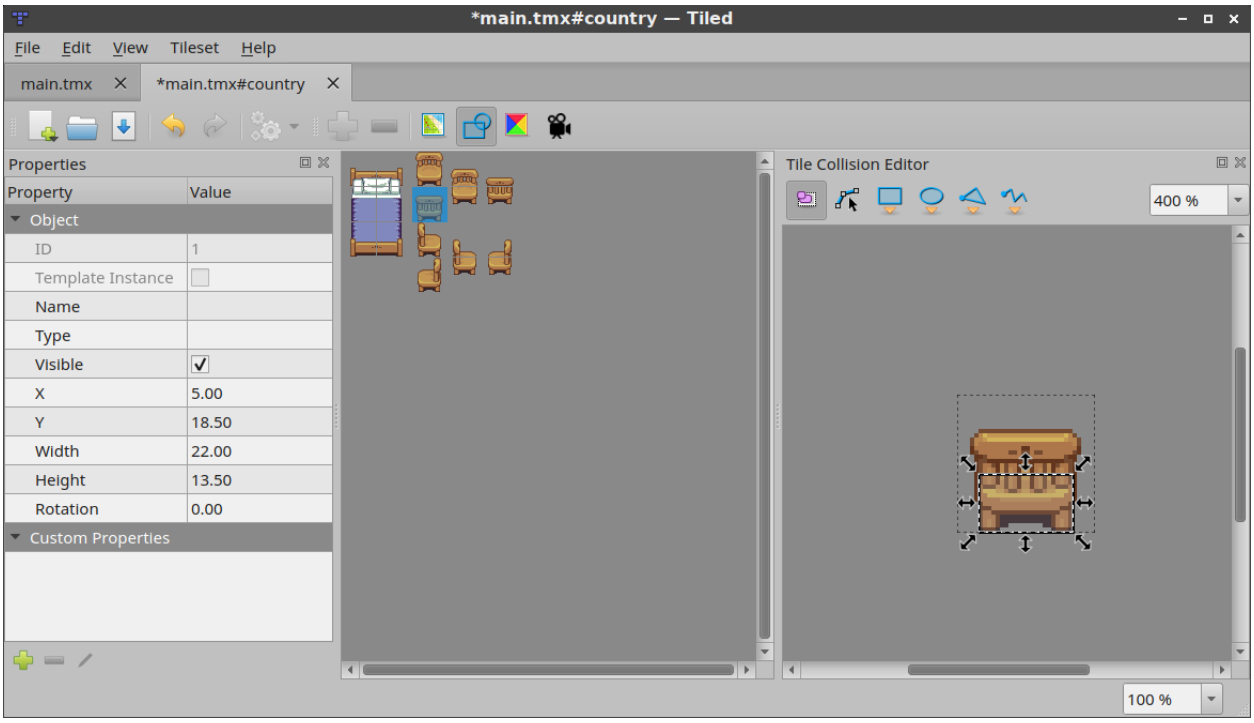



Fig. 1: Tile Collision Editor

**Note:** Check out the Tiled2Unity tool by Sean Barton for a great example of what you can do with this information. It can take the collision shapes for all tiles and generate a single collision mesh from it, as demonstrated in the Mega Dad Adventures post.

## 5.7 Tile Animation Editor

The tile animation editor allows defining a single linear looping animation with each tile by referring to other tiles in the tileset as its frames. Open it by clicking the *Tile Animation Editor*  button.

Tile animations can be live-previewed in Tiled, which is useful for getting a feeling of what it would look like in-game. The preview can be turned on or off via *View* > *Show Tile Animations*.

The following steps allow to add or edit a tile animation:

- Select the tile in the main Tiled window. This will make the *Tile Animation Editor* window show the (initially empty) animation associated with that tile, along with all other tiles from the tileset.
- Drag tiles from the tileset view in the Tile Animation Editor into the list on the left to add animation frames. You can drag multiple tiles at the same time. Each new frame gets a default duration of 100 ms.
- Double-click on the duration of a frame to change it.
- Drag frames around in the list to reorder them.

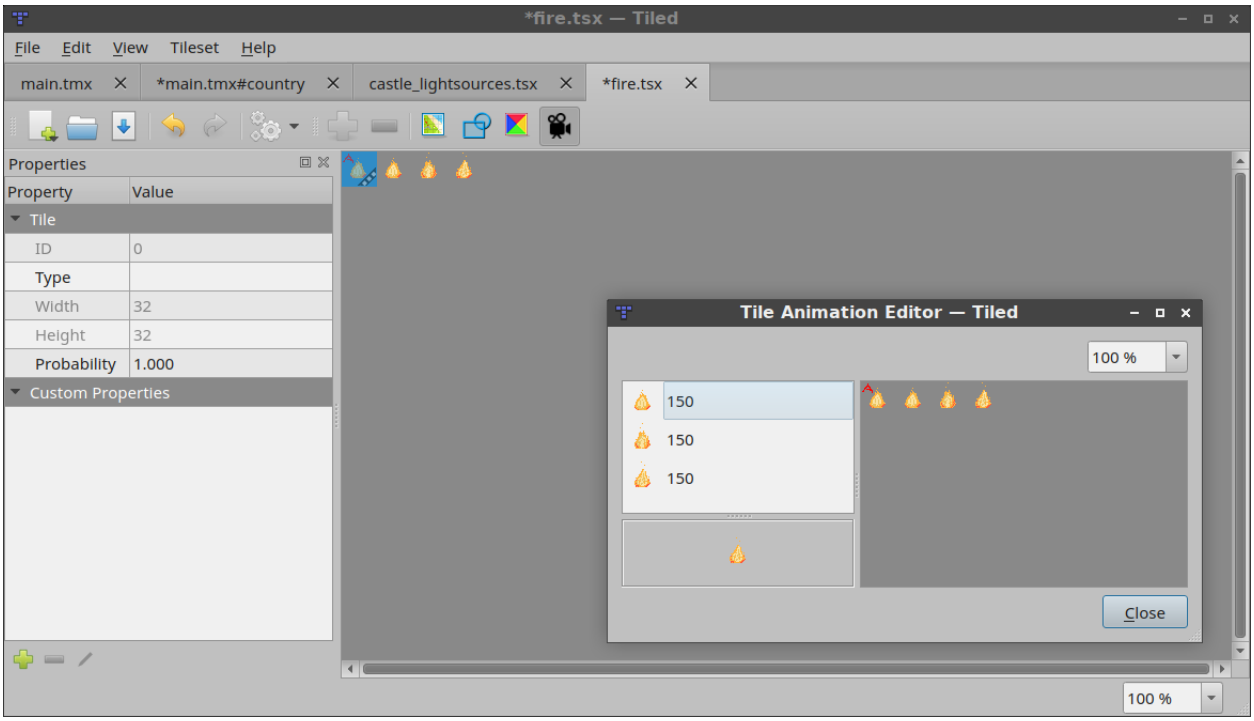



Fig. 2: Tile Animation Editor

A preview of the animation shows in the bottom left corner.

**Future Extensions**

There are many ways in which the tileset editor can be made more efficient, for example:

**Wang Sets**

- Make it easier to set up Wang tiles (#1729)

**Tile Collision Editor**

- Allow setting collisions for multiple tiles at once (#1322)
- Render tile collision shapes to the main map (#799) or to the tileset view (#1281)

**Tile Animation Editor**

- Allow changing the default frame duration (#1631)
- Allow changing the duration of multiple frames at the same time (#1310)
- Support multiple named animations per tile (#986)

If you like any of these plans, please help me getting around to it faster by becoming a patron. The more support I receive the more time I can afford to spend improving Tiled!

CHAPTER 6

# Custom Properties

One of the major strengths of Tiled is that it allows setting custom properties on all of its basic data structures. This way it is possible to include many forms of custom information, which can later be used by your game or by the framework you're using to integrate Tiled maps.

Custom properties are displayed in the Properties view. This view is context-sensitive, usually displaying the properties of the last selected object. For tiles in a tileset or objects on an object layer, it also supports multi-selection.

## 6.1 Adding Properties

When you add a property (using the '+' button at the bottom of the Properties view), you are prompted for its name and its type. Currently Tiled supports the following basic property types:

- **bool** (true or false)
- **color** (a 32-bit color value)
- **file** (a relative path referencing a file)
- **float** (a floating point number)
- **int** (a whole number)
- **object** (a reference to an object) - *Since Tiled 1.4*
- **string** (any text, including multi-line text)

The property type is used to choose a custom editor in the Properties view. Choosing a number or boolean type also avoids that the value will get quoted in JSON and Lua exports.

## 6.2 Tile Property Inheritance

When custom properties are added to a tile, these properties will also be visible when an object instance of that tile is selected. This enables easy per-object overriding of certain default properties associated with a tile. This becomes

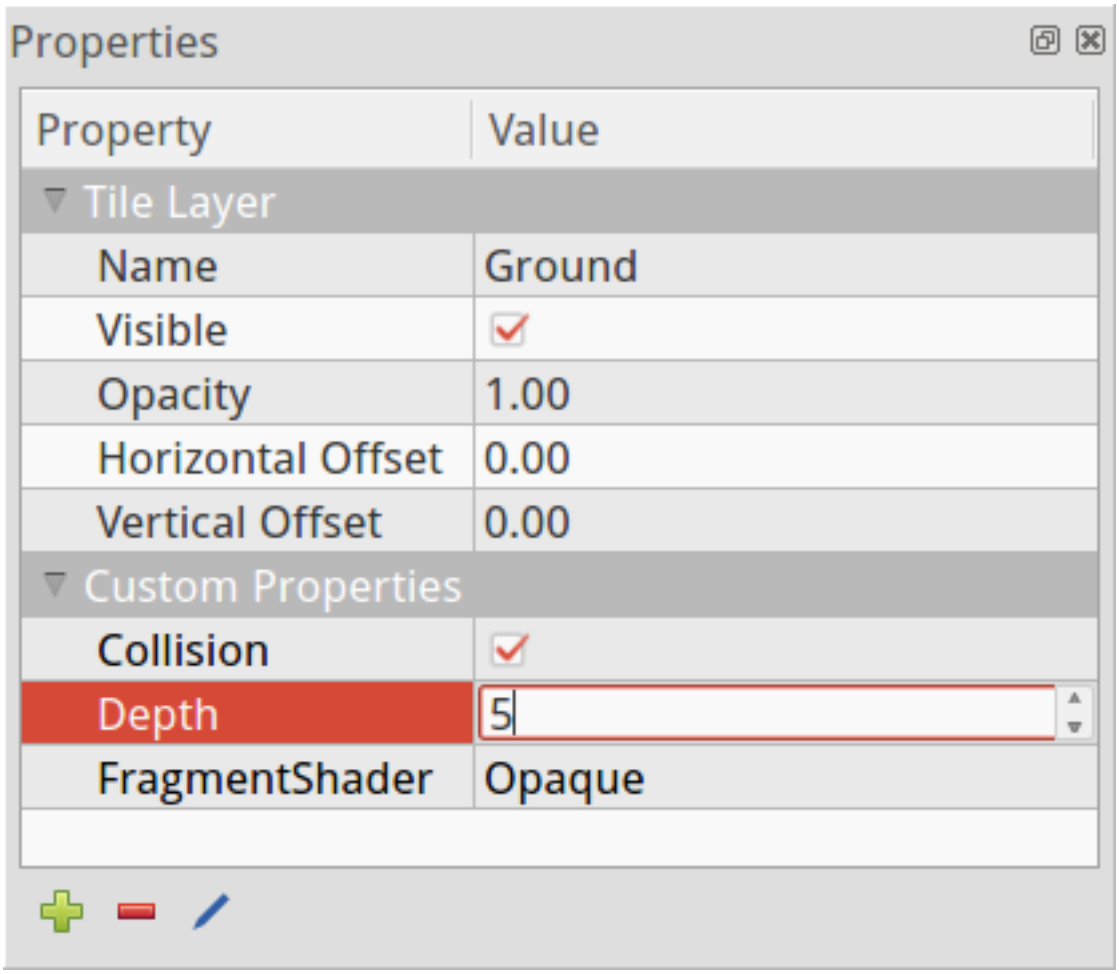



Fig. 1: Properties View

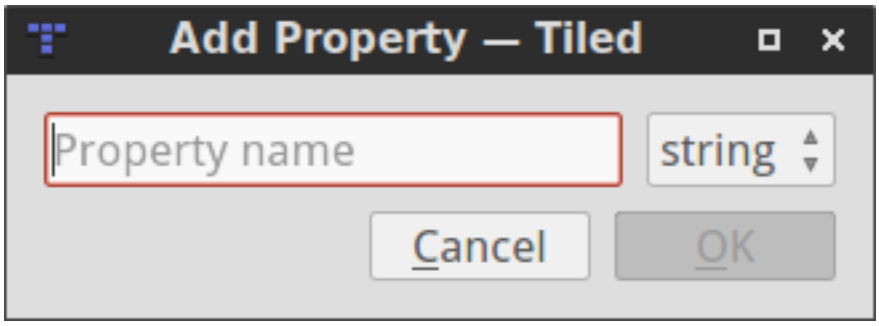



Fig. 2: Add Property Dialog

especially useful when combined with *Typed Tiles*.

Inherited properties will be displayed in gray (disabled text color), whereas overridden properties will be displayed in black (usual text color).

## 6.3 Predefining Properties

### 6.3.1 General Setup

Usually you only use a limited set of object types in your game, and each type of object has a fixed set of possible properties, with specific types and default values. To save you time, Tiled allows predefining these properties based on the "Type" field for objects. You can set this up using the Object Types Editor, available from the *View* menu.

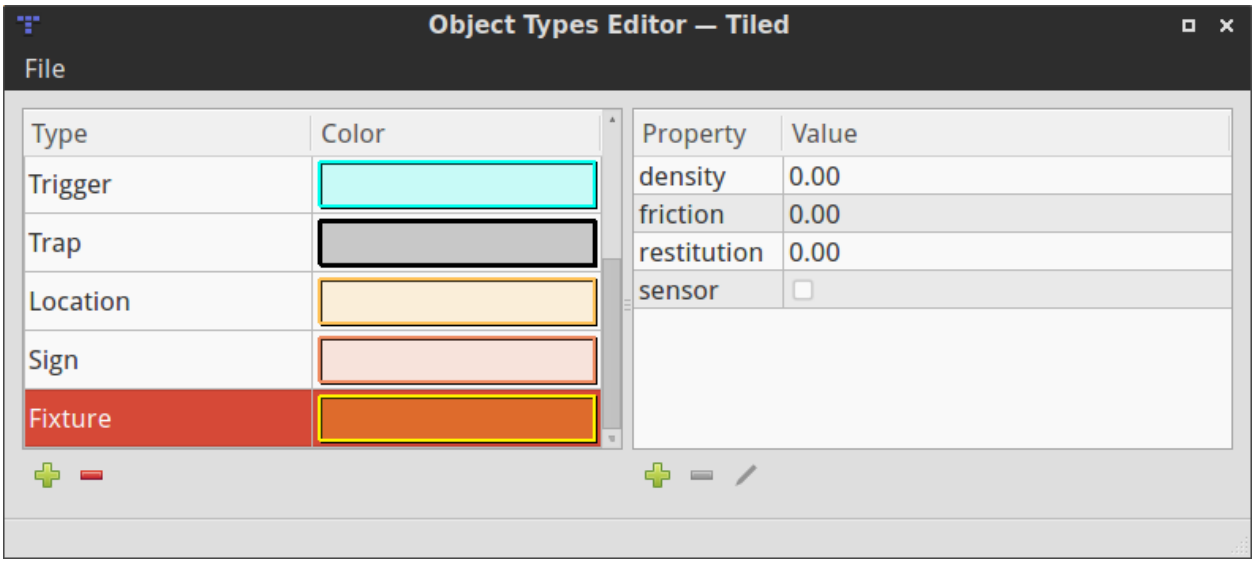



Fig. 3: Object Types Editor

By default, Tiled stores these object types in the user settings. However, since you'll often want to share them with other people in your project, you can export your object types or change the storage location of the object types file. A simple XML or JSON file with self-explanatory contents is used to store your object types.

The color not only affects the rendering of the various shapes of objects, but is also the color of the label which will show up if you give your object a name.

To make the predefined properties show up in the Properties view, all you need to do is to enter the name of the type in the built-in "Type" property. Usually this is what you're doing already anyway to tell your engine what kind of object it is dealing with.

### 6.3.2 Typed Tiles

If you're using *tile objects*, you can set the type on the tile to avoid having to set it on each object instance. Setting the type on the tile makes the predefined properties visible when having the tile selected, allowing to override the values. It also makes those possibly overridden values visible when having a tile object instance selected, again allowing you to override them.

An example use-case for this would be to define custom types like “NPC”, “Enemy” or “Item” with properties like “name”, “health” or “weight”. You can then specify values for these on the tiles representing these entities. And when placing those tiles as objects, you can override those values if you need to.

**Future Extensions**

There are several types of custom properties I’d like to add:

- **Enumerations**, where you can predefine all possible values and it forms a combo box (#1211).
- **Object references**, which would allow easily linking objects together and Tiled could display such connections (#707).
- **Array properties**, which would be properties having a list of values (#1493).
- **Dictionary properties**, which would be properties that can contain any number of other properties as children (#489).

It would also be nice to add support for **limiting property values**, like the length of string properties or a minimum/maximum on number values.

Apart from predefining properties based on object type, I’d like to add support for **predefining the properties for each data type**. So defining which custom properties are valid for maps, tilesets, layers, etc. (#1410)

Finally, the predefined properties would work very well together with explicit **support for projects**. Then you could switch between different projects or get started on an existing project, without needing to configure Tiled to use the right object type definitions.

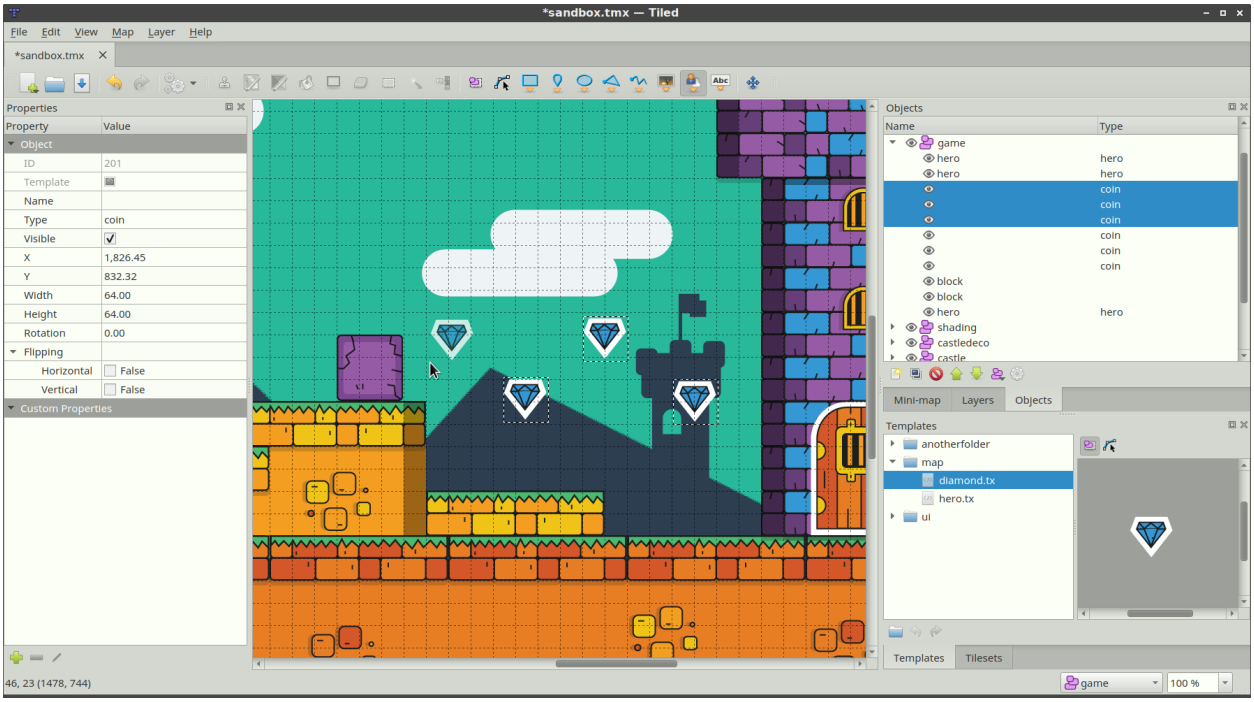
If you like any of these plans, please help me getting around to it faster by becoming a patron. The more support I receive the more time I can afford to spend improving Tiled!

CHAPTER 7

# Using Templates

Any created object can be saved as a template. These templates can then be instantiated elsewhere as objects that inherit the template's properties. This can save a lot of tedious work of setting up the object type and properties, or even just finding the right tile in the tileset.

Each template is stored in its own file and they can be organized in directories. You can save templates in either XML or JSON format, just like map and tileset files.



## 7.1 Creating Templates

A template can be created by right clicking on any object in the map and selecting "Save As Template". You will be asked to choose the file name and the format to save the template in. If the object already has a name the suggested file name will be based on that.

---

**Note:** You can't create a template from a tile object that uses a tile from an embedded tileset, because *template files* do not support referring to such tilesets.

---

## 7.2 The Templates View

Working with templates is done through the Templates view. The Templates view is divided into two parts: the left part is a tree view that shows the template files in a selected directory and the right part shows a preview of the selected template.

## 7.3 Creating Template Instances

Shortcut: V

Template instantiation works by either dragging and dropping the template from the list of templates to the map, or by using the "Insert Template" tool by selecting a template and clicking on the map which is more convenient when you want to create many instances.

## 7.4 Editing Templates

Selecting a template will show an editable preview in the Templates view and will show the template's properties in the Properties view where they can be edited. Changes to the template are saved automatically.

All template instances are linked to their template, so all edits will be immediately reflected upon all the template instances on the map.

If a property of a template instance is changed, it will be internally marked as an overridden property and won't be changed when the template changes.

## 7.5 Detaching Template Instances

Detaching a template instance will disconnect it from its template, so any further edits to the template will not affect the detached instance.

To detach an instance, right click on it and select *Detach*.

**Future Extensions**

- Resetting overridden properties individually (#1725).
- Locking template properties (#1726).
- Handling wrong file paths (#1732).
- Managing the templates folder, e.g. moving, renaming or deleting a template or a sub-folder (#1723).

CHAPTER 8

# Using the Terrain Brush

The *Terrain Brush* was added to make editing tile maps easier when using terrain transitions. There are of course multiple ways to do transitions between tiles. This tool supports transition tiles that have a well-defined terrain type at each of their 4 corners, which seems to be the most common method.

To demonstrate how to use this tool we describe the steps necessary to reproduce the `desert.tmx` example map, which now also includes terrain information in its tileset.

## 8.1 Create a New Map and Add a Tileset

First of all, follow the *Getting Started* instructions to set up the map and the tileset.

The `tmw_desert_spacing.png` tileset we just set up has 4 different terrain types. Traditionally editing a map with these tiles meant that you had to carefully connect the right transitions to avoid broken edges. Now we will define the terrain information for this tileset, which the *Terrain Brush* will use to automatically place the right transitions.

## 8.2 Define the Terrain Information

First of all, switch to the tileset file. If you're looking at the map and have the tileset selected, you can do this by clicking the small *Edit Tileset* button below the Tilesets view.

Then, activate the terrain editing mode by clicking on the *Terrains*  button on the tool bar.

In this mode, the list of terrain types is displayed and you can mark corners of the tiles in your tileset as belonging to a certain terrain type. To start with, add each of the 4 terrain types. The fastest way is by right-clicking on a tile representing a certain terrain and choosing *Add Terrain Type*. This automatically sets the tile as the image representing the terrain.

Give each of the terrains an appropriate name. Once you're done, select the Sand terrain and mark all corners in the tileset with this type of terrain. The result should look like this:

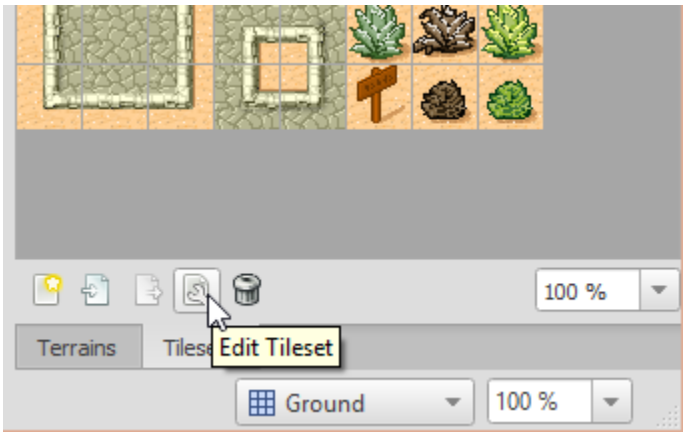



Fig. 1: Edit Tileset button

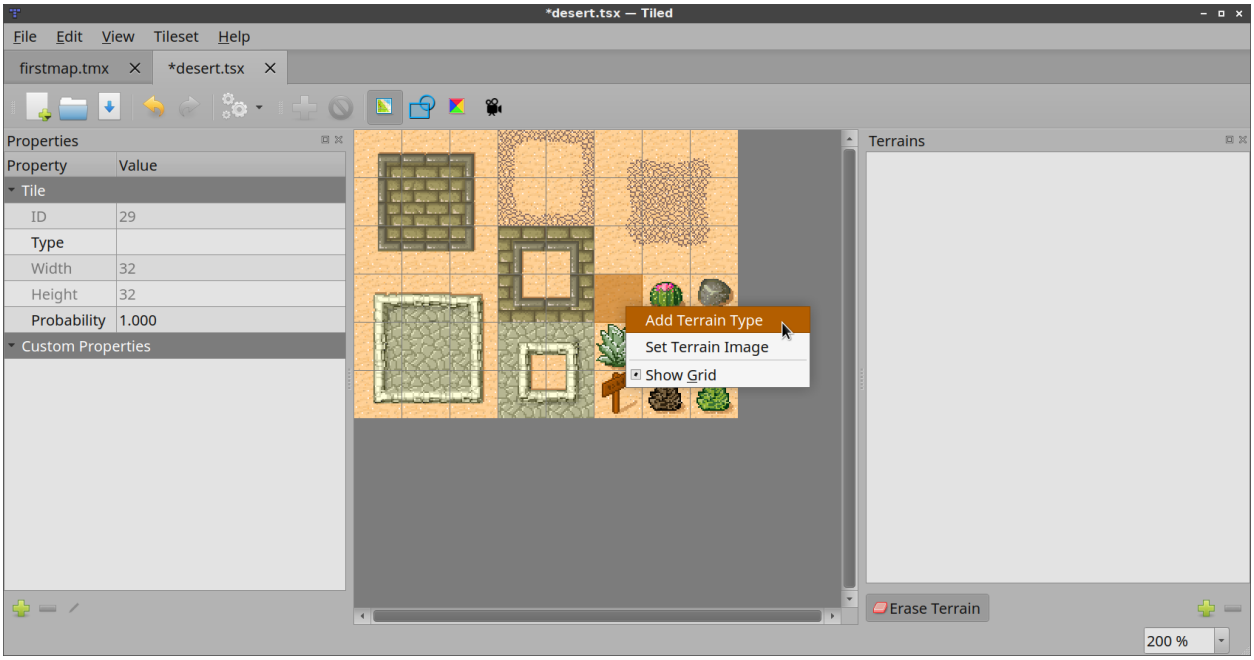



Fig. 2: Adding Terrain Type

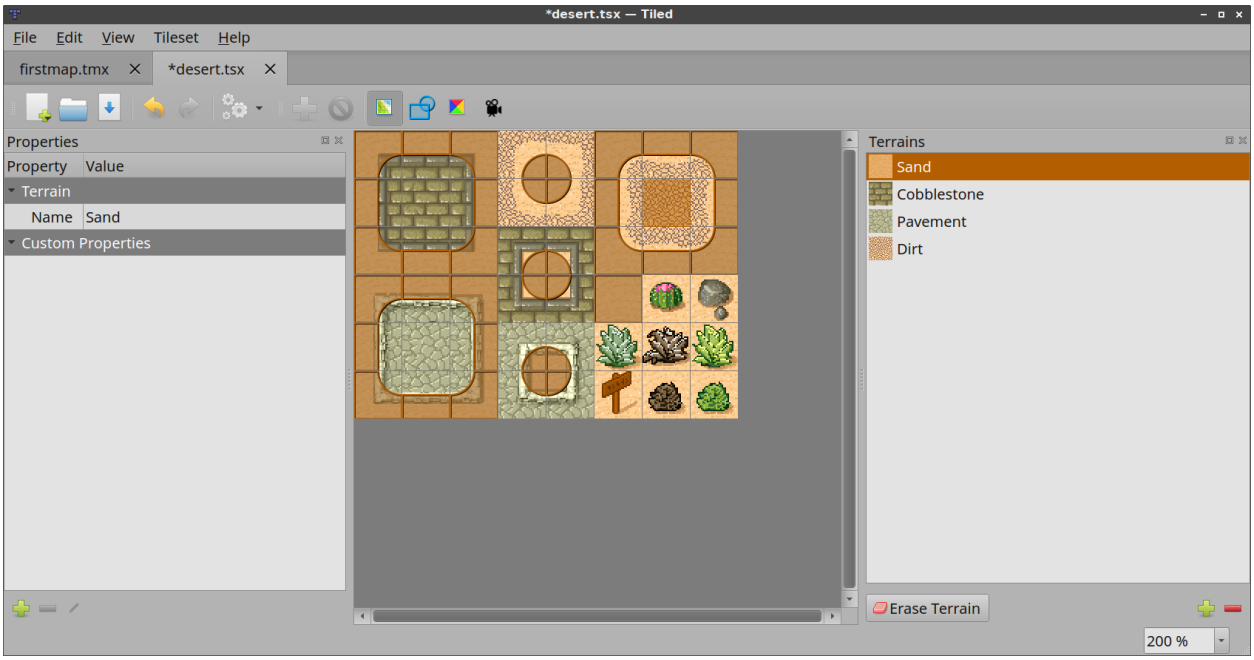



Fig. 3: Sand Terrain Marked

To understand how we did this, we take a look at a single tile that transitions between Sand and Dirt. With the Sand terrain selected, we clicked and dragged to mark the top-left, top-right and bottom-left corners of this tile as "Sand". The bottom-right corner is not yet marked, we'll get to that one once we're marking all Dirt corners.

If you make a mistake, just use Undo (or press `Ctrl+Z`). Or if you notice a mistake later, either use *Erase Terrain* to clear a terrain type from a corner or select the correct terrain type and paint over it. Each corner can only have one type of terrain associated with it.

Now do the same for each of the other terrain types. Eventually you'll have marked all tiles apart from the special objects.

Now you can disable the *Terrains*  mode by clicking the tool bar button again.

## 8.3 Editing with the Terrain Brush

Switch back to the map and then activate the Terrains window. You should see the 4 terrain types represented in a list. Click on the Sand terrain and start painting. You may immediately notice that nothing special is happening. This is because there are no other tiles on the map yet so the terrain tool doesn't really know how to help (because we have no transitions to "nothing" in our tileset). Assuming we're out to create a desert map, it's better to start by filling your entire map with sand. Just switch back to the Tilesets window for a moment, select the sand tile and then use the *Bucket Fill Tool*.

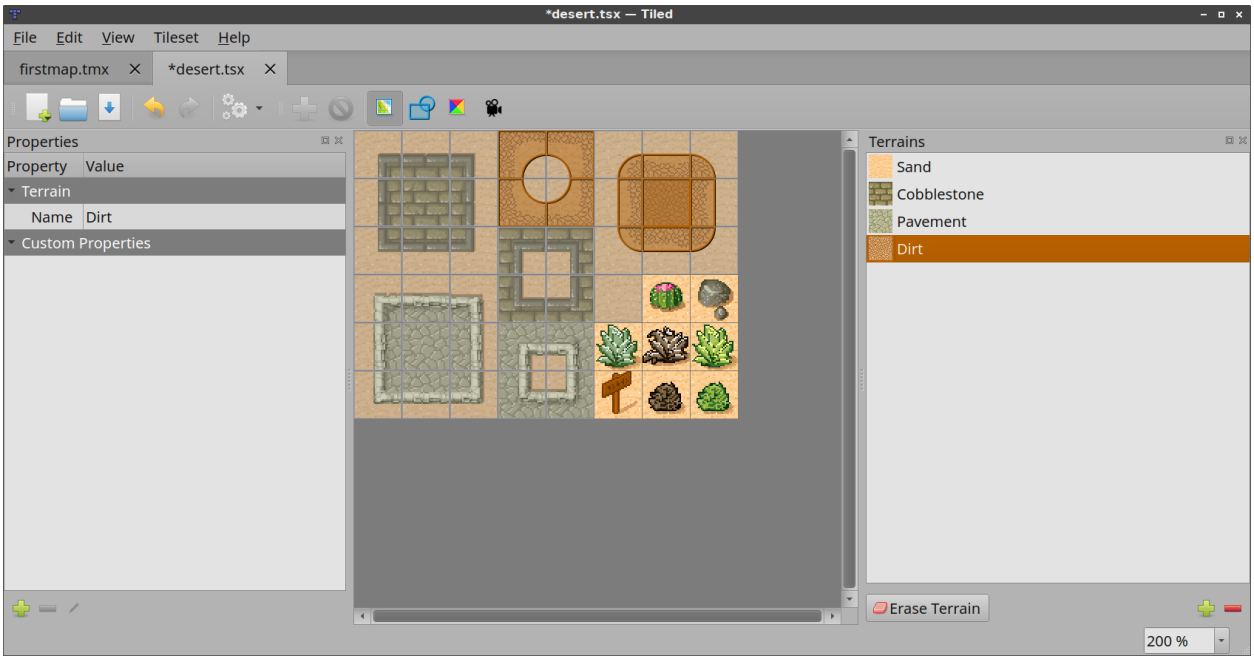



Fig. 4: Done Marking Terrain

Let's switch back to the Terrains window and draw some cobblestones. Now you can see the tool in action!

Try holding `Control` (`Command` on a Mac) while drawing. This reduces the modified area to just the closest corner to the mouse, allowing for precision work.

Finally, see what happens when you try drawing some dirt on the cobblestone. Because there are no transitions from dirt directly to cobblestone, the Terrain tool first inserts transitions to sand and from there to cobblestone. Neat!

## 8.4 Final Words

Now you should have a pretty good idea about how to use this tool in your own project. A few things to keep in mind:

- Currently the tool requires all terrain types to be part of the same tileset. You can have multiple tilesets with terrain in your map, but the tool can't perform automatic transitions from a terrain from one tileset to a terrain in another tileset. This usually means you may have to combine several tiles into one image.
- Since defining the terrain information can be somewhat laboursome, you'll want to avoid using embedded tilesets so that terrain information can be shared among several maps.
- The Terrain tool works fine with isometric maps as well. To make sure the terrain overlay is displayed correctly, set up the *Orientation*, *Grid Width* and *Grid Height* in the tileset properties.
- The tool will handle any number of terrain types and each corner of a tile can have a different type of terrain. Still, there are other ways of dealing with transitions that this tool can't handle. Also, it is not able to edit multiple layers at the same time. For a more flexible, but also more complicated way of automatic tile placement, check out *Automapping*.
- I'm maintaining a collection of tilesets that contain transitions that are compatible with this tool on OpenGameArt.org.

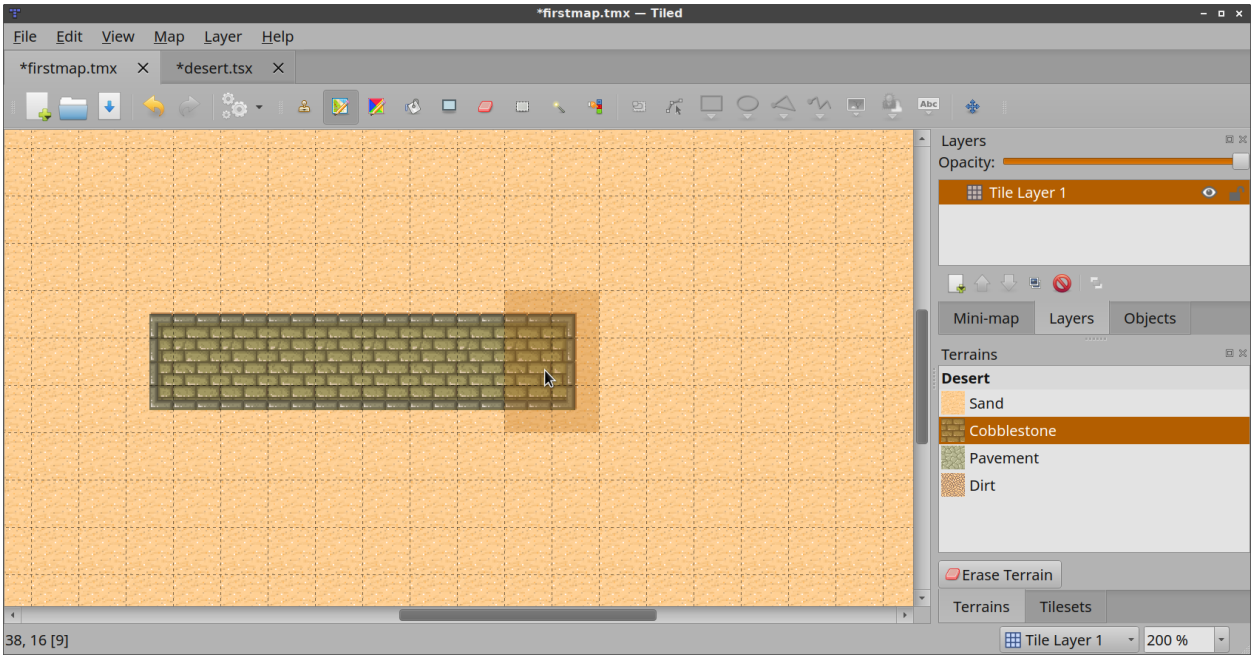



Fig. 5: Drawing Cobblestone

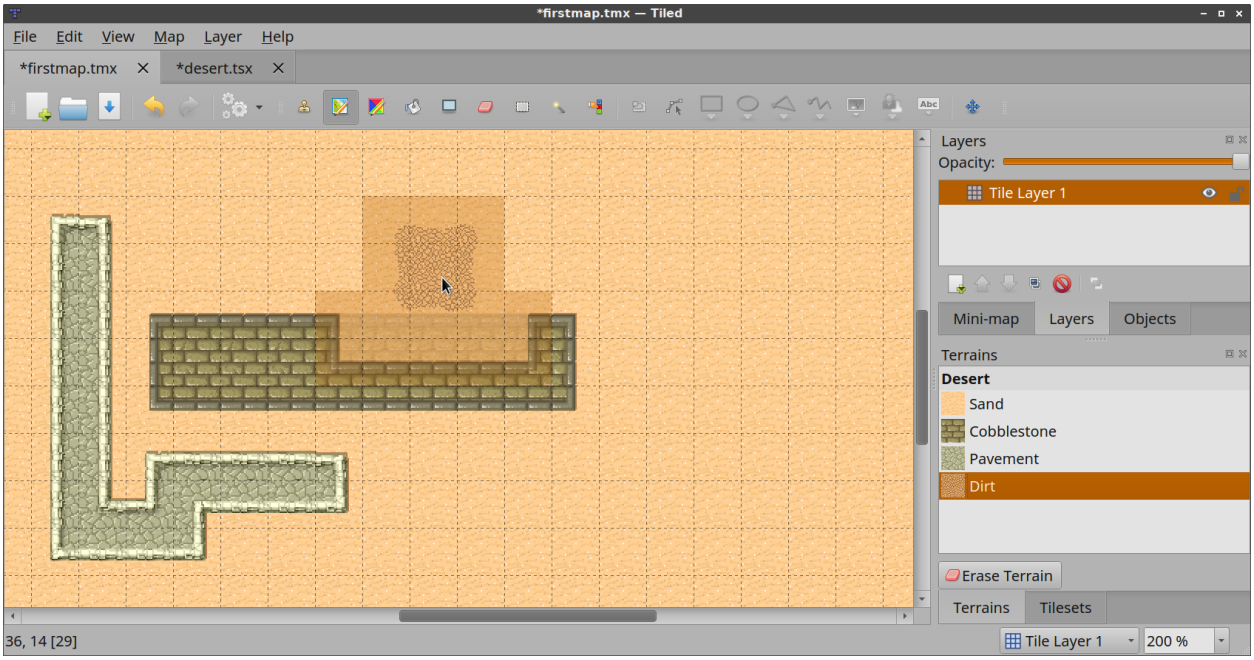



Fig. 6: Drawing Dirt

CHAPTER 9

# Using Wang Tiles

Wang tiles are similar in concept to *Terrains*. They are, however, more focused on filling larger areas without repetition. One defines the edge and corner colors of tiles in a tileset. This information can then be used when filling, or brushing to allow for smooth, non-repetitive transitions between tiles. In most cases this tiling is random, and based on color probability. More info on Wang tiles can be found here.

To demonstrate how to use Wang tiles, we will describe the steps necessary to recreate `walkways.tsx` example tileset.

## 9.1 Defining Wang Tile Info

After making the tileset, from the tileset editor, click the Wang Sets button.

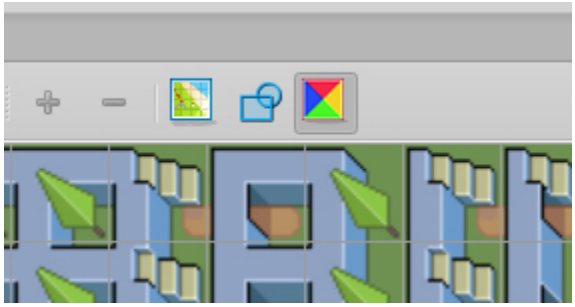


Fig. 1: Wang Set Button

A single tileset can have many Wang sets. Create a new Wang set using the plus button at the bottom of the Wang set view.

You can now edit the properties of the Wang set. Important for us is edge and corner count. This will determine how the set is defined, and how it behaves. This tileset is a 3 edge Wang set.

Now in the complete pattern set will generate in the *Patterns* tab below the Wang set view. For the set to be complete (though this is unnecessary), each pattern must be used at least once.


Fig. 2: Wang Set View

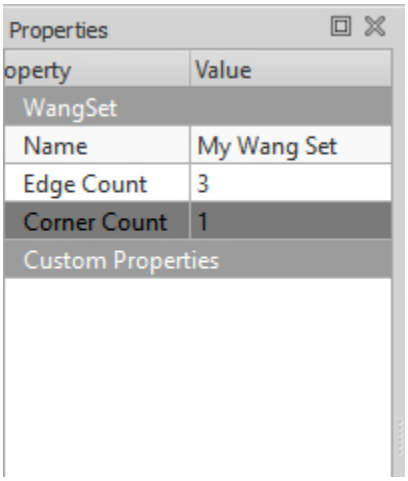



Fig. 3: Wang Set Properties

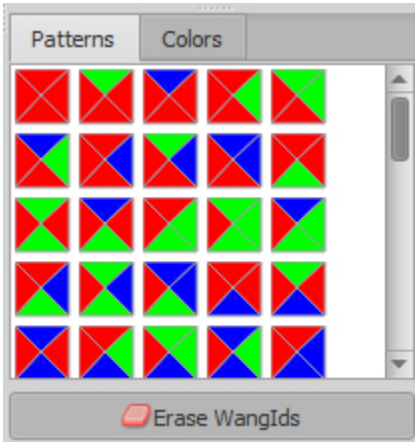



Fig. 4: Pattern View

Once a pattern is selected, you can paint it directly onto the tileset. Similar to when using the Stamp Brush, `Z` and `Shift + Z` can be used to rotate the pattern 90 degrees clockwise and counterclockwise respectively. `X` and `Y` flip the pattern horizontally, and vertically respectively.

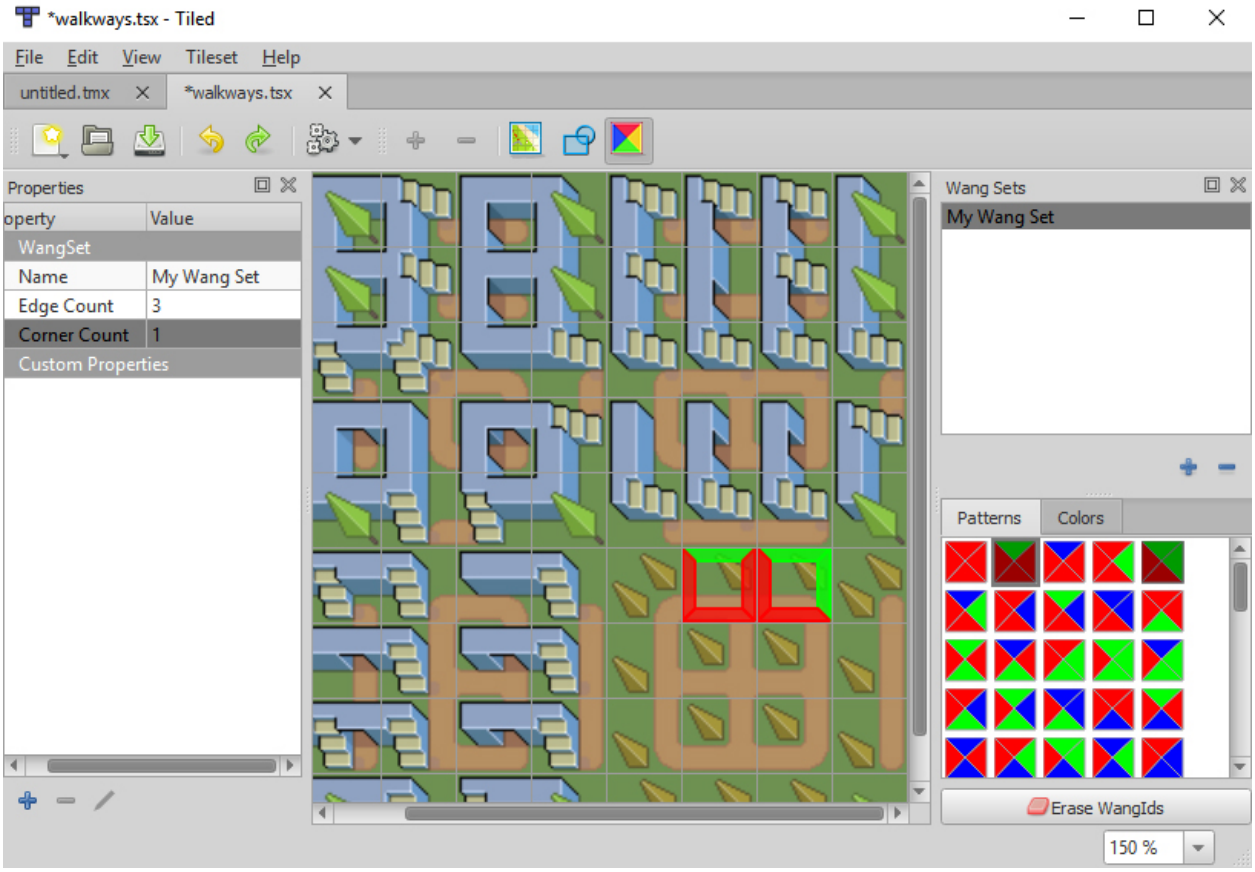



Fig. 5: Painting on a Pattern

In the other tab, there is the *Colors* view. This gives you access to edit properties and assign with each individual color of a set.

Using these methods, assign each tile matching all the edges. After this is done, the set is ready to be used with all the Wang methods.

## 9.2 Editing With Wang Methods

There are two main ways to use Wang tiles:

- Activating the *Wang mode*
- Using the *Wang brush*

### 9.2.1 Wang Mode

Similar to the random mode, the Stamp Brush, and Bucket Fill tools can use Wang methods to fill. With the Wang mode activated, each cell will be randomly chosen from all those in the Wang set which match all adjacent edge/corner colors.

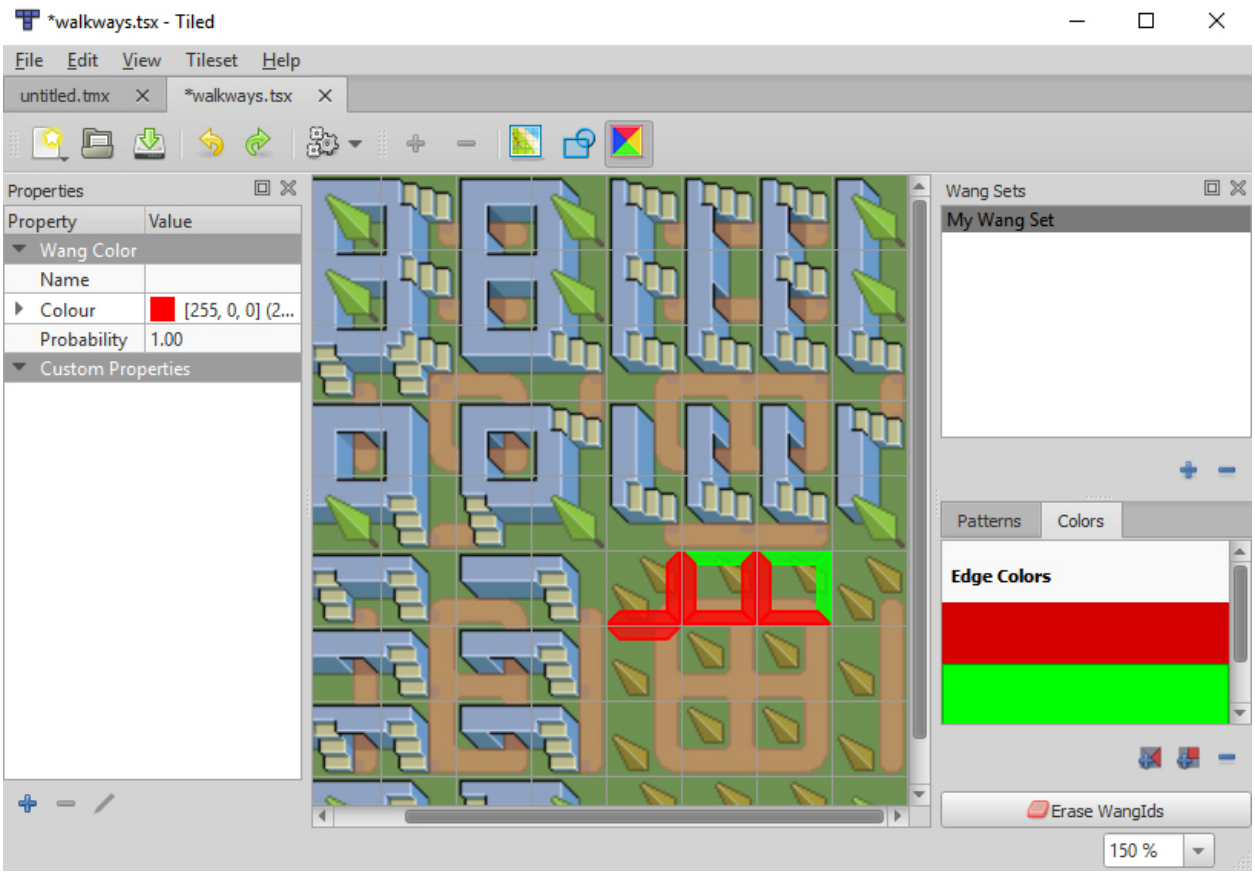



Fig. 6: Painting Individual Edge

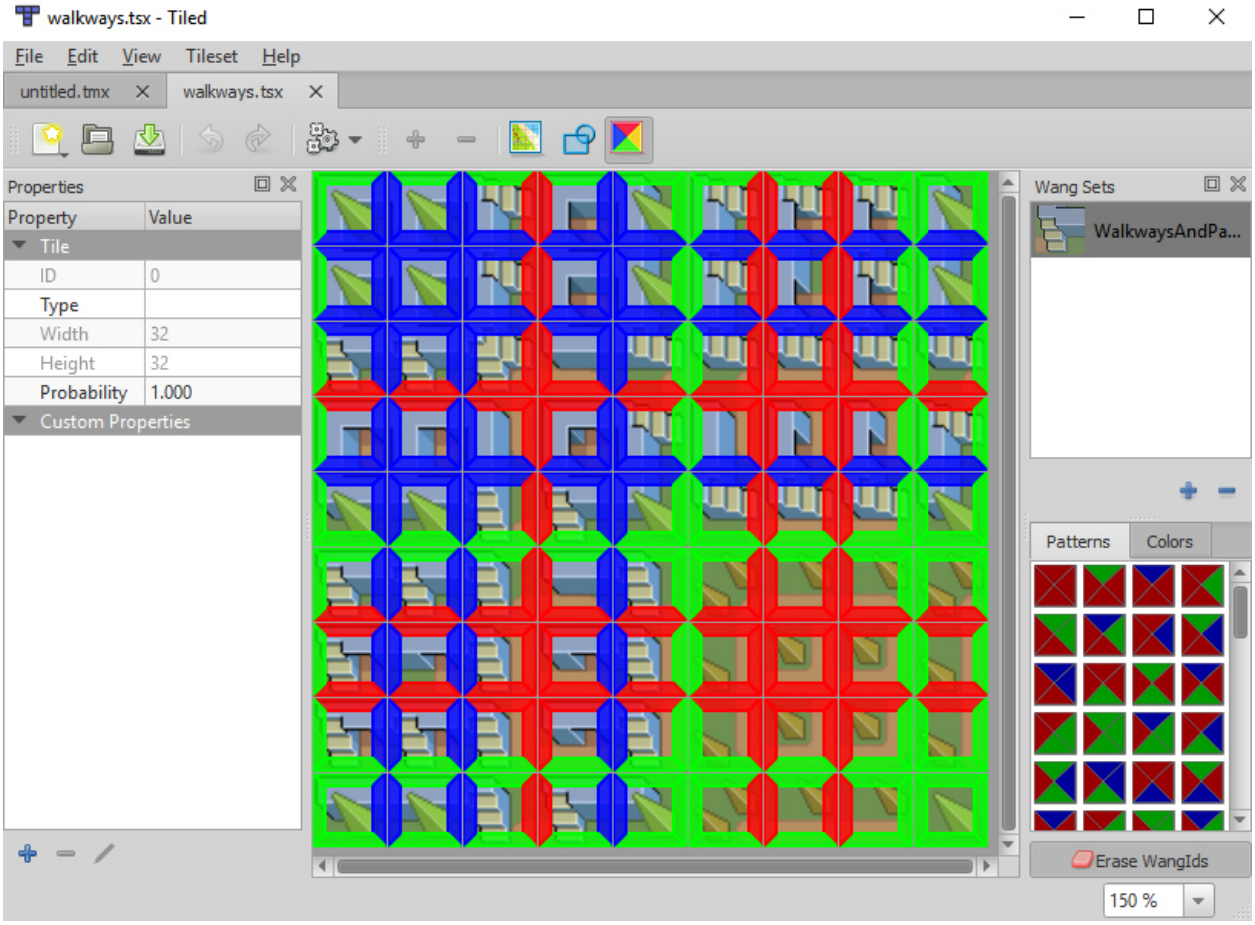



Fig. 7: Completely Assigned Wang Set

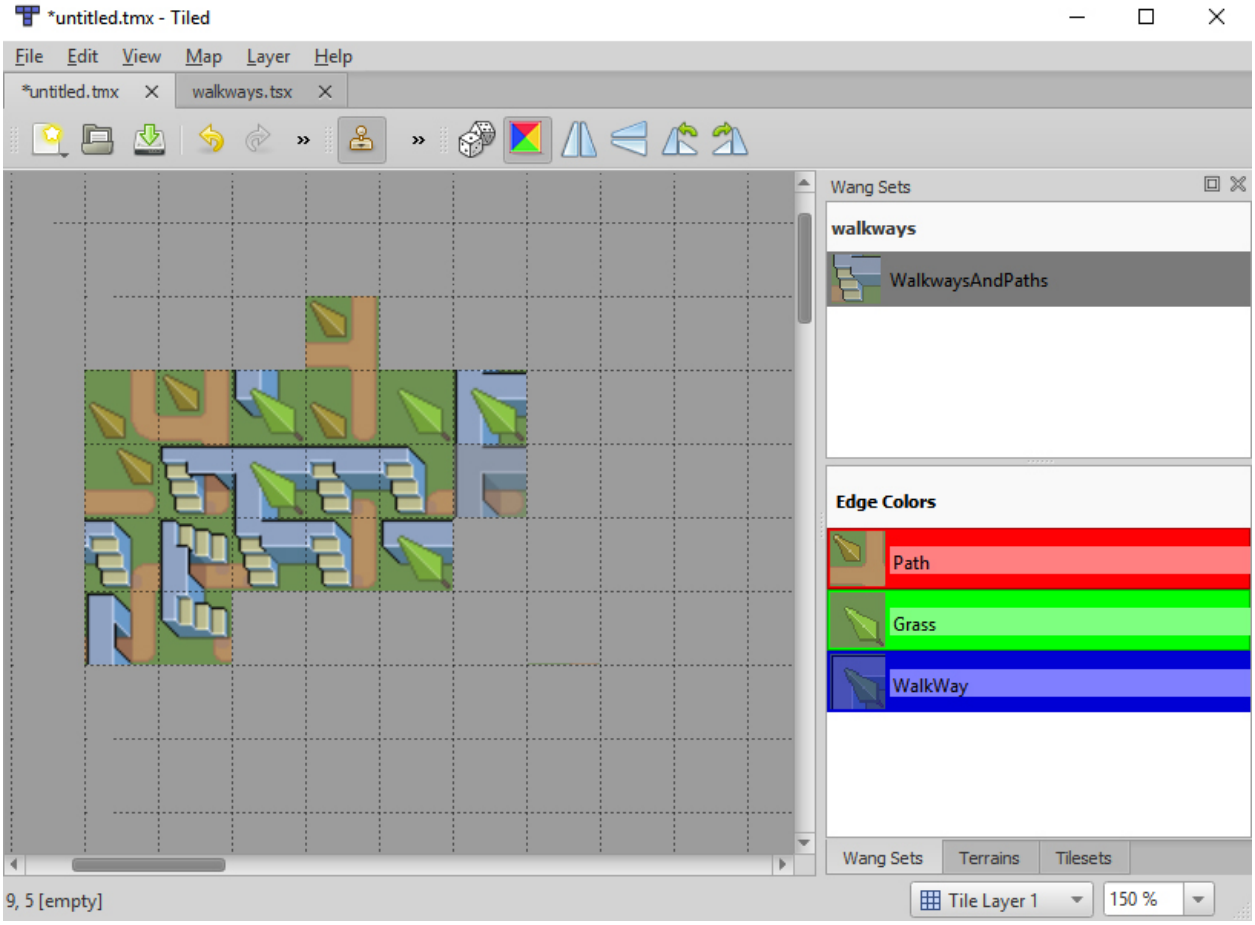



Fig. 8: Stamp Brush with Wang Fill Mode Enabled

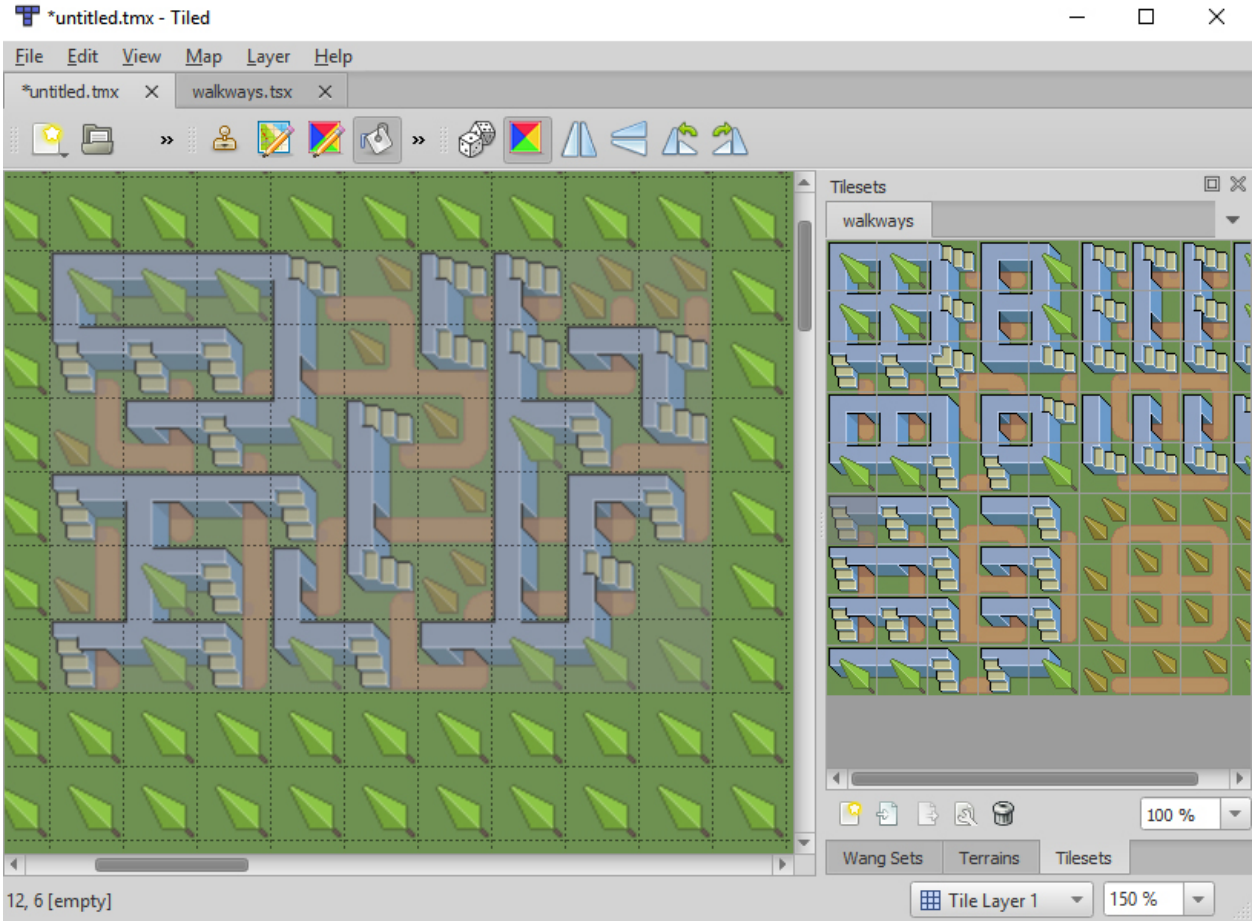



Fig. 9: Bucket Fill with Wang Fill Mode Enabled

### 9.2.2 Wang Brush

There is also the *Wang Brush*, which works very much like the *Terrain Brush*. This tool changes the edge/color patterns of the adjacent cells, to match a selected color. If no tiles exist in the Wang set of a particular pattern, the area can not be painted.

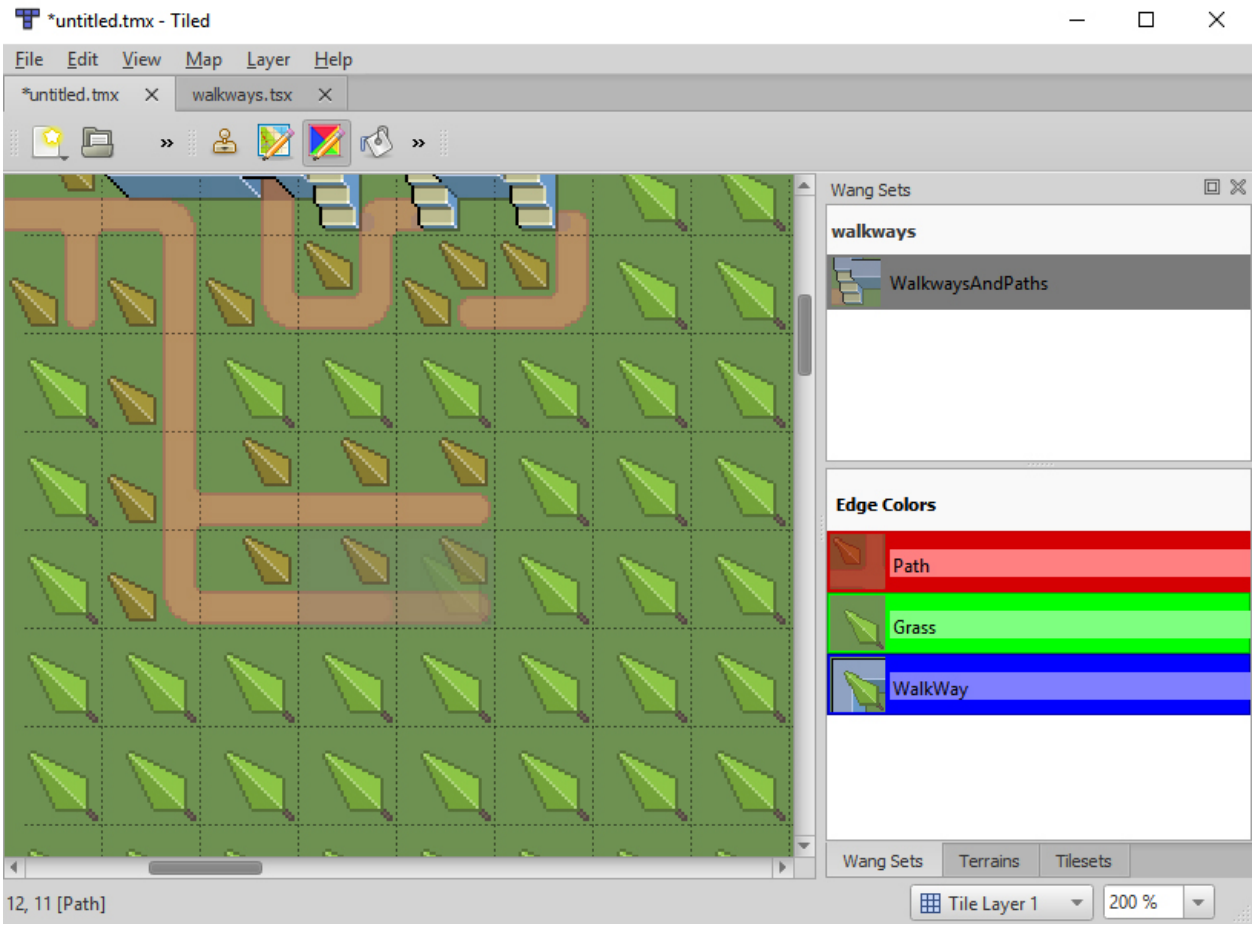



Fig. 10: Wang Brush

## 9.3 Customizing Wang Colors

Each Wang color can be customized to become more recognizable. As well, the probability of each color can be adjusted, such that with the Wang mode it will show up more often in filling or brushing.

### 9.3.1 Color Appearance

The name, image, and of course color can be changed to alter the appearance of a Wang color. This image can be changed be selecting a color, then right clicking on the tile whose image is desired, and selecting *Set Wang Color Image*.

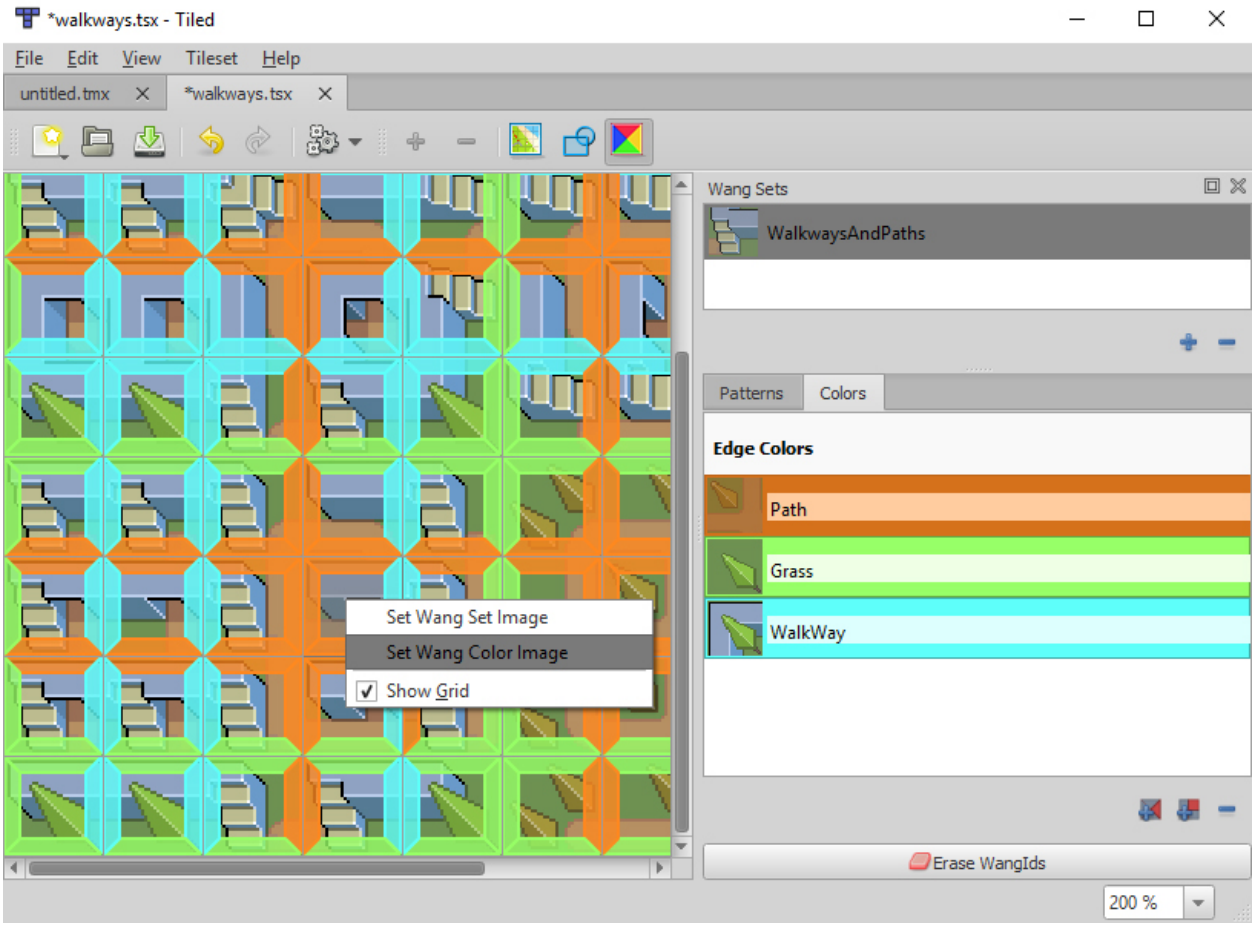


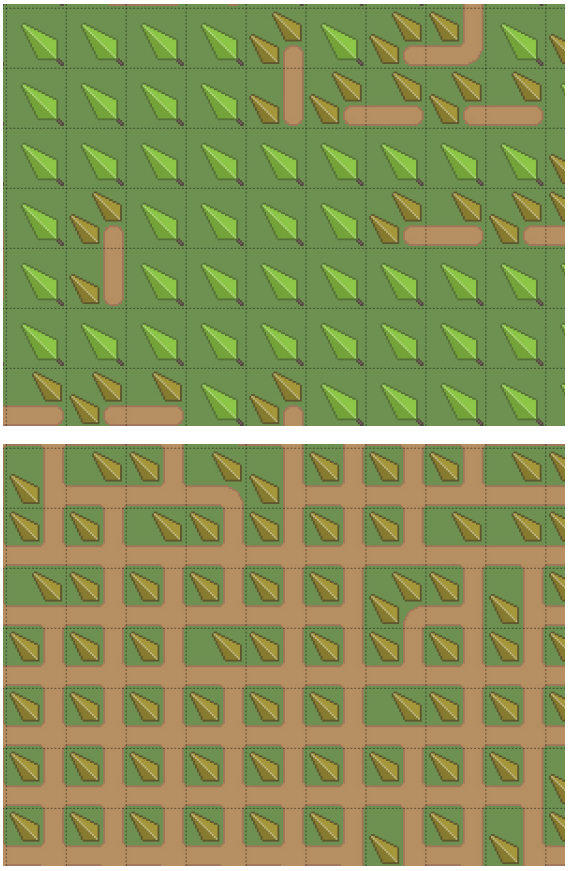

Fig. 11: Selecting Wang color image

| Property | Value |
|---|---|
| Wang Color | |
| Name | Grass |
| Colour | [0, 255, 0] (255) |
| Probability | 0.10 |
| Custom Properties | |

Fig. 12: The other values can be changed from the properties view.

### 9.3.2 Probability

When choosing a tile with Wang methods, all tiles with a valid Wang pattern are considered. They are given a weight based on their edge/corner colors' probabilities. Then one is selected at random, while considering this weight. The weight is the product of all the probabilities.



Left shows path with probability 0.1, right shows path with probability 10.

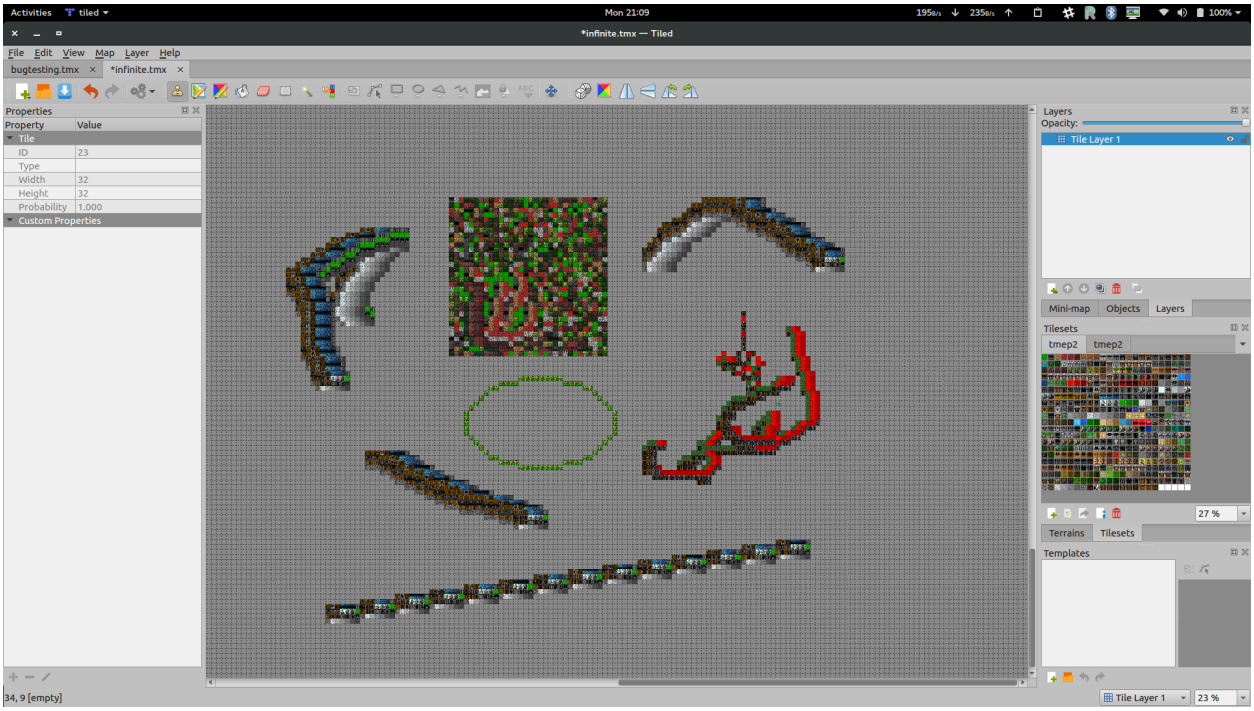
## 9.4 Standard Wang Sets

Some typical Wang sets are 2-corner, 2-edge, and blob. Wang tiles in Tiled support up to 15 edge and 15 corner colors in a single set.

CHAPTER 10

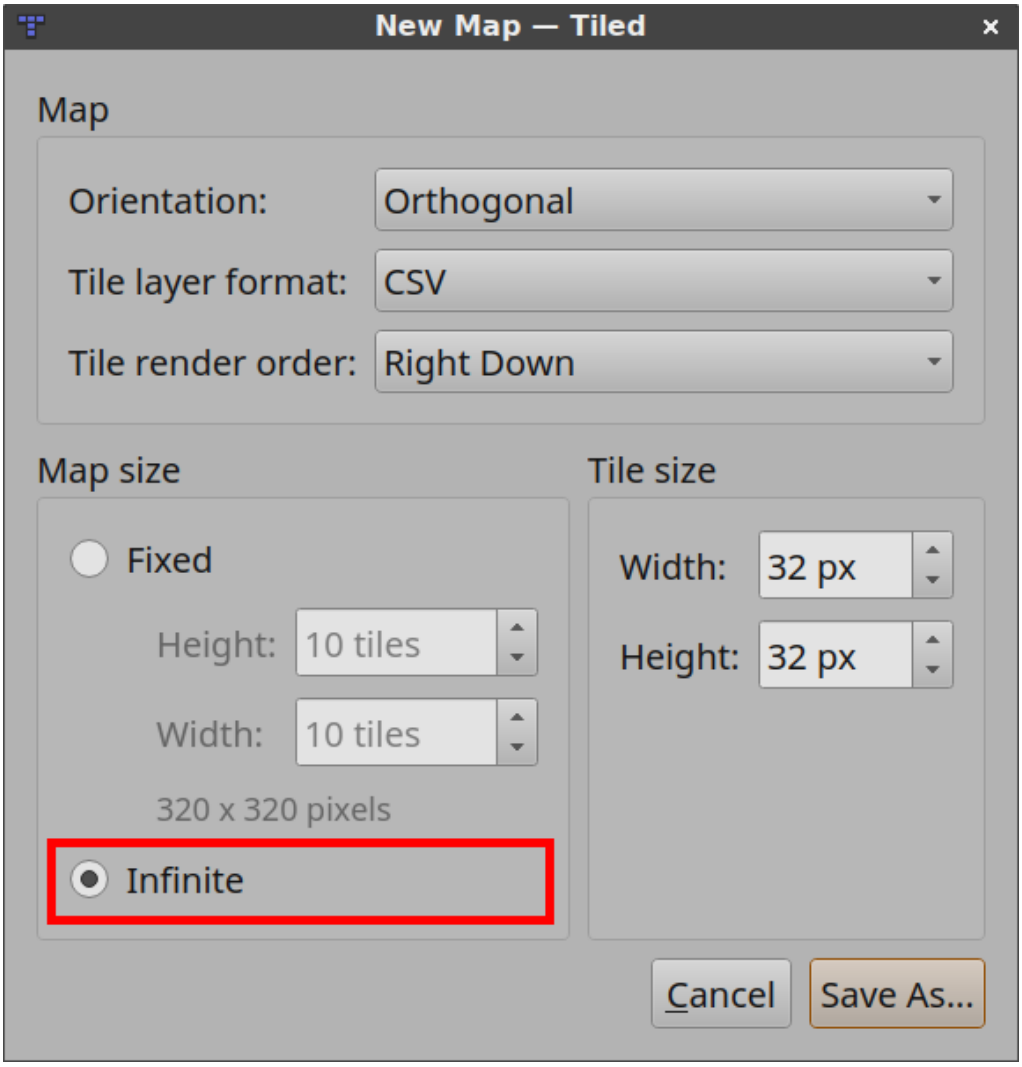
# Using Infinite Maps

Infinite maps give you independence from bounds of the map. The canvas is “auto-growing”, which basically means, that you have an infinite grid which can be painted upon without worrying about the width and height of the map. The bounds of a particular layer get expanded whenever tiles are painted outside the current bounds.




## 10.1 Creating an Infinite Map

In the order to create an infinite map, make sure the ‘Infinite’ option is selected in New Map dialog.


New Map — Tiled
Map
Orientation:
Orthogonal
Tile layer format:
CSV
Tile render order:
Right Down
Map size
Fixed
Height:
10 tiles
Width:
10 tiles
320 x 320 pixels
Infinite
Tile size
Width:
32 px
Height:
32 px
Cancel
Save As...

The newly created map will then have an infinite canvas.

## 10.2 Editing the Infinite Map

Except for the *Bucket Fill Tool*, all tools works exactly in the same way as in the fixed-size maps. The Bucket Fill Tool fills the current bounds of that particular tile layer. These bounds get increased upon further painting of that tile layer.

## 10.3 Conversion from Infinite to Finite Map and Vice Versa

In the map properties, you can toggle whether the map should be infinite or not. When converting from infinite to a finite map, the width and height of the final map are chosen on the basis of bounds of all the tile layers.

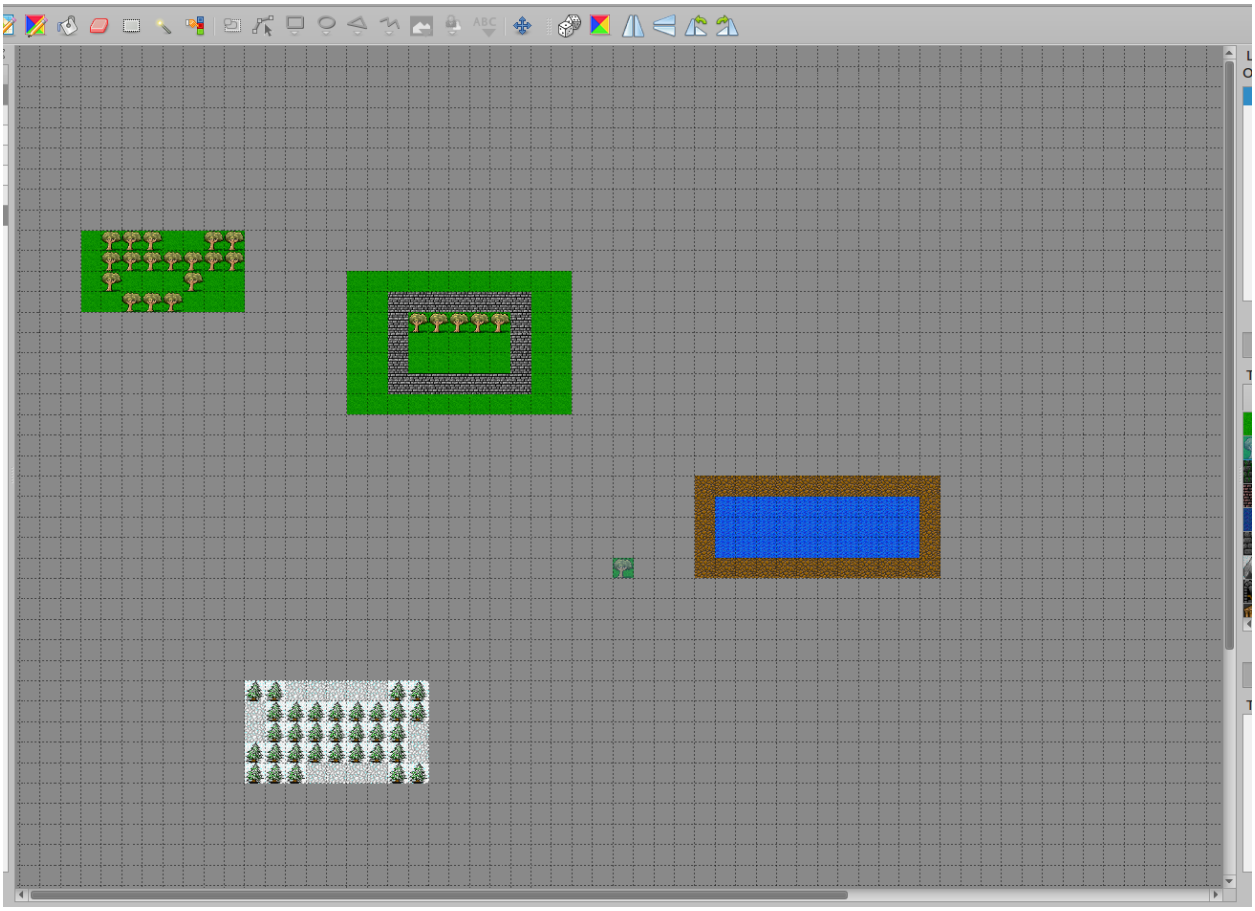


Fig. 1: The Initial Infinite Map

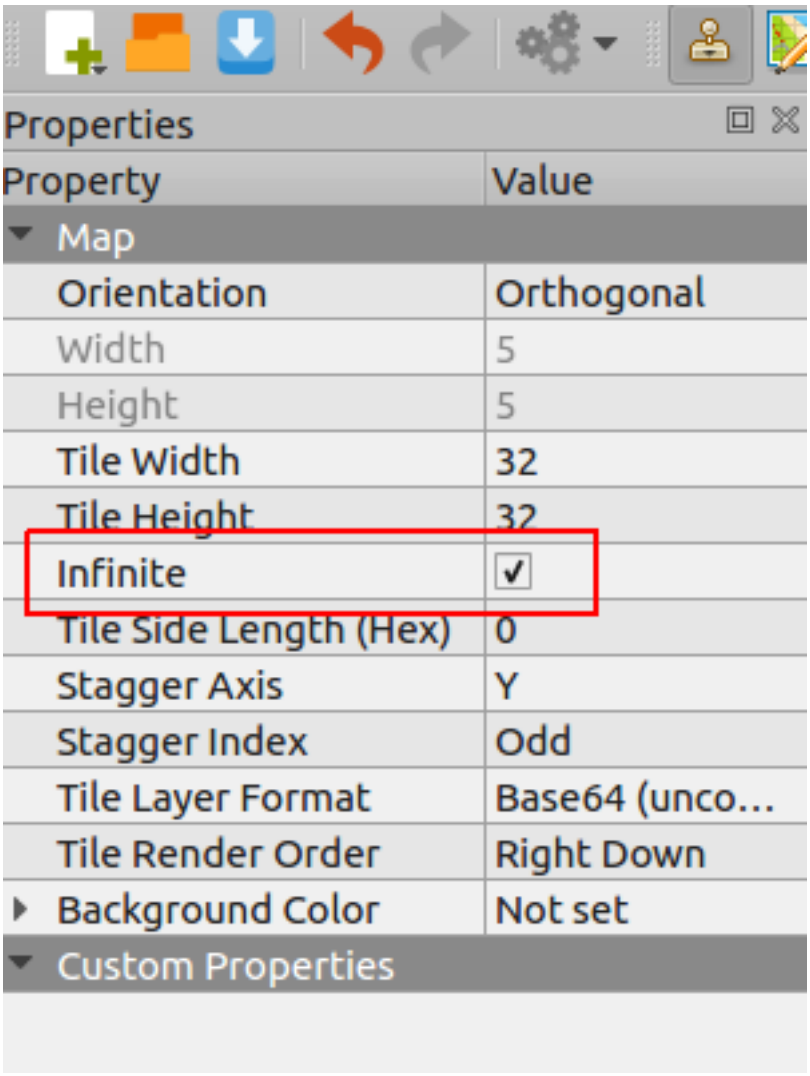



Fig. 2: Unchecking the Infinite property in Map Properties

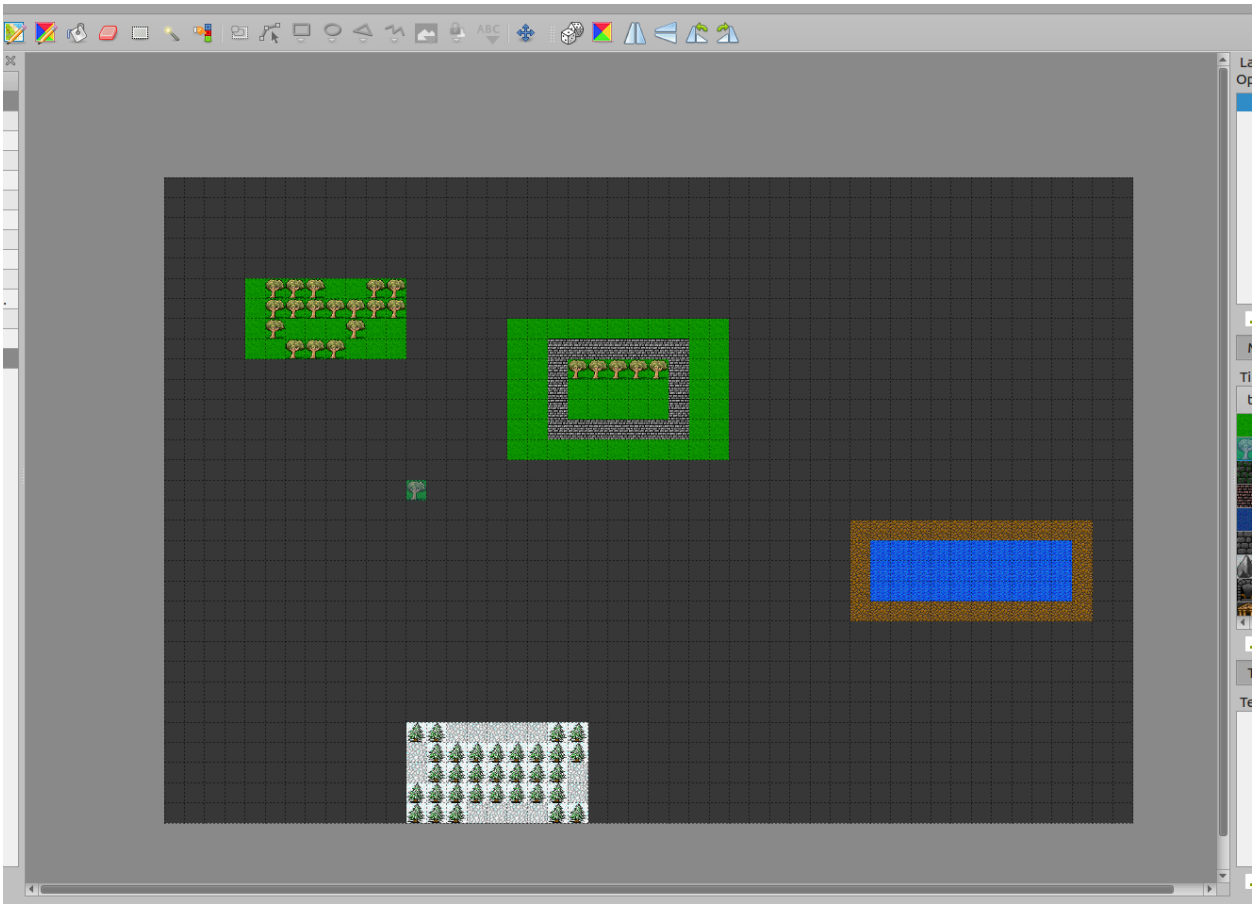


Fig. 3: The Converted Map

CHAPTER 11

# Working with Worlds

Sometimes a game has a large world which is split over multiple maps to make the world more digestible by the game (less memory usage) or easier to edit by multiple people (avoiding merge conflicts). It would be useful if the maps from such a world could be seen within the same view, and to be able to quickly switch between editing different maps. Defining a world allows you to do exactly that.

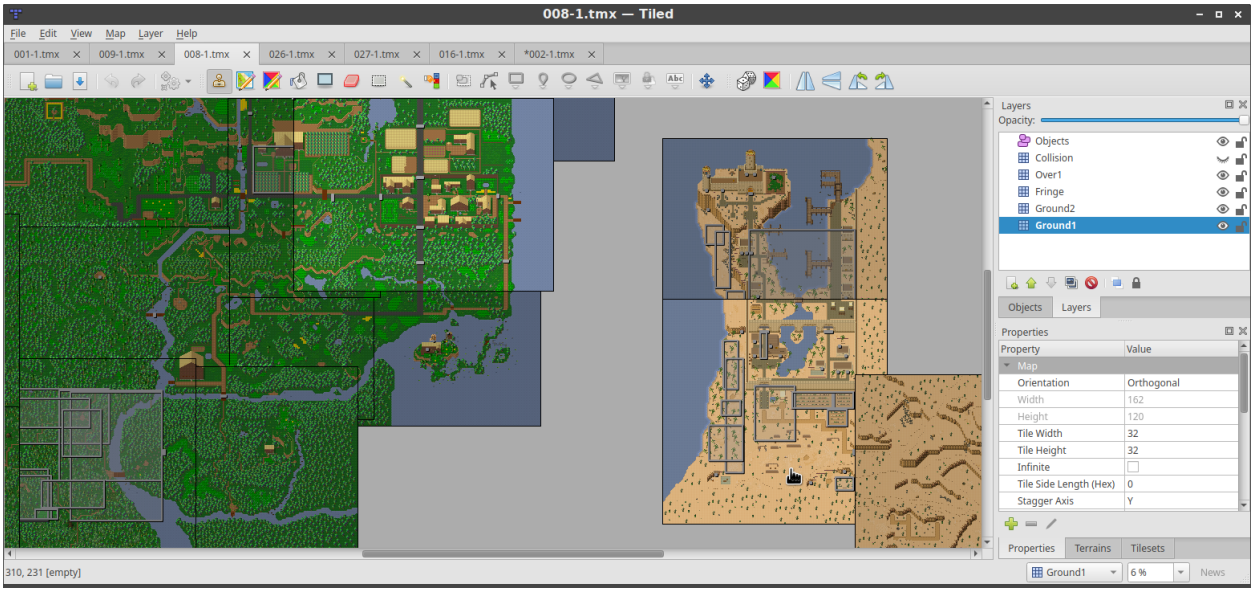



Fig. 1: Many maps from The Mana World shown at once.

## 11.1 Defining a World

A world is defined in a `.world` file, which is a JSON file that tells Tiled which maps are part of the world and at what location. Worlds can be created by using the *Map* > *New World...* action.

You may also create *.world files* by hand. Here is a simple example of a world definition, which defines the global position (in pixels) of three maps:

```
{
    "maps": [
        {
            "fileName": "001-1.tmx",
            "x": 0,
            "y": 0
        },
        {
            "fileName": "002-1.tmx",
            "x": 0,
            "y": 3200
        },
        {
            "fileName": "006-1.tmx",
            "x": 3840,
            "y": 4704
        }
    ],
    "type": "world"
}
```

Once defined, a world needs to be loaded by choosing *Map* > *Load World...* from the menu. Multiple worlds can be loaded at the same time, and worlds will be automatically loaded again when Tiled is restarted.

When is map is opened, Tiled checks whether it is part of any of the loaded worlds. If so, any other maps in the same world are loaded as well and displayed alongside the opened map. You can click any of the other maps to open them for editing, which will switch files while keeping the view in the same position.

Worlds are reloaded automatically when their file is changed on disk.

## 11.2 Editing Worlds

Once you have loaded a world, you can select the 'World Tool' from the toolbar to add, remove and move maps within the world.

**Adding Maps** Click the 'Add the current map to a loaded world' button on the toolbar, from the dropdown menu select the world you want to add it to. To add a different map to the current world, you can use the 'Add another map to the current world' button from the toolbar. Alternatively, both actions can be accessed by rightclicking in the map editor.

**Removing Maps** Hit the 'Remove the current map from the current world' button on the toolbar. Alternatively, rightclick a map in the map editor and select the 'Remove ... from World ...' action from the context menu.

**Moving Maps** Simply drag around maps within the map editor. You can abort moving a map by hitting 'Escape' or by right-clicking.

Alternatively you can use the arrow keys to move the current selected map - holding Shift will perform bigger steps.

**Saving World files** You can save manipulated world files by using the *Map* > *Save World* menu. Worlds will also automatically be saved if you launch any external tool that has the 'Save Map Before Executing' option enabled.

## 11.3 Using Pattern Matching

For projects where the maps follow a certain naming style that allows the location of each map in the world to be derived from the file name, a regular expression can be used in combination with a multiplier and an offset.

---

**Note:** Currently no interface exists in Tiled to define a world using pattern matching, nor can it be modified. World files with patterns have to be manually edited.

---

Here is an example:

```
{
    "patterns": [
        {
            "regexp": "ow-p0*(\\d+)-n0*(\\d+)-o0000\\.tmx",
            "multiplierX": 6400,
            "multiplierY": 6400,
            "offsetX": -6400,
            "offsetY": -6400
        }
    ],
    "type": "world"
}
```

The regular expression is matched on all files that live in the same directory as the world file. It captures two numbers, the first is taken as `x` and the second as `y`. These will then be multiplied by `multiplierX` and `multiplierY` respectively, and finally `offsetX` and `offsetY` are added. The offset exists mainly to allow multiple sets of maps in the same world to be positioned relative to each other. The final value becomes the position (in pixels) of each map.

A world definition can use a combination of manually defined maps and patterns.

## 11.4 Showing Only Direct Neighbors

Tiled takes great care to only load each map, tileset and image once, but sometimes the world is just too large for it to be loaded completely. Maybe there is not enough memory, or rendering the entire map is too slow.

In this case, there is an option to only load the direct neighbors of the current map. Add `"onlyShowAdjacentMaps": true` to the top-level JSON object.

To make this possible, not only the position but also the size of each map needs to be defined. For individual maps, this is done using `width` and `height` properties. For patterns, the properties are `mapWidth` and `mapHeight`, which default to the defined multipliers for convenience. All values are in pixels.

---

**Note:** In the future, I will probably change this option to allow specifying a distance around the current map in which other maps are loaded.

---

CHAPTER 12

# Using Commands

The Command Button allows you to create and run shell commands (other programs) from Tiled.

You may setup as many commands as you like. This is useful if you edit maps for multiple games and you want to set up a command for each game. Or you could setup multiple commands for the same game that load different checkpoints or configurations.

## 12.1 The Command Button

It is located on the main toolbar to the right of the redo button. Clicking on it will run the default command (the first command in the command list). Clicking the arrow next to it will bring down a menu that allows you to run any command you have set up, as well as an option to open the Edit Commands dialog. You can also find all the commands in the File menu.

Apart from this, you can set up custom keyboard shortcuts for each command.

## 12.2 Editing Commands

The 'Edit Commands' dialog contains a list of commands. Each command has several properties:

**Name** The name of the command as it will be shown in the drop down list, so you can easily identify it.

**Executable** The executable to run. It should either be a full path or the name of an executable in the system PATH.

**Arguments** The arguments for running the executable.

**Working directory** The path to the working directory.

**Shortcut** A custom key sequence to trigger the command. You can use 'Clear' to reset the shortcut.

**Output in Debug Console** If this is enabled, then the output (stdout and stderr) of this command will be displayed in the Debug Console. You can find the Debug Console in *View > Views and Toolbars > Debug Console*.

**Save map before executing** If this is enabled, then the current map will be saved before executing the command.

**Enabled** A quick way to disable commands and remove them from the drop down list. The default command is the first enabled command.

Note that if the executable or any of its arguments contain spaces, these parts need to be quoted.

### 12.2.1 Substituted Variables

In the executable, arguments and working directory fields, you can use the following variables:

**`%mapfile`** the current maps full path.

**`%mappath`** the full folder path in which the map is located. (since Tiled 0.18)

**`%objecttype`** the type of the currently selected object, if any. (since Tiled 0.12)

**`%objectid`** the ID of the currently selected object, if any. (since Tiled 0.17)

**`%layername`** the name of the currently selected layer. (since Tiled 0.17)

For the working directory field, you can additionally use the following variable:

**`%executablepath`** the path to the executable.

## 12.3 Example Commands

Launching a custom game called "mygame" with a -loadmap parameter and the mapfile:

```
mygame -loadmap %mapfile
```

On Mac, remember that Apps are folders, so you need to run the actual executable from within the `Contents/MacOS` folder:

```
/Applications/TextEdit.app/Contents/MacOS/TextEdit %mapfile
```

Or use `open` (and note the quotes since one of the arguments contains spaces):

```
open -a "/Applications/CoronaSDK/Corona Simulator.app" /Users/user/Desktop/project/
↪main.lua
```

Some systems also have a command to open files in the appropriate program:

- OSX: `open %mapfile`
- GNOME systems like Ubuntu: `gnome-open %mapfile`
- FreeDesktop.org standard: `xdg-open %mapfile`

CHAPTER 13

# Automapping

## 13.1 What is Automapping?

Automapping is an advanced tool to automatically search certain combinations of tiles across layers in a map and to replace these parts with another combination. This allows the user to draw structures quickly and Automapping will generate a complex scenario from them, which would have taken much more time if manually crafted.

The goal of Automapping is that you only need to draw within one layer and everything else is setup for you. This brings some advantages:

- **Working speed** - you need less time to setup a map.
- **Less errors** - the main reason is to reduce the error rate. If you have setup the rules properly, there are no hidden errors.

### 13.1.1 External Links

- Automapping explained for Tiled 0.9 and later (YouTube)
- Examples on Automapping
- Tiled Map Editor Tutorial Part Three: AutoMap (YouTube)

## 13.2 Setting it Up

The Automapping feature looks for a text file called 'rules.txt' in the folder where the current map is located. Each line in this text file is either

- a path to a **rulefile**
- or a path to another textfile which has the same syntax (i.e. in another directory)
- or is a comment which is indicated by **#** or **//**

A **rulefile** is a standard map file, which can be read and written by tiled (*.tmx). In one rulefile there can be multiple defined rules.

An automapping **rulefile** consists of 4 major parts:

1. The definition of regions describes which locations of the rulemap are actually used to create Automapping rules.
2. The definition of inputs describes which kind of pattern the working map will be searched for.
3. The definition of outputs describes how the working map is changed when an input pattern is found.
4. The map properties are used to fine-tune the input pattern localization and the output of all rules within this rules file.

### 13.2.1 Defining the Regions

There must be either a tile layer called **regions** or there must be both tile layers **regions_input** and **regions_output**. Using the **regions** layer, the region defined for input and output is the same. Using the different layers **regions_input** and **regions_output** delivers the possibility to have different regions for the input section and the output section. The region layer(s) are only used to mark regions where an Automapping rule exists. Therefore, it does not matter which tiles are used in this layer, since these tiles are just used to define a region. So either use any tile or no tile at a coordinate to indicate if that coordinate belongs to a rule or if it doesn't.

If multiple rules are defined in one rulemap file, the regions must not be adjacent. That means there must be at least one tile of unused space in between two rules. If the regions are adjacent (coherent) then both regions are interpreted as one rule.

#### Multiple Rules in One Rulefile

Multiple rules are possible in one rulemap. However, if you want to have the rules applied in a certain sequence, you should use multiple **rulefiles** and define the sequence within the **rules.txt** file. As of now there also is a certain sequence within one rulemapfile. Generally speaking the regions with small y value come first. If there are regions at the same y value, then the x value is taken into account. On orthogonal maps this ordering scheme is the same as for reading in most western countries (Left to right, top to bottom). The order within one rulemap may be changed later, once tiled is capable of utilizing multiple threads/processors. So if you want to rely on a certain sequence, use different rulemaps and order these in the rules.txt

### 13.2.2 Definition of Inputs

Inputs are generally defined by tile layers which name follows this scheme:

**input[not][index]_name**

where the **[not]** and **[index]** are optional. After the first underscore there will be the name of the input layer. The input layer name can of course include more underscores.

The **name** determines which layer on the working map is examined. So for example the layer *input_Ground* will check the layer called *Ground* in the working map for this rule. *input_test_case* will check the layer *test_case* in the working map for this rule.

Multiple layers having the same name and index is explicitly allowed and is intended. Having multiple layers of the same name and index , will allow you to define different possible tiles per coordinate as input.

The index is used to create complete different input conditions. All layers having the same index are taken into account for forming one condition. Each of these conditions are checked individually.

1. index must not contain an underscore.

2. index must not start with *not*
3. index may be empty.

If there are tiles in the standard input layers one of these tiles must be there to match the rule. The optional **[not]** inverts the meaning of that layer. So if there are **inputnot** layers, the tiles placed on them, must not occur in the working map at the examined region to make a rule match. Within one rule you can combine the usage of both input and inputnot layers to make rules input conditions as accurate as you need or as fuzzy as you need.

### 13.2.3 Definition of Outputs

Outputs are generally defined by layers whichs name follows this scheme:

**output[index]_name**

which is very similar to the input section. At first there must be the word output. Then optionally an **[index]** may occur. After the first underscore there will be the name of the target layer. The target layer name can of course include more underscores.

All layers of the same index are treated as one possible output. So the intention of indexes in the outputs of rules is only used for random output.

The indexes in the output section have nothing to do with the indexes in the input section, they are independent. In the output section they are used for randomness. In the input section they are used to define multiple possible layers as input. So when there are multiple indexes within one rule, the output will be chosen fairly (uniformly distributed) across all indexes. So a dice will be rolled and one index is picked. All of the output layers carrying this index will be put out into the working map then.

Note that the output is not being checked for overlapping on itself. This can be achieved by setting the map property **NoOverlappingRules** to true.

### 13.2.4 Map Properties

The following map properties can be used to customize the behavior of the rules in a **rulefile**:

**DeleteTiles** This map property is a boolean property: it can be true or false. If rules of this rulefile get applied at some location in your map, this map property determines if all other tiles are deleted before applying the rules. Consider a map where you have multiple layers. Not all layers are filled at all places. In that case all tiles of all layers should be cleared, so afterwards there are only the tiles which are defined by the rules. Since when not all tiles are cleared before, you will have still tiles from before at these places, which are not covered by any tile.

**AutomappingRadius** This map property is a number: 1, 2, 3 ... It determines how many tiles around your changes will be checked as well for redoing the Automapping at live Automapping.

**MatchOutsideMap** This map property determines whether rules can match even when their input region falls partially outside of a map. By default it is `false` for bounded maps and `true` for infinite maps. In some cases it can be useful to enable this also for bounded maps. Tiles outside of the map boundaries are simply considered empty unless one of either **OverflowBorder** or **WrapBorder** are also true.

Tiled 1.0 and 1.1 behaved as if this property was `true`, whereas older versions of Tiled have behaved as if this property was `false`.

**OverflowBorder** This map property customizes the behavior intended by the **MatchOutsideMap** property. When this property is `true`, tiles outside of the map boundaries are considered as if they were copies of the nearest inbound tiles, effectively "overflowing" the map's borders to the outside region.

When this property is `true`, it implies **MatchOutsideMap**. Note that this property has no effect on infinite maps (since there is no notion of border).

**WrapBorder** This map property customizes the behavior intended by the **MatchOutsideMap** property. When this property is `true`, the map effectively "wraps" around itself, making tiles on one border of the map influence the regions on the other border and vice versa.

When this property is `true`, it implies **MatchOutsideMap**. Note that this property has no effect on infinite maps (since there is no notion of border).

If both **WrapBorder** and **OverflowBorder** are `true`, **WrapBorder** takes precedence over **OverflowBorder**.

**NoOverlappingRules** This map property is a boolean property: A rule is not allowed to overlap on itself.

These properties are map wide, meaning it applies to all rules which are part of the rulemap. If you need rules with different properties you can use multiple rulemaps.

### 13.2.5 Layer Properties

The following properties are supported on a per-layer basis:

**StrictEmpty** This layer property is a boolean property. It can be added to **input** and **inputnot** layers to customize the behavior for empty tiles within the input region.

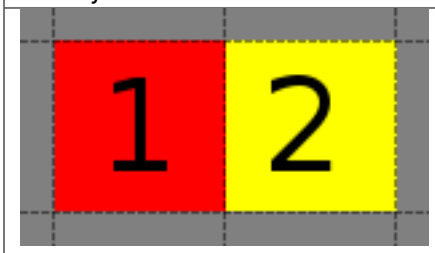
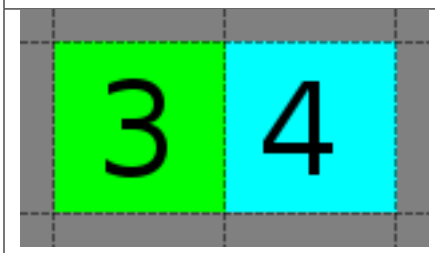
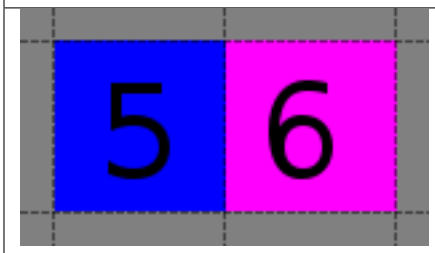
In "StrictEmpty" mode, empty tiles in the input region match empty tiles in the set layer. So when an "input" layer contains an empty tile within the input region, this means an empty tile is allowed at that location. And when an "inputnot" layer contains an empty tile within the input region, it means an empty tile is not allowed at that location.

## 13.3 Examples

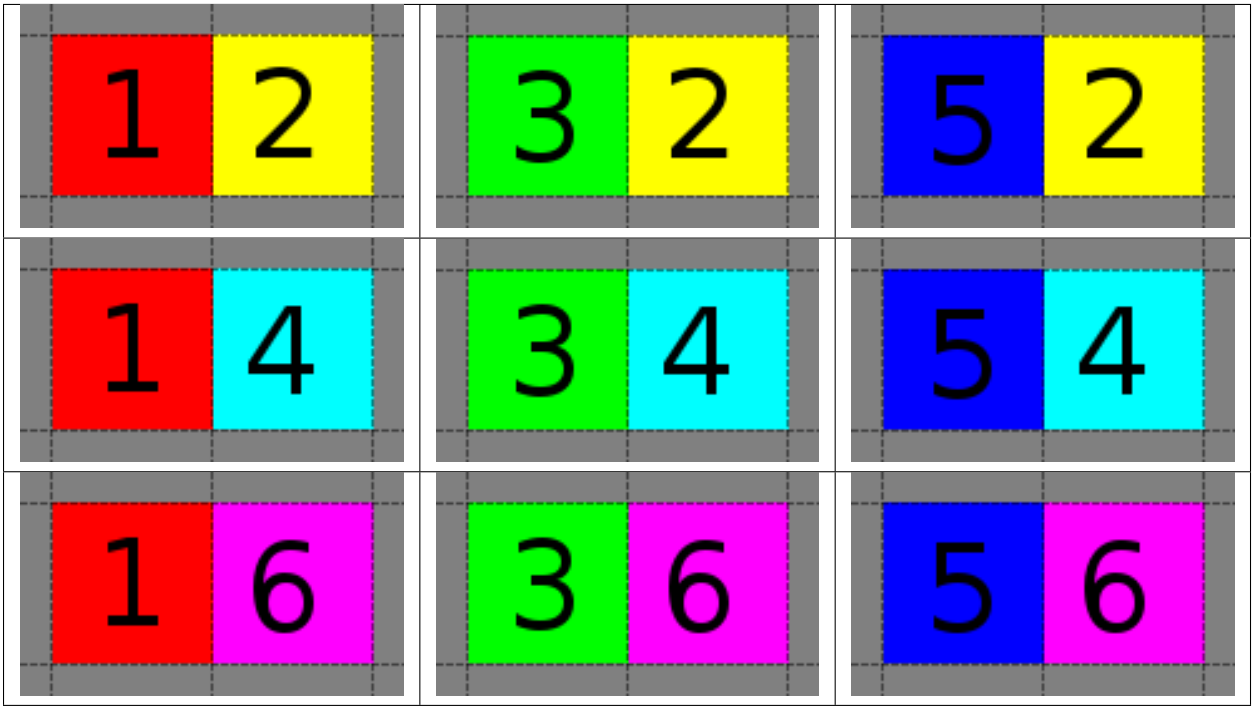
### 13.3.1 Abstract Input Layer Examples

#### Having Multiple Input Layers with the Same Name

Assume the following 3 tile layers as input, which possible inputs are there in the working map?

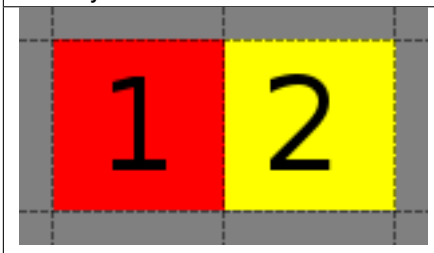
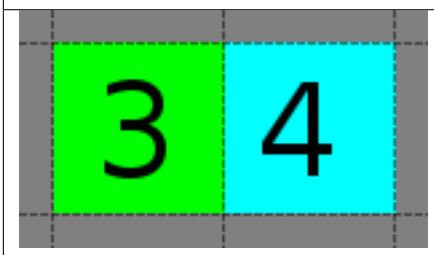
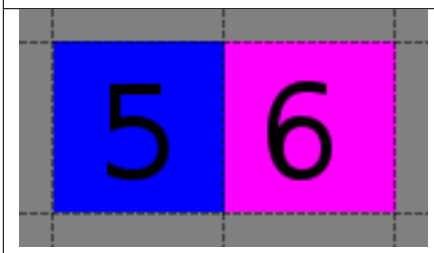
| Tile layer | Name |
| --- | --- |
|  | input_Ground |
|  | input_Ground |
|  | input_Ground |

The following parts would be detected as matches for this rule:

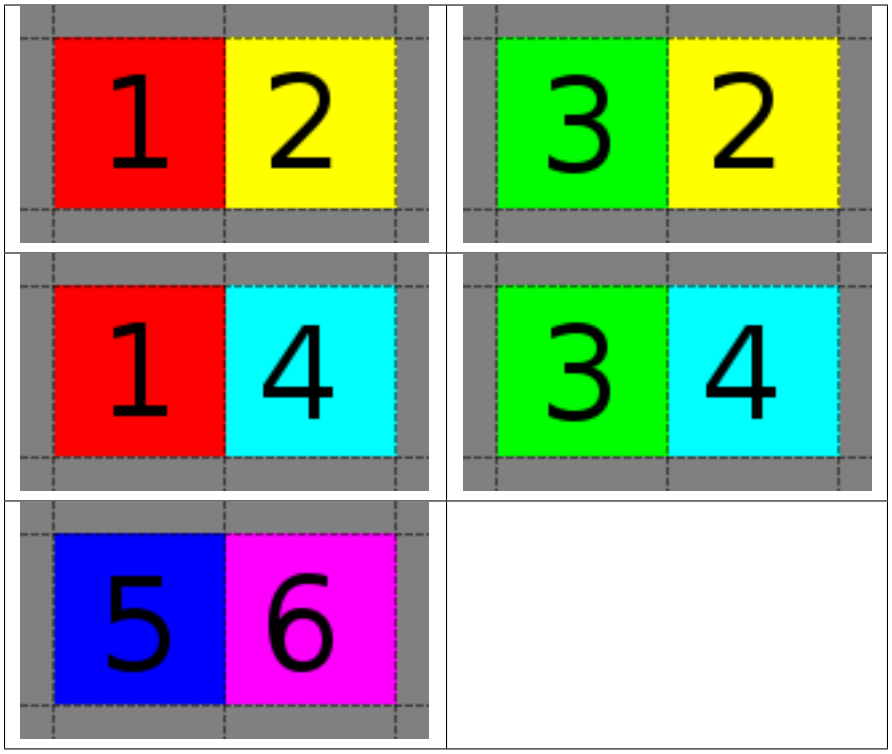


### Input Layers Using Different Indexes

Given the following 3 input tile layers:

| Tile layer | Name |
| --- | --- |
|   | input_Ground |
|   | input_Ground |
|   | input2_Ground |

The last layer has an index unequal to the other indexes (which are empty). All following parts would be recognized as matches within the working map:




### 13.3.2 The Mana World Examples

The Mana world examples will demonstrate quite a lot of different Automapping features. At first a shoreline will be constructed, by first adding all the straight parts and afterwards another rule will correct the corners to make them also fit the given tileset. After the shoreline has been added, the waters will be marked as unwalkable for the game engine. Last but not least the grass should be tiles should be made random by using 5 different grasss tiles.

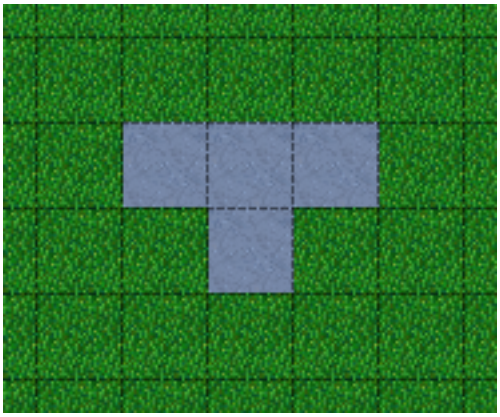


Fig. 1: This is what we want to draw.

#### Basic Shoreline

This example will demonstrate how a straight shoreline can easily be setup between shallow water grass tiles. In this example we will only implement the shoreline, which has grass in southern and water in northern direction.

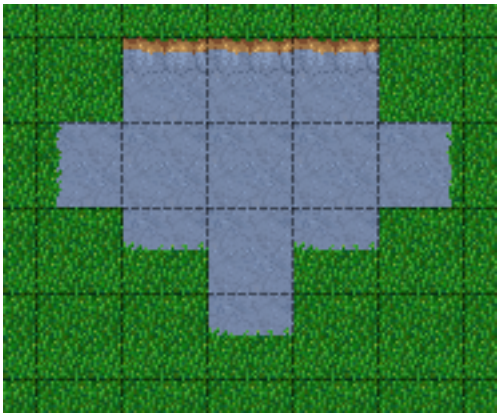


Fig. 2: Here we have straight shorelines applied.

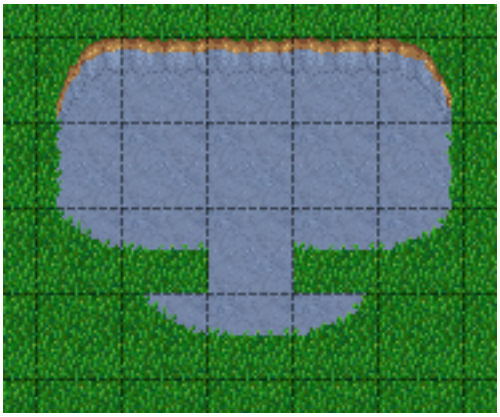


Fig. 3: Here we have some corners.

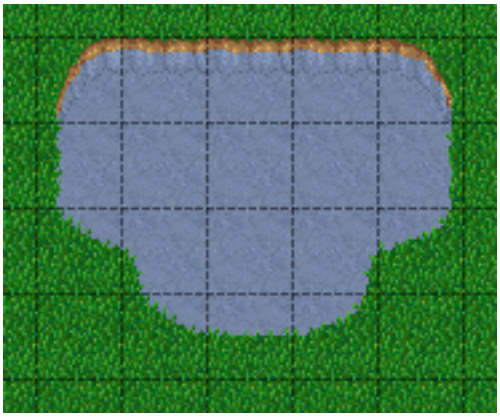


Fig. 4: And corners the other way round as well.

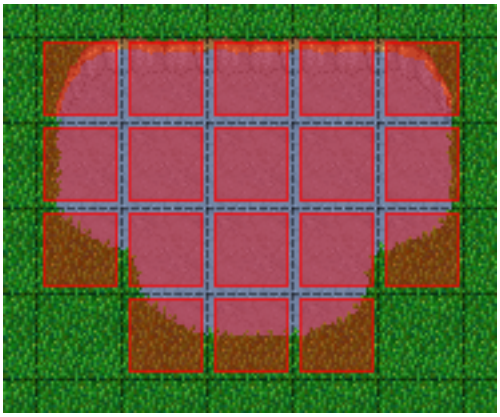


Fig. 5: Here all unwalkable tiles are marked.

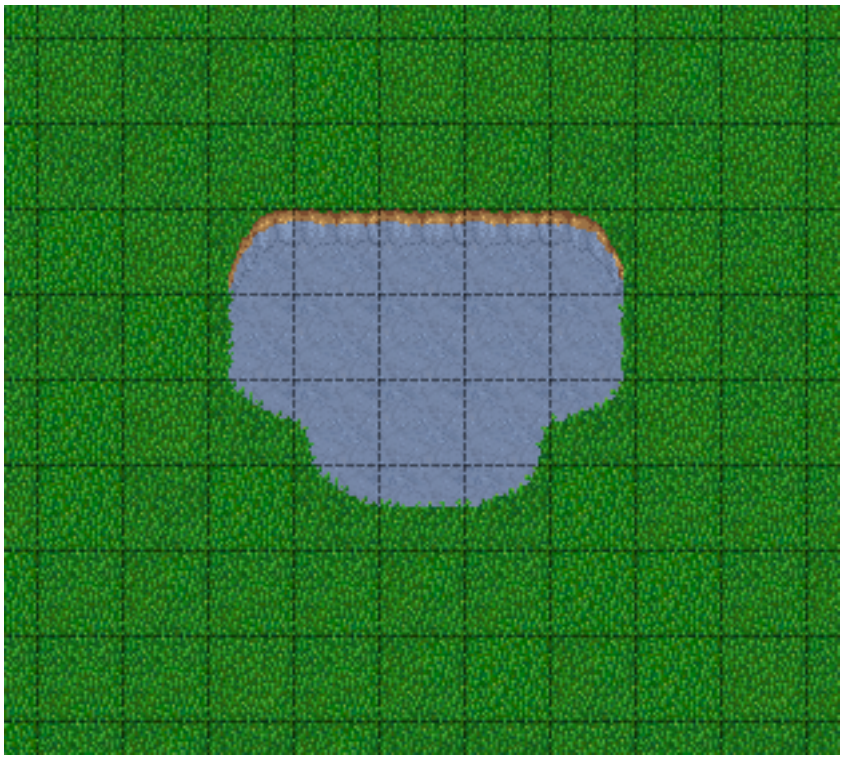
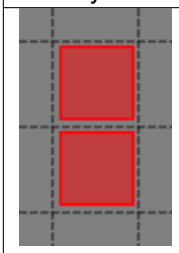
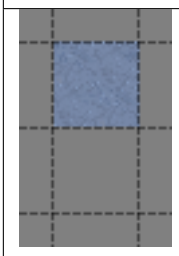
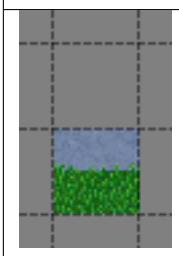


Fig. 6: If you look closely at the grass, you'll see they are now randomized.

So basically the meaning we will define in the input region is: *All tiles which are south of a water tile and are not water tiles themselves, will be replaced by a shoreline tile*

| Tile layer | Name |
| --- | --- |
|  | regions |
|  | input_Ground |
|  | output_Ground |

The region in which this Automapping rule should be defined is of 2 tiles in height and 1 tile in width. Therefore we need a layer called *regions* and it will have 2 tiles placed to indicate this region.

The input layer called *input_Ground* is depicted in the middle. Only the upper tile is filled by the water tile. The lower tile contains no tile. It is not an invisible tile, just no tile at all.

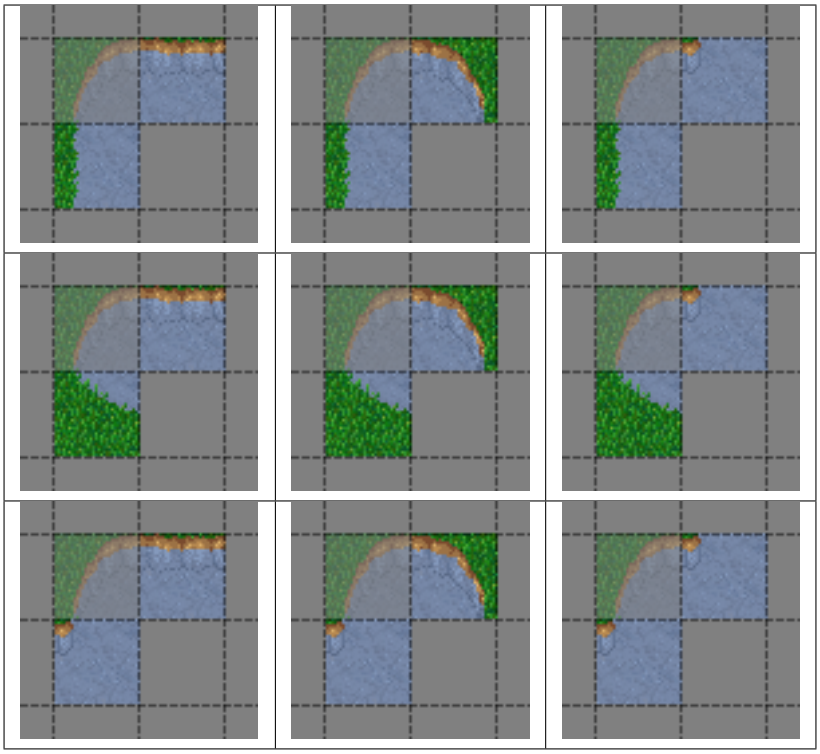
And whenever there is no tile in a place within the rule regions in an input layer, what kind of tiles will be allowed there? There will be allowed any tiles except all used tiles within all input layer with the same index and name.

Here we only have one tile layer as an input layer carrying only the water tile. Hence at the position, where no tile is located, all tiles except that water tile are allowed.

The output layer called *output_Ground* shows the tile which gets placed, if this rule matches.

### Corners on a Shore Line

This example is a continuation of the previous example. Now the corners of the given shoreline should be implemented automatically. Within this article we will just examine the bent in corner shoreline in the topleft corner. The other shoreline corners are constructed the same way. So after the example is applied, we would like to have the corners of the shoreline get suitable tiles. Since we rely on the other example being finished, we will put the rules needed for the corners into another new rulefile. (which is listed afterwards in rules.txt)



The shoreline may have some more corners nearby, which means there may be more different tiles than the straight corner lines. In the figure we see all inputs which should be covered.

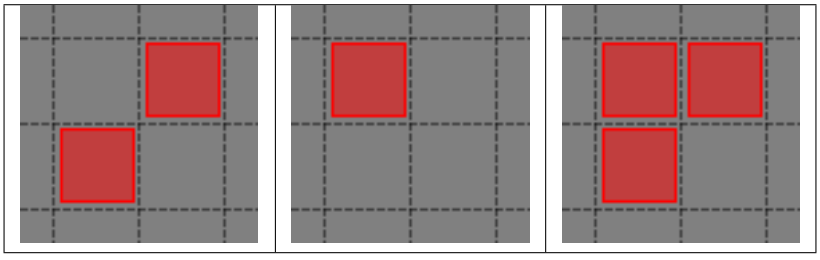
Both the tiles in the top right corner and in the lower left corner are directly adjacent to the desired (slightly transparent) tile in the top left corner.

We can see 3 different tiles for the lower left corner, which is straight shore line, bent inside and bend outside shore lines.

Also we see 3 different inputs for the top right corner, which also is straight, bent in or out shore line.

### regions

So with this rule we want to put the bent in shore line tile in the top left corner, we don't care which tile was there before. Also we don't care about the tile in the lower right corner. (probably water, but can be any decorative watertile, so just ignore it).



Therefore we will need different input and output regions. In the figure we can see the both tilelayers regions input and regions output. The input section covers just these two tiles as we discussed. The output region covers just the

single tile we want to output. Though the input and output region do not overlap, the united region of both the input and the output region is still one coherent region, so it's one rule and works.

Output regions can be larger than absolutely required, since when there are no tiles in the Output section, the tiles in the working map are not overwritten but just kept as is, hence the Output region could also be sized as the united region of both the output and input region.

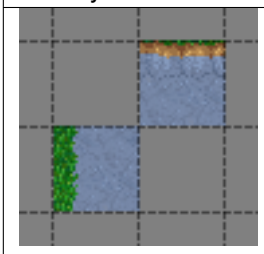
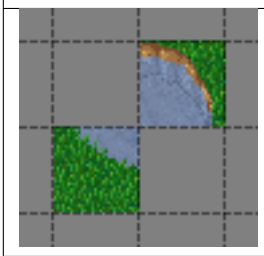
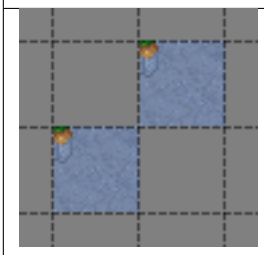
### regions_input

Now we want to put all the nine possible patterns we observed as possible input for this rule. We could of course define nine different layers *input1_Ground* up to *input9_Ground*

Nine TileLayers?! what a mess, we'll do it a better way.

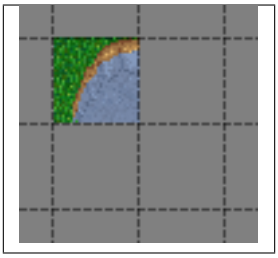
Also, consider having not just 3 possible tiles at the 2 locations but 4. Then we would need 4*4=16 tilelayers to get all conditions. Another downside of this comes with more needed locations: Think of more than 2 locations needed to construct a ruleinput. So for 3 locations, then each location could have the 3 possibilites, hence you need 3*3*3 = 27 tilelayers. It's not getting better...

So let's try a smart way: All input layers have the same name, so at each position any of the three different tiles is valid.

| Tile layer | Name |
|---|---|
|  | input_Ground |
|  | input_Ground |
|  | input_Ground |

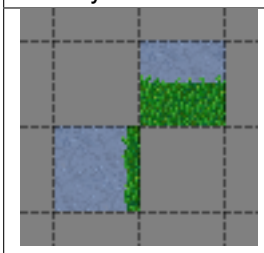
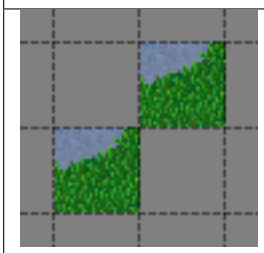
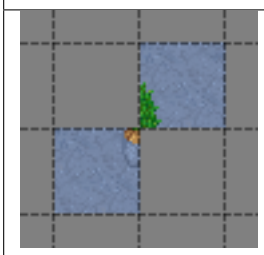
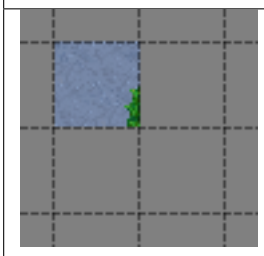
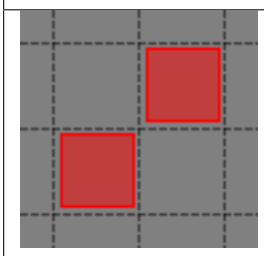
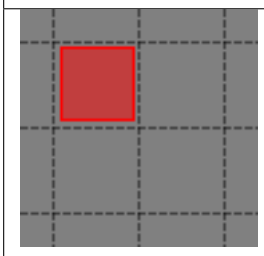
### outputs

The output is straight forward, since only one tile is needed. No randomness is needed, hence the index is not needed to be varied, so it's kept empty. The desired output layer is called Ground, so the over all name of the single output layer will be output Ground. At this single layer at the correct location the correct tile is placed.



### The Other Corners on a Shore Line

This is for corners bent the other way round. Basically it has the same concepts, just other tiles.

| Tile layer | Name |
| --- | --- |
|  | input_Ground |
|  | input_Ground |
|  | input_Ground |
|  | output_Ground |
|  | regions_input |
|  | regions_output |

### Adding Collision Tiles

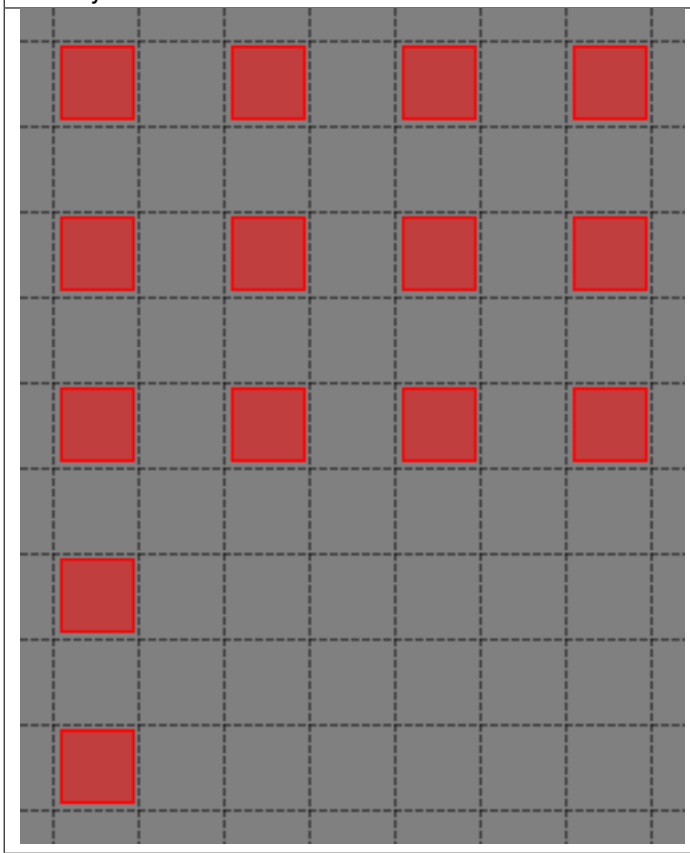
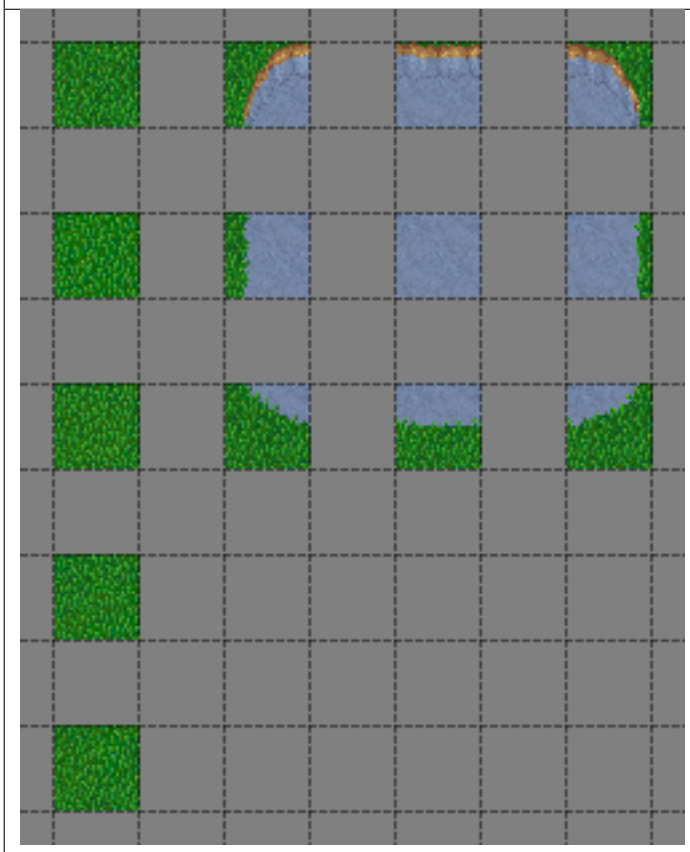
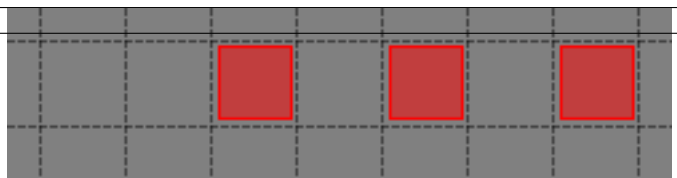
The Mana World uses an extra tile layer called *Collision* to have information about whether a player is able to walk on certain tiles or not. That layer is invisible to the player, but the game engine parses it, whether there is a tile or there is

no tile.

So we need to decide for each position if a player can walk there and put a tile into the *Collision* layer if it is unwalkable.

As *input* layer we will parse the *Ground* layer and put collision tiles where the player should not walk.

Actually this task is a bunch of rules, but each rule itself is very easy:

| Tile layer | Name |
| --- | --- |
|  | regions |
|  | input_Ground |
|  | |

In the above *regions* layer we have 14 different rules, because there are 14 incoherent regions in the *regions* layer. That's 9 different water tiles, which should be unwalkable and 5 different grass tiles which will be placed randomly in the next example.

As input we will have one of all the used tiles and as output there is either a tile in the *Collision* layer or not.

**Do we need the rules with clean output?** No, it is not needed for one run of Automapping. But if you are designing a map, you will likely add areas with collision and then remove some parts of it again and so on.

So we need to also remove the collision tiles from positions, which are not marked by a collision any more. This can be done by adding the map property *DeleteTiles* and setting it to *yes* or *true*. Then all the parts in the *Collision* layer will be erased before the Automapping takes place, so the collision tiles are only placed at real unwalkable tiles and the history of if there has been a collision tile placed is neglected.

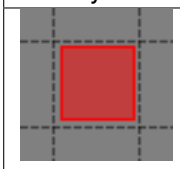
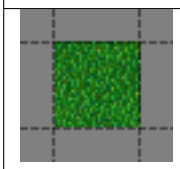
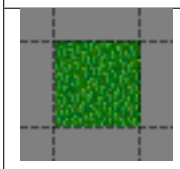
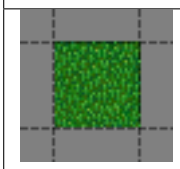
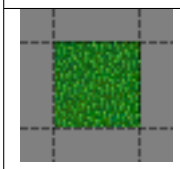
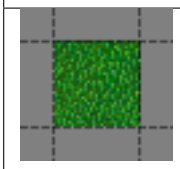
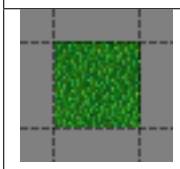
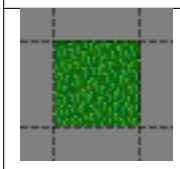
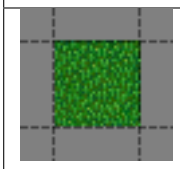
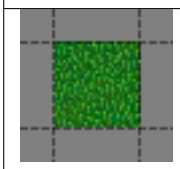
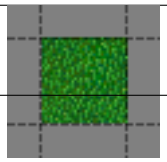
### Random Grass Tiles

In this example we will shuffle all grass tiles, so one grass tiles will be replaced with another randomly chosen tile.

As input we will choose all of our grass tiles. This is done by having each tile in its own input layer, so each grass tile gets accepted for this rule.

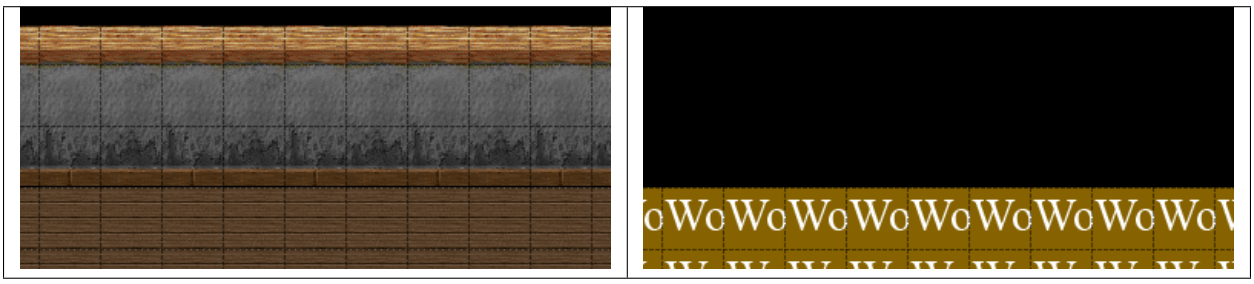
As output we will also put each grass tile into one output layer. To make it random the *index* of the output layers needs to be different for each layer.

The following rule might look the same, but there are different grass tiles. Each grass tile is in both one of the input and one of the output layers (the order of the layers doesn't matter).

| Tile layer | Name |
| --- | --- |
|  | regions |
|  | input_Ground |
|  | input_Ground |
|  | input_Ground |
|  | input_Ground |
|  | input_Ground |
|  | output1_Ground |
|  | output2_Ground |
|  | output3_Ground |
|  | output4_Ground |
|  | output5_Ground |

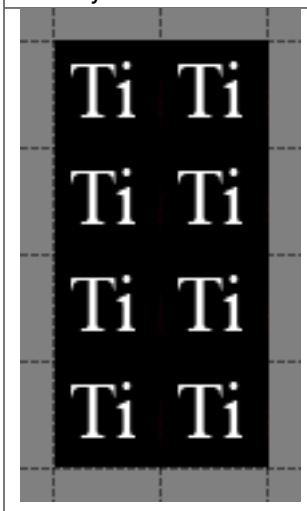
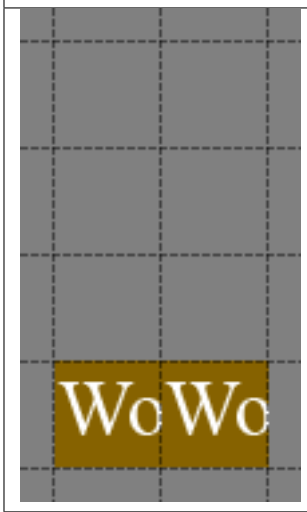
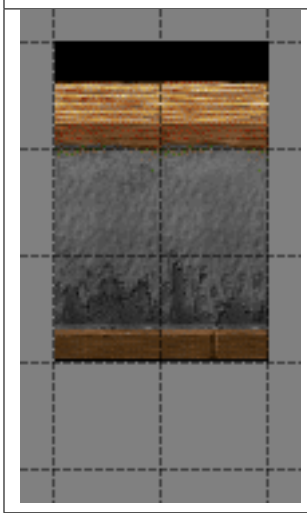
### 13.3.3 An alternating wall

This example will demonstrate how a wall as a transition between a walkable area and the non-walkable black void can easily be setup. As input a dedicated set layer will be used.



In my opinion a dedicated set layer is much easier to use for the rough draft, but for adding details such as collision information on decorative tiles the input should use the decoration.

The structure of the input, output and region layer is very similar to the example of the straight shoreline in The Mana World examples. The main difference is the different size. Since the wall contains multiple tiles in height, the height of the rulelayers is different as well. Vertically the tiles are also alternating. As you can see in the following figure, every second tile displaying the base board of the wall has a notch for example.

| Tile layer | Name |
| --- | --- |
|  | regions |
|  | input_Ground |
|  | output_Walls |

Hence the region in which this Automapping rule should be defined is of 4 tiles in height and 2 tile in width. Therefore we need a layer called *regions* and it will have 8 tiles placed to indicate this region. In the figure the top graphics shows such a region layer.

The input layer has the following meaning:

*If there are 2 vertical adjacent brown tiles in the set layer and in the 3x2 tiles above here are no brown tiles, this rule matches.*

Only the lowest 2 coordinates contain the brown tile. The upper coordinates contain no tile. (It is not an invisible tile, just no tile at all.) The input layer called *Input_set* is depicted in the middle of the figure.

The output consists of only one layer as well called *Output_Walls*. It contains the actual wall tiles.

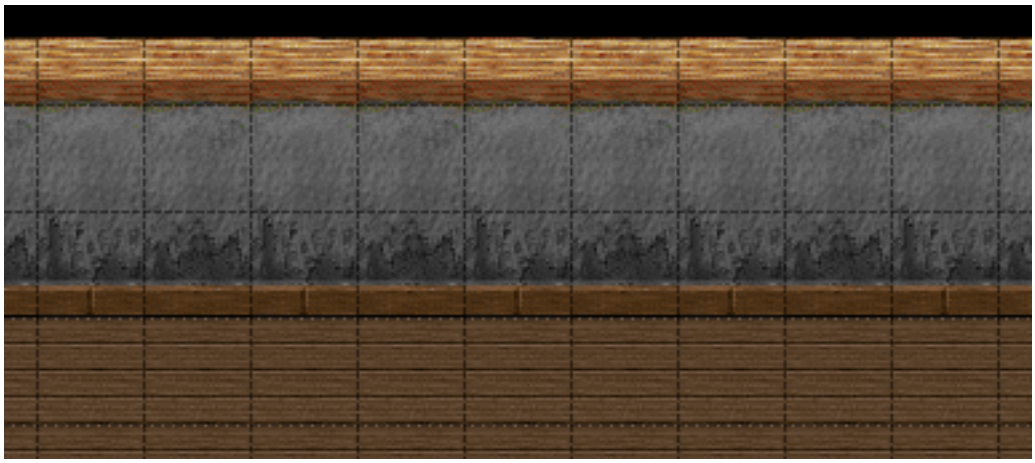


Fig. 7: Vertically the tiles are alternating.

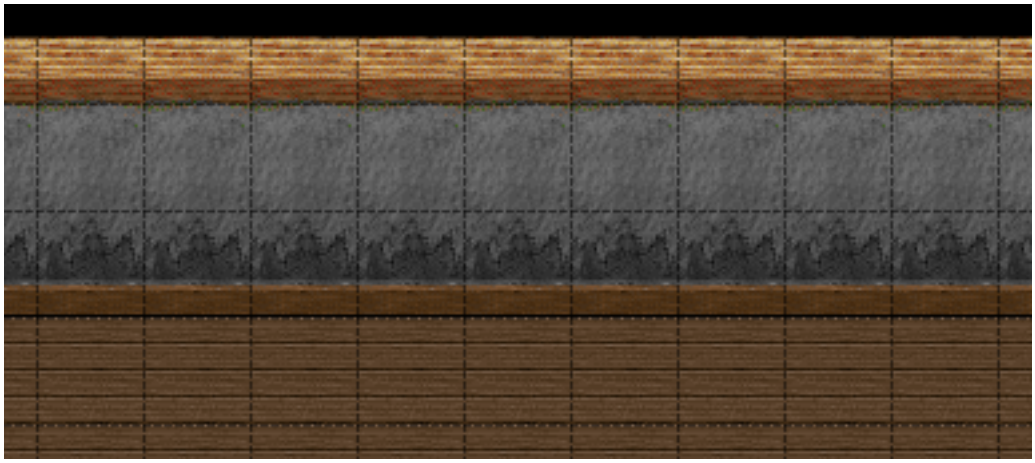


Fig. 8: A broken version of the rule, because *NoOverlappingRules* was not yet set.

When trying to match the input layer to the desired set layer (right picture of the figure at the beginning of the example, you will see it matches all the way along, with no regard of the vertical adjustment.

Hence when we use the rule as discussed now, we will get not the desired result, because this rule overlaps itself. The overlapping problem is shown in figure above.

Since the overlapping is not desired, we can turn it off by adding a map property to the rulemap *NoOverlappingRules* and setting it to *true*

Keep in mind that the map property applies for all rules on that rule map.

CHAPTER 14

# Export Formats

While there are many *libraries and frameworks* that work directly with Tiled maps, Tiled also supports a number of additional file and export formats.

Exporting can be done by clicking *File* > *Export*. When triggering the menu action multiple times, Tiled will only ask for the file name the first time. Exporting can also be automated using the `--export-map` command-line parameter.

Several *Export Options* are available, which are applied to maps or tilesets before they are exported (without affecting the map or tileset itself).

**Note:** When exporting on the command-line on Linux, Tiled will still need an X server to run. To automate exports in a headless environment, you can use a headless X server such as Xvfb. In this case you would run Tiled from the command-line as follows:

```
xvfb-run tiled --export-map ...
```

## 14.1 JSON

*The JSON format* is most common additional file format supported by Tiled. It can be used instead of TMX since Tiled can also open JSON maps and tilesets and the format supports all Tiled features. Especially in the browser and when using JavaScript in general, the JSON format is easier to load.

The JSON format is currently the only additional format supported for tilesets.

## 14.2 Lua

Maps and tilesets can be exported to Lua code. This export option supports most of Tiled's features and is useful when using a Lua-based framework like LÖVE (with Simple Tiled Implementation), Corona (with ponytiled or Dusk Engine) or Defold.

Currently not included are the type of custom properties (though the type does affect how a property value is exported) and information related to recent features like *Wang tiles* and *object templates*.

## 14.3 CSV

The CSV export only supports *tile layers*. Maps containing multiple tile layers will export as multiple files, called `base_<layer-name>.csv`.

Each tile is written out by its ID, unless the tile has a custom property called `name`, in which case its value is used to write out the tile. Using multiple tilesets will lead to ambiguous IDs, unless the custom `name` property is used. Empty cells get the value `-1`.

## 14.4 GameMaker: Studio 1.4

GameMaker: Studio 1.4 uses a custom XML-based format to store its rooms, and Tiled ships with a plugin to export maps in this format. Currently only orthogonal maps will export correctly.

Tile layers and tile objects (when no type is set) will export as "tile" elements. These support horizontal and vertical flipping, but no rotation. For tile objects, scaling is also supported.

**Warning:** The tilesets have to be named the same as the corresponding backgrounds in the GameMaker project. Otherwise GameMaker will pop up an error for each tile while loading the exported `room.gmx` file.

### 14.4.1 Object Instances

GameMaker object instances are created by putting the object name in the "Type" field of the object in Tiled. Rotation is supported here, and for tile objects also flipping and scaling is supported (though flipping in combination with rotation doesn't appear to work in GameMaker).

The following custom properties can be set on objects to affect the exported instance:

- string `code` (instance creation code, default: "")
- float `scaleX` (default: derived from tile or 1.0)
- float `scaleY` (default: derived from tile or 1.0)
- int `originX` (default: 0)
- int `originY` (default: 0)

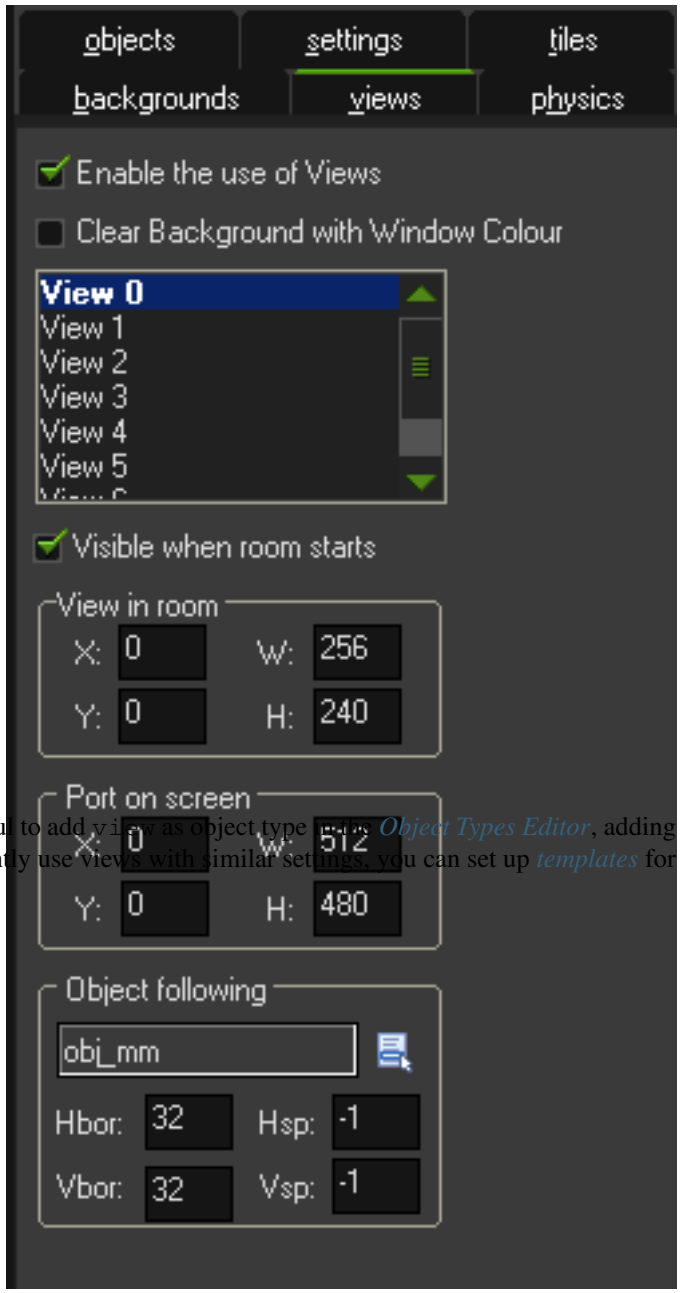
The `scaleX` and `scaleY` properties can be used to override the scale of the instance. However, if the scale is relevant then it will generally be easier to use a tile object, in which case it is automatically derived from the tile size and the object size.

The `originX` and `originY` properties can be used to tell Tiled about the origin of the object defined in GameMaker, as an offset from the top-left. This origin is taken into account when determining the position of the exported instance.

**Hint:** Of course setting the type and/or the above properties manually for each instance will get old fast. Since Tiled 1.0.2, you can instead use tile objects with the type set on the tile, and in Tiled 1.1 you can also use *object templates*.

### 14.4.2 Views

Views can be defined using *rectangle objects* where the Type has been set to `view`. The position and size will be snapped to pixels. Whether the view is visible when the room starts depends on whether the object is visible. The use of views is automatically enabled when any views are defined.




The following custom properties can be used to define the various other properties of the view:

**Port on screen**

- int `xport` (default: 0)
- int `yport` (default: 0)
- int `wport` (default: 1024)
- int `hport` (default: 768)

**Object following**

- string `objName`
- int `hborder` (default: 32)
- int `vborder` (default: 32)
- int `hspeed` (default: -1)
- int `vspeed` (default: -1)

---

**Hint:** When you're defining views in Tiled, it is useful to add `view` as object type in the *Object Types Editor*, adding the above properties for ease of access. If you frequently use views with similar settings, you can set up *templates* for them.

---

### 14.4.3 Map Properties

#### General

- int `speed` (default: 30)
- bool `persistent` (default: false)
- bool `clearDisplayBuffer` (default: true)
- bool `clearViewBackground` (default: false)
- string `code` (map creation code, default: "")

#### Physics

- bool `PhysicsWorld` (default: false)
- int `PhysicsWorldTop` (default: 0)

- int `PhysicsWorldLeft` (default: 0)
- int `PhysicsWorldRight` (default: width of map in pixels)
- int `PhysicsWorldBottom` (default: height of map in pixels)
- float `PhysicsWorldGravityX` (default: 0.0)
- float `PhysicsWorldGravityY` (default: 10.0)
- float `PhysicsWorldPixToMeters` (default: 0.1)

### 14.4.4 Layer Properties

Both tile layers and object layers may produce "tile" elements in the exported room file. Their depth is set automatically, with tiles from the bottom-most layer getting a value of 10000000 (the GameMaker default) and counting up from there. If you want to set a custom depth value you can set the following property on the layer:

- int `depth` (default: 10000000 + N)

## 14.5 tBIN

The tBIN map format is a binary format used by the tIDE Tile Map Editor. tIDE was used by Stardew Valley, a successful game that spawned many community mods.

Tiled ships with a plugin that enables direct editing of Stardew Valley maps (and any other maps using the tBIN format). This plugin needs to be enabled in *Edit* > *Preferences* > *Plugins*. It is not enabled by default because it won't store everything (most notably it doesn't support object layers in general, nor external tilesets), so you need to know what you are doing.

---

**Note:** The tBIN format supports setting custom properties on the tiles of a tile layer. Since Tiled does not support this directly, "TileData" objects are created that match the location of the tile, on which such properties are then stored.

---

## 14.6 Defold

Tiled can export to Defold using one of the two supplied plugins. Both are disabled by default.

**defold**

This plugin exports a map to a Defold Tile Map (*.tilemap). It only supports tile layers and only a single tileset may be used.

Upon export, the `tile_set` property of the Tile Map is left empty, so it will need to be set up in Defold after each export.

**defoldcollection**

This plugin exports a map to a Defold Collection (*.collection), while also creating multiple .tilemap files.

It supports:

- Group layers (**only top-level group layers are supported, not nested ones!**)
- Multiple Tilesets per Tilemap

Upon export:

- The `Path` property of each Tileset may need to be set up manually in Defold after each export. However, Tiled will attempt to find the .tilesource file corresponding with the name your Tileset in Tiled in your project's `/tilesources/` directory. If one is found, manual adjustments won't be necessary.
- If you create custom properties on your map called `x-offset` and `y-offset`, these values will be used as coordinates for your top-level GameObject in the Collection. This is useful when working with *Worlds*.

All layers of a Tilemap will have Z-index property assigned with values ranging between 0 and 0.1. The plugin supports the use of 9999 Group Layers and 9999 Tile Layers per Group Layer.

When any additional information from the map is needed, the map can be exported in *Lua format* and loaded as Defold script.

## 14.7 Other Formats

A few other plugins ship with Tiled to support various games:

**droidcraft** Adds support for editing DroidCraft maps (*.dat)

**flare** Adds support for editing Flare Engine maps (*.txt)

**replicaisland** Adds support for editing Replica Island maps (*.bin)

**tengine** Adds support for exporting to T-Engine4 maps (*.lua)

These plugins are disabled by default. They can be enabled in *Edit* > *Preferences* > *Plugins*.

### 14.7.1 JavaScript

It is possible to add custom export formats using *scripting* (by calling *tiled.registerMapFormat*).

### 14.7.2 Python Scripts

It is also possible to write *Python scripts* to add support for importing or exporting custom map formats.

CHAPTER 15

# Keyboard Shortcuts

Try out these keyboard shortcuts to help save you time.

On Mac, replace `Ctrl` with the `Command` key.

## 15.1 General

- `Right Click on Tile` - Captures the tile under the mouse (drag to capture larger areas).
- `Ctrl + MouseWheel` - Zoom in/out on tileset and map
- `Ctrl + Plus/Minus` - Zoom in/out on map
- `Ctrl + 0` - Reset zoom on map
- `Ctrl + Object Move` - Toggles "Snap to Grid" temporarily
- `Ctrl + Object Resize` - Keep aspect ratio
- `Alt + Object Resize` - Toggles "Snap to Grid" temporarily
- `Middle Click` or `Space Bar` - Hold to pan the map view
- `Ctrl + X` - Cut (tiles, objects or properties)
- `Ctrl + C` - Copy (tiles, objects or properties)
- `Ctrl + V` - Paste (tiles, objects or properties)
- `Del` - Delete (tiles or objects)
- `H` - Toggle highlighting of the current layer
- `A` - Invokes *Automapping*
- `Alt + C` - Copy current position of mouse cursor to clipboard (in tile coordinates)
- `Ctrl + D` - Duplicate selected objects (since Tiled 1.0, before it was Delete)
- `Ctrl + Shift + D` - Duplicate active layer

- `F2` - Rename (if applicable in context)
- `Tab` - Hide docks and tool bars (since Tiled 1.0)
- `Ctrl + PgUp` - Select previous layer (above current layer)
- `Ctrl + PgDown` - Select next layer (below current layer)
- `Ctrl + Shift + Up` - Move current layer up
- `Ctrl + Shift + Down` - Move current layer down
- `Ctrl + Shift + H` - Show/Hide all other layers (only active layer visible / all layers visible)
- `Ctrl + Tab` - Switch to left document
- `Ctrl + Shift + Tab` - Switch to right document
- `Alt + Left` - Switch to left document
- `Alt + Right` - Switch to right document
- `Ctrl + G` - Toggle displaying of the tile grid
- `Ctrl + W` - Close current document
- `Ctrl + Shift + W` - Close all documents
- `Ctrl + E` - Export current document
- `Ctrl + Shift + E` - Export current document to another file
- `Ctrl + Q` - Quit Tiled
- `Ctrl + R` - Reload current document
- `Ctrl + T` - Force-reload all tilesets used by the current map (mainly useful when not using the automatic reloading)

## 15.2 When a tile layer is selected

- `D` - Toggle Random Mode
- `B` - Activate *Stamp Brush*
  - `Shift + Click` - Line mode, places tiles on a line between two clicked locations
  - `Ctrl + Shift + Click` - Circle mode, places tiles around the clicked center
- `T` - Activate *Terrain Brush*
- `G` - Activate *Wang Brush* (since Tiled 1.1)
- `F` - Activate *Bucket Fill Tool*
- `P` - Activate *Shape Fill Tool*
- `E` - Activate *Eraser*
- `R` - Activate Rectangular Select
- `W` - Activate Magic Wand
- `S` - Activate Select Same Tile
- `Ctrl + 1-9` - Store current tile selection (similar to `Ctrl + C`)
- `1-9` recall the previous selection (similar to `Ctrl + V`)

- `Ctrl + A` - Select the whole layer
- `Ctrl + Shift + A` - Select nothing

Changing the active stamp:

- `X` - Flip active stamp horizontally
- `Y` - Flip active stamp vertically
- `Z` - Rotate active stamp clockwise
- `Shift + Z` - Rotate active stamp counterclockwise

## 15.3 When an object layer is selected

- `S` - Activate *Select Objects*
  - `PgUp` - Raise selected objects (with Manual object drawing order)
  - `PgDown` - Lower selected objects (with Manual object drawing order)
  - `Home` - Move selected objects to Top (with Manual object drawing order)
  - `End` - Move selected objects to Bottom (with Manual object drawing order)
- `O` - Activate *Edit Polygons* (was `E` until Tiled 1.0)
- `R` - Activate *Insert Rectangle*
- `I` - Activate *Insert Point*
- `C` - Activate *Insert Ellipse*
- `P` - Activate *Insert Polygon*
  - `Enter` - Finish creating object
  - `Escape` - Cancel creating object
- `T` - Activate *Insert Tile*
- `V` - Activate *Insert Template* (since Tiled 1.1)
- `E` - Activate *Insert Text* (since Tiled 1.0)
- `Ctrl + A` - Select all objects in the current object layer
- `Ctrl + Shift + A` - Clear object selection

## 15.4 In the Properties dialog

- `Del` or `Backspace` - Deletes a property

# CHAPTER 16

# User Preferences

There are only a few options located in the Preferences, accessible though the menu via *Edit* > *Preferences*. Most other options, like whether to draw the grid, what kind snapping to do or the last used settings when creating a new map are simply remembered persistently.

The preferences are stored in a system-dependent format and location:

| | |
|---|---|
| **Windows** | Registry key `HKEY_CURRENT_USER\SOFTWARE\mapeditor.org\Tiled` |
| **macOS** | `~/Library/Preferences/org.mapeditor.Tiled.plist` |
| **Linux** | `~/.config/mapeditor.org/tiled.conf` |

## 16.1 General

### 16.1.1 Saving and Loading

**Include DTD reference in saved maps**
This option is not enabled by default, since it is of very little use whereas it can in some environments cause problems. Feel free to enable it if it helps with validation for example, but note that the referenced DTD is likely out of date (there is a somewhat more up-to-date XSD file available in the repository).

**Reload tileset images when they change**
This is very useful while working on the tiles or when the tiles might change as a result of a source control system.

**Open last files on startup**
Generally a useful thing to keep enabled.

**Use safe writing of files** This setting causes files to be written to a temporary file, and when all went well, to be swapped with the target file. This avoids data getting lost due to errors while saving or due to insufficient disk space. Unfortunately, it is known to cause issues when saving files to a Dropbox folder or a network drive, in which case it helps to disable this feature.

### 16.1.2 Export Options

The following export options are applied each time a map or tileset gets exported, without affecting the map or tileset itself.

**Embed tilesets** All tilesets are embedded in the exported map. Useful for example when you are exporting to JSON and loading an external tileset is not desired.

**Detach templates** All template instances are detached. Useful when you want to use the templates feature but can't or don't want to load the external template object files.

**Resolve object types and properties**
Stores effective object type and properties with each object. Object properties

are inherited from a tile (in case of a tile object) and from the default properties of their type.

**Minimize output** Omits unnecessary whitespace in the output file. This option is supported for XML (TMX and TSX), JSON and Lua formats.

These options are also available as options when exporting using the command-line.

## 16.2 Interface

### 16.2.1 Interface

**Language** By default the language tries to match that of the system, but if it picks the wrong one you can change it here.

**Grid colour** Because black is not always the best color for the grid.

**Fine grid divisions** The tile grid can be divided further using this setting, which affects the "Snap to Fine Grid" setting in the *View* > *Snapping* menu.

**Object line width** Shapes are by default rendered with a 2 pixel wide line, but some people like it thinner or even thicker. On some systems the DPI-based scaling will affect this setting as well.

**Hardware accelerated drawing (OpenGL)** This enables a rather unoptimized way of rendering the map using OpenGL. It's usually not an improvement and may lead to crashes, but in some scenarios it

can make editing more responsive.

**Mouse wheel zooms by default**
This option causes the mouse wheel to zoom without the need to hold Control (or Command on macOS). It can be a convenient way to navigate the map, but it can also interfere with panning on a touchpad.

### 16.2.2 Updates

By default, Tiled checks for news and new versions and highlights any updates in the status bar. Here you can disable this functionality. It is recommended to keep at least one of these enabled.

If you disable displaying of new versions, you can still manually check whether a new version is available by opening the *About Tiled* dialog.

## 16.3 Keyboard

Here you can add, remove or change the keyboard shortcuts of most available actions.

Conflicting keybindings are highlighted in red. They will not work until you resolve the conflict.

If you customize multiple shortcuts, it is recommended to use the export functionality to save the keybindings somewhere, so that you can easily recover that setup or copy it to other Tiled installations.

## 16.4 Theme

On Windows and Linux, the default style used by Tiled is "Tiled Fusion". This is a customized version of the "Fusion" style that ships with Qt. On macOS, this style can also be used, but because it looks so out of place the default is "Native" there.

The "Tiled Fusion" style allows customizing the base color. When choosing a dark base color, the text automatically switches to white and some other adjustments are made to keep things readable. You can also choose a custom selection color.

The "Native" style tries to fit in with the operating system, and is available since it is in some cases preferable to the custom style. The base color and selection color can't be changed when using this style, as they depend on the system.

## 16.5 Plugins

Here you can choose which plugins are enabled, as well as opening the *scripted extensions* folder.

Plugins add support for map and/or tileset file formats. Some generic plugins are enabled by default, while more specific ones need to be manually enabled.

There is no need to restart Tiled when enabling or disabling plugins. When a plugin fails to load, try hovering its icon to see if the

tool tip displays a useful error message.

See *Export Formats* for more information about supported file formats.

# CHAPTER 17

# Python Scripts

**Note:** Since Tiled 1.3, Tiled can be *extended using JavaScript*. The JavaScript API provides a lot more opportunity for extending Tiled's functionality than just adding custom map formats. It is fully documented and works out of the box on all platforms. It should be preferred over the Python plugin when possible.

Tiled ships with a plugin that enables you to use Python 3 to add support for custom map formats. This is nice especially since you don't need to compile Tiled yourself and the scripts are easy to deploy to any platform.

For the scripts to get loaded, they should be placed in `~/.tiled`. Tiled watches this directory for changes, so there is no need to restart Tiled after adding or changing scripts (though the directory needs to exist when you start Tiled).

There are several example scripts available in the repository.

**Note:** To create the `~/.tiled` folder on Windows, open command prompt (`cmd.exe`), which should start in your home folder by default, then type `mkdir .tiled` to create the folder.

On Linux, folders starting with a dot are hidden by default. In most file managers you can toggle showing of hidden files using `Ctrl+H`.

**Note:** Since Tiled 1.2.4, the Python plugin is disabled by default, because depending on which Python version is installed on the system the loading of this plugin may cause a crash (#2091). To use the Python plugin, first enable it in the Preferences.

**Warning:** On Windows, Python is not installed by default. For the Tiled Python plugin to work, you'll need to install Python 3.7 (get it from https://www.python.org/). You will also need to check the box "Add Python 3.7 to PATH" in the installer:

On Linux you will also need to install the appropriate package. However, currently Linux builds are done on Ubuntu 16.04 against Python 3.5, and you'd need to install the same version somehow.

The Python plugin is currently not enabled for macOS releases. We'll need to find out how to build it against Python 3, while macOS only ships with Python 2.7 by default. If you rely on this plugin on macOS you'll need to use Tiled 1.1 for now.

## 17.1 Example Export Plugin

Suppose you'd like to have a map exported in the following format:

```
29,29,29,29,29,29,32,-1,34,29,29,29,29,29,29,
29,29,29,29,29,29,32,-1,34,29,29,29,29,29,29,
29,29,29,29,29,29,32,-1,34,29,29,29,29,29,29,
29,29,29,29,29,29,32,-1,34,29,29,29,29,29,29,
25,25,25,25,25,25,44,-1,34,29,29,29,29,29,29,
-1,-1,-1,-1,-1,-1,-1,-1,34,29,29,29,29,29,29,
41,41,41,41,41,41,41,41,42,29,29,24,25,25,25,
29,29,29,29,29,29,29,29,29,29,29,32,-1,-1,-1,
29,29,29,29,29,29,39,29,29,29,29,32,-1,35,41,
29,29,29,29,29,29,29,29,29,29,29,32,-1,34,29,
29,29,29,29,29,29,29,29,37,29,29,32,-1,34,29;
```

You can achieve this by saving the following `example.py` script in the scripts directory:

```
from tiled import *

class Example(Plugin):
    @classmethod
    def nameFilter(cls):
        return "Example files (*.example)"

    @classmethod
    def shortName(cls):
        return "example"

    @classmethod
    def write(cls, tileMap, fileName):
        with open(fileName, 'w') as fileHandle:
            for i in range(tileMap.layerCount()):
                if isTileLayerAt(tileMap, i):
                    tileLayer = tileLayerAt(tileMap, i)
                    for y in range(tileLayer.height()):
                        tiles = []
                        for x in range(tileLayer.width()):
                            if tileLayer.cellAt(x, y).tile() != None:
                                tiles.append(str(tileLayer.cellAt(x, y).tile().id()))
                            else:
                                tiles.append(str(-1))
                        line = ','.join(tiles)
                        if y == tileLayer.height() - 1:
                            line += ';'
                        else:
                            line += ','
                        print(line, file=fileHandle)


        return True
```

Then you should see an "Example files" entry in the type dropdown when going to *File* > *Export*, which allows you to export the map using the above script.

---

**Note:** This example does not support the use of group layers, and in fact the script API doesn't support this yet either. Any help with maintaining the Python plugin would be very appreciated. See open issues related to Python support.

---

## 17.2 Debugging Your Script

Any errors that happen while parsing or running the script are printed to the Debug Console, which can be enabled in *View* > *Views and Toolbars* > *Debug Console*.

## 17.3 API Reference

It would be nice to have the full API reference documented here, but for now please check out the source file for available classes and methods.

CHAPTER 18

# Libraries and Frameworks

There are many libraries available for reading and/or writing Tiled maps (either stored in the *TMX Map Format* or the *JSON Map Format*) as well as many development frameworks that include support for Tiled maps. This list is divided into two sections:

- *Support by Language*
- *Support by Framework*

The first list is for developers who plan on implementing their own renderer. The second list is for developers already using (or considering) a particular game engine / graphics library who would rather pass on having to write their own tile map renderer.

**Note:** For updates to this page please open a pull request or issue on GitHub, thanks!

## 18.1 Support by Language

These libraries typically include only a TMX parser, but no rendering support. They can be used universally and should not require a specific game engine or graphics library.

### 18.1.1 C

- TMX - TMX map loader with Allegro5 and SDL2 examples (BSD).

### 18.1.2 C++

- C++/TinyXML based tmxparser (BSD)
- C++/Qt based libtiled, used by Tiled itself and included at src/libtiled (BSD)
- C++11x/TinyXml2 libtmx-parser by halsafar. (zlib/tinyxml2)

- C++11/TinyXml2 libtmx by jube, for reading only (ISC licence). See documentation.
- TMXParser General *.tmx tileset data loader. Intended to be used with TSXParser for external tileset loading. (No internal tileset support)
- TSXParser General *.tsx tileset data loader. Intended to be used with TMXParser.
- TMXLoader based on RapidXml. Limited functionality (check the website for details).
- tmxlite C++14 map parser with compressed map support but no external linking required. Includes examples for SFML and SDL2 rendering. Currently has full tmx support up to 0.16. (Zlib/libpng)
- Tileson - A Tiled JSON parser for modern C++ (C++17) by Robin Berg Pettersen (BSD)

### 18.1.3 C#/.NET

- MonoGame.Extended has a Tiled map loader and renderer that works with MonoGame on all platforms that support portable class libraries.
- XNA map loader by Kevin Gadd, extended by Stephen Belanger and Zach Musgrave (has dependency on XNA but supposedly can be turned into a standalone parser easily)
- TiledSharp: Yet another C# TMX importer library, with Tiled 0.11 support. TiledSharp is a generic parser which can be used in any framework, but it cannot be used to render the maps. Available via NuGet.
- NTiled: Generic parser for 0.9.1 tiled maps. Available via NuGet.
- TmxCSharp: Useful for multi-layer orthographic tile engines. No framework dependencies, used with a custom OpenTK tile engine soon to be open source, tested with Tiled 0.8.1 (multiple output formats). MIT license.
- tmx-mapper-pcl: PCL library for parsing Tiled map TMX files. This library could be used with MonoGame and Windows Runtime Universal apps.

### 18.1.4 Clojure

- tile-soup: Parses and validates a TMX file into a map. Automatically decodes Base64 and CSV formatted data and coerces numbers when necessary. Works on both the JVM and in browsers via ClojureScript.

### 18.1.5 D

- tiledMap.d simple single-layer and single-tileset example to load a map and its tileset in D language. It also contains basic rendering logic using DSFML
- dtiled can load JSON-formatted Tiled maps. It also provides general tilemap-related functions and algorithms.

### 18.1.6 Go

- github.com/lafriks/go-tiled
- github.com/salviati/go-tmx/tmx

### 18.1.7 Haskell

- htiled (TMX) by Christian Rødli Amble.
- aeson-tiled (JSON) by Schell Scivally.

### 18.1.8 Java

- A library for loading TMX files is included with Tiled at util/java/libtiled-java.
- Android-Specific:
  - AndroidTMXLoader loads TMX data into an object and renders to an Android Bitmap (limited functionality)
  - libtiled-java port is a port of the libtiled-java to be used on Android phones.

### 18.1.9 PHP

- PHP TMX Viewer by sebbu : render the map as an image (allow some modifications as well)

### 18.1.10 Pike

- TMX parser: a simple loader for TMX maps (CSV format only).

### 18.1.11 Processing

- linux-man/ptmx: Add Tiled maps to your Processing sketch.

### 18.1.12 Python

- pytiled-parser: Python parser for TMX maps
- Arcade: 2D game library that uses pytiled-parser for easy loading of TMX maps into a game. Arcade TMX Examples
- pytmxlib: library for programmatic manipulation of TMX maps
- python-tmx: a simple library for reading and writing TMX files.

### 18.1.13 Ruby

- tmx gem by erisdiscord

### 18.1.14 Vala

- librpg A library to load and handle spritesets (own format) and orthogonal TMX maps.

## 18.2 Support by Framework

Following entries are integrated solutions for specific game engines. They are typically of little to no use if you're not using said game engine.

### 18.2.1 AndEngine

- AndEngine by Nicolas Gramlich supports rendering TMX maps

### 18.2.2 Allegro

- allegro_tiled integrates Tiled support with Allegro 5.

### 18.2.3 Castle Game Engine (Object Pascal)

- Castle Game Engine has native support for Tiled maps (see the CastleTiledMap unit)

### 18.2.4 cocos2d

- cocos2d (Python) supports loading Tiled maps through its `cocos.tiles` module.
- cocos2d-x (C++) supports loading TMX maps through the CCTMXTiledMap class.
- cocos2d-objc (Objective-C, Swift) (previously known as: cocos2d-iphone, cocos2d-swift, cocos2d-spritebuilder) supports loading TMX maps through CCTiledMap
- TilemapKit is a tilemapping framework for Cocos2D. It supports all TMX tilemap types, including staggered iso and all hex variations. No longer in development.

### 18.2.5 Construct 2 - Scirra

- Construct 2, since the Beta Release 149, officially supports TMX maps, and importing it by simple dragging the file inside the editor. Official Note

### 18.2.6 Corona SDK

- ponytiled is a simple Tiled Map Loader for Corona SDK (forum announcement)
- Dusk Engine is a fully featured Tiled map game engine for Corona SDK (no longer maintained, but may still be useful)
- Berry is a simple Tiled Map Loader for Corona SDK.
- Qiso is an isometric engine for Corona SDK that supports loading Tiled maps, and also handles things like path-finding for you.

### 18.2.7 Flixel

- Lithander demonstrated his Flash TMX parser combined with Flixel rendering

### 18.2.8 Game Maker

- Tiled ships with a plug-in that can *export a map to a GameMaker: Studio 1.4 room file*
- Tiled2GM Converter by Dmi7ry

### 18.2.9 Godot

- Tiled Map Importer imports each map as Godot scene which can be instanced or inherited (forum announcement)

### 18.2.10 Haxe

- HaxePunk Tiled Loader for HaxePunk
- HaxeFlixel
- OpenFL "openfl-tiled" is a library, which gives OpenFL developers the ability to use the Tiled Map Editor.
- OpenFL + Tiled + Flixel Experimental glue to use "openfl-tiled" with HaxeFlixel

### 18.2.11 HTML5 (multiple engines)

- Canvas Engine A framework to create video games in HTML5 Canvas
- chem-tmx Plugin for chem game engine.
- chesterGL A simple WebGL/canvas game library
- Crafty JavaScript HTML5 Game Engine; supports loading Tiled maps through an external component TiledMapBuilder.
- GameJs JavaScript library for game programming; a thin wrapper to draw on HTML5 canvas and other useful modules for game development
- KineticJs-Ext A multi-canvas based game rendering library
- melonJS A lightweight HTML5 game engine
- Panda 2, a HTML5 Game Development Platform for Mac, Windows and Linux. Has a plugin for rendering Tiled maps, both orthogonal and isometric.
- Phaser A fast, free and fun open source framework supporting both JavaScript and TypeScript (Tiled tutorial)
- linux-man/p5.tiledmap adds Tiled maps to p5.js.
- Platypus Engine A robust orthogonal tile game engine with game entity library.
- sprite.js A game framework for image sprites.
- TMXjs A JavaScript, jQuery and RequireJS-based TMX (Tile Map XML) parser and renderer.

### 18.2.12 indielib-crossplatform

- indielib cross-platform supports loading TMX maps through the C++/TinyXML based tmx-parser by KonoM (BSD)

### 18.2.13 LibGDX

- libgdx, a Java-based Android/desktop/HTML5 game library, provides a packer, loader and renderer for TMX maps

### 18.2.14 LITIengine

- LITIengine is a 2D Java Game Engine that supports loading, saving and rendering maps in the .tmx format.

### 18.2.15 LÖVE

- Simple Tiled Implementation Lua loader for the LÖVE (Love2d) game framework.

### 18.2.16 MOAI SDK

- Hanappe Framework for MOAI SDK.
- Rapanui Framework for MOAI SDK.

### 18.2.17 Monkey X

- bit.tiled Loads TMX file as objects. Aims to be fully compatible with native TMX files.
- Diddy is an extensive framework for Monkey X that contains a module for loading and rendering TMX files. Supports orthogonal and isometric maps as both CSV and Base64 (uncompressed).

### 18.2.18 Node.js

- node-tmx-parser - loads the TMX file into a JavaScript object

### 18.2.19 Oak Nut Engine (onut)

- Oak Nut Engine supports Tiled maps through Javascript and C++. (see TiledMap Javascript or C++ samples)

### 18.2.20 Orx Portable Game Engine

- TMX to ORX Converter Tutorial and converter download for Orx.

### 18.2.21 Pygame

- Pygame map loader by dr0id
- PyTMX by Leif Theden (bitcraft)
- tmx.py by Richard Jones, from his 2012 PyCon 'Introduction to Game Development' talk.
- TMX, a fork of tmx.py and a port to Python3. A demo called pylletTown can be found here.

### 18.2.22 Pyglet

- JSON map loader/renderer for pyglet by Juan J. Martínez (reidrac)
- PyTMX by Leif Theden (bitcraft)

### 18.2.23 PySDL2

- PyTMX by Leif Theden (bitcraft)

### 18.2.24 RPG Maker MV

- Tiled Plugin by Dr.Yami & Archeia, from RPG Maker Web

### 18.2.25 SDL

- C++/TinyXML/SDL based loader example by Rohin Knight (limited functionality)

### 18.2.26 SFML

- STP (SFML TMX Parser) by edoren
- C++/SFML Tiled map loader by fallahn. (Zlib/libpng)
- C++/SfTileEngine by Tresky (currently limited functionality)

### 18.2.27 Slick2D

- Slick2D supports loading TMX maps through TiledMap.

### 18.2.28 Sprite Kit Framework

- SKTilemap is built from the ground up in Swift. It's up to date, full of features and easy to integrate into any Sprite Kit project. Supports iOS and OSX.
- SKTiled - A Swift framework for working with Tiled assets in SpriteKit.
- TilemapKit is a tilemapping framework for Sprite Kit. It supports all TMX tilemap types, including staggered iso and all hex variations. No longer in development.
- JSTileMap is a lightweight SpriteKit implementation of the TMX format supporting iOS 7 and OS X 10.9 and above.

### 18.2.29 TERRA Engine (Delphi/Pascal)

- TERRA Engine supports loading and rendering of TMX maps.

### 18.2.30 Unity 3D

- Orthello Pro (2D framework) offers Tiled map support.
- Tiled To Unity is a 3D pipeline for Tiled maps. It uses prefabs as tiles, and can place decorations dynamically on tiles. Supports multiple layers (including object layers).
- Tiled2Unity exports TMX files to Unity with support for (non-simple) collisions.
- UniTiled, a native TMX importer for Unity.

- UniTMX imports TMX files into a mesh.
- X-UniTMX supports almost all Tiled 0.11 features. Imports TMX/XML files into Sprite Objects or Meshes.
- Tiled TMX Importer, imports into Unity 2017.2's new native Tilemap system.

### 18.2.31 Unreal Engine 4

- Paper2D provides built-in support for tile maps and tile sets, importing JSON exported from Tiled.

### 18.2.32 Urho3D

- Urho3D natively supports loading Tiled maps as part of the Urho2D sublibrary (Documentation, HTML5 example).

### 18.2.33 XNA

- FlatRedBall Glue tool ships with a Tiled plugin that loads TMX maps into the FlatRedBall engine, providing rich integration with its features.
- XTiled by Michael C. Neel and Dylan Wolf, XNA library for loading and rendering TMX maps
- XNA map loader by Kevin Gadd, extended by Stephen Belanger and Zach Musgrave

# CHAPTER 19

# TMX Map Format

**Version 1.1**

The TMX (Tile Map XML) map format used by Tiled is a flexible way to describe a tile based map. It can describe maps with any tile size, any amount of layers, any number of tile sets and it allows custom properties to be set on most elements. Beside tile layers, it can also contain groups of objects that can be placed freely.

Note that there are many *libraries and frameworks* available that can work with TMX maps.

In this document we'll go through each element found in this map format. The elements are mentioned in the headers and the list of attributes of the elements are listed right below, followed by a short explanation. Attributes or elements that are deprecated or unsupported by the current version of Tiled are formatted in italics. All optional attributes are either marked as optional, or have a default value to imply that they are optional.

Have a look at the *changelog* when you're interested in what changed between Tiled versions.

*A DTD-file (Document Type Definition) is served at http://mapeditor.org/dtd/1.0/map.dtd. This file is not up-to-date but might be useful for XML-namespacing anyway.*

*Note to implementors: When parsing TMX files, follow XML parsing guidelines. If an invalid element is found, it should generally be ignored (or cause a warning). When there are multiple copies of an element that should only appear once, use the first parsed option. Unknown attributes or element tags should also be ignored. These behaviors make adding future features easier without breaking backwards compatibility, and allows custom variants and additions to work with existing tools.*

## 19.1 <map>

- **version:** The TMX format version. Was "1.0" so far, and will be incremented to match minor Tiled releases.
- **tiledversion:** The Tiled version used to save the file (since Tiled 1.0.1). May be a date (for snapshot builds). (optional)
- **orientation:** Map orientation. Tiled supports "orthogonal", "isometric", "staggered" and "hexagonal" (since 0.11).

- **renderorder:** The order in which tiles on tile layers are rendered. Valid values are `right-down` (the default), `right-up`, `left-down` and `left-up`. In all cases, the map is drawn row-by-row. (only supported for orthogonal maps at the moment)
- **compressionlevel:** The compression level to use for tile layer data (defaults to -1, which means to use the algorithm default).
- **width:** The map width in tiles.
- **height:** The map height in tiles.
- **tilewidth:** The width of a tile.
- **tileheight:** The height of a tile.
- **hexsidelength:** Only for hexagonal maps. Determines the width or height (depending on the staggered axis) of the tile's edge, in pixels.
- **staggeraxis:** For staggered and hexagonal maps, determines which axis ("x" or "y") is staggered. (since 0.11)
- **staggerindex:** For staggered and hexagonal maps, determines whether the "even" or "odd" indexes along the staggered axis are shifted. (since 0.11)
- **backgroundcolor:** The background color of the map. (optional, may include alpha value since 0.15 in the form `#AARRGGBB`. Defaults to fully transparent.)
- **nextlayerid:** Stores the next available ID for new layers. This number is stored to prevent reuse of the same ID after layers have been removed. (since 1.2) (defaults to the highest layer id in the file + 1)
- **nextobjectid:** Stores the next available ID for new objects. This number is stored to prevent reuse of the same ID after objects have been removed. (since 0.11) (defaults to the highest object id in the file + 1)
- **infinite:** Whether this map is infinite. An infinite map has no fixed size and can grow in all directions. Its layer data is stored in chunks. (`0` for false, `1` for true, defaults to 0)

The `tilewidth` and `tileheight` properties determine the general grid size of the map. The individual tiles may have different sizes. Larger tiles will extend at the top and right (anchored to the bottom left).

A map contains three different kinds of layers. Tile layers were once the only type, and are simply called `layer`, object layers have the `objectgroup` tag and image layers use the `imagelayer` tag. The order in which these layers appear is the order in which the layers are rendered by Tiled.

The `staggered` orientation refers to an isometric map using staggered axes.

Can contain at most one: *<properties>*

Can contain any number: *<tileset>*, *<layer>*, *<objectgroup>*, *<imagelayer>*, *<group>* (since 1.0), *<editorsettings>* (since 1.3)

## 19.2 <editorsettings>

This element contains various editor-specific settings, which are generally not relevant when reading a map.

Can contain: *<chunksize>*, *<export>*

### 19.2.1 <chunksize>

- **width:** The width of chunks used for infinite maps (default to 16).
- **height:** The width of chunks used for infinite maps (default to 16).

### 19.2.2 <export>

- **target:** The last file this map was exported to.
- **format:** The short name of the last format this map was exported as.

## 19.3 <tileset>

- **firstgid:** The first global tile ID of this tileset (this global ID maps to the first tile in this tileset).
- **source:** If this tileset is stored in an external TSX (Tile Set XML) file, this attribute refers to that file. That TSX file has the same structure as the `<tileset>` element described here. (There is the firstgid attribute missing and this source attribute is also not there. These two attributes are kept in the TMX map, since they are map specific.)
- **name:** The name of this tileset.
- **tilewidth:** The (maximum) width of the tiles in this tileset.
- **tileheight:** The (maximum) height of the tiles in this tileset.
- **spacing:** The spacing in pixels between the tiles in this tileset (applies to the tileset image, defaults to 0)
- **margin:** The margin around the tiles in this tileset (applies to the tileset image, defaults to 0)
- **tilecount:** The number of tiles in this tileset (since 0.13)
- **columns:** The number of tile columns in the tileset. For image collection tilesets it is editable and is used when displaying the tileset. (since 0.15)
- **objectalignment:** Controls the alignment for tile objects. Valid values are `unspecified`, `topleft`, `top`, `topright`, `left`, `center`, `right`, `bottomleft`, `bottom` and `bottomright`. The default value is `unspecified`, for compatibility reasons. When unspecified, tile objects use `bottomleft` in orthogonal mode and `bottom` in isometric mode. (since 1.4)

If there are multiple `<tileset>` elements, they are in ascending order of their `firstgid` attribute. The first tileset always has a `firstgid` value of 1. Since Tiled 0.15, image collection tilesets do not necessarily number their tiles consecutively since gaps can occur when removing tiles.

Image collection tilesets have no `<image>` tag. Instead, each tile has an `<image>` tag.

Can contain at most one: *<image>*, *<tileoffset>*, *<grid>* (since 1.0), *<properties>*, *<terraintypes>*, *<wangsets>* (since 1.1),

Can contain any number: *<tile>*

### 19.3.1 <tileoffset>

- **x:** Horizontal offset in pixels. (defaults to 0)
- **y:** Vertical offset in pixels (positive is down, defaults to 0)

This element is used to specify an offset in pixels, to be applied when drawing a tile from the related tileset. When not present, no offset is applied.

### 19.3.2 <grid>

- **orientation:** Orientation of the grid for the tiles in this tileset (`orthogonal` or `isometric`, defaults to `orthogonal`)
- **width:** Width of a grid cell
- **height:** Height of a grid cell

This element is only used in case of isometric orientation, and determines how tile overlays for terrain and collision information are rendered.

### 19.3.3 <image>

- **format:** Used for embedded images, in combination with a `data` child element. Valid values are file extensions like `png`, `gif`, `jpg`, `bmp`, etc.
- *id:* Used by some versions of Tiled Java. Deprecated and unsupported by Tiled Qt.
- **source:** The reference to the tileset image file (Tiled supports most common image formats). Only used if the image is not embedded.
- **trans:** Defines a specific color that is treated as transparent (example value: "#FF00FF" for magenta). Up until Tiled 0.12, this value is written out without a # but this is planned to change. (optional)
- **width:** The image width in pixels (optional, used for tile index correction when the image changes)
- **height:** The image height in pixels (optional)

Note that it is not currently possible to use Tiled to create maps with embedded image data, even though the TMX format supports this. It is possible to create such maps using `libtiled` (Qt/C++) or tmxlib (Python).

Can contain at most one: *<data>*

### 19.3.4 <terraintypes>

This element defines an array of terrain types, which can be referenced from the `terrain` attribute of the `tile` element.

Can contain any number: *<terrain>*

#### <terrain>

- **name:** The name of the terrain type.
- **tile:** The local tile-id of the tile that represents the terrain visually.

Can contain at most one: *<properties>*

### 19.3.5 <tile>

- **id:** The local tile ID within its tileset.
- **type:** The type of the tile. Refers to an object type and is used by tile objects. (optional) (since 1.0)
- **terrain:** Defines the terrain type of each corner of the tile, given as comma-separated indexes in the terrain types array in the order top-left, top-right, bottom-left, bottom-right. Leaving out a value means that corner has no terrain. (optional)

- **probability:** A percentage indicating the probability that this tile is chosen when it competes with others while editing with the terrain tool. (defaults to 0)

Can contain at most one: *<properties>*, *<image>* (since 0.9), *<objectgroup>*, *<animation>*

#### <animation>

Contains a list of animation frames.

Each tile can have exactly one animation associated with it. In the future, there could be support for multiple named animations on a tile.

Can contain any number: *<frame>*

#### <frame>

- **tileid:** The local ID of a tile within the parent *<tileset>*.
- **duration:** How long (in milliseconds) this frame should be displayed before advancing to the next frame.

### 19.3.6 <wangsets>

Contains the list of Wang sets defined for this tileset.

Can contain any number: *<wangset>*

#### <wangset>

Defines a list of corner colors and a list of edge colors, and any number of Wang tiles using these colors.

- **name**: The name of the Wang set.
- **tile**: The tile ID of the tile representing this Wang set.

Can contain at most one: *<properties>*

Can contain up to 15 (each): *<wangcornercolor>*, *<wangedgecolor>*

Can contain any number: *<wangtile>*

#### <wangcornercolor>

A color that can be used to define the corner of a Wang tile.

- **name**: The name of this color.
- **color**: The color in `#RRGGBB` format (example: `#c17d11`).
- **tile**: The tile ID of the tile representing this color.
- **probability**: The relative probability that this color is chosen over others in case of multiple options. (defaults to 0)

#### <wangedgecolor>

A color that can be used to define the edge of a Wang tile.

- **name**: The name of this color.
- **color**: The color in `#RRGGBB` format (example: `#c17d11`).
- **tile**: The tile ID of the tile representing this color.
- **probability**: The relative probability that this color is chosen over others in case of multiple options. (defaults to 0)

#### <wangtile>

Defines a Wang tile, by referring to a tile in the tileset and associating it with a certain Wang ID.

- **tileid**: The tile ID.
- **wangid**: The Wang ID, which is a 32-bit unsigned integer stored in the format `0xCECECECE` (where each C is a corner color and each E is an edge color, from right to left clockwise, starting with the top edge)
- **hflip**: Whether the tile is flipped horizontally. This only affects the tile image, it does not change the meaning of the wangid. See *Tile flipping* for more info. (defaults to false)
- **vflip**: Whether the tile is flipped vertically. This only affects the tile image, it does not change the meaning of the wangid. See *Tile flipping* for more info. (defaults to false)
- **dflip**: Whether the tile is flipped on its diagonal. This only affects the tile image, it does not change the meaning of the wangid. See *Tile flipping* for more info. (defaults to false)

## 19.4 <layer>

All *<tileset>* tags shall occur before the first *<layer>* tag so that parsers may rely on having the tilesets before needing to resolve tiles.

- **id:** Unique ID of the layer. Each layer that added to a map gets a unique id. Even if a layer is deleted, no layer ever gets the same ID. Can not be changed in Tiled. (since Tiled 1.2)
- **name:** The name of the layer. (defaults to "")
- *x:* The x coordinate of the layer in tiles. Defaults to 0 and can not be changed in Tiled.
- *y:* The y coordinate of the layer in tiles. Defaults to 0 and can not be changed in Tiled.
- **width:** The width of the layer in tiles. Always the same as the map width for fixed-size maps.
- **height:** The height of the layer in tiles. Always the same as the map height for fixed-size maps.
- **opacity:** The opacity of the layer as a value from 0 to 1. Defaults to 1.
- **visible:** Whether the layer is shown (1) or hidden (0). Defaults to 1.
- **offsetx:** Rendering offset for this layer in pixels. Defaults to 0. (since 0.14)
- **offsety:** Rendering offset for this layer in pixels. Defaults to 0. (since 0.14)

Can contain at most one: *<properties>*, *<data>*

### 19.4.1 <data>

- **encoding:** The encoding used to encode the tile layer data. When used, it can be "base64" and "csv" at the moment. (optional)
- **compression:** The compression used to compress the tile layer data. Tiled supports "gzip", "zlib", and "zstd". (zstd supported since 1.3)

When no encoding or compression is given, the tiles are stored as individual XML `tile` elements. Next to that, the easiest format to parse is the "csv" (comma separated values) format.

The base64-encoded and optionally compressed layer data is somewhat more complicated to parse. First you need to base64-decode it, then you may need to decompress it. Now you have an array of bytes, which should be interpreted as an array of unsigned 32-bit integers using little-endian byte ordering.

Whatever format you choose for your layer data, you will always end up with so called "global tile IDs" (gids). They are global, since they may refer to a tile from any of the tilesets used by the map. In order to find out from which tileset the tile is you need to find the tileset with the highest `firstgid` that is still lower or equal than the gid. The tilesets are always stored with increasing `firstgid`s.

Can contain any number: *<tile>*, *<chunk>*

#### Tile flipping

The highest three bits of the gid store the flipped states. Bit 32 is used for storing whether the tile is horizontally flipped, bit 31 is used for the vertically flipped tiles and bit 30 indicates whether the tile is flipped (anti) diagonally, enabling tile rotation. These bits have to be read and cleared before you can find out which tileset a tile belongs to.

When rendering a tile, the order of operation matters. The diagonal flip (x/y axis swap) is done first, followed by the horizontal and vertical flips.

The following C++ pseudo-code should make it all clear:

```
// Bits on the far end of the 32-bit global tile ID are used for tile flags
const unsigned FLIPPED_HORIZONTALLY_FLAG = 0x80000000;
const unsigned FLIPPED_VERTICALLY_FLAG   = 0x40000000;
const unsigned FLIPPED_DIAGONALLY_FLAG   = 0x20000000;

...

// Extract the contents of the <data> element
string tile_data = ...

unsigned char *data = decompress(base64_decode(tile_data));
unsigned tile_index = 0;

// Here you should check that the data has the right size
// (map_width * map_height * 4)

for (int y = 0; y < map_height; ++y) {
  for (int x = 0; x < map_width; ++x) {
    unsigned global_tile_id = data[tile_index] |
                              data[tile_index + 1] << 8 |
                              data[tile_index + 2] << 16 |
                              data[tile_index + 3] << 24;
    tile_index += 4;

    // Read out the flags
```

(continues on next page)

(continued from previous page)

```
    bool flipped_horizontally = (global_tile_id & FLIPPED_HORIZONTALLY_FLAG);
    bool flipped_vertically = (global_tile_id & FLIPPED_VERTICALLY_FLAG);
    bool flipped_diagonally = (global_tile_id & FLIPPED_DIAGONALLY_FLAG);

    // Clear the flags
    global_tile_id &= ~(FLIPPED_HORIZONTALLY_FLAG |
                        FLIPPED_VERTICALLY_FLAG |
                        FLIPPED_DIAGONALLY_FLAG);

    // Resolve the tile
    for (int i = tileset_count - 1; i >= 0; --i) {
      Tileset *tileset = tilesets[i];

      if (tileset->first_gid() <= global_tile_id) {
        tiles[y][x] = tileset->tileAt(global_tile_id - tileset->first_gid());
        break;
      }
    }
  }
}
```

(Since the above code was put together on this wiki page and can't be directly tested, please make sure to report any errors you encounter when basing your parsing code on it, thanks.)

### 19.4.2 <chunk>

- **x:** The x coordinate of the chunk in tiles.
- **y:** The y coordinate of the chunk in tiles.
- **width:** The width of the chunk in tiles.
- **height:** The height of the chunk in tiles.

This is currently added only for infinite maps. The contents of a chunk element is same as that of the `data` element, except it stores the data of the area specified in the attributes.

Can contain any number: *<tile>*

### 19.4.3 <tile>

- **gid:** The global tile ID (default: 0).

Not to be confused with the `tile` element inside a `tileset`, this element defines the value of a single tile on a tile layer. This is however the most inefficient way of storing the tile layer data, and should generally be avoided.

## 19.5 <objectgroup>

- **id:** Unique ID of the layer. Each layer that added to a map gets a unique id. Even if a layer is deleted, no layer ever gets the same ID. Can not be changed in Tiled. (since Tiled 1.2)
- **name:** The name of the object group. (defaults to "")
- **color:** The color used to display the objects in this group. (defaults to gray ("#a0a0a4"))

- *x:* The x coordinate of the object group in tiles. Defaults to 0 and can no longer be changed in Tiled.
- *y:* The y coordinate of the object group in tiles. Defaults to 0 and can no longer be changed in Tiled.
- *width:* The width of the object group in tiles. Meaningless.
- *height:* The height of the object group in tiles. Meaningless.
- **opacity:** The opacity of the layer as a value from 0 to 1. (defaults to 1)
- **visible:** Whether the layer is shown (1) or hidden (0). (defaults to 1)
- **offsetx:** Rendering offset for this object group in pixels. (defaults to 0) (since 0.14)
- **offsety:** Rendering offset for this object group in pixels. (defaults to 0) (since 0.14)
- **draworder:** Whether the objects are drawn according to the order of appearance ("index") or sorted by their y-coordinate ("topdown"). (defaults to "topdown")

The object group is in fact a map layer, and is hence called "object layer" in Tiled.

Can contain at most one: *<properties>*

Can contain any number: *<object>*

### 19.5.1 <object>

- **id:** Unique ID of the object. Each object that is placed on a map gets a unique id. Even if an object was deleted, no object gets the same ID. Can not be changed in Tiled. (since Tiled 0.11)
- **name:** The name of the object. An arbitrary string. (defaults to "")
- **type:** The type of the object. An arbitrary string. (defaults to "")
- **x:** The x coordinate of the object in pixels. (defaults to 0)
- **y:** The y coordinate of the object in pixels. (defaults to 0)
- **width:** The width of the object in pixels. (defaults to 0)
- **height:** The height of the object in pixels. (defaults to 0)
- **rotation:** The rotation of the object in degrees clockwise around (x, y). (defaults to 0)
- **gid:** A reference to a tile. (optional)
- **visible:** Whether the object is shown (1) or hidden (0). (defaults to 1)
- **template:** A reference to a *template file*. (optional)

While tile layers are very suitable for anything repetitive aligned to the tile grid, sometimes you want to annotate your map with other information, not necessarily aligned to the grid. Hence the objects have their coordinates and size in pixels, but you can still easily align that to the grid when you want to.

You generally use objects to add custom information to your tile map, such as spawn points, warps, exits, etc.

When the object has a `gid` set, then it is represented by the image of the tile with that global ID. The image alignment currently depends on the map orientation. In orthogonal orientation it's aligned to the bottom-left while in isometric it's aligned to the bottom-center. The image will rotate around the bottom-left or bottom-center, respectively.

When the object has a `template` set, it will borrow all the properties from the specified template, properties saved with the object will have higher priority, i.e. they will override the template properties.

Can contain at most one: *<properties>*, *<ellipse>* (since 0.9), *<point>* (since 1.1), *<polygon>*, *<polyline>*, *<text>* (since 1.0)

### 19.5.2 <ellipse>

Used to mark an object as an ellipse. The existing `x`, `y`, `width` and `height` attributes are used to determine the size of the ellipse.

### 19.5.3 <point>

Used to mark an object as a point. The existing `x` and `y` attributes are used to determine the position of the point.

### 19.5.4 <polygon>

- **points:** A list of x,y coordinates in pixels.

Each `polygon` object is made up of a space-delimited list of x,y coordinates. The origin for these coordinates is the location of the parent `object`. By default, the first point is created as 0,0 denoting that the point will originate exactly where the `object` is placed.

### 19.5.5 <polyline>

- **points:** A list of x,y coordinates in pixels.

A `polyline` follows the same placement definition as a `polygon` object.

### 19.5.6 <text>

- **fontfamily:** The font family used (defaults to "sans-serif")
- **pixelsize:** The size of the font in pixels (not using points, because other sizes in the TMX format are also using pixels) (defaults to 16)
- **wrap:** Whether word wrapping is enabled (1) or disabled (0). (defaults to 0)
- **color:** Color of the text in `#AARRGGBB` or `#RRGGBB` format (defaults to #000000)
- **bold:** Whether the font is bold (1) or not (0). (defaults to 0)
- **italic:** Whether the font is italic (1) or not (0). (defaults to 0)
- **underline:** Whether a line should be drawn below the text (1) or not (0). (defaults to 0)
- **strikeout:** Whether a line should be drawn through the text (1) or not (0). (defaults to 0)
- **kerning:** Whether kerning should be used while rendering the text (1) or not (0). (defaults to 1)
- **halign:** Horizontal alignment of the text within the object (`left`, `center`, `right` or `justify`, defaults to `left`) (since Tiled 1.2.1)
- **valign:** Vertical alignment of the text within the object (`top` , `center` or `bottom`, defaults to `top`)

Used to mark an object as a text object. Contains the actual text as character data.

For alignment purposes, the bottom of the text is the descender height of the font, and the top of the text is the ascender height of the font. For example, `bottom` alignment of the word "cat" will leave some space below the text, even though it is unused for this word with most fonts. Similarly, `top` alignment of the word "cat" will leave some space above the "t" with most fonts, because this space is used for diacritics.

If the text is larger than the object's bounds, it is clipped to the bounds of the object.

## 19.6 <imagelayer>

- **id:** Unique ID of the layer. Each layer that added to a map gets a unique id. Even if a layer is deleted, no layer ever gets the same ID. Can not be changed in Tiled. (since Tiled 1.2)
- **name:** The name of the image layer. (defaults to "")
- **offsetx:** Rendering offset of the image layer in pixels. (defaults to 0) (since 0.15)
- **offsety:** Rendering offset of the image layer in pixels. (defaults to 0) (since 0.15)
- *x:* The x position of the image layer in pixels. (defaults to 0, deprecated since 0.15)
- *y:* The y position of the image layer in pixels. (defaults to 0, deprecated since 0.15)
- **opacity:** The opacity of the layer as a value from 0 to 1. (defaults to 1)
- **visible:** Whether the layer is shown (1) or hidden (0). (defaults to 1)

A layer consisting of a single image.

Can contain at most one: *<properties>*, *<image>*

## 19.7 <group>

- **id:** Unique ID of the layer. Each layer that added to a map gets a unique id. Even if a layer is deleted, no layer ever gets the same ID. Can not be changed in Tiled. (since Tiled 1.2)
- **name:** The name of the group layer. (defaults to "")
- **offsetx:** Rendering offset of the group layer in pixels. (defaults to 0)
- **offsety:** Rendering offset of the group layer in pixels. (defaults to 0)
- **opacity:** The opacity of the layer as a value from 0 to 1. (defaults to 1)
- **visible:** Whether the layer is shown (1) or hidden (0). (defaults to 1)

A group layer, used to organize the layers of the map in a hierarchy. Its attributes `offsetx`, `offsety`, `opacity` and `visible` recursively affect child layers.

Can contain at most one: *<properties>*

Can contain any number: *<layer>*, *<objectgroup>*, *<imagelayer>*, *<group>*

## 19.8 <properties>

Wraps any number of custom properties. Can be used as a child of the `map`, `tileset`, `tile` (when part of a `tileset`), `terrain`, `layer`, `objectgroup`, `object`, `imagelayer` and `group` elements.

Can contain any number: *<property>*

### 19.8.1 <property>

- **name:** The name of the property.
- **type:** The type of the property. Can be `string` (default), `int`, `float`, `bool`, `color`, `file` or `object` (since 0.16, with `color` and `file` added in 0.17, and `object` added in 1.4).

- **value:** The value of the property. (default string is “”, default number is 0, default boolean is “false”, default color is #00000000, default file is “.” (the current file’s parent directory))

Boolean properties have a value of either “true” or “false”.

Color properties are stored in the format `#AARRGGBB`.

File properties are stored as paths relative from the location of the map file.

Object properties can reference any object on the same map and are stored as an integer (the ID of the referenced object, or 0 when no object is referenced). When used on objects in the Tile Collision Editor, they can only refer to other objects on the same tile.

When a string property contains newlines, the current version of Tiled will write out the value as characters contained inside the `property` element rather than as the `value` attribute. It is possible that a future version of the TMX format will switch to always saving property values inside the element rather than as an attribute.

## 19.9 Template Files

Templates are saved in their own file, and are referenced by *objects* that are template instances.

### 19.9.1 <template>

The template root element contains the saved *map object* and a *tileset* element that points to an external tileset, if the object is a tile object.

Example of a template file:

```
<?xml version="1.0" encoding="UTF-8"?>
<template>
 <tileset firstgid="1" source="desert.tsx"/>
 <object name="cactus" gid="31" width="81" height="101"/>
</template>
```

Can contain at most one: *<tileset>*, *<object>*

---

Fig. 1: Creative Commons License

CHAPTER 20

# TMX Changelog

Below are described the changes/additions that were made to the *TMX Map Format* for recent versions of Tiled.

## 20.1 Tiled 1.4

- Added the `objectalignment` attribute to the *<tileset>* element, allowing the tileset to control the alignment used for tile objects.

## 20.2 Tiled 1.3

- Added an *<editorsettings>* element, which is used to store editor specific options that are generally not relevant when loading a map.

## 20.3 Tiled 1.2.1

- Text objects can now get their horizontal alignment saved as `justify`. This option existed in the UI before but wasn't saved properly.

## 20.4 Tiled 1.2

- Added an `id` attribute to the *<layer>*, *<objectgroup>*, *<imagelayer>* and *<group>* elements, which stores a map-unique ID of the layer.
- Added a `nextlayerid` attribute to the *<map>* element, which stores the next available ID for new layers. This number is stored to prevent reuse of the same ID after layers have been removed.

## 20.5 Tiled 1.1

- Added a *map.infinite* attribute, which indicates whether the map is considered unbounded. Tile layer data for infinite maps is stored in chunks.
- A new <*chunk*> element was added for infinite maps which contains the similar content as <*data*>, except it stores the data of the area specified by its `x`, `y`, `width` and `height` attributes.
- *Templates* were added, a template is an *external file* referenced by template instance objects:

```
<object id="3" template="diamond.tx" x="200" y="100"/>
```

- Tilesets can now contain *Wang tiles*. They are saved in the new <*wangsets*> element.
- A new <*point*> child element was added to <*object*>, which marks point objects. Point objects do not have a size or rotation.

## 20.6 Tiled 1.0

- A new <*group*> element was added which is a group layer that can have other layers as child elements. This means layers now form a hierarchy.
- Added Text objects, identified by a new <*text*> element which is used as a child of the <*object*> element.
- Added a *tile.type* attribute for supporting *Typed Tiles*.

## 20.7 Tiled 0.18

*No file format changes.*

## 20.8 Tiled 0.17

- Added `color` and `file` as possible values for the *property.type* attribute.
- Added support for editing multi-line string properties, which are written out differently.

## 20.9 Tiled 0.16

- The <*property*> element gained a `type` attribute, storing the type of the value. Currently supported types are `string` (the default), `int`, `float` and `bool`.

## 20.10 Tiled 0.15

- The `offsetx` and `offsety` attributes are now also used for <*imagelayer*> elements, replacing the `x` and `y` attributes previously used. This change was made for consistency with the other layer types.
- The tiles in an image collection tileset are no longer guaranteed to be consecutive, because removing tiles from the collection will no longer change the IDs of other tiles.

- The pure XML and Gzip-compressed tile layer data formats were deprecated, since they didn't have any advantage over other formats. Remaining formats are CSV, base64 and Zlib-compressed layer data.
- Added `columns` attribute to the *<tileset>* element, which specifies the number of tile columns in the tileset. For image collection tilesets it is editable and is used when displaying the tileset.
- The `backgroundcolor` attribute of the *<map>* element will now take the format `#AARRGGBB` when its alpha value differs from 255. Previously the alpha value was silently discarded.

## 20.11 Tiled 0.14

- Added optional `offsetx` and `offsety` attributes to the `layer` and `objectgroup` elements. These specify an offset in pixels that is to be applied when rendering the layer. The default values are 0.

## 20.12 Tiled 0.13

- Added an optional `tilecount` attribute to the `tileset` element, which is written by Tiled to help parsers determine the amount of memory to allocate for tile data.

## 20.13 Tiled 0.12

- Previously tile objects never had `width` and `height` properties, though the format technically allowed this. Now these properties are used to store the size the image should be rendered at. The default values for these attributes are the dimensions of the tile image.

## 20.14 Tiled 0.11

- Added `hexagonal` to the supported values for the `orientation` attribute on the `map` element. This also adds `staggerindex` (`even` or `odd`) and `staggeraxis` (`x` or `y`) and `hexsidelength` (integer value) attributes to the `map` element, in order to support the many variations of staggered hexagonal. The new `staggerindex` and `staggeraxis` attributes are also supported when using the `staggered` map orientation.
- Added an `id` attribute to the `object` element, which stores a map-unique ID of the object.
- Added a `nextobjectid` attribute to the `map` element, which stores the next available ID for new objects. This number is stored to prevent reuse of the same ID after objects have been removed.

## 20.15 Tiled 0.10

- Tile objects can now be horizontally or vertically flipped. This is stored in the `gid` attribute using the same mechanism as for regular tiles. The image is expected to be flipped without affecting its position, same way as flipped tiles.
- Objects can be rotated freely. The rotation is stored in degrees as a `rotation` attribute, with positive rotation going clockwise.

- The render order of the tiles on tile layers can be configured in a number of ways through a new `renderorder` property on the `map` element. Valid values are `right-down` (the default), `right-up`, `left-down` and `left-up`. In all cases, the map is drawn row-by-row. This is only supported for orthogonal maps at the moment.
- The render order of objects on object layers can be configured to be either sorted by their y-coordinate (previous behavior and still the default) or simply the order of appearance in the map file. The latter enables manual control over the drawing order with actions that "Raise" and "Lower" selected objects. It is controlled by the `draworder` property on the `objectgroup` element, which can be either `topdown` (default) or `index`.
- Tiles can have an `objectgroup` child element, which can contain objects that define the collision shape to use for that tile. This information can be edited in the new Tile Collision Editor.
- Tiles can have a single looping animation associated with them using an `animation` child element. Each frame of the animation refers to a local tile ID from this tileset and defines the frame duration in milliseconds. Example:

```
<tileset ...>
...
<tile id="[n]">
    <animation>
        <frame tileid="0" duration="100"/>
        <frame tileid="1" duration="100"/>
        <frame tileid="2" duration="100"/>
    </animation>
</tile>
</tileset>
```

## 20.16 Tiled 0.9

- Per-object visibility flag is saved (defaults to 1):

```
<object visible="0|1">
```

- Terrain information was added to tileset definitions (this is generally not very relevant for games):

```
<tileset ...>
...
<terraintypes>
    <terrain name="Name" tile="local_id"/>
</terraintypes>
<tile id="local_id" terrain="[n],[n],[n],[n]" probability="percentage"/>
...
</tileset>
```

- There is preliminary support for a "staggered" (isometric) projection (new value for the `orientation` attribute of the `map` element).
- A basic image layer type was added:

```
<imagelayer ...>
<image source="..."/>
</imagelayer>
```

- Added ellipse object shape. Same parameters as rectangular objects, but marked as ellipse with a child element:

```
<object ...>
<ellipse/>
</object>
```

- Added map property for specifying the background color:

```
<map ... backgroundcolor="#RRGGBB">
```

- Added initial (non-GUI) support for individual and/or embedded tile images (since there is no way to set this up in Tiled Qt but only in Tiled Java or with pytmxlib, this is not very important to support at the moment):

```
<tileset ...>
<tile id="[n]">
    <!-- an embedded image -->
    <image format="png">
        <data encoding="base64">
            ...
        </data>
    </image>
</tile>
<tile id="[n]">
    <!-- an individually referenced image for a single tile -->
    <image source="file.png"/>
</tile>
...
</tileset>
```

## 20.17 Tiled 0.8

- Tilesets can now have custom properties (using the `properties` child element, just like everything else).
- Tilesets now support defining a drawing offset in pixels, which is to be used when drawing any tiles from that tileset. Example:

```
<tileset name="perspective_walls" tilewidth="64" tileheight="64">
<tileoffset x="-32" y="0"/>
...
</tileset>
```

- Support for tile rotation in 90-degree increments was added by using the third most significant bit in the global tile id. This new bit means "anti-diagonal flip", which swaps the x and y axis when rendering a tile.

CHAPTER 21

# JSON Map Format

Tiled can export maps as JSON files. To do so, simply select "File > Export As" and select the JSON file type. You can export json from the command line with the `--export-map` option.

The fields found in the JSON format differ slightly from those in the *TMX Map Format*, but the meanings should remain the same.

The following fields can be found in a Tiled JSON file:

## 21.1 Map

| Field | Type | Description |
|---|---|---|
| backgroundcolor | string | Hex-formatted color (#RRGGBB or #AARRGGBB) (optional) |
| height | int | Number of tile rows |
| hexsidelength | int | Length of the side of a hex tile in pixels (hexagonal maps only) |
| infinite | bool | Whether the map has infinite dimensions |
| layers | array | Array of *Layers* |
| nextlayerid | int | Auto-increments for each layer |
| nextobjectid | int | Auto-increments for each placed object |
| orientation | string | `orthogonal`, `isometric`, `staggered` or `hexagonal` |
| properties | array | Array of *Properties* |
| renderorder | string | `right-down` (the default), `right-up`, `left-down` or `left-up` (orthogonal maps only) |
| staggeraxis | string | `x` or `y` (staggered / hexagonal maps only) |
| staggerindex | string | `odd` or `even` (staggered / hexagonal maps only) |
| tiledversion | string | The Tiled version used to save the file |
| tileheight | int | Map grid height |
| tilesets | array | Array of *Tilesets* |
| tilewidth | int | Map grid width |
| type | string | `map` (since 1.0) |
| version | number | The JSON format version |
| width | int | Number of tile columns |

### 21.1.1 Map Example

```
{
  "backgroundcolor":"#656667",
  "height":4,
  "layers":[ ],
  "nextobjectid":1,
  "orientation":"orthogonal",
  "properties":[
    {
      "name":"mapProperty1",
      "type":"one",
      "value":"string"
    },
    {
      "name":"mapProperty2",
      "type":"two",
      "value":"string"
    }],
  "renderorder":"right-down",
  "tileheight":32,
  "tilesets":[ ],
  "tilewidth":32,
  "version":1,
  "tiledversion":"1.0.3",
  "width":4
}
```

## 21.2 Layer

| Field | Type | Description |
|---|---|---|
| chunks | array | Array of *chunks* (optional). `tilelayer` only. |
| compression | string | `zlib`, `gzip` or empty (default). `tilelayer` only. |
| data | array or string | Array of `unsigned int` (GIDs) or base64-encoded data. `tilelayer` only. |
| draworder | string | `topdown` (default) or `index`. `objectgroup` only. |
| encoding | string | `csv` (default) or `base64`. `tilelayer` only. |
| height | int | Row count. Same as map height for fixed-size maps. |
| id | int | Incremental id - unique across all layers |
| image | string | Image used by this layer. `imagelayer` only. |
| layers | array | Array of *layers*. `group` only. |
| name | string | Name assigned to this layer |
| objects | array | Array of *objects*. `objectgroup` only. |
| offsetx | double | Horizontal layer offset in pixels (default: 0) |
| offsety | double | Vertical layer offset in pixels (default: 0) |
| opacity | double | Value between 0 and 1 |
| properties | array | Array of *Properties* |
| startx | int | X coordinate where layer content starts (for infinite maps) |
| starty | int | Y coordinate where layer content starts (for infinite maps) |
| transparentcolor | string | Hex-formatted color (#RRGGBB) (optional). `imagelayer` only. |
| type | string | `tilelayer`, `objectgroup`, `imagelayer` or `group` |
| visible | bool | Whether layer is shown or hidden in editor |
| width | int | Column count. Same as map width for fixed-size maps. |
| x | int | Horizontal layer offset in tiles. Always 0. |
| y | int | Vertical layer offset in tiles. Always 0. |

### 21.2.1 Tile Layer Example

```
{
  "data":[1, 2, 1, 2, 3, 1, 3, 1, 2, 2, 3, 3, 4, 4, 4, 1],
  "height":4,
  "name":"ground",
  "opacity":1,
  "properties":[
    {
      "name":"tileLayerProp",
      "type":"int",
      "value":1
    }],
  "type":"tilelayer",
  "visible":true,
  "width":4,
  "x":0,
  "y":0
}
```

### 21.2.2 Object Layer Example

```
{
  "draworder":"topdown",
  "height":0,
  "name":"people",
  "objects":[ ],
  "opacity":1,
  "properties":[
    {
      "name":"layerProp1",
      "type":"string",
      "value":"someStringValue"
    }],
  "type":"objectgroup",
  "visible":true,
  "width":0,
  "x":0,
  "y":0
}
```

## 21.3 Chunk

Chunks are used to store the tile layer data for *infinite maps*.

| Field | Type | Description |
|---|---|---|
| data | array or string | Array of `unsigned int` (GIDs) or base64-encoded data |
| height | int | Height in tiles |
| width | int | Width in tiles |
| x | int | X coordinate in tiles |
| y | int | Y coordinate in tiles |

### 21.3.1 Chunk Example

```
{
  "data":[1, 2, 1, 2, 3, 1, 3, 1, 2, 2, 3, 3, 4, 4, 4, 1, ...],
  "height":16,
  "width":16,
  "x":0,
  "y":-16,
}
```

## 21.4 Object

| Field | Type | Description |
|---|---|---|
| ellipse | bool | Used to mark an object as an ellipse |
| gid | int | Global tile ID, only if object represents a tile |
| height | double | Height in pixels. |
| id | int | Incremental id, unique across all objects |
| name | string | String assigned to name field in editor |
| point | bool | Used to mark an object as a point |
| polygon | array | Array of *Points*, in case the object is a polygon |
| polyline | array | Array of *Points*, in case the object is a polyline |
| properties | array | Array of *Properties* |
| rotation | double | Angle in degrees clockwise |
| template | string | Reference to a template file, in case object is a *template instance* |
| text | *Text* | Only used for text objects |
| type | string | String assigned to type field in editor |
| visible | bool | Whether object is shown in editor. |
| width | double | Width in pixels. |
| x | double | X coordinate in pixels |
| y | double | Y coordinate in pixels |

### 21.4.1 Object Example

```
{
  "gid":5,
  "height":0,
  "id":1,
  "name":"villager",
  "properties":[
    {
      "name":"hp",
      "type":"int",
      "value":12
    }],
  "rotation":0,
  "type":"npc",
  "visible":true,
  "width":0,
  "x":32,
  "y":32
}
```

### 21.4.2 Ellipse Example

```
{
  "ellipse":true,
  "height":152,
  "id":13,
  "name":"",
  "rotation":0,
```

(continues on next page)

(continued from previous page)

```
  "type":"",
  "visible":true,
  "width":248,
  "x":560,
  "y":808
}
```

### 21.4.3 Rectangle Example

```
{
  "height":184,
  "id":14,
  "name":"",
  "rotation":0,
  "type":"",
  "visible":true,
  "width":368,
  "x":576,
  "y":584
}
```

### 21.4.4 Point Example

```
{
  "point":true,
  "height":0,
  "id":20,
  "name":"",
  "rotation":0,
  "type":"",
  "visible":true,
  "width":0,
  "x":220,
  "y":350
}
```

### 21.4.5 Polygon Example

```
{
  "height":0,
  "id":15,
  "name":"",
  "polygon":[
  {
    "x":0,
    "y":0
  },
  {
    "x":152,
    "y":88
```

(continues on next page)

(continued from previous page)

```
  },
  {
    "x":136,
    "y":-128
  },
  {
    "x":80,
    "y":-280
  },
  {
    "x":16,
    "y":-288
  }],
  "rotation":0,
  "type":"",
  "visible":true,
  "width":0,
  "x":-176,
  "y":432
}
```

### 21.4.6 Polyline Example

```
{
  "height":0,
  "id":16,
  "name":"",
  "polyline":[
  {
    "x":0,
    "y":0
  },
  {
    "x":248,
    "y":-32
  },
  {
    "x":376,
    "y":72
  },
  {
    "x":544,
    "y":288
  },
  {
    "x":656,
    "y":120
  },
  {
    "x":512,
    "y":0
  }],
  "rotation":0,
  "type":"",
  "visible":true,
```

(continues on next page)

(continued from previous page)

```
  "width":0,
  "x":240,
  "y":88
}
```

### 21.4.7 Text Example

```
{
  "height":19,
  "id":15,
  "name":"",
  "text":
  {
    "text":"Hello World",
    "wrap":true
  },
  "rotation":0,
  "type":"",
  "visible":true,
  "width":248,
  "x":48,
  "y":136
}
```

## 21.5 Text

| Field | Type | Description |
|---|---|---|
| bold | bool | Whether to use a bold font (default: `false`) |
| color | string | Hex-formatted color (#RRGGBB or #AARRGGBB) (default: `#000000`) |
| fontfamily | string | Font family (default: `sans-serif`) |
| halign | string | Horizontal alignment (`center`, `right`, `justify` or `left` (default)) |
| italic | bool | Whether to use an italic font (default: `false`) |
| kerning | bool | Whether to use kerning when placing characters (default: `true`) |
| pixelsize | int | Pixel size of font (default: 16) |
| strikeout | bool | Whether to strike out the text (default: `false`) |
| text | string | Text |
| underline | bool | Whether to underline the text (default: `false`) |
| valign | string | Vertical alignment (`center`, `bottom` or `top` (default)) |
| wrap | bool | Whether the text is wrapped within the object bounds (default: `false`) |

## 21.6 Tileset

| Field | Type | Description |
|---|---|---|
| backgroundcolor | string | Hex-formatted color (#RRGGBB or #AARRGGBB) (optional) |
| columns | int | The number of tile columns in the tileset |
| firstgid | int | GID corresponding to the first tile in the set |
| grid | *Grid* | (optional) |
| image | string | Image used for tiles in this set |
| imageheight | int | Height of source image in pixels |
| imagewidth | int | Width of source image in pixels |
| margin | int | Buffer between image edge and first tile (pixels) |
| name | string | Name given to this tileset |
| objectalignment | string | Alignment to use for tile objects (`unspecified` (default), `topleft`, `top`, `topright`, `left`, `center`, `right`, `bottomleft`, `bottom` or `bottomright`) (since 1.4) |
| properties | array | Array of *Properties* |
| source | string | The external file that contains this tilesets data |
| spacing | int | Spacing between adjacent tiles in image (pixels) |
| terrains | array | Array of *Terrains* (optional) |
| tilecount | int | The number of tiles in this tileset |
| tiledversion | string | The Tiled version used to save the file |
| tileheight | int | Maximum height of tiles in this set |
| tileoffset | *Tile Offset* | (optional) |
| tiles | array | Array of *Tiles* (optional) |
| tilewidth | int | Maximum width of tiles in this set |
| transparentcolor | string | Hex-formatted color (#RRGGBB) (optional) |
| type | string | `tileset` (for tileset files, since 1.0) |
| version | number | The JSON format version |
| wangsets | array | Array of *Wang sets* (since 1.1.5) |

Each tileset has a `firstgid` (first global ID) property which tells you the global ID of its first tile (the one with local tile ID 0). This allows you to map the global IDs back to the right tileset, and then calculate the local tile ID by subtracting the `firstgid` from the global tile ID. The first tileset always has a `firstgid` value of 1.

### 21.6.1 Grid

Specifies common grid settings used for tiles in a tileset. See *<grid>* in the TMX Map Format.

| Field | Type | Description |
|---|---|---|
| height | int | Cell height of tile grid |
| orientation | string | `orthogonal` (default) or `isometric` |
| width | int | Cell width of tile grid |

### 21.6.2 Tile Offset

See *<tileoffset>* in the TMX Map Format.

| Field | Type | Description |
|---|---|---|
| x | int | Horizontal offset in pixels |
| y | int | Vertical offset in pixels (positive is down) |

### 21.6.3 Tileset Example

```
{
 "columns":19,
 "firstgid":1,
 "image":"..\/image\/fishbaddie_parts.png",
 "imageheight":480,
 "imagewidth":640,
 "margin":3,
 "name":"",
 "properties":[
   {
     "name":"myProperty1",
     "type":"string",
     "value":"myProperty1_value"
   }],
 "spacing":1,
 "tilecount":266,
 "tileheight":32,
 "tilewidth":32
}
```

### 21.6.4 Tile (Definition)

| Field | Type | Description |
|---|---|---|
| animation | array | Array of *Frames* |
| id | int | Local ID of the tile |
| image | string | Image representing this tile (optional) |
| imageheight | int | Height of the tile image in pixels |
| imagewidth | int | Width of the tile image in pixels |
| objectgroup | *Layer* | Layer with type `objectgroup`, when collision shapes are specified (optional) |
| probability | double | Percentage chance this tile is chosen when competing with others in the editor (optional) |
| properties | array | Array of *Properties* |
| terrain | array | Index of terrain for each corner of tile (optional) |
| type | string | The type of the tile (optional) |

A tileset that associates information with each tile, like its image path or terrain type, may include a `tiles` array property. Each tile has an `id` property, which specifies the local ID within the tileset.

For the terrain information, each value is a length-4 array where each element is the index of a *terrain* on one corner of the tile. The order of indices is: top-left, top-right, bottom-left, bottom-right.

Example:

```
"tiles":[
  {
    "id":0,
    "properties":[
      {
        "name":"myProperty1",
        "type":"string",
        "value":"myProperty1_value"
      }],
    "terrain":[0, 0, 0, 0]
  },
  {
    "id":11,
    "properties":[
      {
        "name":"myProperty2",
        "type":"string",
        "value":"myProperty2_value"
      }],
    "terrain":[0, 1, 0, 1]
  },
  {
    "id":12,
    "properties":[
      {
        "name":"myProperty3",
        "type":"string",
        "value":"myProperty3_value"
      }],
    "terrain":[1, 1, 1, 1]
  }
]
```

### 21.6.5 Frame

| Field | Type | Description |
|---|---|---|
| duration | int | Frame duration in milliseconds |
| tileid | int | Local tile ID representing this frame |

### 21.6.6 Terrain

| Field | Type | Description |
|---|---|---|
| name | string | Name of terrain |
| properties | array | Array of *Properties* |
| tile | int | Local ID of tile representing terrain |

Example:

```
"terrains":[
{
  "name":"ground",
```

(continues on next page)

(continued from previous page)

```
  "tile":0
},
{
  "name":"chasm",
  "tile":12
},
{
  "name":"cliff",
  "tile":36
}],
```

### 21.6.7 Wang Set

| Field | Type | Description |
|---|---|---|
| cornercolors | array | Array of *Wang colors* |
| edgecolors | array | Array of *Wang colors* |
| name | string | Name of the Wang set |
| properties | array | Array of *Properties* |
| tile | int | Local ID of tile representing the Wang set |
| wangtiles | array | Array of *Wang tiles* |

#### Wang Color

| Field | Type | Description |
|---|---|---|
| color | string | Hex-formatted color (#RRGGBB or #AARRGGBB) |
| name | string | Name of the Wang color |
| probability | double | Probability used when randomizing |
| tile | int | Local ID of tile representing the Wang color |

Example:

```
{
  "color": "#d31313",
  "name": "Rails",
  "probability": 1,
  "tile": 18
}
```

#### Wang Tile

| Field | Type | Description |
|---|---|---|
| dflip | bool | Tile is flipped diagonally (default: `false`) |
| hflip | bool | Tile is flipped horizontally (default: `false`) |
| tileid | int | Local ID of tile |
| vflip | bool | Tile is flipped vertically (default: `false`) |
| wangid | array | Array of Wang color indexes (`uchar[8]`) |

Example:

```
{
  "dflip": false,
  "hflip": false,
  "tileid": 0,
  "vflip": false,
  "wangid": [2, 0, 1, 0, 1, 0, 2, 0]
}
```

## 21.7 Object Template

An object template is written to its own file and referenced by any instances of that template.

| Field | Type | Description |
|---|---|---|
| type | string | `template` |
| tileset | *Tileset* | External tileset used by the template (optional) |
| object | *Object* | The object instantiated by this template |

## 21.8 Property

| Field | Type | Description |
|---|---|---|
| name | string | Name of the property |
| type | string | Type of the property (`string` (default), `int`, `float`, `bool`, `color` or `file` (since 0.16, with `color` and `file` added in 0.17)) |
| value | value | Value of the property |

## 21.9 Point

A point on a polygon or a polyline, relative to the position of the object.

| Field | Type | Description |
|---|---|---|
| x | double | X coordinate in pixels |
| y | double | Y coordinate in pixels |

## 21.10 Changelog

### 21.10.1 Tiled 1.4

- Added `objectalignment` to the *Tileset* object.

### 21.10.2 Tiled 1.2

- Added `nextlayerid` to the *Map* object.
- Added `id` to the *Layer* object.

- The tiles in a *Tileset* are now stored as an array instead of an object. Previously the tile IDs were stored as string keys of the "tiles" object, now they are stored as `id` property of each *Tile* object.
- Custom tile properties are now stored within each *Tile* instead of being included as `tileproperties` in the *Tileset* object.
- Custom properties are now stored in an array instead of an object where the property names were the keys. Each property is now an object that stores the name, type and value of the property. The separate `propertytypes` and `tilepropertytypes` attributes have been removed.

### 21.10.3 Tiled 1.1

- Added a *chunked data format*, currently used for *infinite maps*.
- *Templates* were added. Templates can be stored as JSON files with an *Object Template* object.
- *Tilesets* can now contain *Wang tiles*. They are saved in the new *Wang Set* object (since Tiled 1.1.5).

CHAPTER 22

# Scripting

## 22.1 Introduction

Tiled can be extended with the use of JavaScript. Scripts can be used to implement *custom map formats*, *custom actions* and *new tools*. Scripts can also *automate actions based on signals*.

On startup, Tiled will execute any script files present in *extensions*. In addition it is possible to run scripts directly from *the console*. All scripts share a single JavaScript context.

**Note:** A few example scripts and links to existing Tiled extensions are provided at the Tiled Extensions repository: https://github.com/mapeditor/tiled-extensions

**Warning:** Most builds of Tiled use Qt 5.12 or later, which support ECMAScript 7. However, the builds for Windows XP and the snap release of Tiled currently rely on Qt 5.6 and Qt 5.9 respectively. ECMAScript 7 features will not work there and some other functionality is missing as well, like interrupting scripts on API usage errors and the creation of new Tiled-specific objects.

### 22.1.1 Scripted Extensions

Extensions are placed in a system-specific location. This folder can be opened from the Plugins tab in the *Preferences dialog*.

| | |
|---|---|
| **Windows** | `C:/Users/<USER>/AppData/Local/Tiled/extensions/` |
| **macOS** | `~/Library/Preferences/Tiled/extensions/` |
| **Linux** | `~/.config/tiled/extensions/` |

An extension can be placed directly in the extensions directory, or in a sub-directory. All scripts files found in these directories are executed on startup.

When any loaded script is changed or when any files are added/removed from the extensions directory, the script engine is automatically reinstantiated and the scripts are reloaded. This way there is no need to restart Tiled when installing extensions. It also makes it quick to iterate on a script until it works as intended.

Apart from scripts, extensions can include images that can be used as the icon for scripted actions or tools.

### 22.1.2 Console View

In the Console view (*View* > *Views and Toolbars* > *Console*) you will find a text entry where you can write or paste scripts to evaluate them.

You can use the Up/Down keys to navigate through previously entered script expressions.

### 22.1.3 Connecting to Signals

The script API provides signals to which functions can be connected. Currently, the tiled module has the most useful *set of signals*.

Properties usually will have related signals which can be used to detect changes to that property, but most of those are currently not implemented.

To connect to a signal, call its `connect` function and pass in a function object. In the following example, newly created maps automatically get their first tile layer removed:

```
tiled.assetCreated.connect(function(asset) {
    if (asset.layerCount > 0) {
        asset.removeLayerAt(0)
        tiled.log("assetCreated: Removed automatically added tile layer.")
    }
})
```

In some cases it will be necessary to later disconnect the function from the signal again. This can be done by defining the function separately and passing it into the `disconnect` function:

```
function onAssetCreated(asset) {
    // Do something...
}
```

(continues on next page)

(continued from previous page)

```
tiled.assetCreated.connect(onAssetCreated)
// ...
tiled.assetCreated.disconnect(onAssetCreated)
```

## 22.2 API Reference

### 22.2.1 tiled module

The `tiled` module is the main entry point and provides properties, functions and signals which are documented below.

#### Properties

| | |
|---|---|
| **version** : string *[read-only]* | Currently used version of Tiled. |
| **platform** : string *[read-only]* | Operating system. One of `windows`, `macos`, `linux` or `unix` (for any other UNIX-like system). |
| **arch** : string *[read-only]* | Processor architecture. One of `x64`, `x86` or `unknown`. |
| **actions** : [string] *[read-only]* | Available actions for *tiled.trigger()*. |
| **menus** : [string] *[read-only]* | Available menus for *tiled.extendMenu()*. |
| **activeAsset** : *Asset* | Currently selected asset, or `null` if no file is open. Can be assigned any open asset in order to change the active asset. |
| **openAssets** : array *[read-only]* | List of currently opened *assets*. |
| **mapEditor** : *MapEditor* | Access the editor used when editing maps. |
| **tilesetEditor** : *TilesetEditor* | Access the editor used when editing tilesets. |
| **tilesetFormats** : [string] *[read-only]* | List of supported tileset format names. Use *tilesetFormat* to get the corresponding format object to read and write files. (Since 1.4) |
| **mapFormats** : [string] *[read-only]* | List of supported map format names. Use *mapFormat* to get the corresponding format object to read and write files. (Since 1.4) |

#### Functions

**tiled.trigger(action** [string)][void] This function can be used to trigger any registered action. This includes most actions you would normally trigger through the menu or by using their shortcut.

Use the `tiled.actions` property to get a list of all available actions.

Actions that are checkable will toggle when triggered.

**tiled.executeCommand(name** [string, inTerminal][bool)][void] Executes the first custom command with the given name, as if it was triggered manually. Works also with commands that are not currently enabled.

Raises a script error if the command is not found.

**tiled.open(fileName** [string)][*Asset*] Requests to open the asset with the given file name. Returns a reference to the opened asset, or `null` in case there was a problem.

**tiled.close(asset** [*Asset*)][bool] Closes the given asset without checking for unsaved changes (to confirm the loss of any unsaved changes, set `activeAsset` and trigger the "Close" action instead).

**tiled.reload(asset** [*Asset*)][*Asset*] Reloads the given asset from disk, without checking for unsaved changes. This invalidates the previous script reference to the asset, hence the new reference is returned for convenience. Returns `null` if reloading failed.

**tiled.alert(text** [string [, title][string])][void] Shows a modal warning dialog to the user with the given text and optional title.

**tiled.confirm(text** [string [, title][string])][bool] Shows a yes/no dialog to the user with the given text and optional title. Returns `true` or `false`.

**tiled.prompt(label** [string [, text][string [, title][string]])][string] Shows a dialog that asks the user to enter some text, along with the given label and optional title. The optional `text` parameter provides the initial value of the text. Returns the entered text.

**tiled.log(text** [string)][void] Outputs the given text in the Console window as regular text.

**tiled.warn(text** [string, activated][function)][void] Outputs the given text in the Console window as warning message and creates an issue in the Issues window.

When the issue is activated (with double-click or Enter key) the given callback function is invoked.

**tiled.error(text** [string, activated][function)][void] Outputs the given text in the Console window as error message and creates an issue in the Issues window.

When the issue is activated (with double-click or Enter key) the given callback function is invoked.

**tiled.registerAction(id** [string, callback][function)][*Action*] Registers a new action with the given `id` and `callback` (which is called when the action is triggered). The returned action object can be used to set (and update) various properties of the action.

Example:

```
var action = tiled.registerAction("CustomAction", function(action) {
    tiled.log(action.text + " was " + (action.checked ? "checked" : "unchecked"))
})

action.text = "My Custom Action"
action.checkable = true
action.shortcut = "Ctrl+K"
```

The shortcut will currently only work when the action is added to a menu using *tiled.extendMenu()*.

**tiled.registerMapFormat(shortName** [string, mapFormat][object)][void] Registers a new map format that can then be used to open and/or save maps in that format.

If a map format is already registered with the same `shortName`, the existing format is replaced. The short name can also be used to specify the format when using `--export-map` on the command-line, in case the file extension is ambiguous or a different one should be used.

The `mapFormat` object is expected to have the following properties:

| | |
|---|---|
| **name** : string | Name of the format as shown in the file dialog. |
| **extension** : string | The file extension used by the format. |
| **read** : function(fileName : string) : *TileMap* | A function that reads a map from the given file. Can use *TextFile* or *BinaryFile* to read the file. |
| **write** : function(map : *TileMap*, fileName : string) : string \| undefined | A function that writes a map to the given file. Can use *TextFile* or *BinaryFile* to write the file. When a non-empty string is returned, it is shown as error message. |
| **outputFiles** : function(map : *TileMap*, fileName : string) : [string] | A function that returns the list of files that will be written when exporting the given map (optional). |

Example that produces a simple JSON representation of a map:

```
var customMapFormat = {
    name: "Custom map format",
    extension: "custom",

    write: function(map, fileName) {
        var m = {
            width: map.width,
            height: map.height,
            layers: []
        };

        for (var i = 0; i < map.layerCount; ++i) {
            var layer = map.layerAt(i);
            if (layer.isTileLayer) {
                var rows = [];
                for (y = 0; y < layer.height; ++y) {
                    var row = [];
                    for (x = 0; x < layer.width; ++x)
                        row.push(layer.cellAt(x, y).tileId);
                    rows.push(row);
                }
                m.layers.push(rows);
            }
        }

        var file = new TextFile(fileName, TextFile.WriteOnly);
        file.write(JSON.stringify(m));
        file.commit();
    },
}

tiled.registerMapFormat("custom", customMapFormat)
```

**tiled.registerTilesetFormat(shortName** [string, tilesetFormat][object)][void] Like *registerMapFormat*, but registers a custom tileset format instead.

The `tilesetFormat` object is expected to have the following properties:

| | |
|---|---|
| **name** : string | Name of the format as shown in the file dialog. |
| **extension** : string | The file extension used by the format. |
| **read** : function(fileName : string) : *Tileset* | A function that reads a tileset from the given file. Can use *TextFile* or *BinaryFile* to read the file. |
| **write** : function(tileset : *Tileset*, fileName : string) : string \| undefined | A function that writes a tileset to the given file. Can use *TextFile* or *BinaryFile* to write the file. When a non-empty string is returned, it is shown as error message. |

**tiled.registerTool(shortName** [string, tool][object)][object] Registers a custom tool that will become available on the Tools tool bar of the Map Editor.

If a tool is already registered with the same `shortName` the existing tool is replaced.

The `tool` object has the following properties:

| | |
|---|---|
| **name** : string | Name of the tool as shown on the tool bar. |
| **map** : *TileMap* | Currently active tile map. |
| **selectedTile** : *Tile* | The last clicked tile for the active map. See also the `currentBrush` property of *MapEditor*. |
| **preview** : *TileMap* | Get or set the preview for tile layer edits. |
| **tilePosition** : *point* | Mouse cursor position in tile coordinates. |
| **statusInfo** : string | Text shown in the status bar while the tool is active. |
| **enabled** : bool | Whether this tool is enabled. |
| **activated** : function() : void | Called when the tool was activated. |
| **deactivated** : function() : void | Called when the tool was deactivated. |
| **keyPressed** : function(key, modifiers) : void | Called when a key was pressed while the tool was active. |
| **mouseEntered** : function() : void | Called when the mouse entered the map view. |
| **mouseLeft** : function() : void | Called when the mouse left the map view. |
| **mouseMoved** : function(x, y, modifiers) : void | Called when the mouse position in the map scene changed. |
| **mousePressed** : function(button, x, y, modifiers) : void | Called when a mouse button was pressed. |
| **mouseReleased** : function(button, x, y, modifers) : void | Called when a mouse button was released. |
| **mouseDoubleClicked** : function(button, x, y, modifiers) : void | Called when a mouse button was double-clicked. |
| **modifiersChanged** : function(modifiers) : void | Called when the active modifier keys changed. |
| **languageChanged** : function() : void | Called when the language was changed. |
| **mapChanged** : function(oldMap : *TileMap*, newMap : *TileMap*) : void | Called when the active map was changed. |
| **tilePositionChanged** : function() : void | Called when the hovered tile position changed. |
| **updateStatusInfo** : function() : void | Called when the hovered tile position changed. Used to override the default updating of the status bar text. |
| **updateEnabledState** : function() : void | Called when the map or the current layer changed. |

Here is an example tool that places a rectangle each time the mouse has moved by 32 pixels:

```
var tool = tiled.registerTool("PlaceRectangles", {
    name: "Place Rectangles",

    mouseMoved: function(x, y, modifiers) {
        if (!this.pressed)
            return

        var dx = Math.abs(this.x - x)
        var dy = Math.abs(this.y - y)

        this.distance += Math.sqrt(dx*dx + dy*dy)
        this.x = x
        this.y = y
```

(continues on next page)

(continued from previous page)

```
        if (this.distance > 32) {
            var objectLayer = this.map.currentLayer

            if (objectLayer && objectLayer.isObjectLayer) {
                var object = new MapObject(++this.counter)
                object.x = Math.min(this.lastX, x)
                object.y = Math.min(this.lastY, y)
                object.width = Math.abs(this.lastX - x)
                object.height = Math.abs(this.lastY - y)
                objectLayer.addObject(object)
            }

            this.distance = 0
            this.lastX = x
            this.lastY = y
        }
    },

    mousePressed: function(button, x, y, modifiers) {
        this.pressed = true
        this.x = x
        this.y = y
        this.distance = 0
        this.counter = 0
        this.lastX = x
        this.lastY = y
    },

    mouseReleased: function(button, x, y, modifiers) {
        this.pressed = false
    },
})
```

**tiled.extendMenu(id** [string, items][array | object)][void] Extends the menu with the given ID. Supports both a list of items or a single item. Available menu IDs can be obtained using the `tiled.menus` property.

A menu item is defined by an object with the following properties:

| | |
|---|---|
| **action** : string | ID of a registered action that the menu item will represent. |
| **before** : string | ID of the action before which this menu item should be added (optional). |
| **separator** : bool | Set to `true` if this item is a menu separator (optional). |

If a menu item does not include a `before` property, the value is inherited from the previous item. When this property is not set at all, the items are appended to the end of the menu.

Example that adds a custom action to the "Edit" menu, before the "Select All" action and separated by a separator:

```
tiled.extendMenu("Edit", [
    { action: "CustomAction", before: "SelectAll" },
    { separator: true }
]);
```

The "CustomAction" will need to have been registered before using *tiled.registerAction()*.

**tiled.tilesetFormat(shortName** [string)][*TilesetFormat*] Returns the tileset format object with the given name, or *undefined* if no object was found. See the *tilesetFormats* property for more info.

**tiled.tilesetFormatForFile(fileName** [string)][*TilesetFormat*] Returns the tileset format object that can read the given file, or *undefined* if no object was found.

**tiled.mapFormat(shortName** [string)][*MapFormat*] Returns the map format object with the given name, or *undefined* if no object was found. See the *mapFormats* property for more info.

**tiled.mapFormatForFile(fileName** [string)][*MapFormat*] Returns the map format object that can read the given file, or *undefined* if no object was found.

**tiled.filePath(path** [url)][*FilePath*] Creates a *FilePath* object with the given URL.

**tiled.objectRef(id** [int)][*ObjectRef*] Creates an *ObjectRef* object with the given ID.

#### Signals

**tiled.assetCreated(asset** [*Asset*)] A new asset has been created.

**tiled.assetOpened(asset** [*Asset*)] An asset has been opened.

**tiled.assetAboutToBeSaved(asset** [*Asset*)] An asset is about to be saved. Can be used to make last-minute changes.

**tiled.assetSaved(asset** [*Asset*)] An asset has been saved.

**tiled.assetAboutToBeClosed(asset** [*Asset*)] An asset is about to be closed.

**tiled.activeAssetChanged(asset** [*Asset*)] The currently active asset has changed.

### 22.2.2 Action

An action that was registered with *tiled.registerAction()*. This class is used to change the properties of the action. It can be added to a menu using *tiled.extendMenu()*.

#### Properties

| | |
|---|---|
| **checkable** : bool | Whether the action can be checked. |
| **checked** : bool | Whether the action is checked. |
| **enabled** : bool | Whether the action is enabled. |
| **icon** : string | File name of an icon. |
| **iconVisibleInMenu** : bool | Whether the action should show an icon in a menu. |
| **id** : string *[read-only]* | The ID this action was registered with. |
| **shortcut** : QKeySequence | The shortcut (can be assigned a string like "Ctrl+K"). |
| **text** : string | The text used when the action is part of a menu. |
| **visible** : bool | Whether the action is visible. |

#### Functions

**Action.trigger()** [void] Triggers the action.

**Action.toggle()** [void] Changes the checked state to its opposite state.

### 22.2.3 Asset

Inherits *Object*.

Represents any top-level data type that can be saved to a file. Currently either a *TileMap* or a *Tileset*.

For assets that are loaded in the editor, all modifications and modifications to their contained parts create undo commands. This includes both modifying functions that are called as well as simply assigning to a writable property.

#### Properties

| | |
|---|---|
| **fileName** : string *[read-only]* | File name of the asset. |
| **modified** : bool *[read-only]* | Whether the asset was modified after it was saved or loaded. |
| **isTileMap** : bool *[read-only]* | Whether the asset is a *TileMap*. |
| **isTileset** : bool *[read-only]* | Whether the asset is a *Tileset*. |

#### Functions

**Asset.macro(text** [string, callback][function)][value] Creates a single undo command that wraps all changes applied to this asset by the given callback. Recommended to avoid spamming the undo stack with small steps that the user does not care about.

Example function that changes visibility of multiple layers in one step:

```
tileMap.macro((visible ? "Show" : "Hide") + " Selected Layers", function() {
    tileMap.selectedLayers.forEach(function(layer) {
        layer.visible = visible
    })
})
```

The returned value is whatever the callback function returned.

**Asset.undo()** [void] Undoes the last applied change. Note that the undo system is only enabled for assets loaded in the editor!

**Asset.redo()** [void] Redoes the last change that was undone. Note that the undo system is only enabled for assets loaded in the editor!

### 22.2.4 FileFormat

Common functionality for file format readers and writers. (Since 1.4)

#### Properties

| | |
|---|---|
| **canRead** : bool *[read-only]* | Whether this format supports reading files. |
| **canWrite** : bool *[read-only]* | Whether this format supports writing files. |

#### Functions

**FileFormat.supportsFile(fileName** [string)][bool] Returns whether the file is readable by this format.

### 22.2.5 GroupLayer

Inherits *Layer*.

#### Properties

| | |
|---|---|
| **layerCount** : int *[read-only]* | Number of child layers the group layer has. |

#### Functions

**new GroupLayer([name** [string])] Constructs a new group layer.

**GroupLayer.layerAt(index** [int)][*Layer*] Returns a reference to the child layer at the given index.

**GroupLayer.removeLayerAt(index** [int)][void] Removes the child layer at the given index. When a reference to the layer still exists and this group layer isn't already standalone, that reference becomes a standalone copy of the layer.

**GroupLayer.removeLayer(layer** [*Layer*)][void] Removes the given layer from the group. If this group wasn't standalone, the reference to the layer becomes a standalone copy.

**GroupLayer.insertLayerAt(index** [int, layer][*Layer*)][void] Inserts the layer at the given index. The layer can't already be part of a map.

**GroupLayer.addLayer(layer** [*Layer*)][void] Adds the layer to the group, above all existing layers. The layer can't already be part of a map.

### 22.2.6 ImageLayer

Inherits *Layer*.

#### Properties

| | |
|---|---|
| **transparentColor** : color | Color used as transparent color when rendering the image. |
| **imageSource** : url | Reference to the image rendered by this layer. |

### 22.2.7 Layer

Inherits *Object*.

### Properties

| | |
|---|---|
| **name** : string | Name of the layer. |
| **opacity** : number | Opacity of the layer, from 0 (fully transparent) to 1 (fully opaque). |
| **visible** : bool | Whether the layer is visible (affects child layer visibility for group layers). |
| **locked** : bool | Whether the layer is locked (affects whether child layers are locked for group layers). |
| **offset** : *point* | Offset in pixels that is applied when this layer is rendered. |
| **map** : *TileMap* | Map that this layer is part of (or `null` in case of a standalone layer). |
| **selected** : bool | Whether the layer is selected. |
| **isTileLayer** : bool *[read-only]* | Whether this layer is a *TileLayer*. |
| **isObjectLayer** : bool *[read-only]* | Whether this layer is an *ObjectGroup*. |
| **isGroupLayer** : bool *[read-only]* | Whether this layer is a *GroupLayer*. |
| **isImageLayer** : bool *[read-only]* | Whether this layer is an *ImageLayer*. |

## 22.2.8 MapObject

Inherits *Object*.

### Properties

| | |
|---|---|
| **id** : int *[read-only]* | Unique (map-wide) ID of the object. |
| **shape** : int | *Shape* of the object. |
| **name** : string | Name of the object. |
| **type** : string | Type of the object. |
| **x** : number | X coordinate of the object in pixels. |
| **y** : number | Y coordinate of the object in pixels. |
| **pos** : *point* | Position of the object in pixels. |
| **width** : number | Width of the object in pixels. |
| **height** : number | Height of the object in pixels. |
| **size** : *size* | Size of the object in pixels. |
| **rotation** : number | Rotation of the object in degrees clockwise. |
| **visible** : bool | Whether the object is visible. |
| **polygon** : *Polygon* | Polygon of the object. |
| **text** : string | The text of a text object. |
| **font** : *Font* | The font of a text object. |
| **textAlignment** : *Alignment* | The alignment of a text object. |
| **wordWrap** : bool | Whether the text of a text object wraps based on the width of the object. |
| **textColor** : color | Color of a text object. |
| **tile** : *Tile* | Tile of the object. |
| **tileFlippedHorizontally** : bool | Whether the tile is flipped horizontally. |
| **tileFlippedVertically** : bool | Whether the tile is flipped vertically. |
| **selected** : bool | Whether the object is selected. |
| **layer** : *ObjectGroup [read-only]* | Layer this object is part of (or `null` in case of a standalone object). |
| **map** : *TileMap [read-only]* | Map this object is part of (or `null` in case of a standalone object). |

| MapObject.Shape |
| --- |
| MapObject.Rectangle |
| MapObject.Polygon |
| MapObject.Polyline |
| MapObject.Ellipse |
| MapObject.Text |
| MapObject.Point |

#### Functions

**new MapObject([name** [string])] Constructs a new map object, which can be added to an *ObjectGroup*.

### 22.2.9 MapEditor

#### Properties

| | |
| --- | --- |
| **currentBrush** : *TileMap* | Get or set the currently used tile brush. |
| **currentMapView** : *MapView* *[read-only]* | Access the current map view. |
| **tilesetsView** : *TilesetsView* *[read-only]* | Access the Tilesets view. |

### 22.2.10 MapFormat

This is an object that can read or write map files. (Since 1.4)

Inherits *FileFormat*.

#### Functions

**MapFormat.read(fileName** [string)][*TileMap*] Read the given file as a map. This function will throw an error if reading is not supported.

**MapFormat.write(map** [*TileMap*, fileName][string)][string] Write the given map to a file. This function will throw an error if writing is not supported. If there is an error writing the file, it will return a description of the error; otherwise, it will return "".

### 22.2.11 MapView

The view displaying the map.

#### Properties

| | |
| --- | --- |
| **scale** : number | Get or set the scale of the view. |

### Functions

**MapView.centerOn(x** [number, y][number)][void] Centers the view at the given location in screen coordinates.

## 22.2.12 Object

The base of most data types in Tiled. Provides the ability to associate custom properties with the data.

### Properties

| | |
|---|---|
| **asset** : *Asset [read-only]* | The asset this object is part of, or `null`. |
| **readOnly** : bool *[read-only]* | Whether the object is read-only. |

### Functions

**Object.property(name** [string)][variant] Returns the value of the custom property with the given name, or `undefined` if no such property is set on the object.

*Note:* Currently it is not possible to inspect the value of `file` properties.

**Object.setProperty(name** [string, value][variant)][void] Sets the value of the custom property with the given name. Supported types are `bool`, `number` and `string`. When setting a `number`, the property type will be set to either `int` or `float`, depending on whether it is a whole number.

*Note:* Support for `color` and `file` properties is currently missing.

**Object.properties()** [object] Returns all custom properties set on this object. Modifications to the properties will not affect the original object.

**Object.setProperties(properties** [object)][void] Replaces all currently set custom properties with a new set of properties.

**Object.removeProperty(name** [string)][void] Removes the custom property with the given name.

## 22.2.13 ObjectGroup

Inherits *Layer*.

The "ObjectGroup" is a type of layer that can contain objects. It will henceforth be referred to as a layer.

### Properties

| | |
|---|---|
| **objects** : [*MapObject*] *[read-only]* | Array of all objects on this layer. |
| **objectCount** : int *[read-only]* | Number of objects on this layer. |
| **color** : color | Color of shape and point objects on this layer (when not set by object type). |

### Functions

**new ObjectGroup([name** [string])] Constructs a new object layer, which can be added to a *TileMap*.

**ObjectGroup.objectAt(index** [int)][*MapObject*] Returns a reference to the object at the given index. When the object is removed, the reference turns into a standalone copy of the object.

**ObjectGroup.removeObjectAt(index** [int)][void] Removes the object at the given index.

**ObjectGroup.removeObject(object** [*MapObject*)][void] Removes the given object from this layer. The object reference turns into a standalone copy of the object.

**ObjectGroup.insertObjectAt(index** [int, object][*MapObject*)][void] Inserts the object at the given index. The object can't already be part of a layer.

**ObjectGroup.addObject(object** [*MapObject*)][void] Adds the given object to the layer. The object can't already be part of a layer.

## 22.2.14 SelectedArea

### Properties

| | |
|---|---|
| **boundingRect** : *rect [read-only]* | Bounding rectangle of the selected area. |

### Functions

**SelectedArea.get()** [*region*] Returns the selected region.

**SelectedArea.set(rect** [*rect*)][void] Sets the selected area to the given rectangle.

**SelectedArea.set(region** [*region*)][void] Sets the selected area to the given region.

**SelectedArea.add(rect** [*rect*)][void] Adds the given rectangle to the selected area.

**SelectedArea.add(region** [*region*)][void] Adds the given region to the selected area.

**SelectedArea.subtract(rect** [*rect*)][void] Subtracts the given rectangle from the selected area.

**SelectedArea.subtract(region** [*region*)][void] Subtracts the given region from the selected area.

**SelectedArea.intersect(rect** [*rect*)][void] Sets the selected area to the intersection of the current selected area and the given rectangle.

**SelectedArea.intersect(region** [*region*)][void] Sets the selected area to the intersection of the current selected area and the given region.

## 22.2.15 Terrain

Inherits *Object*.

### Properties

| | |
|---|---|
| **id** : int *[read-only]* | ID of this terrain. |
| **name** : string | Name of the terrain. |
| **imageTile** : *Tile* | The tile representing the terrain (needs to be from the same tileset). |
| **tileset** : *Tileset [read-only]* | The tileset of the terrain. |

### 22.2.16 Tile

Inherits *Object*.

#### Properties

| | |
|---|---|
| **id** : int *[read-only]* | ID of this tile within its tileset. |
| **width** : int *[read-only]* | Width of the tile in pixels. |
| **height** : int *[read-only]* | Height of the tile in pixels. |
| **size** : *size* *[read-only]* | Size of the tile in pixels. |
| **type** : string | Type of the tile. |
| **imageFileName** : string | File name of the tile image (when the tile is part of an image collection tileset). |
| **terrain** : *Terrains* | An object specifying the terrain at each corner of the tile. |
| **probability** : number | Probability that the tile gets chosen relative to other tiles. |
| **objectGroup** : *ObjectGroup* | The *ObjectGroup* associated with the tile in case collision shapes were defined. Returns `null` if no collision shapes were defined for this tile. |
| **frames** : *[frame]* | This tile's animation as an array of frames. |
| **animated** : bool *[read-only]* | Indicates whether this tile is animated. |
| **tileset** : *Tileset* *[read-only]* | The tileset of the tile. |

| Tile.Flags |
|---|
| Tile.FlippedHorizontally |
| Tile.FlippedVertically |
| Tile.FlippedAntiDiagonally |
| Tile.RotatedHexagonal120 |

| Tile.Corner |
|---|
| Tile.TopLeft |
| Tile.TopRight |
| Tile.BottomLeft |
| Tile.BottomRight |

#### Functions

**Tile.terrainAtCorner(corner** [*Corner*)][*Terrain*] Returns the terrain used at the given corner.

**Tile.setTerrainAtCorner(corner** [*Corner*, *Terrain*)][void] Sets the terrain used at the given corner.

### 22.2.17 TileCollisionEditor

#### Properties

| | |
|---|---|
| **selectedObjects** : [*MapObject*] | Selected objects. |
| **view** : [*MapView*] | The map view used by the Collision Editor. |

### Functions

**TileCollisionEditor.focusObject(object** [*MapObject*)][void] Focuses the given object in the collision editor view and makes sure its visible in its objects list. Does not automatically select the object.

## 22.2.18 TileLayer

Inherits *Layer*.

Note that while tile layers have a size, the size is generally ignored on infinite maps. Even for fixed size maps, nothing in the scripting API stops you from changing the layer outside of its boundaries and changing the size of the layer has no effect on its contents. If you want to change the size while affecting the contents, use the `resize` function.

### Properties

| | |
|---|---|
| **width** : int | Width of the layer in tiles (only relevant for non-infinite maps). |
| **height** : int | Height of the layer in tiles (only relevant for non-infinite maps). |
| **size** : *size* | Size of the layer in tiles (has `width` and `height` members) (only relevant for non-infinite maps). |

### Functions

**new TileLayer([name** [string])] Constructs a new tile layer, which can be added to a *TileMap*.

**TileLayer.region()** [*region*] Returns the region of the layer that is covered with tiles.

**TileLayer.resize(size** [*size*, offset][*point*)][void] Resizes the layer, erasing the part of the contents that falls outside of the layer's new size. The offset parameter can be used to shift the contents by a certain distance in tiles before applying the resize.

**TileLayer.cellAt(x** [int, y][int)][*cell*] Returns the value of the cell at the given position. Can be used to query the flags and the tile ID, but does not currently allow getting a tile reference.

**TileLayer.flagsAt(x** [int, y][int)][int] Returns the *flags* used for the tile at the given position.

**TileLayer.tileAt(x** [int, y][int)][*Tile*] Returns the tile used at the given position, or `null` for empty spaces.

**TileLayer.edit()** [*TileLayerEdit*] Returns an object that enables making modifications to the tile layer.

## 22.2.19 TileLayerEdit

This object enables modifying the tiles on a tile layer. Tile layers can't be modified directly for reasons of efficiency. The *apply()* function needs to be called when you're done making changes.

An instance of this object is created by calling *TileLayer.edit()*.

### Properties

| | |
|---|---|
| **target** : *TileLayer [read-only]* | The target layer of this edit object. |
| **mergeable** : bool | Whether applied edits are mergeable with previous edits. Starts out as `false` and is automatically set to `true` by *apply()*. |

### Functions

**TileLayerEdit.setTile(x** [int, y][int, tile][*Tile* [, flags][int = 0])][void] Sets the tile at the given location, optionally specifying *tile flags*.

**TileLayerEdit.apply()** [void] Applies all changes made through this object. This object can be reused to make further changes.

## 22.2.20 TileMap

Inherits *Asset*.

### Properties

| | |
|---|---|
| **width** : int | Width of the map in tiles (only relevant for non-infinite maps). |
| **height** : int | Height of the map in tiles (only relevant for non-infinite maps). |
| **size** : *size [read-only]* | Size of the map in tiles (only relevant for non-infinite maps). |
| **tileWidth** : int | Tile width (used by tile layers). |
| **tileHeight**: int | Tile height (used by tile layers). |
| **infinite** : bool | Whether this map is infinite. |
| **hexSideLength** : int | Length of the side of a hexagonal tile (used by tile layers on hexagonal maps). |
| **staggerAxis** : *StaggerAxis* | For staggered and hexagonal maps, determines which axis (X or Y) is staggered. |
| **orientation** : *Orientation* | General map orientation |
| **renderOrder** : *RenderOrder* | Tile rendering order (only implemented for orthogonal maps) |
| **staggerIndex** : *StaggerIndex* | For staggered and hexagonal maps, determines whether the even or odd indexes along the staggered axis are shifted. |
| **backgroundColor** : color | Background color of the map. |
| **layerDataFormat** : *LayerDataFormat* | The format in which the layer data is stored, taken into account by TMX, JSON and Lua map formats. |
| **layerCount** : int *[read-only]* | Number of top-level layers the map has. |
| **tilesets** : [*Tileset*] | The list of tilesets referenced by this map. To determine which tilesets are actually used, call *usedTilesets()*. |
| **selectedArea** : *SelectionArea* | The selected area of tiles. |
| **currentLayer** : *Layer* | The current layer. |
| **selectedLayers** : [*Layer*] | Selected layers. |
| **selectedObjects** : [*MapObject*] | Selected objects. |

| TileMap.Orientation |
|---|
| TileMap.Unknown |
| TileMap.Orthogonal |
| TileMap.Isometric |
| TileMap.Staggered |
| TileMap.Hexagonal |

| TileMap.LayerDataFormat |
| --- |
| TileMap.XML |
| TileMap.Base64 |
| TileMap.Base64Gzip |
| TileMap.Base64Zlib |
| TileMap.Base64Zstandard |
| TileMap.CSV |

| TileMap.RenderOrder |
| --- |
| TileMap.RightDown |
| TileMap.RightUp |
| TileMap.LeftDown |
| TileMap.LeftUp |

| TileMap.StaggerAxis |
| --- |
| TileMap.StaggerX |
| TileMap.StaggerY |

| TileMap.StaggerIndex |
| --- |
| TileMap.StaggerOdd |
| TileMap.StaggerEven |

### Functions

**new TileMap()** Constructs a new map.

**TileMap.autoMap([rulesFile** [string])][void] Applies *Automapping* using the given rules file, or using the default rules file is none is given.

*This operation can only be applied to maps loaded from a file.*

**TileMap.autoMap(region** [*region* | *rect* [, rulesFile][string])][void] Applies *Automapping* in the given region using the given rules file, or using the default rules file is none is given.

*This operation can only be applied to maps loaded from a file.*

**TileMap.setSize(width** [int, height][int)][void] Sets the size of the map in tiles. This does not affect the contents of the map.

See also *resize*.

**TileMap.setTileSize(width** [int, height][int)][void] Sets the tile size of the map in pixels. This affects the rendering of all tile layers.

**TileMap.layerAt(index** [int)][*Layer*] Returns a reference to the top-level layer at the given index. When the layer gets removed from the map, the reference changes to a standalone copy of the layer.

**TileMap.removeLayerAt(index** [int)][void] Removes the top-level layer at the given index. When a reference to the layer still exists, that reference becomes a standalone copy of the layer.

**TileMap.removeLayer(layer** [*Layer*)][void] Removes the given layer from the map. The reference to the layer becomes a standalone copy.

**TileMap.insertLayerAt(index** [int, layer][*Layer*)][void] Inserts the layer at the given index. The layer can't already be part of a map.

**TileMap.addLayer(layer** [*Layer*)][void] Adds the layer to the map, above all existing layers. The layer can't already be part of a map.

**TileMap.addTileset(tileset** [*Tileset*)][bool] Adds the given tileset to the list of tilesets referenced by this map. Returns `true` if the tileset was added, or `false` if the tileset was already referenced by this map.

**TileMap.replaceTileset(oldTileset** [*Tileset*, newTileset][*Tileset*)][bool] Replaces all occurrences of `oldTileset` with `newTileset`. Returns `true` on success, or `false` when either the old tileset was not referenced by the map, or when the new tileset was already referenced by the map.

**TileMap.removeTileset(tileset** [*Tileset*)][bool] Removes the given tileset from the list of tilesets referenced by this map. Returns `true` on success, or `false` when the given tileset was not referenced by this map or when the tileset was still in use by a tile layer or tile object.

**TileMap.usedTilesets()** [[*Tileset*]] Returns the list of tilesets actually used by this map. This is generally a subset of the tilesets referenced by the map (the `TileMap.tilesets` property).

**TileMap.merge(map** [*TileMap* [, canJoin][bool = false])][void] Merges the tile layers in the given map with this one. If only a single tile layer exists in the given map, it will be merged with the `currentLayer`.

If `canJoin` is `true`, the operation joins with the previous one on the undo stack when possible. Useful for reducing the amount of undo commands.

*This operation can currently only be applied to maps loaded from a file.*

**TileMap.resize(size** [*size* [, offset][*point* [, removeObjects][bool = false])][void] Resizes the map to the given size, optionally applying an offset (in tiles).

*This operation can currently only be applied to maps loaded from a file.*

See also *setSize*.

**TileMap.screenToTile(x** [number, y][number)][*point*] Converts the given position from screen to tile coordinates.

**TileMap.screenToTile(position** [*point*)][*point*] Converts the given position from screen to tile coordinates.

**TileMap.tileToScreen(x** [number, y][number)][*point*] Converts the given position from tile to screen coordinates.

**TileMap.tileToScreen(position** [*point*)][*point*] Converts the given position from tile to screen coordinates.

**TileMap.screenToPixel(x** [number, y][number)][*point*] Converts the given position from screen to pixel coordinates.

**TileMap.screenToPixel(position** [*point*)][*point*] Converts the given position from screen to pixel coordinates.

**TileMap.pixelToScreen(x** [number, y][number)][*point*] Converts the given position from pixel to screen coordinates.

**TileMap.pixelToScreen(position** [*point*)][*point*] Converts the given position from pixel to screen coordinates.

**TileMap.pixelToTile(x** [number, y][number)][*point*] Converts the given position from pixel to tile coordinates.

**TileMap.pixelToTile(position** [*point*)][*point*] Converts the given position from pixel to tile coordinates.

**TileMap.tileToPixel(x** [number, y][number)][*point*] Converts the given position from tile to pixel coordinates.

**TileMap.tileToPixel(position** [*point*)][*point*] Converts the given position from tile to pixel coordinates.

### 22.2.21 Tileset

Inherits *Asset*.

### Properties

| | |
|---|---|
| **name** : string | Name of the tileset. |
| **image** : string | The file name of the image used by this tileset. Empty in case of image collection tilesets. |
| **tiles**: [*Tile*] *[read-only]* | Array of all tiles in this tileset. Note that the index of a tile in this array does not always match with its ID. |
| **terrains**: [*Terrain*] *[read-only]* | Array of all terrains in this tileset. |
| **tileCount** : int | The number of tiles in this tileset. |
| **nextTileId** : int | The ID of the next tile that would be added to this tileset. All existing tiles have IDs that are lower than this ID. |
| **tileWidth** : int | Tile width for tiles in this tileset in pixels. |
| **tileHeight** : int | Tile Height for tiles in this tileset in pixels. |
| **tileSize** : *size* | Tile size for tiles in this tileset in pixels. |
| **imageWidth** : int *[read-only]* | Width of the tileset image in pixels. |
| **imageHeight** : int *[read-only]* | Height of the tileset image in pixels. |
| **imageSize** : *size* *[read-only]* | Size of the tileset image in pixels. |
| **tileSpacing** : int *[read-only]* | Spacing between tiles in this tileset in pixels. |
| **margin** : int *[read-only]* | Margin around the tileset in pixels (only used at the top and left sides of the tileset image). |
| **objectAlignment** : *Alignment* | The alignment to use for tile objects (when `Unspecified`, uses `Bottom` alignment on isometric maps and `BottomLeft` alignment for all other maps). |
| **tileOffset** : *point* | Offset in pixels that is applied when tiles from this tileset are rendered. |
| **orientation** : *Orientation* | The orientation of this tileset (used when rendering overlays and in the tile collision editor). |
| **backgroundColor** : color | Background color for this tileset in the *Tilesets* view. |
| **isCollection** : bool *[read-only]* | Whether this tileset is a collection of images (same as checking whether `image` is an empty string). |
| **selectedTiles** : [*Tile*] | Selected tiles (in the tileset editor). |

| Tileset.Alignment |
|---|
| Tileset.Unspecified |
| Tileset.TopLeft |
| Tileset.Top |
| Tileset.TopRight |
| Tileset.Left |
| Tileset.Center |
| Tileset.Right |
| Tileset.BottomLeft |
| Tileset.Bottom |
| Tileset.BottomRight |

| Tileset.Orientation |
|---|
| Tileset.Orthogonal |
| Tileset.Isometric |

#### Functions

**new Tileset([name** [string])] Constructs a new tileset.

**Tileset.tile(id** [int)][*Tile*] Returns a reference to the tile with the given ID. Raises an error if no such tile exists. When the tile gets removed from the tileset, the reference changes to a standalone copy of the tile.

Note that the tiles in a tileset are only guaranteed to have consecutive IDs for tileset-image based tilesets. For image collection tilesets there will be gaps when tiles have been removed from the tileset.

**Tileset.setTileSize(width** [int, height][int)][void] Sets the tile size for this tileset. If an image has been specified as well, the tileset will be (re)loaded. Can't be used on image collection tilesets.

**Tileset.addTile()** [*Tile*] Adds a new tile to this tileset and returns it. Only works for image collection tilesets.

**Tileset.removeTiles(tiles** [[*Tile*])][void] Removes the given tiles from this tileset. Only works for image collection tilesets.

### 22.2.22 TilesetEditor

#### Properties

| | |
|---|---|
| **collisionEditor** : *TileCollisionEditor* | Access the collision editor within the tileset editor. |

### 22.2.23 TilesetFormat

This is an object that can read or write tileset files. (Since 1.4)

Inherits *FileFormat*.

#### Functions

**TilesetFormat.read(fileName** [string)][*Tileset*] Read the given file as a tileset. This function will throw an error if reading is not supported.

**TilesetFormat.write(tileset** [*Tileset*, fileName][string)][string] Write the given tileset to a file. This function will throw an error if writing is not supported. If there is an error writing the file, it will return a description of the error; otherwise, it will return "".

### 22.2.24 TilesetsView

#### Properties

| | |
|---|---|
| **currentTileset** : *Tileset* | Access or change the currently displayed tileset. |
| **selectedTiles** : [*Tile*] | A list of the tiles that are selected in the current tileset. |

### 22.2.25 Basic Types

Some types are provided by the Qt Scripting Engine and others are added based on the needs of the data types above. In the following the most important ones are documented.

### Alignment

| Qt.Alignment | |
|---|---|
| Qt.AlignLeft | 0x0001 |
| Qt.AlignRight | 0x0002 |
| Qt.AlignHCenter | 0x0004 |
| Qt.AlignJustify | 0x0008 |
| Qt.AlignTop | 0x0020 |
| Qt.AlignBottom | 0x0040 |
| Qt.AlignVCenter | 0x0080 |
| Qt.AlignCenter | Qt.AlignVCenter \| Qt.AlignHCenter |

### cell

A cell on a *TileLayer*.

**Properties**

| | |
|---|---|
| **tileId** : int | The local tile ID of the tile, or -1 if the cell is empty. |
| **empty** : bool | Whether the cell is empty. |
| **flippedHorizontally** : bool | Whether the tile is flipped horizontally. |
| **flippedVertically** : bool | Whether the tile is flipped vertically. |
| **flippedAntiDiagonally** : bool | Whether the tile is flipped anti-diagonally. |
| **rotatedHexagonal120** : bool | Whether the tile is rotated by 120 degrees (for hexagonal maps, the anti-diagonal flip is interpreted as a 60-degree rotation). |

### FilePath

Used as the value for custom 'file' properties. Can be created with *tiled.filePath*.

| | |
|---|---|
| **url** : url | The URL of the file. |

### Font

| | |
|---|---|
| **family** : string | The font family. |
| **pixelSize** : int | Font size in pixels. |
| **bold** : bool | Whether the font is bold. |
| **italic** : bool | Whether the font is italic. |
| **underline** : bool | Whether the text is underlined. |
| **strikeOut** : bool | Whether the text is striked through. |
| **kerning** : bool | Whether to use kerning when rendering the text. |

### Frames

An array of frames, which are objects with the following properties:

| | |
|---|---|
| **tileId** : int | The local tile ID used to represent the frame. |
| **duration** : int | Duration of the frame in milliseconds. |

### ObjectRef

The value of a property of type 'object', which refers to a *MapObject* by its ID. Generally only used as a fallback when an object property cannot be resolved to an actual object. Can be created with *tiled.objectRef*.

| | |
|---|---|
| **id** : int | The ID of the referenced object. |

### point

`Qt.point(x, y)` can be used to create a point object.

**Properties**

| | |
|---|---|
| **x** : number | X coordinate of the point. |
| **y** : number | Y coordinate of the point. |

### Polygon

A polygon is not strictly a custom type. It is an array of objects that each have an `x` and `y` property, representing the points of the polygon.

To modify the polygon of a *MapObject*, change or set up the polygon array and then assign it to the object.

### rect

`Qt.rect(x, y, width, height)` can be used to create a rectangle.

**Properties**

| | |
|---|---|
| **x** : int | X coordinate of the rectangle. |
| **y** : int | Y coordinate of the rectangle. |
| **width** : int | Width of the rectangle. |
| **height** : int | Height of the rectangle. |

### region

**Properties**

| | |
|---|---|
| **boundingRect** : *rect [read-only]* | Bounding rectangle of the region. |

### size

`Qt.size(width, height)` can be used to create a size object.

**Properties**

| | |
|---|---|
| **width** : number | Width. |
| **height** : number | Height. |

### Terrains

An object specifying the terrain for each corner of a tile:

| |
|---|
| **topLeft** : *Terrain* |
| **topRight** : *Terrain* |
| **bottomLeft** : *Terrain* |
| **bottomRight** : *Terrain* |

### TextFile

The TextFile object is used to read and write files in text mode.

When using `TextFile.WriteOnly`, you need to call `commit()` when you're done writing otherwise the operation will be aborted without effect.

**Properties**

| | |
|---|---|
| **filePath** : string *[read-only]* | The path of the file. |
| **atEof** : bool *[read-only]* | True if no more data can be read. |
| **codec** : string | The text codec. |

| TextFile.OpenMode | |
|---|---|
| TextFile.ReadOnly | 0x0001 |
| TextFile.WriteOnly | 0x0002 |
| TextFile.ReadWrite | TextFile.ReadOnly \| TextFile.WriteOnly |
| TextFile.Append | |

**Functions**

**new TextFile(fileName** [string [, mode][OpenMode = ReadOnly])] Opens a text file in the given mode.

**TextFile.readLine()** [string] Reads one line of text from the file and returns it. The returned string does not contain the newline characters.

**TextFile.readAll()** [string] Reads all data from the file and returns it.

**TextFile.truncate()** [void] Truncates the file, that is, gives it the size of zero, removing all content.

**TextFile.write(text** [string)][void] Writes a string to the file.

**TextFile.writeLine(text** [string)][void] Writes a string to the file and appends a newline character.

**TextFile.commit()** [void] Commits all written text to disk and closes the file. Should be called when writing to files in WriteOnly mode. Failing to call this function will result in cancelling the operation, unless safe writing to files is disabled.

**TextFile.close()** [void] Closes the file. It is recommended to always call this function as soon as you are finished with the file.

### BinaryFile

The BinaryFile object is used to read and write files in binary mode.

When using `BinaryFile.WriteOnly`, you need to call `commit()` when you're done writing otherwise the operation will be aborted without effect.

### Properties

| | |
|---|---|
| **filePath** : string *[read-only]* | The path of the file. |
| **atEof** : bool *[read-only]* | True if no more data can be read. |
| **size** : number | The size of the file (in bytes). |
| **pos** : number | The position that data is written to or read from. |

| BinaryFile.OpenMode | |
|---|---|
| BinaryFile.ReadOnly | 0x0001 |
| BinaryFile.WriteOnly | 0x0002 |
| BinaryFile.ReadWrite | BinaryFile.ReadOnly \| BinaryFile.WriteOnly |

### Functions

**new BinaryFile(filePath** [string [, mode][OpenMode = ReadOnly])] Opens a binary file in the given mode.

**BinaryFile.resize(size** [int)][void] Sets the file size (in bytes). If size is larger than the file currently is, the new bytes will be set to 0; if size is smaller, the file is truncated.

**BinaryFile.seek(pos** [int)][void] Sets the current position to *pos*.

**BinaryFile.read(size** [int)][ArrayBuffer] Reads at most *size* bytes of data from the file and returns it as an ArrayBuffer.

**BinaryFile.readAll()** [ArrayBuffer] Reads all data from the file and returns it as an ArrayBuffer.

**BinaryFile.write(data** [ArrayBuffer)][void] Writes *data* into the file at the current position.

**BinaryFile.commit()** [void] Commits all written data to disk and closes the file. Should be called when writing to files in WriteOnly mode. Failing to call this function will result in cancelling the operation, unless safe writing to files is disabled.

**BinaryFile.close()** [void] Closes the file. It is recommended to always call this function as soon as you are finished with the file.