Tiled Documentation

Release 1.12.2

Thorbjørn Lindeijer

May 27, 2026

USER MANUAL

Note

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

CHAPTER

ONE

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, staggered hexagonal and oblique 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, *extending Tiled* with JavaScript, the tile stamp memory, *tile animation support* and the *tile collision editor*.

1.2 Getting Started

1.2.1 Setting up a New Project

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

To make all our assets readily accessible from the *Project* view, as well as to be able to quickly switch between multiple projects, it is recommended to first set up a *Tiled project*. This is however an entirely optional step that can be skipped when desired.

Choose *File -> New -> New Project...* to create a new project file. It is recommended to save this file in the root of your project. The directory in which you store the project will be automatically added, so that its files are visible in the Project view.

When necessary, you can add additional folders to the project or replace the one added by default. For example, you could choose to add several top-level folders like “tilesets”, “maps”, “templates”, etc. Right-click in the Project view and choose *Add Folder to Project...* to add the relevant folders.

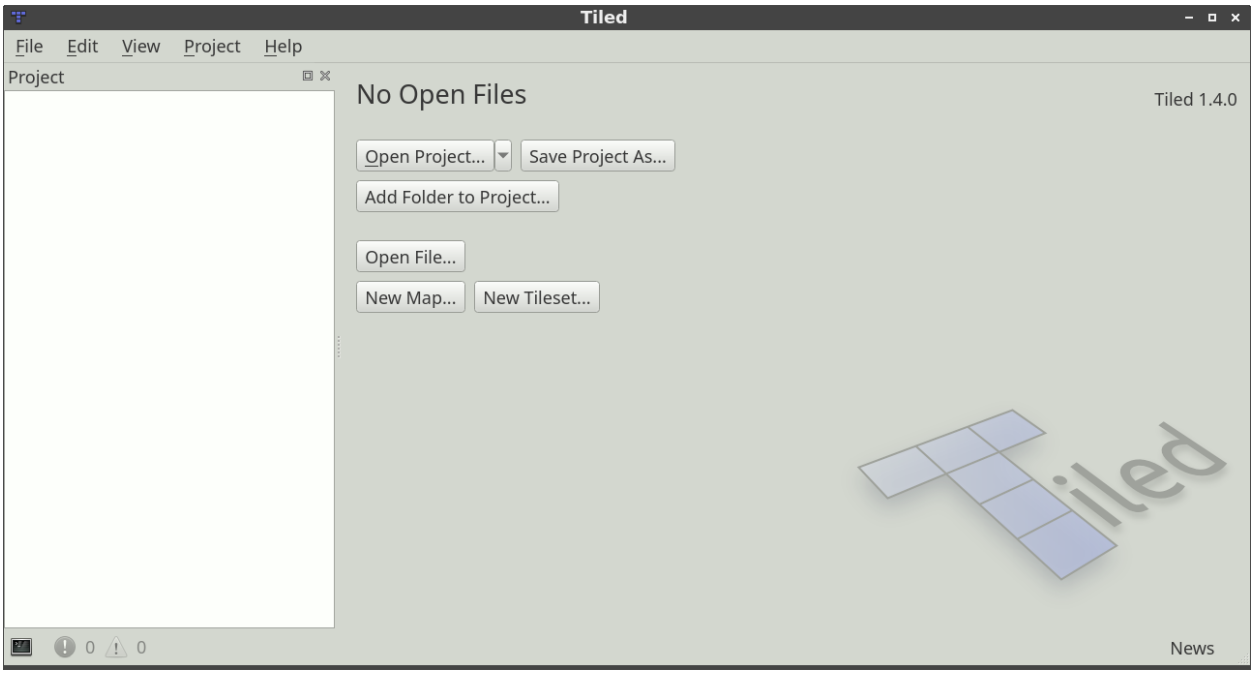


Fig. 1: Tiled Window

Hint

You can press `Ctrl+Shift+P` to open the action search widget, which can provide a faster way to get to actions than looking for them in the menus!

1.2.2 Creating a New Map

To create a new map, choose *File -> New -> New Map...* (`Ctrl+N`). The following dialog will pop up:

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

Note

If you set up a project, make sure to save the map to a folder that you had added to your project. This will make it quickly accessible using *File -> Open File in Project* (`Ctrl+P`).

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).

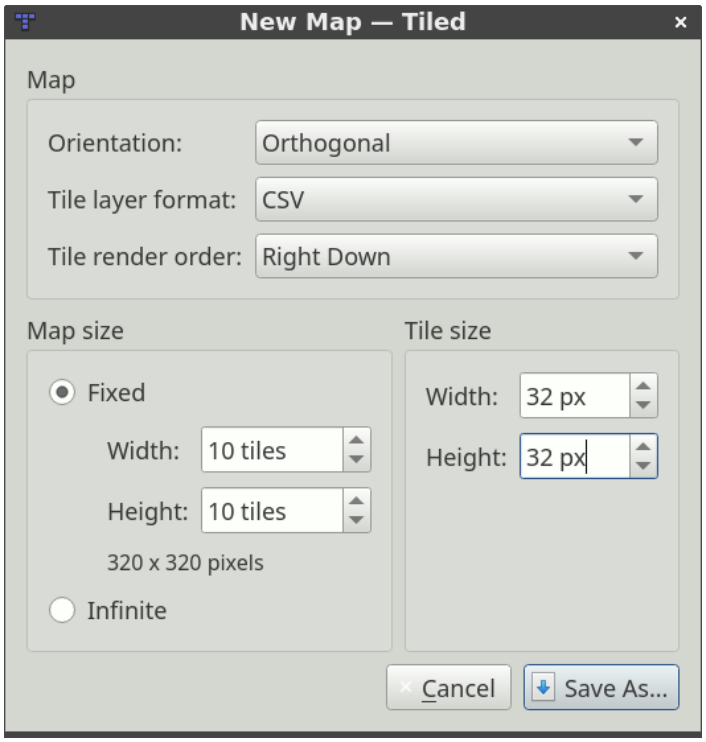


Fig. 2: New Map

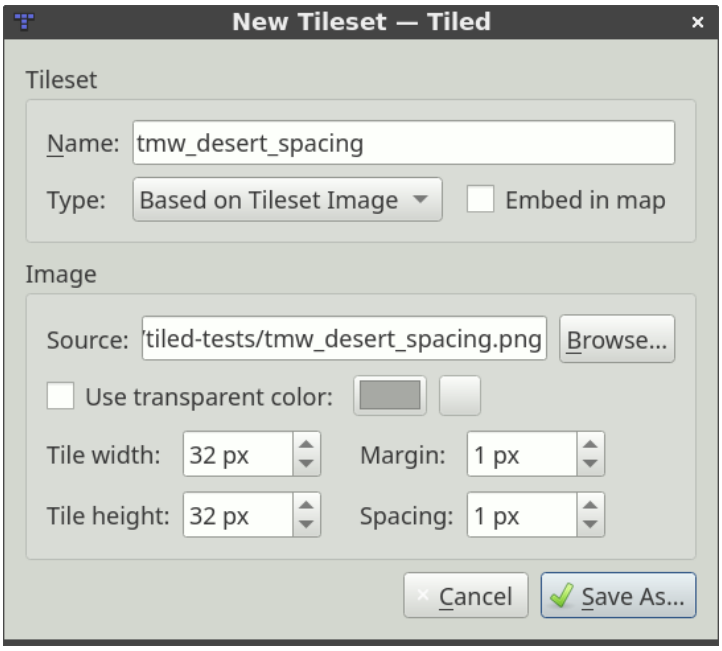


Fig. 3: New Tileset

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:

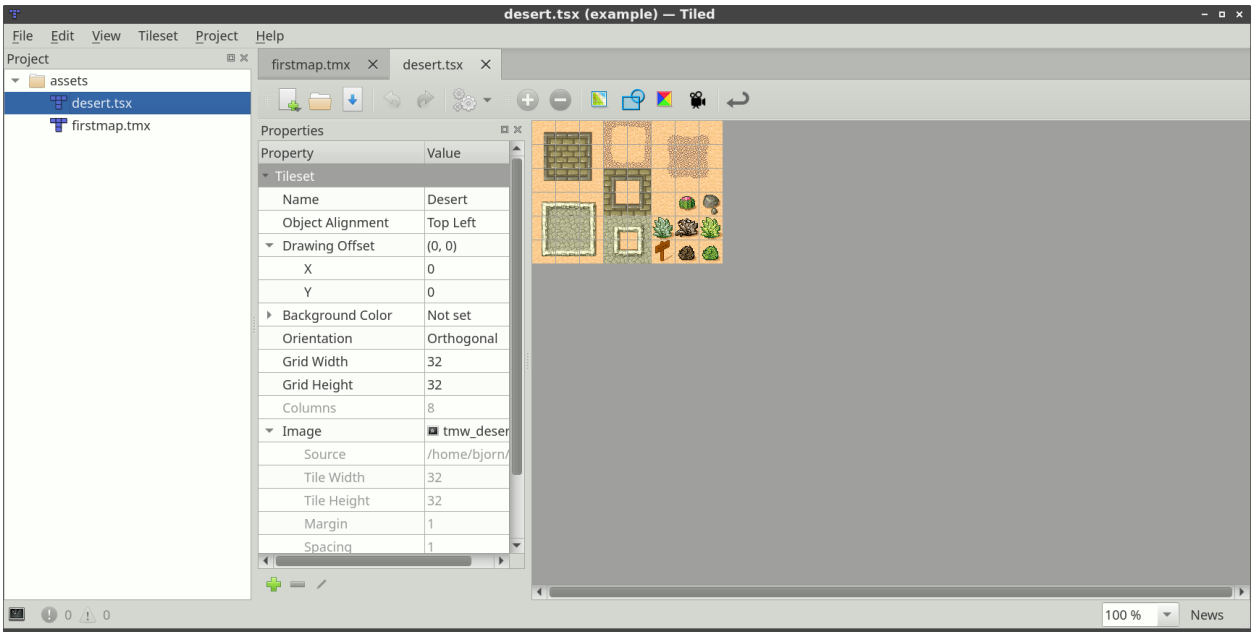


Fig. 4: Tileset Created

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.

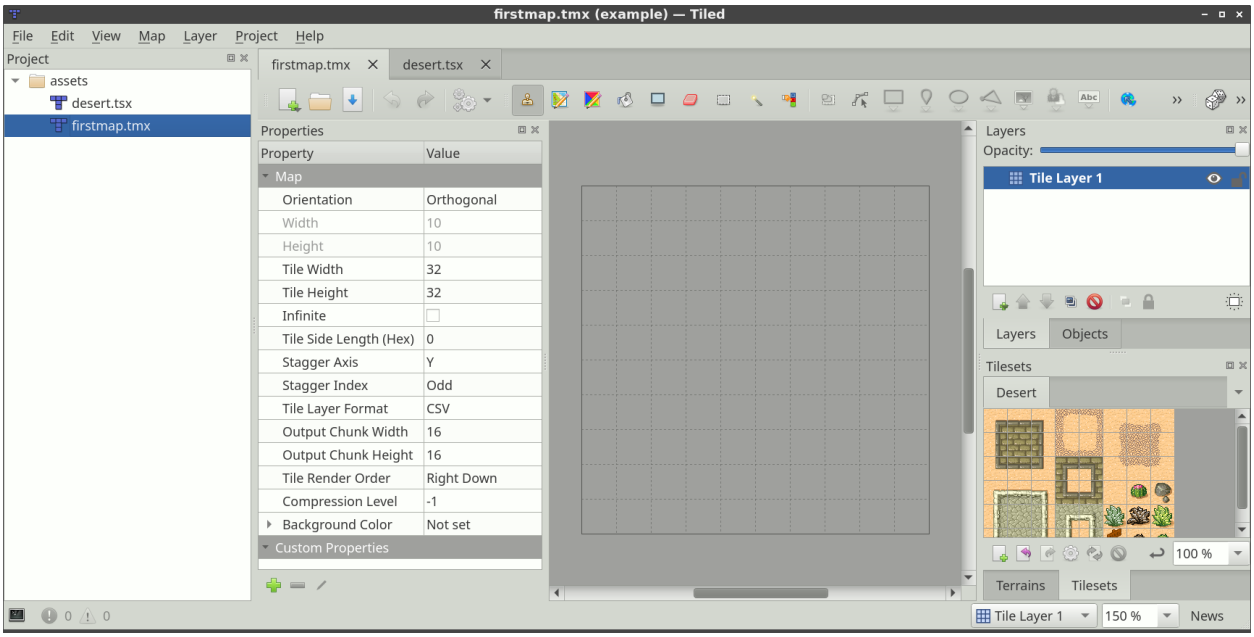


Fig. 5: Tileset Usable on the Map

CHAPTER

TWO

PROJECTS

2.1 What's in a Project

A Tiled project file primarily defines the list of folders containing the assets belonging to that project. In addition, it provides an anchor for the *session file*.

Apart from the list of folders, a project currently has the following properties, which can be changed through the *Project -> Project Properties...* dialog.

Compatibility Version
: The Tiled version to target when saving or exporting files. Can be used to maintain compatibility with earlier versions of Tiled or with *Libraries and Frameworks* that do not yet support certain backwards-incompatible changes.

Extensions Directory
: A project-specific directory where you can put *Tiled extensions*. It defaults to simply `extensions`, so when you have a directory called "extensions" alongside your project file it will be picked up automatically.

  The directory is loaded in addition to the global extensions.

Automapping Rules File
: Refers to an *Automapping* rules file, or a single rule map, that should be used for all maps while this project is loaded. It is ignored for maps that have a `rules.txt` file saved alongside them.

Any types defined in the *Custom Types Editor* are also saved in the project.

2.2 Sessions

Each project file gets an associated *.tiled-session* file, stored alongside it. The session file should generally not be shared with others and stores your last opened files, part of their last editor state, last used parameters in dialogs, etc.

When switching projects Tiled automatically switches to the associated session, so you can easily resume where you left off. When no project is loaded a global session file is used.

2.3 Opening a File in the Project

Another advantage of setting up a project is that you can quickly open any file with a recognized extension located in one of the folders of the project. Use *File -> Open File in Project* (`Ctrl`+`P`) to open the file filter and just type the name of the file you'd like to open.

Future Extensions

There are many ways in which the projects could be made more powerful:

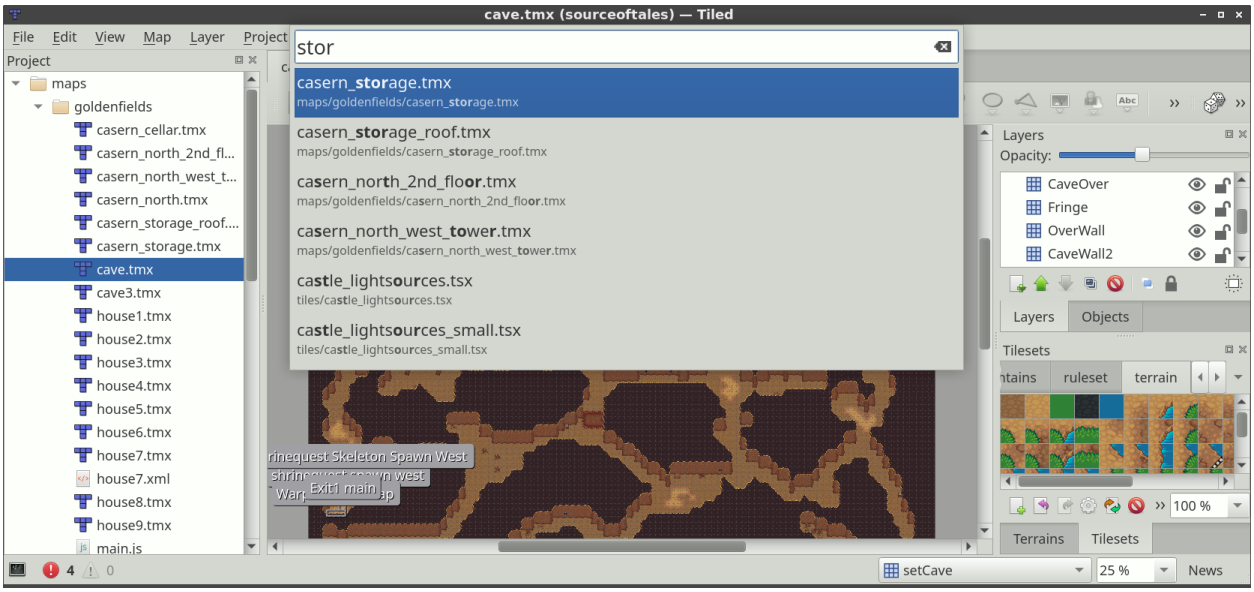


Fig. 1: Open File in Project

- Make the custom types defined in the project accessible through the *scripting API* (#3419).
- Allow turning off features on a per-project basis, to simplify the UI and reduce the chance of accidentally doing something your project doesn't support.
- Recognizing the various assets in your project, so that selection of images, tilesets and templates can be made more efficient (potentially replacing the system file dialog).

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

CHAPTER

THREE

EDITING MAPS

A map is a top-level asset file that contains *layers* and references *tileset files* (which can optionally be embedded). It defines the coordinate system, tile size, orientation, render order and other map-wide configuration options that affect how the content is displayed and exported.

3.1 Creating a Map

You can create a new map via *File -> New -> New Map...*. The New Map dialog lets you set:

- **Orientation** (see *Map Orientations*)
- **Tile Layer Format** (the storage format for tile data)
- **Render Order** (only for orthogonal and oblique maps)
- **Map Size** (fixed width/height or infinite)
- **Tile Size** (tile width/height in pixels)

All of these options can be changed later in the **Properties** panel, via *Map -> Map Properties...*.

3.2 Map Orientations

The map orientation controls how the tile grid is projected and how tiles are positioned relative to each other. Tiled supports the following orientations.

3.2.1 Orthogonal

The classic top-down grid where rectangular tiles are arranged in straight rows and columns. This is the most straightforward orientation.

3.2.2 Isometric

Isometric maps are projected to give a 3D-like appearance. Tiles are still arranged in a grid, but are drawn as diamonds (though Tiled doesn't transform your art, you'll need to provide the diamond shaped tiles). Object positions are stored in a projected coordinate space (see *Object Layers*).

3.2.3 Isometric (Staggered)

Staggered isometric maps also use diamond-shaped tiles, but the grid is staggered every other row or column. The exact staggering is controlled by the **Stagger Axis** and **Stagger Index** map properties.

This orientation allows a map based on isometric tiles to still have an overall rectangular shape.

3.2.4 Hexagonal (Staggered)

Hexagonal maps use hex tiles arranged in a staggered grid. Like staggered isometric maps, the staggering is controlled by **Stagger Axis** and **Stagger Index**. Hexagonal maps also support a **Hex Side Length** that defines the length of the straight edges of the hex tile.

The following table shows how **Stagger Axis** relates to the two common hex tile orientations:

Stagger Axis	Hex Tile Orientation
X	Pointy-top
Y	Straight-top

3.2.5 Oblique

Oblique maps apply a skew transform to achieve a pseudo-3D projection. The amount of skew is controlled by the **Skew** map property, in pixels.

Skew X determines the horizontal offset applied for each row, while **Skew Y** determines the vertical offset applied for each column.

3.3 Map Properties

The following properties are especially relevant when configuring a map.

3.3.1 Map Size (Fixed vs. Infinite)

Maps can be **fixed size** or **infinite**. A fixed map has a set width and height in tiles, while an infinite map has an auto-growing canvas that expands as you paint. The choice impacts how tile layers are stored.

For details on working with infinite maps and converting between the two, see *Using Infinite Maps*.

3.3.2 Tile Size

The tile width and height define the size of each tile in pixels. These values affect the tile grid and the map bounds, but does not affect the size at which tiles are rendered (unless the relevant tileset has **Tile Render Size** set to **Map Grid Size**).

3.3.3 Parallax Origin

The parallax origin defines the reference point for *parallax scrolling factor* on layers. It is stored per map and defaults to (0, 0), which is the top-left of the map's bounding box.

3.3.4 Tile Layer Format

The tile layer format determines how tile data is stored when saving or exporting the map. Some formats are more compact or faster to parse than others. You can change the format at any time in the **Properties** panel, via *Map -> Map Properties...*.

CHAPTER

FOUR

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 and a *parallax scrolling factor*, which can be used to position them independently of each other, for example to fake depth. Finally their contents can be tinted by multiplying with a custom *tint color*.

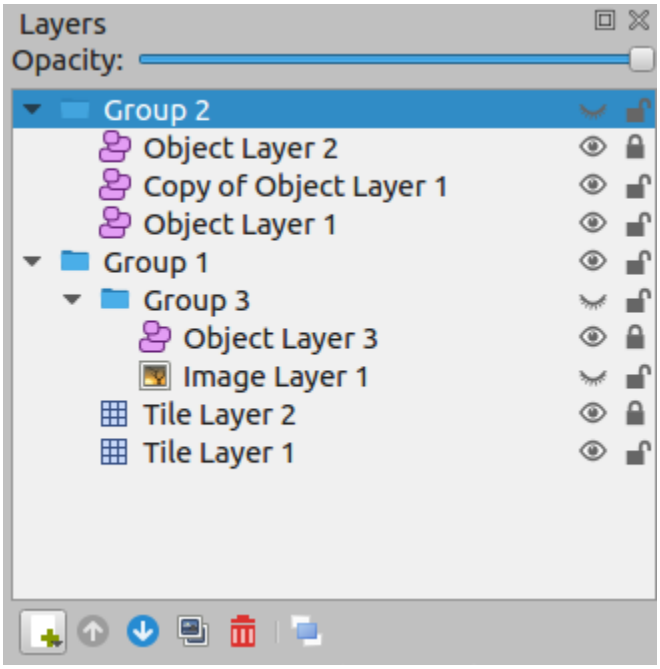


Fig. 1: The eye and lock icon toggle the visibility and locked state of a layer respectively.

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

4.1 Layer Types

4.1.1 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.

4.1.2 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
- **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 *class*, which is useful since it can be used to customize the color of their label and the available *custom properties* for this object. For tile objects, the class 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.

4.1.3 Image Layers

Image layers provide a way to quickly include a single image as foreground or background of your map. They currently have limited functionality and you may consider adding the image as a Tileset instead and place it as a *Tile Object*. This way, you gain the ability to freely scale and rotate the image.

However, image layers can be repeated along the respective axes through their *Repeat X* and *Repeat Y* properties.

The other 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.

4.1.4 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, lock and *tint color* 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.

4.2 Parallax Scrolling Factor

The parallax scrolling factor determines the amount by which the layer moves in relation to the camera.

By default its value is 1, which means its position on the screen changes at the same rate as the position of the camera (in opposite direction). A lower value makes it move slower, simulating a layer that is further away, whereas a higher value makes it move faster, simulating a layer positioned in between the screen and the camera.

A value of 0 makes the layer not move at all, which can be useful to include some pieces of your ingame UI or to mark its general viewport boundaries.

Negative values make the layer move in opposite direction, though this is rarely useful.

When the parallax scrolling factor is set on a group layer, it applies to all its child layers. The effective parallax scrolling factor of a layer is determined by multiplying the parallax scrolling factor by the scrolling factors of all parent layers.

4.2.1 Parallax Reference Point

To match not only the scrolling speed but also the positioning of layers, we need to use the same points of reference. In Tiled these are the parallax origin and the center of the view. The parallax origin is stored per map and defaults to (0,0), which is the top-left of the maps bounding box. The distance between these two points is multiplied by the parallax factor to determine the final position on the screen for each layer. For example:

- If the parallax origin is in the center of the view, the distance is (0,0) and none of the parallax factors have any effect. The layers are rendered where they would have been, if parallax was disabled.
- Now, when the map is scrolled right by 10 pixels, the distance between the parallax origin and the center of the view is 10. So a layer with a parallax factor of 0.7 will have moved just `0.7 * 10 = 7` pixels.

Quite often, a viewport transform is used to scroll the entire map. In this case, one may need to adjust the position of each layer to take its parallax factor into account. Instead of multiplying the distance with the parallax factor directly, we now multiply by `1 - parallaxFactor` to get the layer position. For example:

- When the camera moves right by 10 pixels, the layer will have moved 10 pixels to the left (-10), so by positioning the layer at `10 * (1 - 0.7) = 3`, we're making sure that it only moves 7 pixels to the left.

4.3 Tinting Layers

When you set the *Tint Color* property of a layer, this affects the way images are rendered. This includes tiles, tile objects and the image of an *Image Layer*.

Each pixel color value is multiplied by the tint color. This way you can darken or colorize your graphics in various ways without needing to set up separate images for it.

The tint color can also be set on a *Group Layer*, in which case it is inherited by all layers in the group.

4.4 Blend Modes

Tiled provides support for several common blend modes (also called compositing operators) for layers. These modes allow you to modify the appearance of a layer by blending it with the layers beneath it in various ways. By default, layers in Tiled use the Normal blend mode.

Below is the full list of blend modes available in Tiled, along with links to their equivalents in the SVG Compositing Specification, where you can see examples and calculation details.

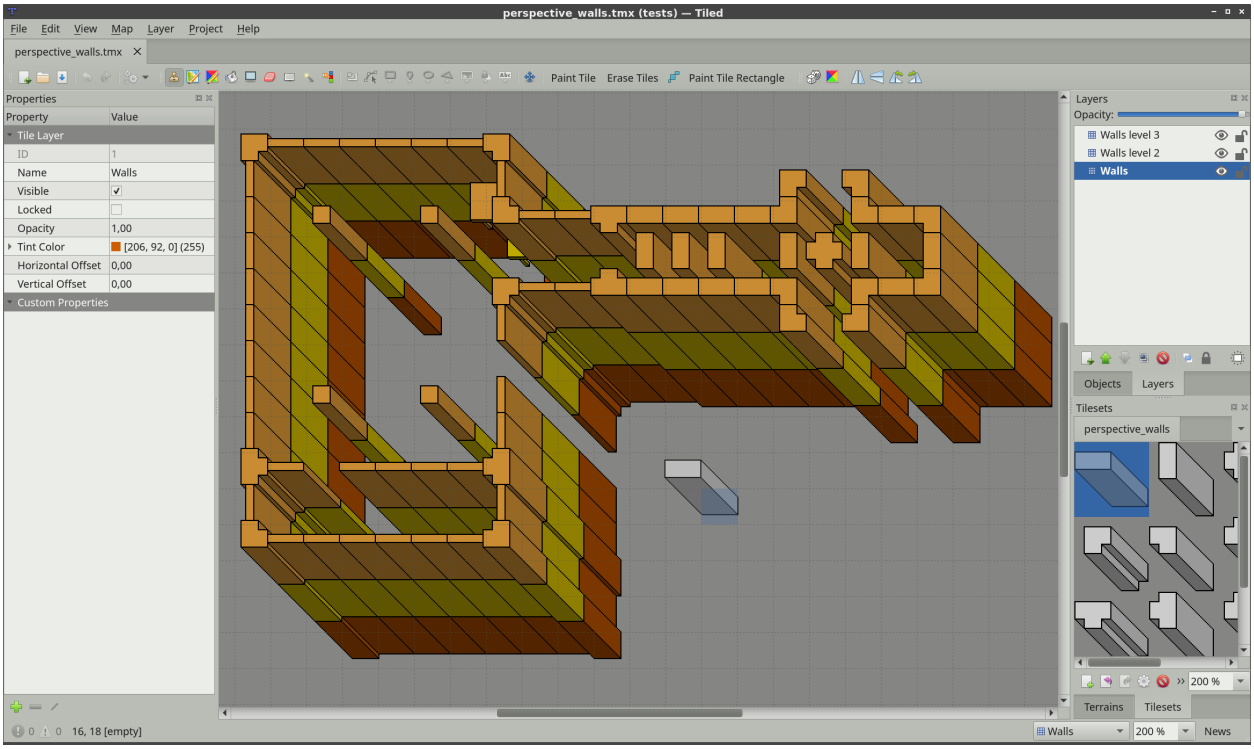


Fig. 2: A gray tileset rendered in a different color for each layer.

Mode	SVG equivalent
Normal	src-over
Add	plus
Multiply	multiply
Screen	screen
Overlay	overlay
Darken	darken
Lighten	lighten
Color Dodge	color-dodge
Color Burn	color-burn
Hard Light	hard-light
Soft Light	soft-light
Difference	difference
Exclusion	exclusion

In OpenGL, these blend modes can be implemented using `glBlendEquation` with values from the KHR_blend_equation_advanced extension. In Vulkan, they are part of the VK_EXT_blend_operation_advanced extension.

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 sponsoring Tiled development. The more support I receive the more time I can afford to spend improving Tiled!

CHAPTER

FIVE

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.

5.1 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 *Terrain Fill Mode* using the Terrain tile  button on the tool bar to have the Stamp Brush paint using random terrain tiles. This makes adjacent tiles match edge and corner terrains to be placed. Terrain tiles are described in detail in *Using Terrains*.
- 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.

5.2 Terrain Brush

Shortcut: `T` 

The Terrain Brush allows for efficient editing with certain types of terrain transitions (corner-based, edge-based or a combination). Setting it up requires associating terrain information with your tiles, which is described in detail in *Using Terrains*.

Similarly to the *Stamp Brush*, you can draw lines by holding `Shift`. When holding `Ctrl`, the size of the edited area is increased to cover an entire tile rather than just one corner or edge.

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 increasing the edited area.

5.3 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 *Terrain 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*.

5.4 Shape Fill Tool

Shortcut: `P` 

This tool provides a quick way to fill rectangles or ellipses with a certain tile or pattern.

- Holding `Shift` fills an exact square or circle.
- Holding `Alt` draws the rectangle or ellipse centered around the starting location.

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

5.5 Eraser

Shortcut: `E` 

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

- Holding `Shift` erases on all layers.

5.6 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 the selection mode before starting the selection:

- Hold `Shift` to expand the current selection with the new area
- Hold `Ctrl` to subtract the new area from the current selection
- Hold `Ctrl` and `Shift` to select 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.

While selecting an area, the following modifiers can be used:

- Hold `Shift` to constrain the selection to a square.
- Hold `Ctrl` to expand the selection from the starting location.

5.7 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. When no tile drawing tool is selected, tries to capture the current tile selection (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

SIX

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.

By using *tile objects*, objects of various types can be made easy to recognize or they can be used for purely graphical purposes. In some cases they can replace the use of tile layers entirely, as demonstrated by the "Sticker Knight" example shipping with Tiled.

All objects can have *custom properties*, which can also be used to create *connections between objects*.

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

6.1 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.

6.1.1 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.

Rectangle objects have their origin in the top-left. However, 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.

6.1.2 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.

6.1.3 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.

6.1.4 Insert Capsule

Shortcut: Shift+C 

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

6.1.5 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.

6.1.6 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 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*.

You can customize the alignment of tile objects using the *Object Alignment* property on the *Tileset*. For compatibility reasons this property is set to *Unspecified* by default, in which case tile objects are bottom-left aligned in all orientations except on *Isometric* maps, where they are bottom-center aligned. Setting this property to *Top Left* makes the alignment of tile objects consistent with that of *rectangle objects*.

6.1.7 Insert Template

Shortcut: `V` 

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

6.1.8 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.

6.2 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.

6.2.1 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.

6.2.2 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.

6.2.3 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.

6.2.4 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`.

6.2.5 Changing Stacking Order

If the active *Object Layer* has its Draw 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.

6.2.6 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.

6.3 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 polyline.

While creating or extending a polygon/polyline with the *Insert Polygon* tool, you can press `Backspace` to remove the previously added point.

6.4 Connecting Objects

It can often be useful to connect one object with another, like when a switch should open a certain door or an NPC should follow a certain path. To do this, add a custom property of type `object` to the source object. This property can then be set to the desired target object in several ways.

Make sure the property value is selected, as seen on the following screenshot:

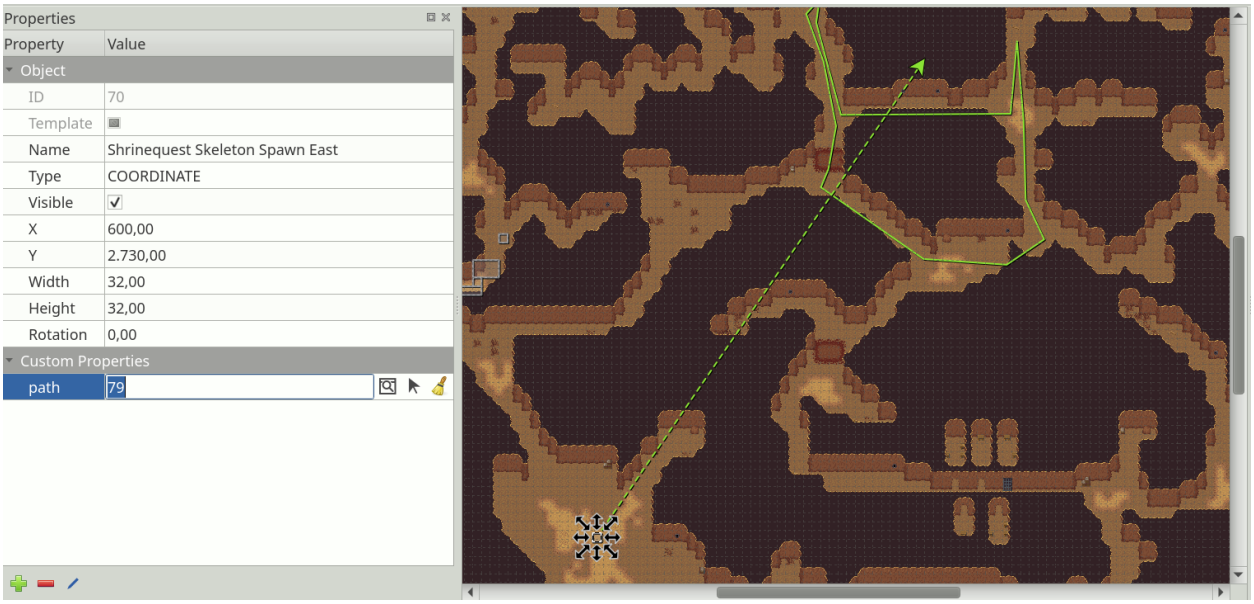


Fig. 1: Object Connection Property

Then, you can set the connection by either:

- Typing in the ID of the target object.
- Clicking the icon with the window and magnifier, to open a dialog where you can filter all objects on the map to find your target object.
- Clicking the arrow icon and then clicking an object on the map to set it as the target object.

As shown on the screenshot above, any connections between objects are rendered as arrows, taking the color of their target object (defined as part of the *object class* or by the color of the object layer). You can toggle the display of these arrows using *View -> Show Object References*.

If you'd like to get to the target object, but it's very far away, you can jump there by right-clicking the property and selecting *Go to Object*.

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 sponsoring Tiled development. The more support I receive the more time I can afford to spend improving Tiled!

CHAPTER

SEVEN

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.

7.1 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.

7.2 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.

Object Alignment
: The alignment to use for *tile objects* referring to tiles from this tileset. This affects the placement of the tile relative to the position of the object (the origin) and is also the location around which the rotation is applied.

Possible values are: *Unspecified* (the default), *Top Left*, *Top*, *Top Right*, *Left*, *Center*, *Right*, *Bottom Left*, *Bottom* and *Bottom Right*. When unspecified, tile object alignment is generally *Bottom Left*, except for Isometric maps where it is *Bottom*.

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*. 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.

7.3 Tile Properties

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

Class
This property refers to custom classes defined in the *Custom 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.

7.4 Terrain Information

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

7.5 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.

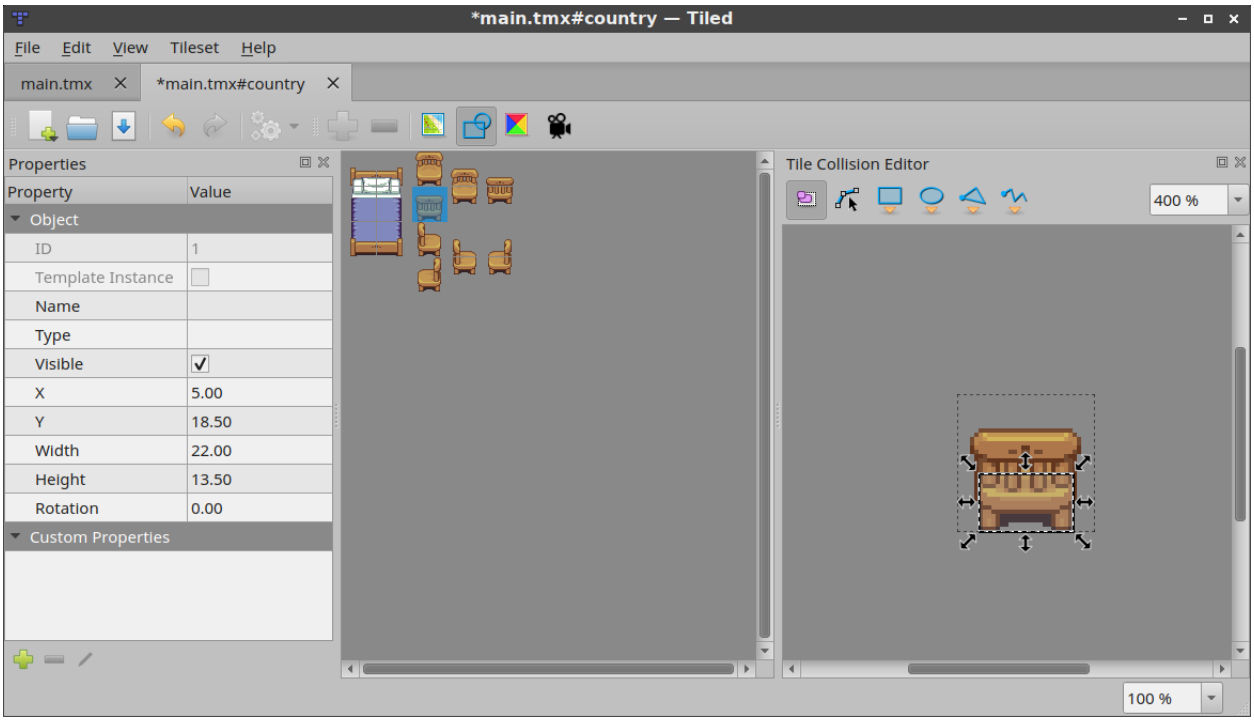


Fig. 1: Tile Collision Editor

To be able to easily check whether your tiles have the right collision shapes set up, they can be rendered on the map. To enable this, check *Show Tile Collision Shapes* in the *View* menu. The collision shapes are rendered for both tile layers and tile objects.

7.6 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 (or other value when set using the *Frame Duration* field at the top).
- Double-click on the duration of a frame to change it.
- Drag frames around in the list to reorder them.

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

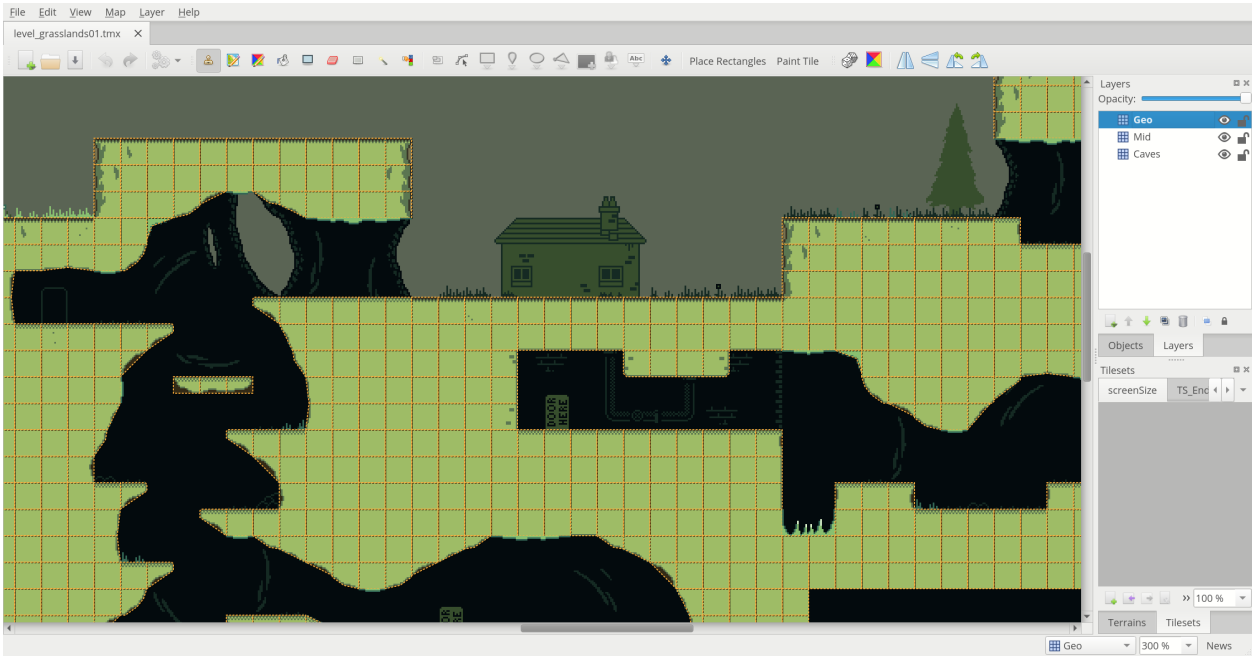


Fig. 2: Collision shapes rendered on the map. This map is from Owyn's Adventure.

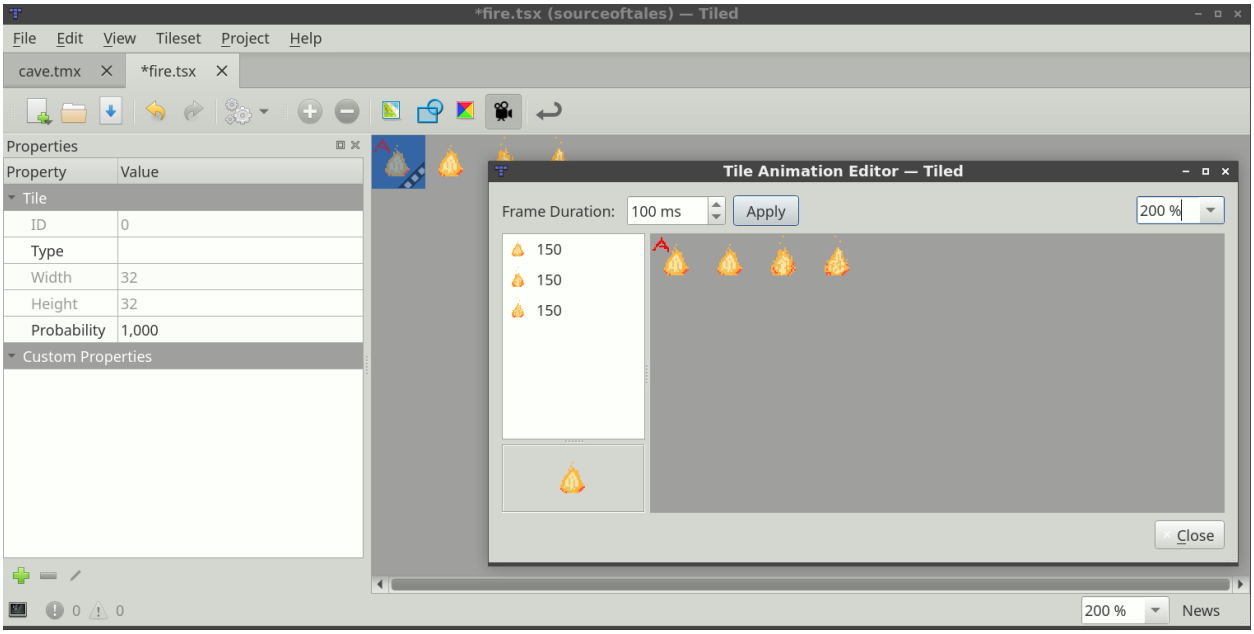


Fig. 3: Tile Animation Editor

You can change the duration of multiple frames at once by selecting them, changing the value in the *Frame Duration* field and then clicking *Apply*.

Future Extensions

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

Terrain Sets

- Make it easier to set up terrain (#1729)

Tile Collision Editor

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

Tile Animation Editor

- Support multiple named animations per tile (#986)
- Make it easier to define animations spanning multiple tiles (#811)

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

CHAPTER

EIGHT

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. It also supports multi-selection, for changing the properties of many objects at once.

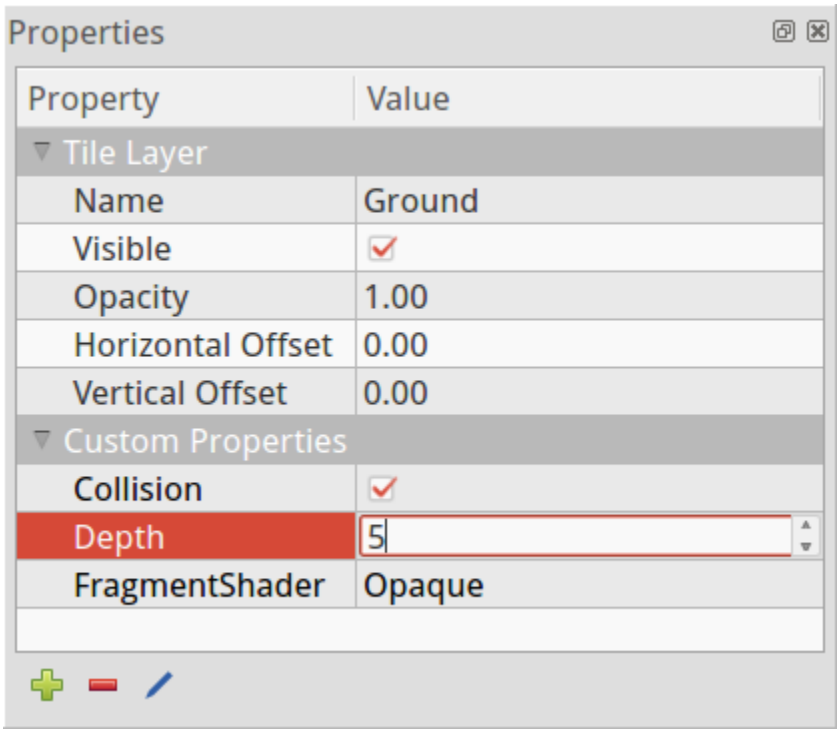


Fig. 1: Properties View

8.1 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. Tiled supports the following basic property types:

- **bool** (true or false)
- **color** (a 32-bit color value)

- **file** (a file reference, which is saved as a relative path)
- **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)

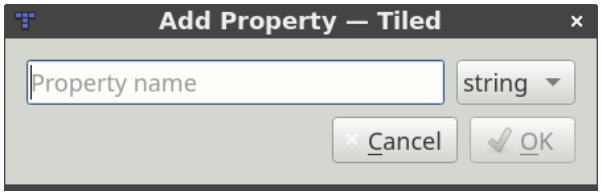


Fig. 2: Add Property Dialog

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.

The context menu for custom file properties provides a quick way to open the file in its preferred editor. For object references, there is an action to quickly jump to the referenced object.

8.2 Custom Types

In addition to the basic property types listed above, you can define custom types in your project. Tiled supports *custom enums* and *custom classes*.

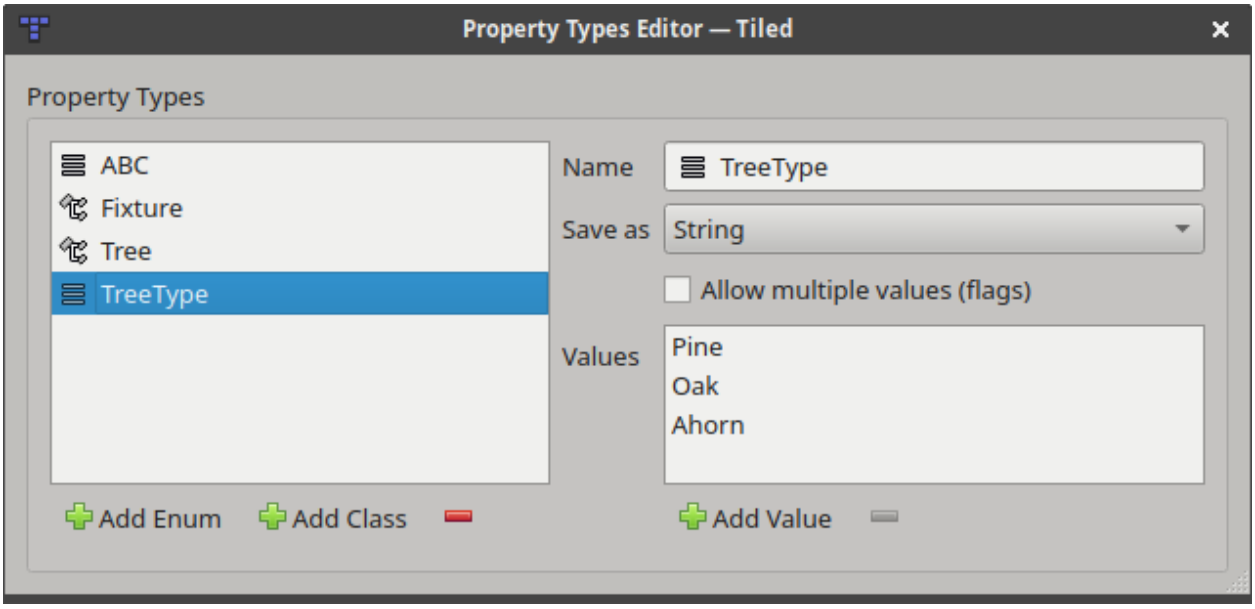


Fig. 3: Custom Types Editor

Note

These types are automatically saved in the *project file*. Hence you need to create a project, before you can set up your custom types.

8.2.1 Custom Enums

An enum is useful if you want to limit the options for a certain property to a fixed set of values.

An enum also defines how its value is saved. It can be saved as a string, saving one of its values directly. Alternatively it can be saved as a number, the index of the current value in the list of values. The former is more readable whereas the latter could easier and more efficient to load.

Finally, an enum can also allow multiple values to be chosen. In this case each option is displayed with a checkbox. When saving as string, a comma-separated list is used and when saving as number the selected indexes are encoded as bitflags. In both cases, the maximum number of flags supported is 31, since internally a 32-bit signed integer is used to store the value.

8.2.2 Custom Classes

A class is useful if you want to be able to add a set of properties at once, with predefined defaults. It can also prevent excessive prefixing of property names. Classes can have members referring to other classes.

Each data type has a "Class" property, which can be used to refer to a custom class. The members of this class will then be directly available as custom properties of that instance (before Tiled 1.9, this feature was only available for objects and tiles as the "Type" property).

Each class can also have a custom color, which is used to make objects more recognizable. The class color is used when rendering shape objects, object name labels and connections between objects.

In the *JSON* and *Lua* file formats, custom class properties used as property values are saved using the native object and table constructs.

8.3 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 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).

8.3.1 Typed Tiles

If you're using *tile objects*, you can set the class on the tile to avoid having to set it on each object instance. Setting the class 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 classes 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:

- **Customized basic properties**, where you can set properties like the minimum or maximum value, the precision or a different default value.
- **Array properties**, which would be properties having a list of values (#1493).

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)

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

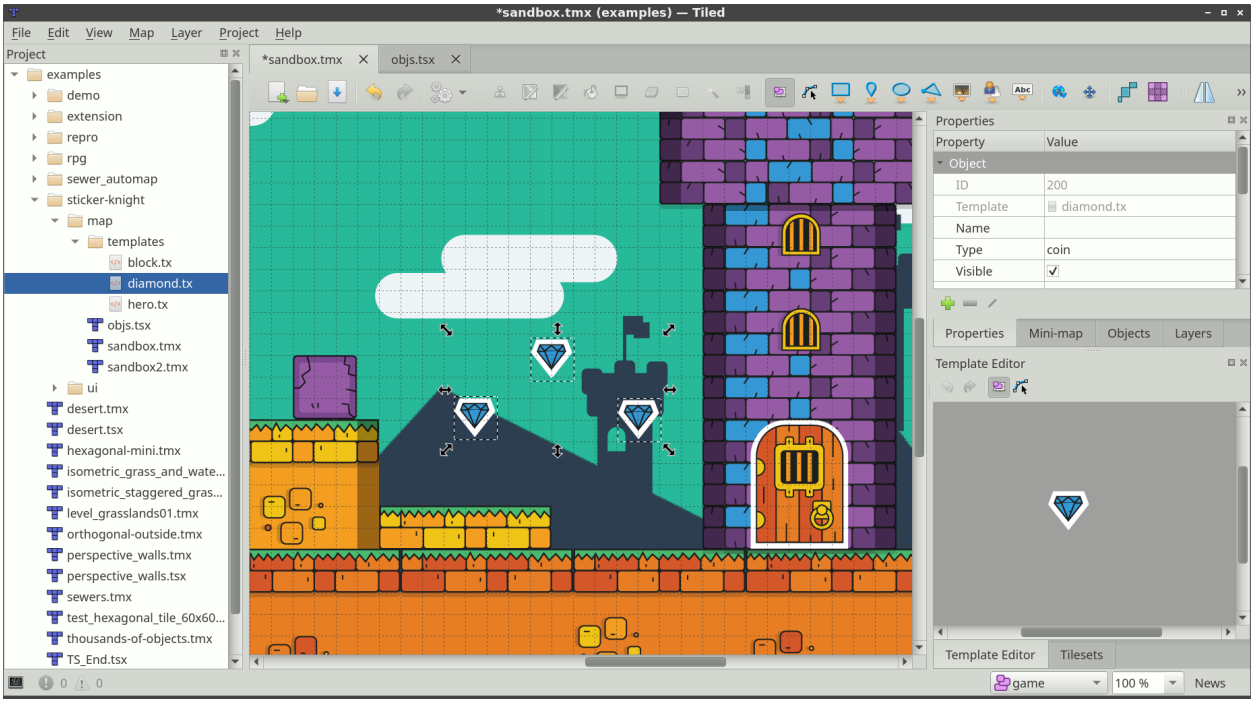
CHAPTER

NINE

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 class and properties, or even just finding the right tile in the tileset.

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



9.1 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.

To be able to select your templates for editing or instantiating you'll generally want to use the *Project view*, so make sure to save your templates in a folder that is part of your project. Dragging in a template from a file manager is also possible.

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.

9.2 Creating Template Instances

Shortcut: `V`

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

9.3 Editing Templates

Editing templates is done using the *Template Editor* view. A template can be opened for editing by selecting it in the Project view or by dragging the template file on the *Template Editor* view. The template can also be selected using the *Open File in Project* action.

When selecting the template in the *Template Editor* view, the *Properties* view will show the template's properties, where they can be edited.

Any changes to the template are saved automatically and are immediately reflected on all template instances.

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.

If a template file changes on disk, it is automatically reloaded and any changes will be reflected in the *Template Editor* as well as on any template instances.

9.4 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*.

If your map loader does not support object templates, but you'd still like to use them, you can enable the *Detach templates* *export option*.

Future Extensions

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

CHAPTER

TEN

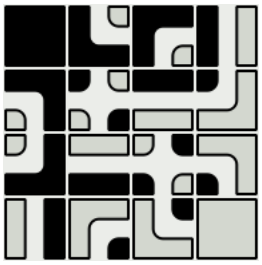
USING TERRAINS

When editing a tile map, sometimes we don't think in terms of *tiles* but rather in terms of *terrains* - areas of tiles with transitions to other kinds of tiles. Say we want to draw a patch of grass, a road or a certain platform. In this case, manually choosing the right tiles for the various transitions or connections quickly gets tedious. The *Terrain Brush* was added to make editing tile maps easier in such cases.

Warning

While Tiled has supported terrains since version 0.9 and later supported a similar feature called "Wang tiles" since version 1.1, both features were unified and extended in Tiled 1.5. As a result, *terrain information defined in Tiled 1.5 can't be used by older versions.*

The Terrain Brush relies on the tileset providing one or more *Terrain Sets* - sets of tiles labeled according to their terrain layouts. Tiled supports the following terrain sets:

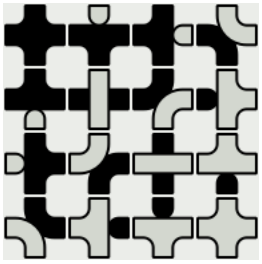


Corner Set
: Tiles that needs to match neighboring tiles at their corners, with a transition from one type of terrain to another in between. A complete set with 2 terrains has 16 tiles.

Edge Set
: Tiles that need to match neighboring tiles at their sides. This is common for roads, fences or platforms. A complete set with 2 terrains has 16 tiles.

Mixed Set
: Tiles that rely on matching neighboring tiles using both their corners and sides. This allows a tileset to provide more variation, at the cost of needing significantly more tiles. A complete set with 2 terrains has 256 tiles, but reduced sets like the 47-tile Blob tileset can be used with this type as well.

Based on the information in a terrain set, the *Terrain Brush* can understand the map and automatically choose the right tiles when making edits. When necessary, it also adjusts neighboring tiles to make sure they correctly connect to the modified area. A terrain set can contain up to 254 terrains.



The *Stamp Brush*, as well as the *Bucket Fill Tool* and the *Shape Fill Tool*, also have a mode where they can *fill an area with random terrain*.

10.1 Define the Terrain Information

10.1.1 Creating the Terrain Set

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.

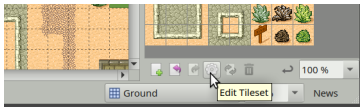


Fig. 1: Edit Tileset button

Then, activate the terrain editing mode by clicking on the *Terrain Sets*  button on the tool bar. With this mode activated, the *Terrain Sets* view will become visible, with a button to add a new set. In this example, we'll define a *Corner Set*.

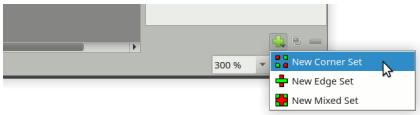


Fig. 2: Adding a Terrain Set

When adding a terrain set, the name of the new set will automatically get focus. Give the set a recognizable name, in the example we'll type "Desert Ground". We can also set one of the tiles as the icon of the set by right-clicking a tile and choosing "Use as Terrain Set Image".

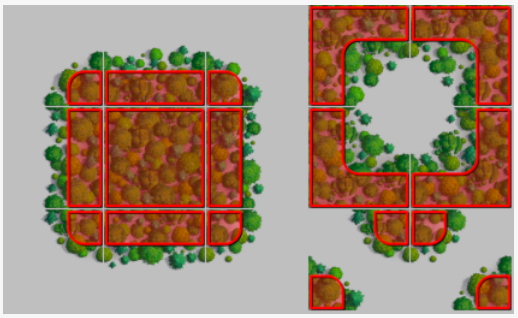
10.1.2 Adding Terrains

The new set will have one terrain added by default. If we already know we need additional ones, click the *Add Terrain* button to add more.

Each terrain has a name, color and can have one of the tiles as its icon it to make it more recognizable. Double-click the terrain to edit its name. To change the color, right-click the terrain and choose "Pick Custom Color". To assign an icon, select the terrain and then right-click a tile, choosing "Use as Terrain Image".

Fig. 3: Our Terrains

Note



We generally don't need to define an explicit terrain for "empty tiles". If you have tiles transitioning to nothing, it should be enough to not mark those areas.

(Example © Game Shovel LLC, license: MIT)

With our terrains set up we're ready to mark each of our tiles.

10.1.3 Marking the Tiles

Note that for a *Corner Set*, we can only mark the corners of the tiles. For a *Edge Set*, we're limited to marking the edges of our tiles. If we need both we need to use a *Mixed Set*. If it turns out that we chose the wrong type of terrain set, we can still change the type in the Properties view (right-click the terrain set and choose *Terrain Set Properties...*).

With the terrain we want to mark selected, click and drag to mark the regions of the tiles that match this terrain.

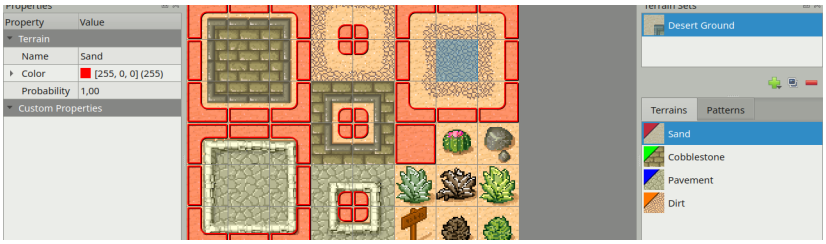


Fig. 4: Here we have marked all the sandy corners in our example tileset.

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.

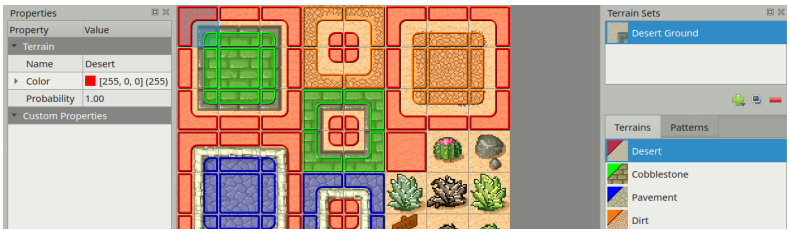


Fig. 5: We're done marking the terrain of our tiles.

Patterns View

Next to the *Terrains* tab there's also a *Patterns* tab. This view can be useful when marking complete sets, since it can highlight still missing patterns. Each pattern which already occurs on a tile in the tileset is darkened, to make the missing patterns stand out. Note though, that it is not necessary for a terrain set to have all possible patterns, especially when using more than 2 terrains.

Fig. 6: Patterns view, showing all possible combinations in the set.

10.2 Editing with the Terrain Brush

Now you can disable the *Terrain Sets*  mode by clicking the tool bar button again. Then switch back to the map and activate the *Terrain Sets* window. Select the terrain set we have just set up, so we can use its terrains.

Click on the Sand terrain and try to paint. You may immediately notice that nothing 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 also have no transitions to "nothing" in our tileset). There are two ways out of this:

- We can hold `Ctrl` (`Command` on a Mac) to paint a slightly larger area. This way we will paint at least a single tile filled with the selected terrain, though this is not convenient for painting larger areas.
- Assuming we're out to create a desert map, it's better to start by filling the 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*.

Once we've painted some sand, let's select the Cobblestone terrain. Now you can see the tool in action!

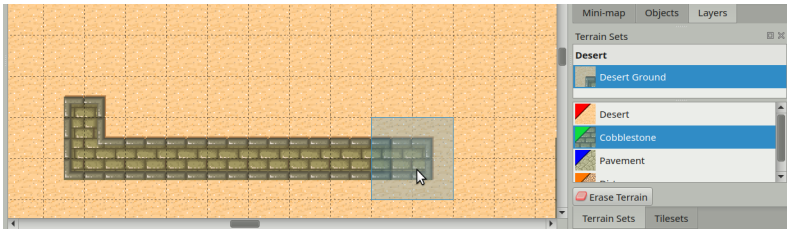


Fig. 7: Drawing cobblestone

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!

Note

An *Erase Terrain* button is provided for the case where your terrain tiles transition to nothing. This allows for erasing parts of your terrain while choosing the right tiles as well. This mode does nothing useful when there are no transitions to nothing in the selected Terrain Set.

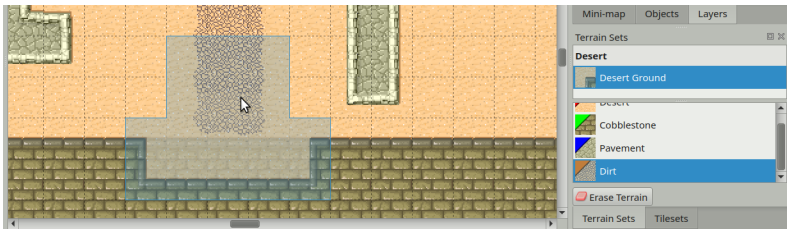


Fig. 8: Drawing dirt

10.3 Terrain Fill Mode

The *Stamp Brush*, *Bucket Fill Tool* and the *Shape Fill Tool* have a *Terrain Fill Mode*, which can be used to paint or fill an area with random terrain. With this mode activated, each cell will be randomly chosen from all those in the selected Terrain Set, making sure to match all adjacent edges and/or corners.

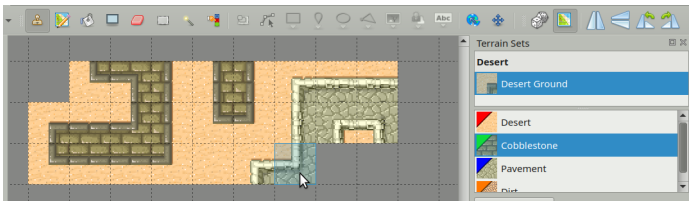


Fig. 9: Stamp Brush with Terrain Fill Mode Enabled

Note that since this mode makes sure that newly placed tiles match up with any already existing tiles, generally nothing will change when painting with the Stamp Brush on existing terrain. The exception is when there are multiple variations of the same tile, in which case it will randomize between those.

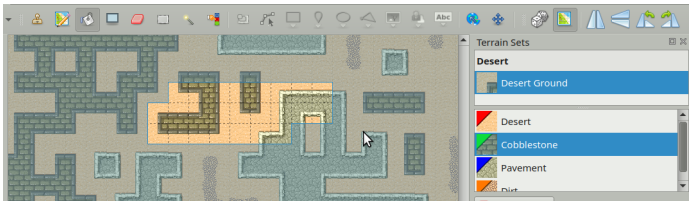


Fig. 10: Bucket Fill with Terrain Fill Mode Enabled

When filling a shape or an area, only the edges of the filled area need to connect to any existing tiles. Internally the area is completely randomized.

10.4 Tile and Terrain Probability

Both the *Terrain Fill Mode* and the Terrain Brush will by default consider all matching tiles with equal probability. Both individual tiles as well as terrains have a *Probability* property, which can be used to change the frequency with which a certain tile or terrain is chosen compared to other valid options.

The relative probability of a tile is the product of its own probability and the probability of the terrain at each corner and/or side.

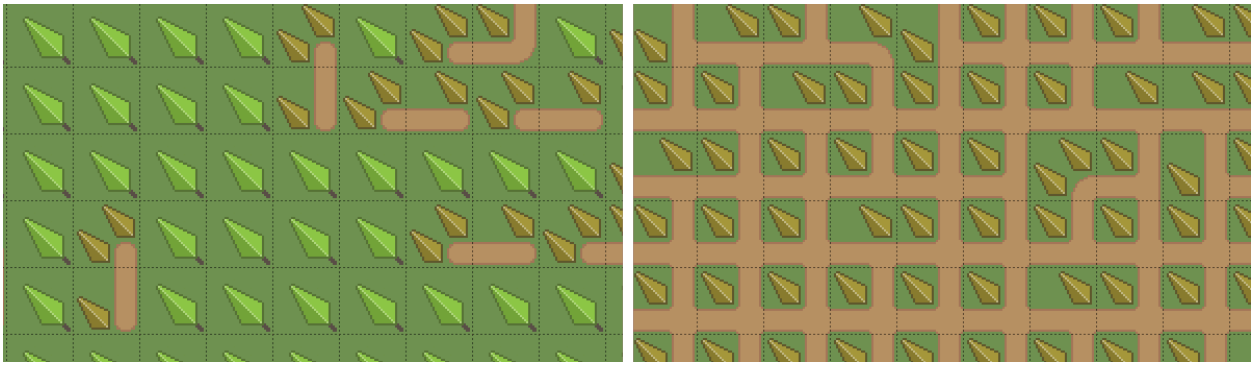


Fig. 11: Left shows “path” with probability 0.1, right shows “path” with probability 10.

10.4.1 Probability for Variations

A common usage for probability, especially at the individual tile level, is to make certain variations of a tile less common than others. Our example tileset contains several bushes and other decorations which we may randomly want to scatter across the desert.

To achieve this, first of all we mark all of them as “sand” tiles, because this is their base terrain. Then, to make them less common than the regular sand tile, we can put their probability on 0.01. This value means they are each 100 times less likely to be chosen than the regular sand tile (which still has its default probability of 1). To edit the *Probability* property of the tiles we need to exit the *Terrain Sets* mode.

Fig. 12: Setting low probability on decoration tiles.

Hint

It is also possible to put the probability to 0, which disables automatic usage of a tile entirely. This can be useful because it still makes the tools aware of the terrain of a certain tile, which is taken into account when modifying neighboring tiles.

10.5 Tile Transformations

Tiled supports flipping and rotating tiles. When using terrains, tiles can be automatically flipped and/or rotated to create variations that would otherwise not be available in a tileset. This can be enabled in the *Tileset Properties*.

The following transformation-related options are available:

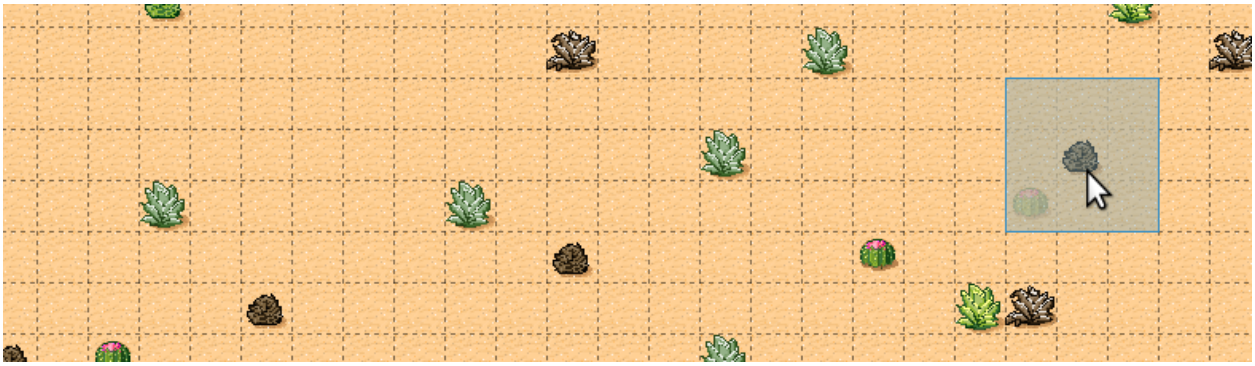


Fig. 13: Random decorative tiles appearing with low probability.

Flip Horizontally
: Allow tiles to be flipped horizontally.

Flip Vertically
: Allow tiles to be flipped vertically. This would be left disabled when the graphics contain shadows in vertical direction, for example.

Rotate
: Allow tiles to be rotated (by 90, 180 or 270-degrees).

Prefer Untransformed Tiles
: When transformations are enabled, it could happen that a certain pattern can be filled by either a regular tile or a transformed tile. With this option enabled, the untransformed tiles will always take precedence. Leaving this option disabled allows transformations to be used to create more variation.

Fig. 14: With rotations enabled, the normally 47-tiles Blob tileset can be reduced to a mere 15 tiles.

10.6 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:

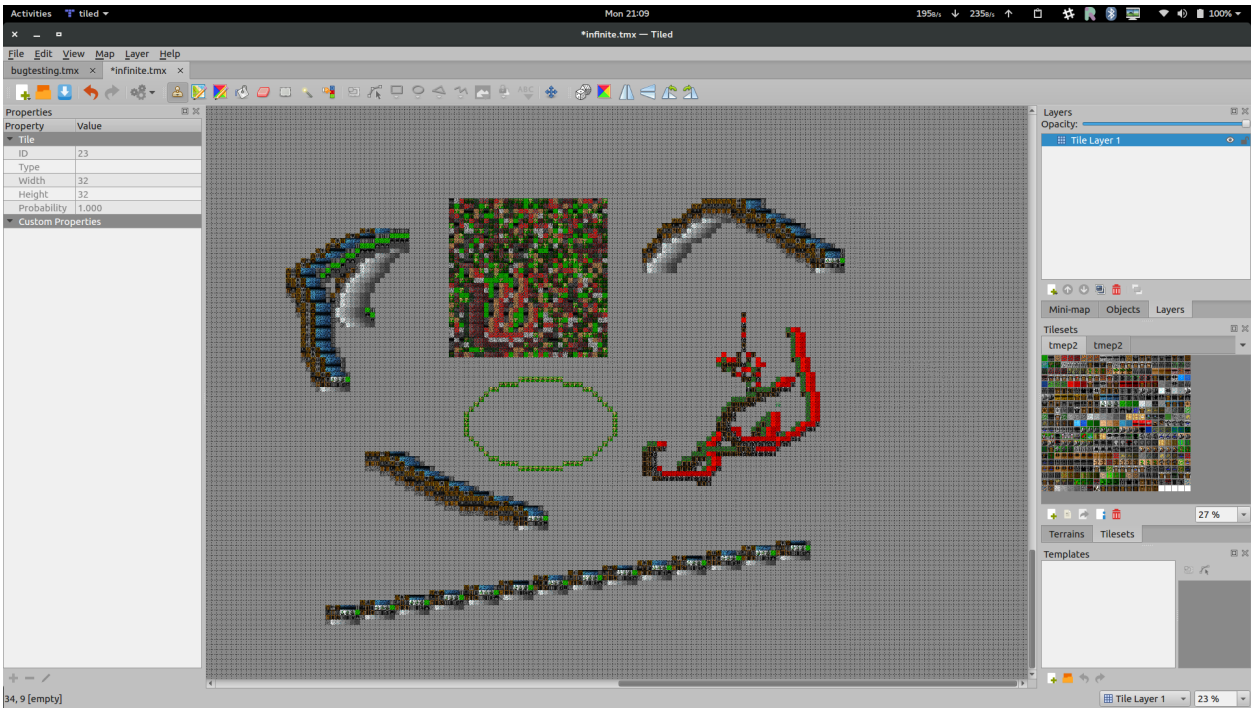
- For one terrain to interact with another, they need to be part of the same *Terrain Set*. This also means all tiles need to be part of the same tileset. If you have tiles in different tilesets that you want to transition to one another, you will need to merge the tilesets into one.
- Since defining the terrain information can be somewhat laborious, 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 terrains (up to 254) 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*.
- There's a collection of tilesets that contain transitions that are compatible with this tool on OpenGameArt.org.

CHAPTER

ELEVEN

USING INFINITE MAPS

Infinite maps in Tiled free you from the constraints of the fixed-size map. With an "auto-growing" canvas, you can paint on an infinite grid without being limited by width and height. This document guides you through creating, editing, and converting infinite maps in Tiled.



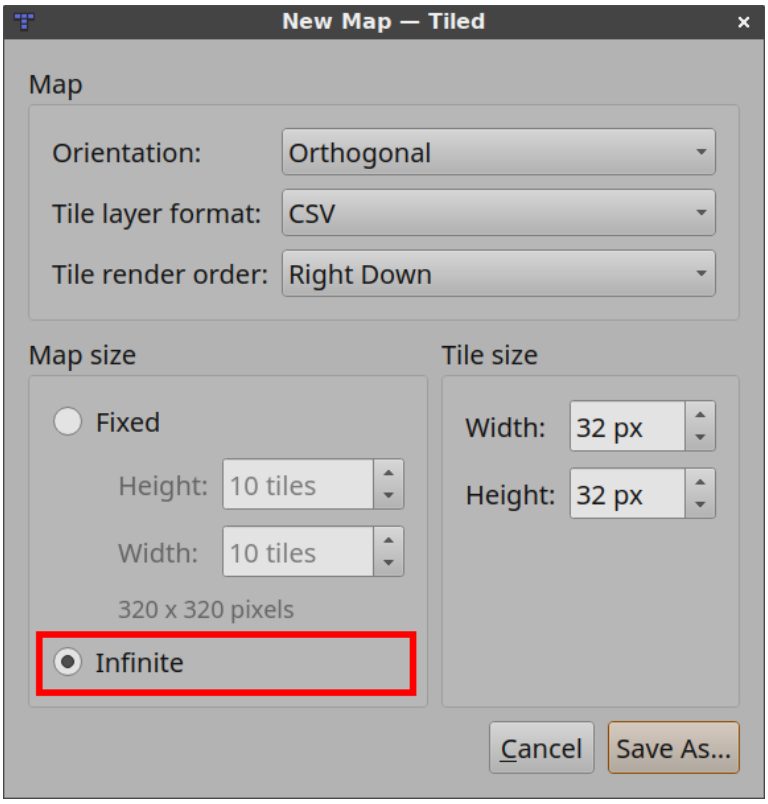
11.1 Creating an Infinite Map

1. Open the New Map dialog (*File -> New -> New Map*).
2. Ensure the 'Infinite' option is selected.

The map you create will have an infinite canvas.

11.2 Editing an Infinite Map

Most tools in Tiled work the same way for infinite maps as they do for fixed-size maps. However, the *Bucket Fill Tool* fills only the current bounds of a tile layer. As you paint, these bounds expand.



11.3 Converting Between Infinite And Fixed-Size Maps

You can toggle between infinite and fixed-size maps in the Map Properties window. When converting an infinite map to a fixed-size map, Tiled determines the final map's width and height based on the bounds of all tile layers.

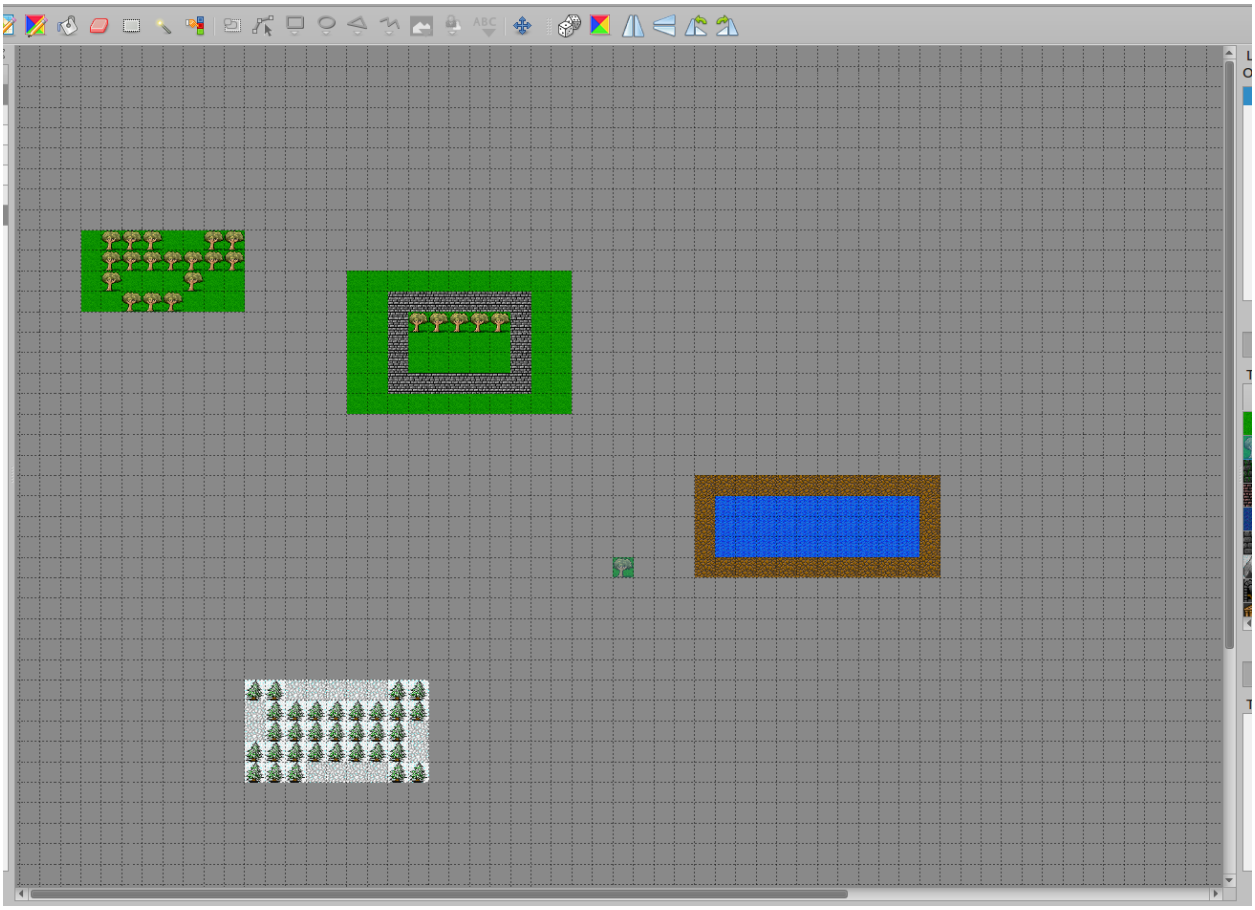


Fig. 1: The Initial Infinite Map

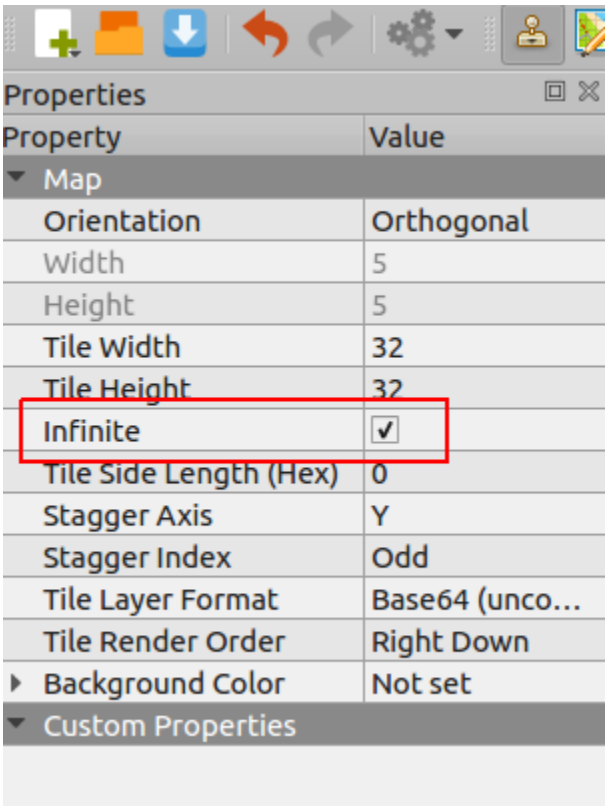


Fig. 2: Unchecking the Infinite property in Map Properties

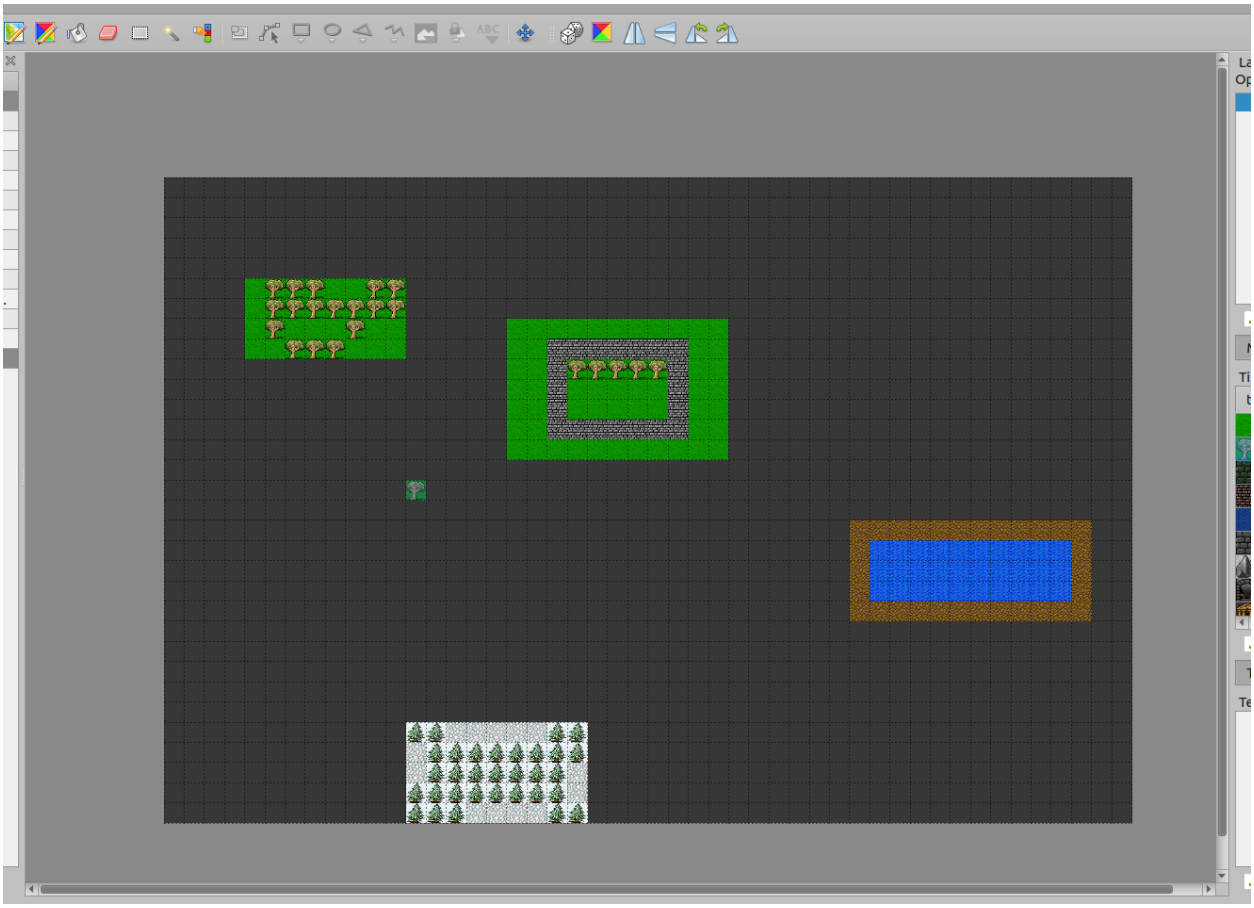


Fig. 3: The Converted Map

CHAPTER

TWELVE

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.

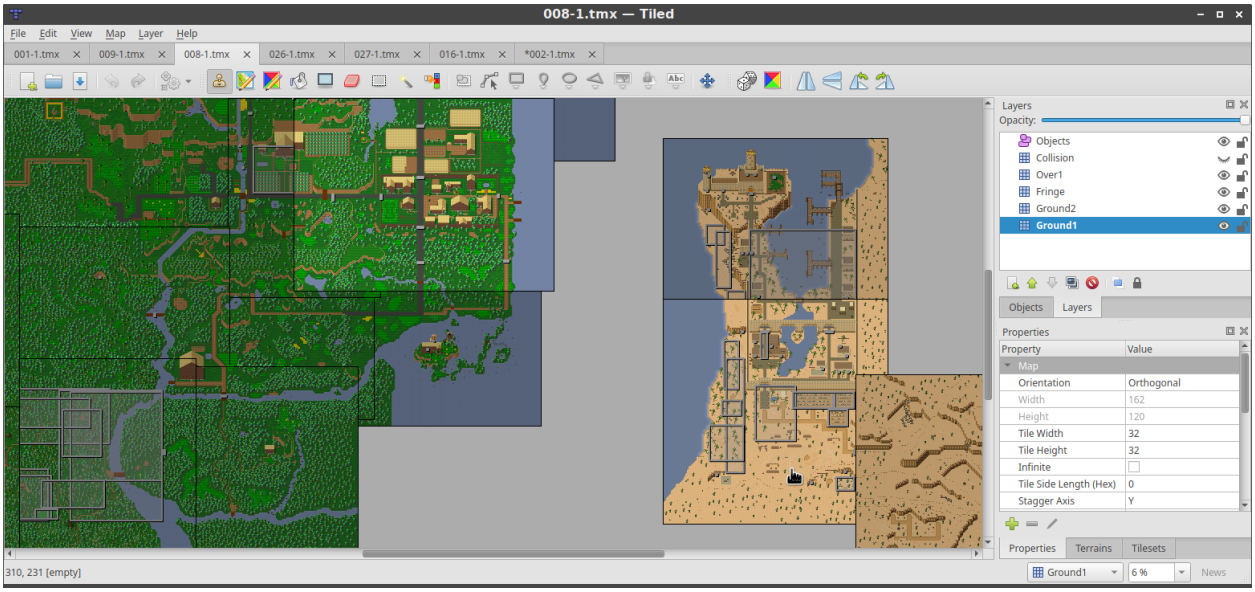


Fig. 1: Many maps from The Mana World shown at once.

12.1 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 *World* > *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
```

(continues on next page)

(continued from previous page)

```
        },
        {
            "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 *World* > *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.

12.2 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 right-clicking in the map editor.

Removing Maps
: Hit the 'Remove the current map from the current world' button on the toolbar. Alternatively, right-click 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 *World* > *Save World* menu. Worlds will also automatically be saved if you launch any external tool that has the 'Save Map Before Executing' option enabled.

12.3 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.

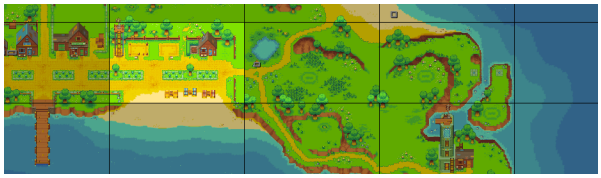


Fig. 2: The island from Alchemic Cutie, using patterns to automatically show each map at the right location.

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

12.4 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, a property could be added to allow specifying a distance around the current map in which other maps are loaded.

CHAPTER

THIRTEEN

USING COMMANDS

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

You may set up 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 set up multiple commands for the same game that load different checkpoints or configurations.

13.1 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.

13.2 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.

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

13.2.1 Substituted Variables

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

%mapfile
: the full path of the current file (either map or tileset).

%mappath
: the path in which the current file is located.

%projectpath
: the path in which the current project is located.

%objectclass
: the class of the currently selected object, if any (also available as `%objecttype` for compatibility with Tiled < 1.9).

%objectid
: the ID of the currently selected object, if any.

%layername
: the name of the currently selected layer.

%tileid
: a comma-separated list with the IDs of the selected tiles, if any.

%worldfile
: the full path of the world the current map is part of, if any.

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

%executablepath
: the path to the executable.

13.3 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

FOURTEEN

AUTOMAPPING

14.1 What is Automapping?

Automapping can automatically place or replace tiles based on rules you define. It looks for tiles in your working map that match each rule's input, and if it finds any, it'll place the corresponding output. This enables complex or repetitive tile placement to be entirely automated, which can make decorating your levels much faster, and can help you automatically correct mistakes.

If your tiles are set up to work as corners and edges of shapes, you may want to look into using *Terrains* instead. Terrains provide a more convenient way to automate placement of such tiles.

Automapping can be applied manually via *Map* > *AutoMap*, or dynamically as you draw on the map if you enable *Map* > *AutoMap While Drawing*.

Note

Automapping changed significantly in Tiled 1.9. It's 10-30x faster and setting up rules is more intuitive, but it behaves differently from the old system in some ways. Old rules should still behave the same, but you may want to take a look at the *section on updating your rules*. If you need help understanding your old rules, the old documentation is available on GitHub.

If you are making new rules, make sure you *do not* have any `regions` layers. These will enable the old Automapping system, and the rules will likely not behave as you intend.

14.2 Setting Up the Rules File

Automapping rules are defined in regular map files, which we'll call **rule maps**. These files are then referenced by a text file, usually called `rules.txt`. The `rules.txt` can list any number of rule maps, in the order in which their rules should be applied.

There are two ways to make the rule maps defined in the `rules.txt` apply to a map:

- Since Tiled 1.4 Open *Project* > *Project Properties* and set the "Automapping rules" property to the `rules.txt` file that you created in your project. If you have only a single rule map, you can also refer to that map file directly.
- Alternatively, you can save your `rules.txt` in the same directory as the map files to which you want the rules to apply. This can also be used to override the project-wide rules for a certain set of maps.

Each line in the `rules.txt` file is either:

- A path to a **rule map**.
- A path to another `.txt` file which has the same syntax (e.g. in another directory).
- Since Tiled 1.9 A map filename filter, enclosed in `[]` and using `*` as a wildcard.

- A comment, when the line starts with # or //.

By default, all Automapping rules will run on any map you Automap. The map filename filters let you restrict which maps rules apply to. For example, any rule maps listed after `[town*]` will only apply to maps whose filenames start with "town". To start applying rules to all maps again, you can use `[*]`, which will match any map name.

14.3 Setting Up a Rule Map

A **rule map** is a standard map file, which can be read and written by Tiled (usually in TMX or TMJ format). A rule map can define any number of rules. At a minimum, a rule map contains:

- One or more `input` layers, describing which kind of pattern(s) the working map will be searched for.
- One or more `output` layers, describing how the working map is changed when an input pattern is found.

In addition, custom properties on the rule map, its layers and on objects can be used to fine-tune the overall behavior or the behavior of specific rules.

Every contiguous region of tiles on the `input` and `output` layers is a rule. Tiles are considered contiguous if they're next to each other vertically, horizontally, or diagonally (8-way connectivity). You can include many rules in one map, as long as you leave space between them. By default all the rules will match simultaneously, and apply their outputs in order from top to bottom, left to right - rules with smaller Y value come first, and if there are rules at the same Y value, then the rules with smaller X come first. If you want the rules to match in order and take previous rules' output into account, you can use the ***MatchInOrder*** map property.

14.3.1 Defining Inputs

The `input` layers define the pattern(s) of tiles that a rule will look for. These are Tile Layers, and their names must follow this scheme:

```
input[not][index]_name
```

After the first underscore there will be the **name** of the target input layer. For example, `input_Ground` will look for tiles on a layer called *Ground*. The input layer name can include more underscores, so `input_test_case` will look for tiles on a layer called *test_case*. If the working map includes multiple layers by this name, the bottom-most one will be used. If the working map does not contain the named target layer, the rule checks against a dummy empty layer.

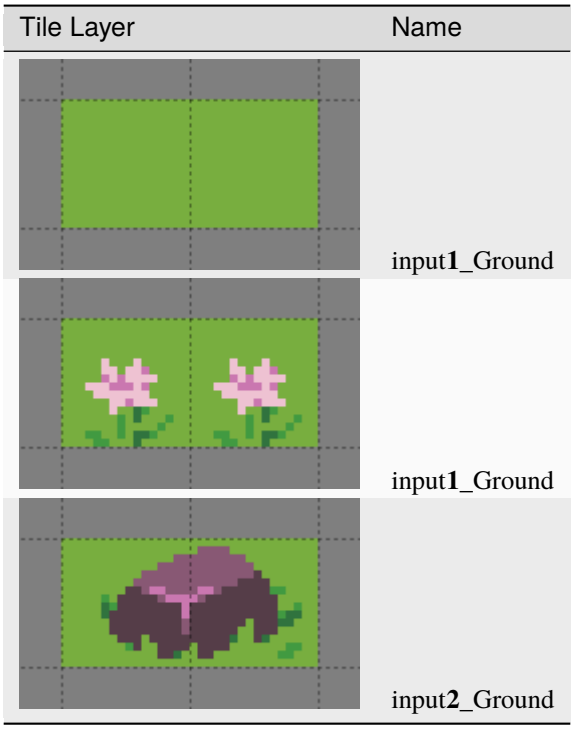
The **not** is optional. If present, it inverts the layer's meaning, so instead of matching the tiles on the layer, Tiled will match anything but those tiles.

The **index** is optional. Indices on `input` layers allow you to create rules that match any of several completely separate inputs. Any inputs with the same index are treated as part of the same condition, and each different index is its own independent set of conditions. Any of these conditions being matched will count as a match for the rule. An index can be empty, or it can be any string that doesn't start with `not` and doesn't contain any underscores.

Multiple input layers having the same name and index is explicitly allowed and is intended. Having multiple input layers of the same name and index allows you to define different possible tiles per coordinate as input, and any combination of those tiles will count as a match.

Input Example

Let's say you want to match two-tile areas of the ground, perhaps to randomise them. You might want to match any combination of grass and flower tiles, but only whole two-tile rocks. You can achieve this like so:



The first two layers both have the index 1, so Automapping will match any combination of those grass and flower tiles. The last layer has the index 2, so its tiles are checked separately. This means these inputs will match any part of the Ground layer that looks like any of these:

Since Tiled 1.9

Matching Special Cases

In some cases, your tiles alone aren't enough to define the scenario you want to match. Tiled provides a built-in "Automapping Rules Tileset" to handle certain special cases, which can be added to your rule map through *Map > Add Automapping Rules Tileset*.

Empty
: This tile matches any empty cell. If used on an `output` layer, this tile will output an empty tile, allowing you to erase tiles with Automapping.

Ignore
: This tile does not affect the rule in any way. Its only function is to allow connecting otherwise disconnected parts into a single rule, but it can also be used for clarity.

NonEmpty
: This tile matches any non-empty cell.

Other
: This tile matches any cell, which contains a tile that is *different* from all the tiles used by the current rule targeting the same input layer. This includes empty cells, unless the Empty tile is explicitly used elsewhere by the rule (since Tiled 1.10).

Negate
: This tile negates the condition at a specific location, making other `input` layers with the same target layer name act like inputnot and vice versa, but only in that location, which can simplify your rules in some cases.

The meaning of these tiles is derived from their custom **MatchType** property. This means that you can set up your own tiles for matching these special cases as well!

14.3.2 Defining Outputs

The `output` layers define what will be output when the input of the rule matches something in the working map. These can be Tile or Object Layers, and their names must follow this scheme, which is similar to that of `input` layer names:

```
output[index]_name
```

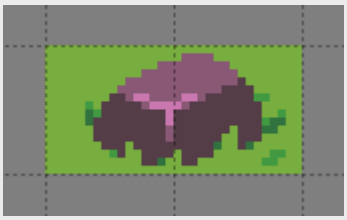
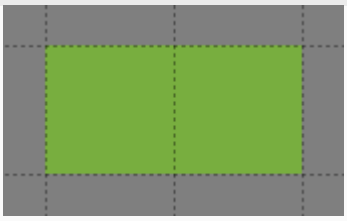
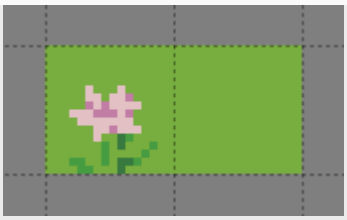
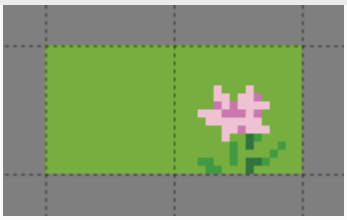
Everything after the first underscore is the **name**, which determines which layer in the working map the tiles or objects will be placed on. If the working map includes multiple layers by this name, the bottom-most one will be used. If the rule matches and the working map does not already contain the named output layer, Automapping will create the layer.

The **index** is optional, and is not related to the input indices. Instead, output indices are used to randomize the output: every time the rule finds a match, a random output index is chosen and only the output layers with that index will have their contents placed into the working map.

New in Tiled 1.11 For convenience, Tiled 1.11 introduced two changes to the behavior related to indexes. If an output index is completely empty for a given rule, it will never be chosen for that rule. This is useful when some rules have more random options than others. Also, when no index is specified, that part of the rule's output will always apply when the rule matches. This can be used to combine an unconditional part of a rule's output with a random part.

Random Output Example

Continuing with the example from before, you can use output layers like these to randomise the Ground layer:

Tile Layer	Name
	output**1**_Ground
	output**2**_Ground
	output**3**_Ground
	output**4**_Ground

By default, the output of a rule is allowed to overlap previous output from the same rule, which isn't always what you want. In the example above, the output rocks can be partially overwritten by subsequent outputs from that rule. You can set the ***NoOverlappingOutput*** map property to `true` to avoid this. This will only apply to rules overlapping their own output, however - outputs from different rules will still be allowed to overlap. If you want to avoid any kind of overlap, you will need to design your inputs such that your inputs are specific enough for different rules to not overlap.

Sometimes, you may want certain outputs to appear more or less frequently than others. The above example would look much nicer if the flowers and rocks didn't appear quite so often. You can control the probability of an output index by setting the ***Probability*** layer property on one of the layers for that index.

Warning

While Automapping can output Objects, there are some caveats when it comes to detecting whether they're part of a given rule's output:

- Object rotation is not taken into account.
- Tile Objects' Object Alignment is not taken into account.
- Ellipse and Text Objects use their bounding rectangles.
- Point positions are checked *exclusively*, a Point must be within a given cell to count as part of it, merely touching the cell is not enough.

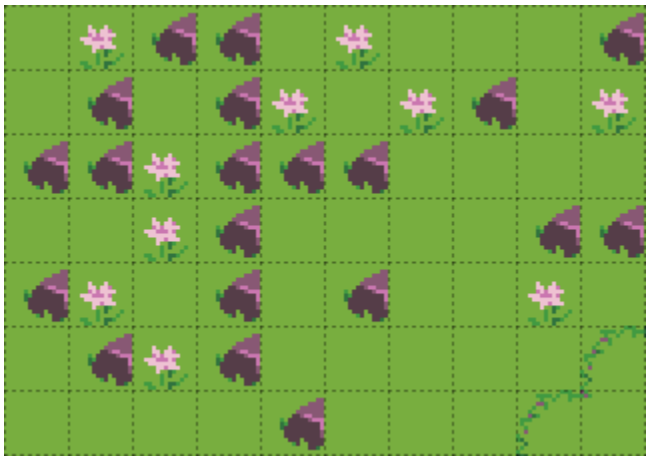


Fig. 1: Because outputs are allowed to overlap each other and the inputs aren't very specific, the two-tile rock outputs are overlapped by subsequent outputs.

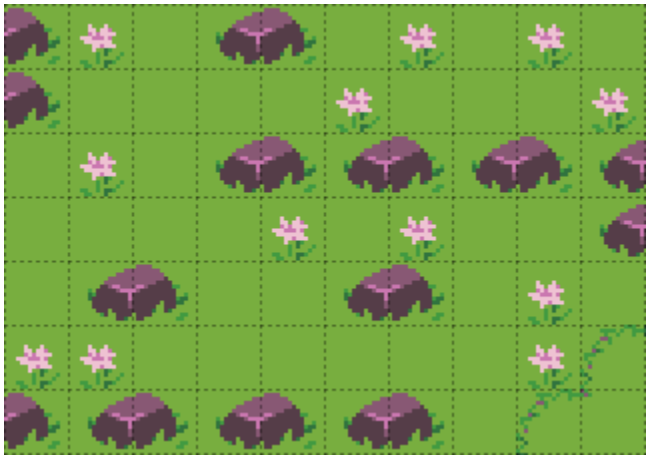


Fig. 2: With **NoOverlappingOutput** set to `true`, outputs don't overlap and all the rocks are whole.

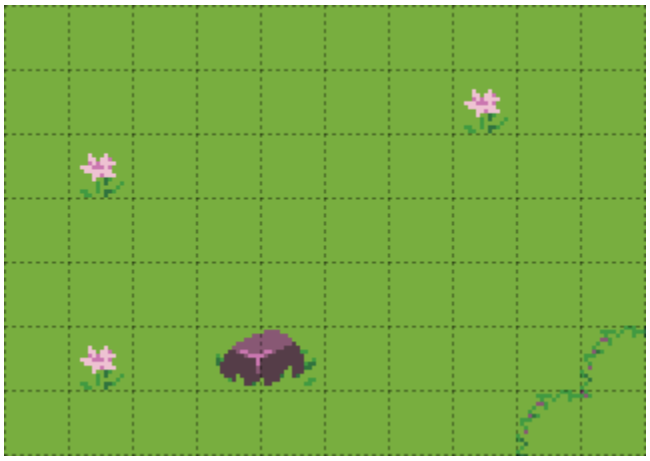


Fig. 3: Setting the **Probability** of the grass output to 20 and the **Probability** of the rock output to 0.5 produces much nicer-looking results.

- Polygons and Polylines are checked as if they were Points at their position, the rest of the shape is not taken into account.

You can ensure these Objects are output by putting Ignore *special tiles* in a tile output layer at their position. You may also need to connect this tile to the rest of the rule with more Ignore tiles to make sure it isn't treated as a separate rule.

Any custom properties set on an output layer (other than **Probability**) will be copied to the target layer when the output is applied. You should normally not need to add any such properties to output layers, but this can be a way to automate setting properties on your layers based on their contents.

14.4 Automapping Properties

The behavior of your rules can be modified by properties on the rules map, input and output layers, and on a per-rule basis using objects.

14.4.1 Map Properties

DeleteTiles
: This is a boolean map property. When this property is `true`, the area covered by tiles in input layers is erased from the output layers before applying the rules. This property is mostly provided for backwards compatibility, because since Tiled 1.9 tiles can be erased by outputting the Empty *special tile*, which is more explicit and more flexible.

  Despite the name, this property affects output Object Layers too, deleting any Objects that fully or partially overlap the erased region. This is currently the only way to delete Objects via Automapping.

> **Warning**
>
> All the caveats of outputting objects also apply when deleting them, see the *warning in the Defining Outputs section*.

AutomappingRadius
: This map property is a number: 1, 2, 3 … When using Automap While Drawing, this property determines how far beyond the tiles affected by your changes Automapping will look for matches.

MatchOutsideMap Since Tiled 1.2
: This boolean 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 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 behaved as if this property was `false`.

OverflowBorder Since Tiled 1.3
: This boolean map property customizes the behavior of 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 Since Tiled 1.3
: This boolean map property customizes the behavior of 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**.

MatchInOrder Since Tiled 1.9
When this boolean map property is set to `true`, each rule is applied immediately after a match is found. This disables concurrent matching of rules, but allows each rule to take into account the output of the previously applied rules (as used to be the case before Tiled 1.9).

Alternatively, you can split up your rules over multiple rule maps. Rule maps are always applied in order, so each rule map can rely on any modifications applied by previous rule maps.

14.4.2 Layer Properties

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

AutoEmpty (alias: StrictEmpty)
This boolean layer property can be added to `input` and `inputnot` layers to customize the behavior for empty tiles within a rule.

Normally, empty tiles are simply ignored. When **AutoEmpty** is `true`, empty tiles within the input region match empty tiles in the target layer. This can only happen when you have multiple input/inputnot layers and some of the tiles that are part of the same rule are empty while others are not. Usually, using the Empty *special tile* is the best way to specify an empty tile, but this property is useful when you have multiple input layers, some of which need to match many empty tiles. Note that the input region is defined by *all* input layers, regardless of index.

IgnoreHorizontalFlip New in Tiled 1.11
This boolean layer property can be added to `input` and `inputnot` layers to also match horizontally flipped versions of the input tile.

IgnoreVerticalFlip
This boolean layer property can be added to `input` and `inputnot` layers to also match vertically flipped versions of the input tile.

IgnoreDiagonalFlip
This boolean layer property can be added to `input` and `inputnot` layers to also match anti-diagonally flipped versions of the input tile. This kind of flip is used for 90-degree rotation of tiles.

IgnoreHexRotate120
This boolean layer property can be added to `input` and `inputnot` layers to also match 120-degree rotated tiles on hexagonal maps. However, note that Automapping currently does not really work for hexagonal maps since it does not take into account the staggered axis.

Probability Since Tiled 1.10
This float layer property can be added to `output` layers to control the probability that a given output index will be chosen. The probabilities for each output index are relative to one another, and default to 1.0. For example, if you have outputA with probability 2 and outputB with probability 0.5, A will be chosen four times as often as B. If multiple output layers with the same index have their **Probability** set, the last (top-most) layer's probability will be used.

Since Tiled 1.9

14.4.3 Object Properties

A number of options can be set on individual rules, even within the same rule map. To do this, add an Object Layer to your rule map called `rule_options`. On this layer, you can create plain rectangle objects and any options you set on these objects will apply to all rules they contain.

The following options are supported per-rule:

ModX
: Only apply a rule every N tiles on the X axis (defaults to 1).

ModY
: Only apply a rule every N tiles on the Y axis (defaults to 1).

OffsetX
: An offset applied in combination with ModX (defaults to 0).

OffsetY
: An offset applied in combination with ModY (defaults to 0).

Probability
: The chance that a rule applies at all, even if its input layers would have matched, from 0 to 1. A value of 0 effectively disables the rule, whereas a value of 1 (the default) means it is never skipped.

Disabled
: A convenient way to (temporarily) disable some rules (defaults to `false`).

NoOverlappingOutput
: When set to `true`, the output of a rule is not allowed to overlap other outputs of the same rule (defaults to `false`).

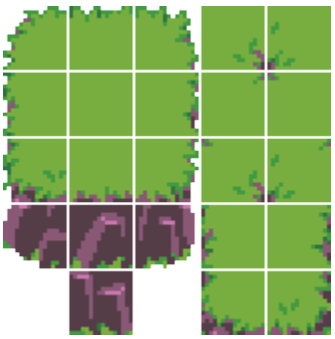
IgnoreLock Since Tiled 1.10
: Since Tiled 1.10, Automapping rules no longer modify locked layers. Set this property to `true` to ignore the lock. This can be useful when you have layers that are only changed by rules and want to keep them locked.

All these options can also be set on the rule map itself, in which case they apply as defaults for all rules, which can then be overridden for specific rules by placing rectangle objects.

14.5 Examples

14.5.1 RPG Cliffs

A common Automapping scenario is to automate the placement of cliff sides. Tilesets will often include cliff tiles like this:



Terrains can be used to place the top of the cliff, but they cannot reliably add the vertical cliffs themselves. Fortunately, they are no problem for Automapping.

The bottom side and bottom corners of the cliff are the only ones that need cliff tiles in this tileset, so only three rules are needed to add those. The rules are shown below, layer by layer.

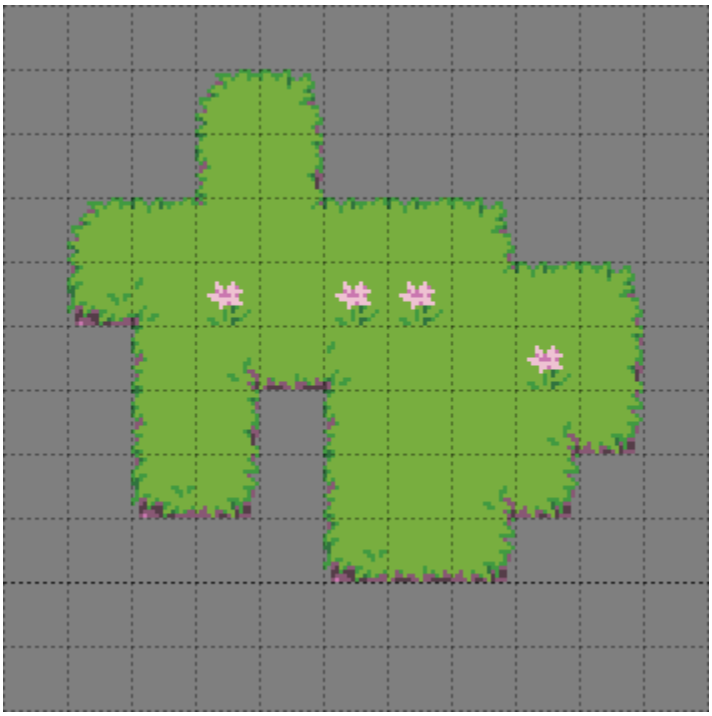


Fig. 4: The starting map: the flat top of a cliff painted using Terrains.

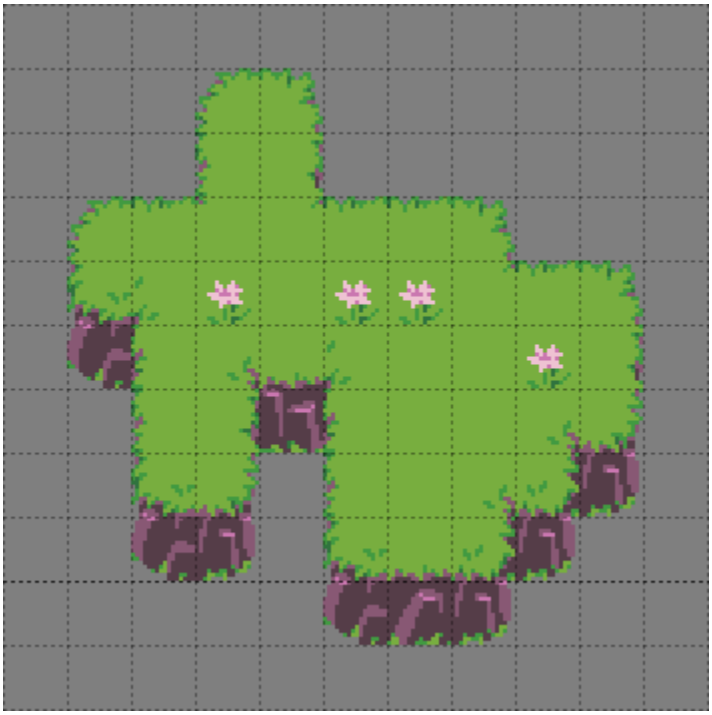
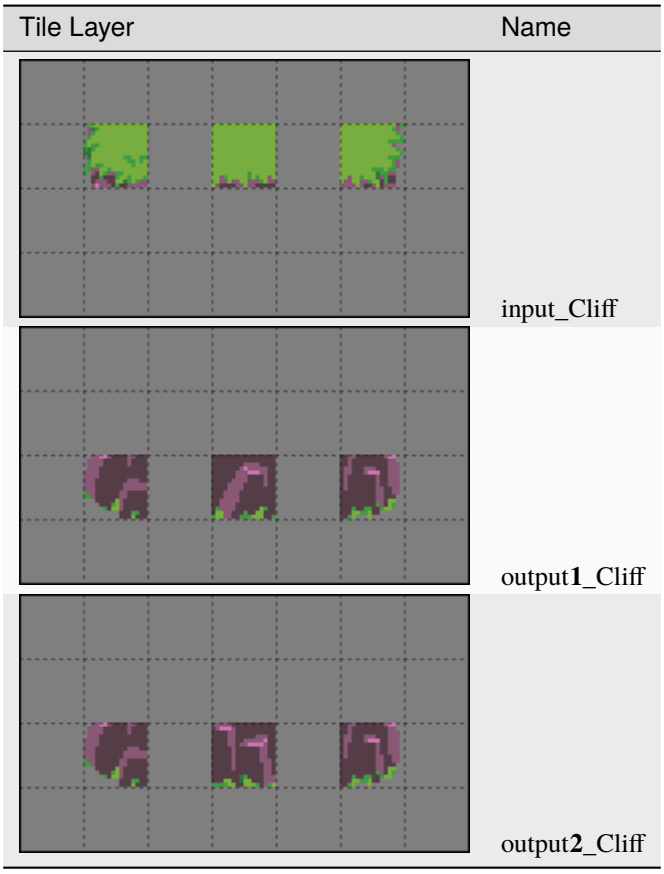


Fig. 5: Automapping can add the appropriate cliff tiles.



The two output layers differ only in which tile is output by the middle rule, the two corner output tiles are the same in both cases. These three rules get us most of the way there, but there are still some small issues:

This tileset includes tiles for the sides and bottom corners of the cliff top when they're next to a cliff, so you can make another rule map to place those. Since there are left and right side tiles and left and right corner tiles, you will need four rules.

You could create rules that check for a literal cliff tiles next to these tiles, but that would require enumerating every tile that counts as a cliff - all the cosmetic variants of straight cliff section, the cliff corners, and if you're not careful, you might still miss some edge cases like two cliff sides facing each other. A simpler approach would be check whether the tile above this side or corner is a concave corner tile: if it is, then you know that the tile next to it will be something with a cliff.

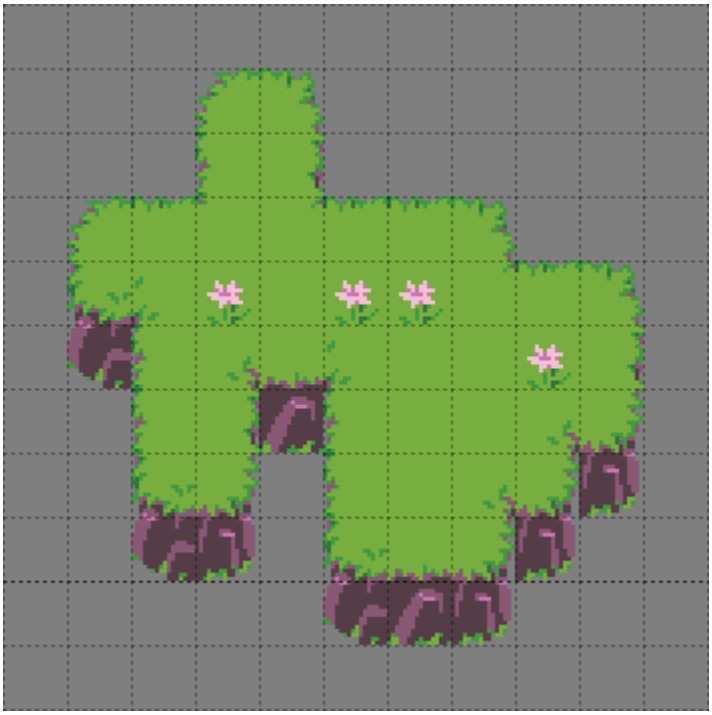


Fig. 6: The result of the rules above.

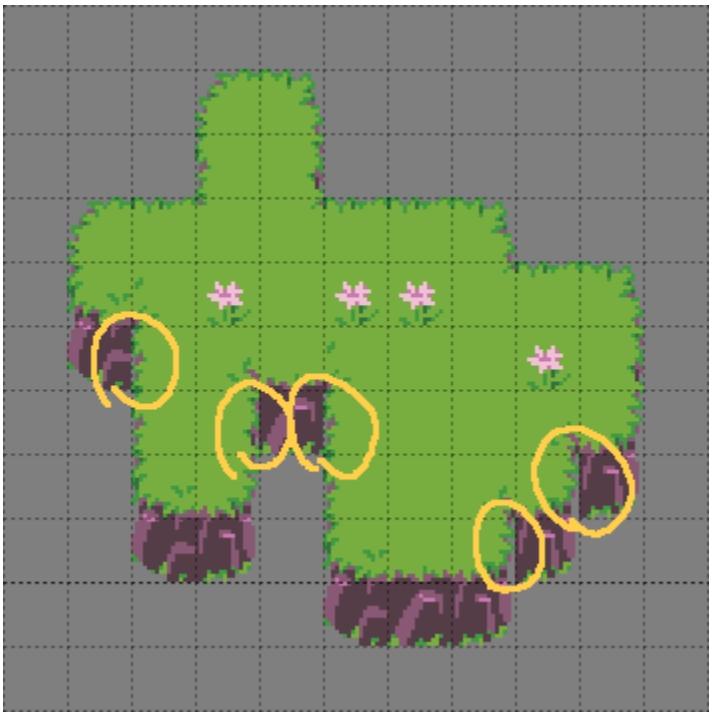
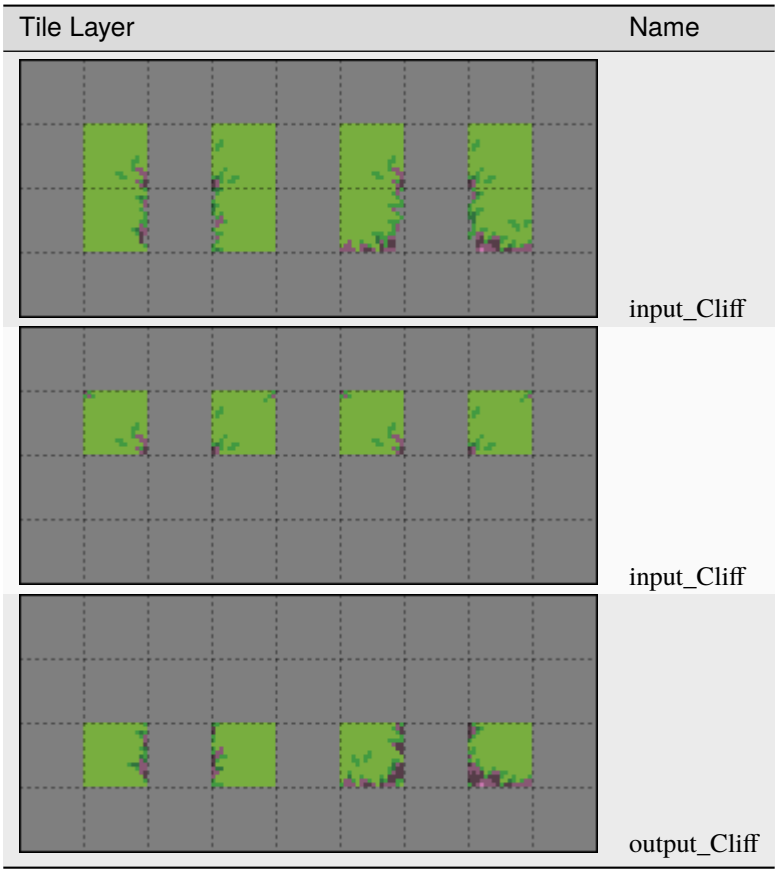


Fig. 7: The bottom corners and sides of the cliff are circled here because they should use different tiles when they're next to a cliff tile.



There is no need to repeat the side and corner tiles on the second "input_Cliff" layer, you can leave those cells empty and only include the extra input tiles that you need.

With these additional rules in place, you should get the result shown at the top of this section: all the cliffs in place, with no transparent holes where sides and corners meet the cliffs.

Since these rules work with a layer called "Cliff", they will not affect cliffs drawn on any other layers. If you want to automap cliffs on several different layers, which may be necessary if you want stacks of cliffs, you'll need to duplicate the rule map and adjust the input and output layer names.

Automap While Drawing

The rules above work well if you draw your cliff tops with Terrains and then manually trigger Automapping, but what if you want to see the cliffs appear as you draw with Terrains, or want to keep drawing with Terrains after automapping manually?

Fig. 8: Without some extra rules, Automap While Drawing can produce messy results.

For this, your rules will need to take into account tiles that may have previously been placed by Automapping.

Hint

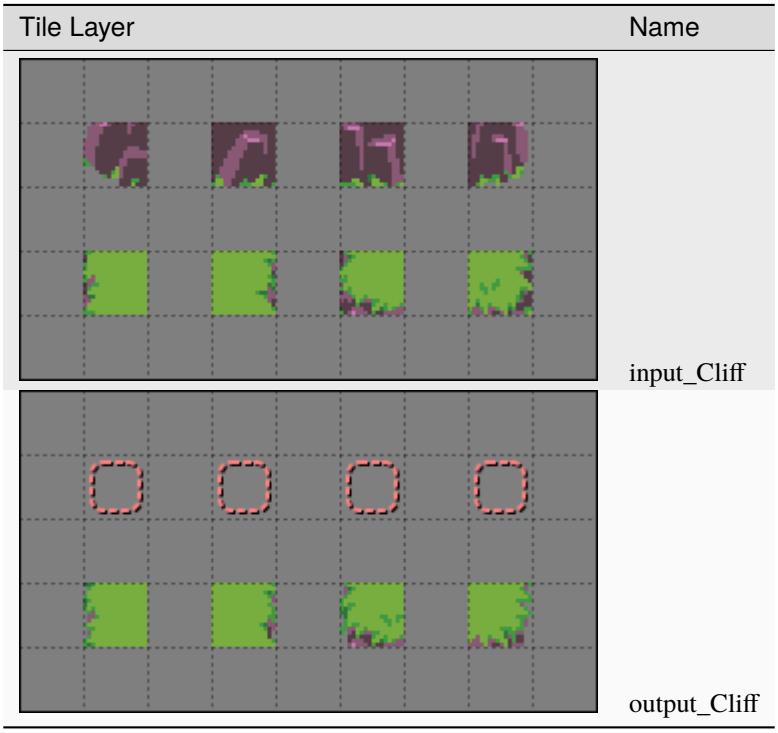
If you're using Automapping While Drawing with Terrains, it also helps for your Terrains to be aware of the tiles that may be output to that same layer by Automapping. In this example, this would mean labeling the side and corner tiles meant to be next to cliffs with the same Terrain labels as their base versions.

This will have the side effect of making Terrains randomly output those tiles where they're not needed, but this can be remedied by setting the tile probability of those tiles to 0 in the Tileset Editor. If you *always* use those Terrains with Automapping, you can also just let Automapping fix the tiles.

There are two approaches you can take to make your Automapping rules take its own output into account:

- Include those tiles as alternate inputs in all the rules, or
- Make another set of rules to reset all the alternate tiles to a uniform condition.

The appropriate option will depend on your specific rules. In this case, the latter is simpler: all you have to do is erase any cliff tiles, and replace the variants meant to be placed next to cliffs with their basic versions. For this purpose, you should create another rule map, and place it *before* the other rules in your `rules.txt`, so that it can prepare the map for those other rules. The actual rules are just simple substitutions:



The output tiles in the top row are the Empty *special tile*, which means the output will erase those tiles.

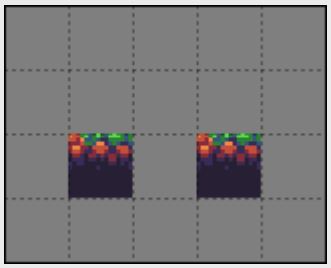
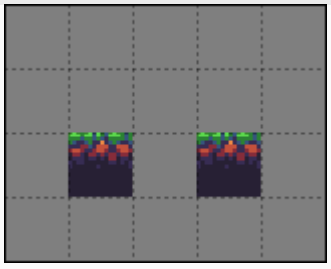
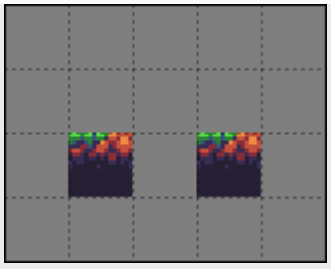
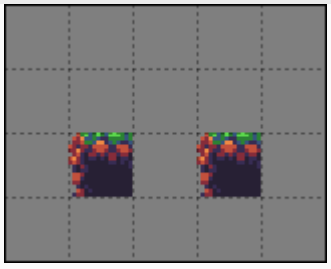
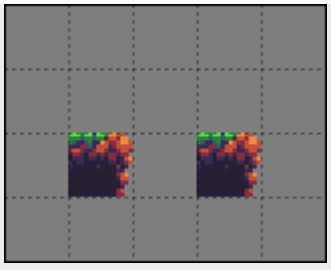
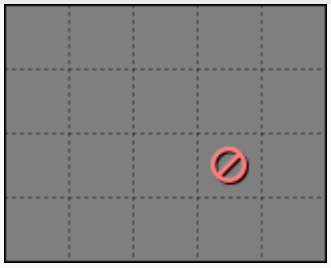
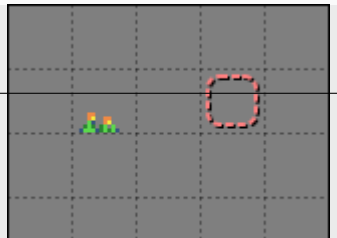
For Automap While Drawing to work correctly, you may also need to increase the ***AutomappingRadius*** property of your rules maps. This is because some of the rules may look only at tiles *near* the ones you change by drawing, such as the rules that erase cliff tiles. In this example, you will probably need to set the **AutomappingRadius** to 1 on the reset rules and on the rules that add cliffs.

Fig. 9: Now, Automap While Drawing produces correct results.

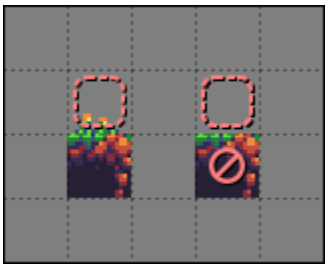
14.5.2 Sidescroller Details

You can use Automapping to add various details to your maps. This small example shows adding foreground details to a sidescroller platforms. This tileset features a number of platform tiles, some of which have rocky tops, and some of which have grassy tops. These two rules will add random grass and flower decorations to a different layer corresponding

to the grassy-topped tiles, and delete any decorations that end up on top of non-grassy tiles. There are many input layers, because there are many grassy-topped tiles to check.

Tile Layer	Name
	input_Platform
	input_Platform
	input_Platform
	input_Platform
	input_Platform
	input_Platform
	

The inputs for these rules are identical except for the last input layer, in which the second rule, which deletes the foreground detail tiles, has the Negate *special tile*. This makes all those `input` layers act like `inputnot` layers, but only in that specific location. This means the first rule matches whenever it encounters any of those grassy-topped tiles, while the second rule matches whenever it encounters *anything other* than those grassy-topped tiles. The second rule could've also been made with a bunch of `inputnot` layers instead, but using the Negate tile reduces how many layers this rule map needs, and it's easier to see that the input tiles are negated when the layers are all viewed together:



The three outputs select a random foreground detail for the first rule, and are all Empty for the second rule. One of the outputs for the first rule is also Empty, just for extra variety.

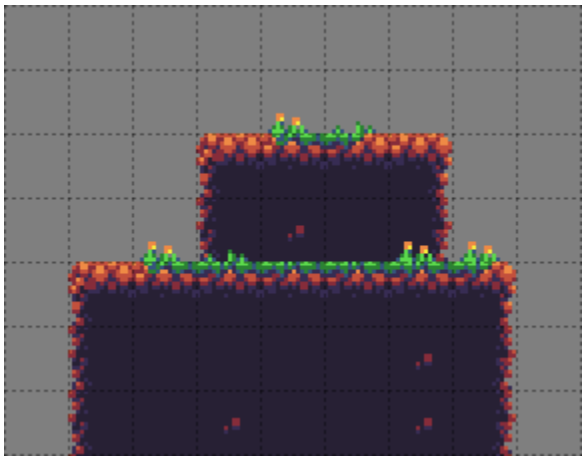


Fig. 10: A result from the two rules above.

14.6 Updating Legacy Rules

If you have some Automapping rules from before Tiled 1.9, they should still work much as they always did in most cases. When Tiled sees that a rule map contains `regions` layers, it will automatically bring back the old behavior - rules will be matched in order by default, cells within input regions that are empty in all the input layers for a given layer and index will be treated as "Other", and completely empty output indices will still be selected as valid outputs.

> **Warning**
>
> In Tiled 1.9.x, the presence of `regions` layers did not imply **MatchInOrder**. If you're using 1.9.x rather than 1.10+ and want to use legacy rules, you'll need to set the **MatchInOrder** map property to `true`.

If you'd like to instead update your rules to not rely on any legacy behavior, that can be as simple as deleting your `regions` layer(s), or it might take some extra work, depending on how exactly your rules are set up:

- If your rules need to take the output of other rules in the same rules map into account, set the ***MatchInOrder*** map property to `true`.

- When deleting your `regions` layers, make sure you weren't relying on them to connect otherwise disconnected areas of tiles. If you were, use the Ignore *special tile* to connect them on one of the `input` layers, so that Tiled knows they're part of the same rule. To make sure the rules behave exactly the same, fill in any part that was previously part of the input region.
- If were using the ***StrictEmpty*** map property to look for empty input tiles, you should now use the Empty special tile instead in the cells you want to check for being empty. You can also continue use the **StrictEmpty** property (or its newer alias, **AutoEmpty**), as long as at least one other input layer is not empty at those locations.
- If were relying on the behavior that any tile which is left empty on all of the input layers for a given index is treated as "any tile not in this rule", you should instead use the Other *special tile* at those locations, and also the Empty *special tile* on an inputnot layer at those same locations. The Empty tile is needed because old-style Other never matched Empty, but the MatchType Other tile does match Empty.
- If you have rules that rely on some output indices being empty to randomly not make any changes, you will need to place ***Ignore*** *special tiles* in at least one layer of each empty output index so that those indices aren't ignored. Alternatively, you can use `rule_options` to give those rules a chance to not run at all.
- If you had rules with random output, but did not specify an index for one of the outputs, this part of the rule's output is now excluded from the options and applied unconditionally instead. If all outputs should be random options, make sure they all have an index. You can automate updating your existing rule maps with the "Add Output Index" script.

14.7 Credits

The *Sidescroller Details* example uses art from A platformer in the forest by Buch.

CHAPTER

FIFTEEN

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, as well as *exporting a map to an image*.

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` and `--export-tileset` command-line parameters.

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

15.1 Generic File Formats

Tiled supports exporting to several generic file formats which are not targeting any specific framework.

15.1.1 JSON

The JSON format is the 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.

15.1.2 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), Solar2D (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 *object templates*.

The tiles are referenced using *Global Tile IDs*, as done in the *TMX* and *JSON* formats.

15.1.3 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`.

When tiles are flipped horizontally, vertically or diagonally, these states are exported using bitflags in the ID, in the same way as done in the *TMX Map Format*.

15.2 Defold

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

15.2.1 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.

Custom Properties

The `tile_set` property of the Tile Map can be set by adding a custom string property to the map named "tile_set" (case sensitive). If left empty, it will need to be set up in Defold after each export.

A custom float property named "z" can be added to set the `z` value for each tile layer. By default, the layers will be exported with incrementing z values, so you only need to set this property in case you need to customize the rendering order.

15.2.2 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

The plugin automatically assigns a Z-index to each layer ranging between 0 and 0.1. It 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.

Custom Properties

- The `tile_set` property of each tilemap 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.

  Alternatively, a custom string property called "tilesource" (case-sensitive) can be added to the *tileset*, which will then be used instead (since Tiled 1.9.2).

- 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*.
- A custom float property named "z" can be added to tile layers to manually specify their `z` value.

15.3 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 class 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.

15.3.1 Object Instances

GameMaker object instances are created by putting the object name in the "Class" 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.

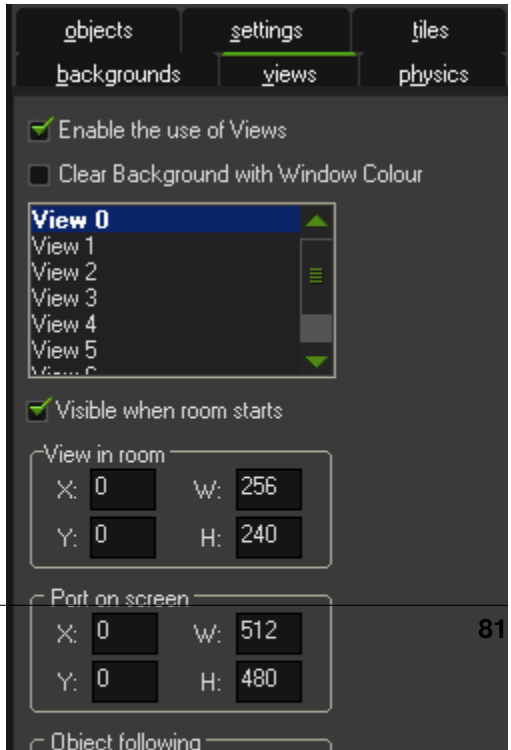
The `originX` and `originY` properties can be used to tell Tiled about the origin of the object defined in GameMaker, as an offset from the top-left. This origin is taken into account when determining the position of the exported instance.

Hint

Of course setting the class 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 class set on the tile, and in Tiled 1.1 you can also use *object templates*.

15.3.2 Views

Views can be defined using *rectangle objects* where the Class 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 class in the *Custom 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.

15.3.3 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)

15.3.4 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)

15.4 GameMaker

GameMaker uses a JSON-based format to store its rooms, and Tiled ships with a plugin to export maps in this format.

This plugin will do its best to export the map as accurately as possible, mapping Tiled's various features to the matching GameMaker features. *Tile layers* get exported as tile layers when possible, but will fall back to asset layers if necessary.

Objects can get exported as instances, but also as tile graphics, sprite graphics or views. *Image layers* get exported as background layers.

Warning

Since it's not possible to add a room to a project by selecting a .yy file, the easiest way to export a Tiled map to your GameMaker project is to create a new room in GameMaker and then overwrite its `room.yy` file when exporting from Tiled.

In 2024, GameMaker made minor but incompatible changes to its file format. Tiled 1.11.2 ships with an updated plugin that uses the new format.

15.4.1 References to Existing Assets

Since Tiled currently only exports a map as a GameMaker room, any sprites, tilesets and objects used by the map are expected to be already available in the GameMaker project.

For sprites, the sprite name is derived by looking for a `*.yy` file in the directory of the image file and up to two parent directories. If such a file is found, it is assumed to be the associated meta file and its name without the file extension is used. If no `*.yy` file can be found, the name of the image file without its file extension is used.

If necessary, the sprite name can be explicitly specified using a custom `sprite` property (supported on tilesets, tiles from image collection tilesets and image layers).

For tilesets, the tileset name entered in Tiled must match the name of the tileset asset in GameMaker.

For object instances, the name of the object should be set in the *Class* field.

15.4.2 Exporting a Tiled Map

A Tiled map contains tile layers, object layers, image layers and group layers. All these layer types are supported.

Tile Layers

When possible, a tile layer will get exported as a tile layer.

When several tilesets are used on the same layer, the layer gets exported as a group with a child tile layer for each tileset, since GameMaker supports only one tileset per tile layer.

When the tile size of a tileset doesn't match the grid size of the map, or when the map orientation is not orthogonal (for example, isometric or hexagonal), the tiles will get exported to an asset layer instead. This layer type is more flexible, though for tile graphics it does not support rotation.

When the layer includes tiles from a collection of images tileset, these will get exported to an asset layer as sprite graphics.

Object Layers

Object layers in Tiled are very flexible since objects take so many forms. As such the export looks at each object to see how it should be exported to the GameMaker room.

When an object has a *Class*, it is exported as an instance on an instance layer, where the class refers to the name of the object to instantiate. Except, when the class is "view", the object is interpreted as *a view*.

When an object has no Class, but it is a tile object, then it is exported as either a tile graphic or a sprite graphic, depending on whether the tile is from a tileset image or a collection of images.

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

- color `colour` (default: based on layer tint color)

- float `scaleX` (default: derived from tile or 1.0)
- float `scaleY` (default: derived from tile or 1.0)
- bool `inheritItemSettings` (default: false)
- int `originX` (default: 0)
- int `originY` (default: 0)
- bool `ignore` (default: whether the object is hidden)

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 sprite 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 class and/or the above properties manually for each instance will get old fast. Instead you can use tile objects with the class set on the tile or use *object templates*.

Object Instances

The following additional custom properties can be set on objects that are exported as object instances:

- bool `hasCreationCode` (default: false)
- int `imageIndex` (default: 0)
- float `imageSpeed` (default: 1.0)
- int `creationOrder` (default: 0)

The `hasCreationCode` property can be set to true. Refers to "InstanceCreationCode_[inst_name].gml" in the room folder which you can create inside GameMaker itself or with an external text editor.

By default the instance creation order is derived from the object positions inside the layer and object hierarchy from Tiled. This can be changed by using the custom property `creationOrder`. Objects with lower values will be created before objects with higher values (so objects with negative values will be created before objects without a `creationOrder` property).

Additional custom properties that are not documented here can be used to override the variable definitions that got set up inside GameMaker for the object.

Note

As of now only variable definitions of the object itself can be overridden. Overriding variable definitions of parent objects is not supported. As a workaround you can use the creation code to override variables of a parent object.

Tile Graphics

For objects exported as tile graphics (aka GMS 1.4 tiles), it should be noted that rotation is not supported on asset layers.

When 90-degree rotation with grid-alignment suffices, these tiles should be placed on tile layers instead. When free placement with rotation is required, a collection of images tileset should be used, so that the objects can be exported as sprite graphics instead.

Sprite Graphics

The following additional custom properties can be set on objects that are exported as sprite graphics:

- float `headPosition` (default: 0.0)
- float `animationSpeed` (default: 1.0)

Image Layers

Image layers are exported as background layers.

The file name of the source image is assumed to be the same as the name of the corresponding sprite asset. Alternatively the custom property `sprite` can be used to explicitly set the name of the sprite asset.

While not supported visually in Tiled, it is possible to create an image layer without an image but with only a tint color. Such layers will get exported as a background layer with just the color set.

The following custom properties can be set on image layers to affect the exported background layers:

- string `sprite` (default: based on image filename)
- bool `htiled` (default: value of Repeat X property)
- bool `vtiled` (default: value of Repeat Y property)
- bool `stretch` (default: false)
- float `hspeed` (default: 0.0)
- float `vspeed` (default: 0.0)
- float `animationFPS` (default: 15.0)
- int `animationSpeedtype` (default: 0)

Even though the custom properties such as `hspeed` and `vspeed` have no visual effect inside Tiled you will see the effect in the exported room inside GameMaker.

15.4.3 Special Cases and Custom Properties

Rooms

If a `Background Color` is set in the map properties of Tiled an extra background layer with the according color is exported as the bottommost layer.

The following custom properties can be set under *Map -> Map Properties*.

General

- string `parent` (default: “Rooms”)
- bool `inheritLayers` (default: false)
- string `tags` (default: “”)

The `parent` property is used to define the parent folder inside GameMakers asset browser.

The `tags` property is used to assign tags to the room. Multiple tags can be separated by commas.

Room Settings

- bool `inheritRoomSettings` (default: false)
- bool `persistent` (default: false)
- bool `clearDisplayBuffer` (default: true)
- bool `inheritCode` (default: false)
- string `creationCodeFile` (default: "")

The `creationCodeFile` property is used to define the path of an existing creation code file, e.g.: "${project_dir}/rooms/room_name/RoomCreationCode.gml".

Viewports and Cameras

General

- bool `inheritViewSettings` (default: false)
- bool `enableViews` (default: true when any "view" objects were found)
- bool `clearViewBackground` (default: false)

Viewport 0 - Viewport 7

You can configure up to 8 viewports by using view objects (see *Views*).

Physics

- bool `inheritPhysicsSettings` (default: false)
- bool `PhysicsWorld` (default: false)
- float `PhysicsWorldGravityX` (default: 0.0)
- float `PhysicsWorldGravityY` (default: 10.0)
- float `PhysicsWorldPixToMeters` (default: 0.1)

Sprite References

As *mentioned above*, references to sprites generally derive the name of the sprite asset from the image file name. The following property can be set on tilesets, tiles from image collection tilesets and image layers to explicitly specify the sprite name:

- string `sprite` (default: based on image filename)

Paths

> **Warning**
>
> Paths are not supported yet, but it's planned to export polyline and polygon objects as paths on path layers in a future update.

Views

Views can be defined using *rectangle objects* where the *Class* 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:

General

- bool `inherit` (default: false)

Camera Properties

The Camera Properties are automatically derived from the position and size of the view object.

Viewport Properties

- int `xport` (default: 0)
- int `yport` (default: 0)
- int `wport` (default: 1366)
- int `hport` (default: 768)

Object following

- string `objectId`
- 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 class in the *Custom 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.

Layers

All layer types support the following custom properties:

- int `depth` (default: auto-assigned, like in GameMaker)
- bool `visible` (default: derived from layer)
- bool `hierarchyFrozen` (default: layer locked state)
- bool `noExport` (default: false)

The `depth` property can be used to assign a specific depth value to a layer.

The `visible` property can be used to override the "Visible" state of the layer if needed.

The `hierarchyFrozen` property can be used to override the "Locked" state of the layer if needed.

The `noExport` property can be used to suppress exporting of an entire layer, including any child layers. This is useful if you use a layer for annotations (like adding background image or text objects) that you do not want exported to GameMaker. Note that any views defined on this layer will then also get ignored.

15.5 Godot 4

Godot 4 revamped its TileMap node, and Tiled ships with a plugin to export maps in this format. For exporting to Godot 3, see the Tiled To Godot Export extension.

The Godot 4 exporter assumes that the generated `.tscn` files and the tileset artwork all share the same file hierarchy. The exporter will search for a common parent folder containing a `.godot` project file and use that folder as the `res://` root for the project. The exporter will search at least two parent folders for a `.godot` file.

15.5.1 Layer Properties

All layer types support the following custom properties:

- bool `ySortEnabled` (default: false)
- int `zIndex` (default: 0)
- bool `noExport` (default: false)
- bool `tilesetOnly` (default: blank)

The `ySortEnabled` property can be used to change the drawing order to allow sprites to be drawn behind tiles based on their Y coordinate.

The `zIndex` property can be used to assign a specific depth value to a layer.

The `noExport` property can be used to suppress exporting of an entire layer, including any child layers. This is useful if you use a layer for annotations (like adding background image or text objects) that you do not want exported to Godot. Note that any views defined on this layer will then also get ignored.

The `tilesetOnly` property can be used if you want to export all the tilesets used in this layer, without actually exporting the layer itself. By default, the exporter will only export tilesets which are actually used in the map, so this property allows you to export tilesets that normally would otherwise get skipped. This is most useful in combination with the *tilesetResPath property*.

15.5.2 Tileset Properties

Tilesets support the following property:

- bool `exportAlternates` (default: false)

Deprecated: The `exportAlternates` property is necessary when using flipped or rotated tiles in Godot 4.0 and 4.1. This will create 7 alternate tiles for each tile, allowing all flipped and rotation combinations. This has been deprecated in Tiled 1.11 in favour of Godot 4.2's native rotation and flipping support.

15.5.3 Tile Properties

All custom properties set on tiles will get exported as Custom Data Layers of the Godot TileSet resource.

15.5.4 Map Properties

Maps support the following custom property:

- string `tilesetResPath` (default: blank)

The `tilesetResPath` property saves the tileset to an external .tres file, allowing it to be shared between multiple maps more efficiently. This path must be in the form of 'res://<path>.tres'. The tileset file will be overwritten every time the map is exported.

> **Note**
>
> Only tilesets that are used in the current map will be exported. You must ensure that every map which uses the same `.tres` file also uses *all* of the same tilesets. You may wish to create a layer with the `tilesetOnly` property to ensure the correct tilesets are exported.

15.5.5 Object Properties

Objects support the following property:

- string `resPath` (required)

The `resPath` property takes the form of 'res://<object path>.tscn' and must be set to the path of the Godot object you wish to replace the object with. Objects without this property set will not be exported.

15.5.6 Limitations

- The Godot 4 exporter does not currently support collection of images tilesets or image layers.
- Godot's hexagonal maps only support *hex side lengths* that are exactly half the tile height. So if, for example, your tile height is 16, then your hex side length must be 8.
- Godot's hexagonal maps do not support 120° tile rotations.
- Animations frames must strictly go from left-to-right and top-to-bottom, without skipping any frames, and animation frames may not be used for anything else.

15.6 tBIN and tIDE

The tBIN map format is a binary format used by the tIDE Tile Map Editor, while the tIDE map format is an XML-based format used by it as well. 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 or tIDE formats). 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

There is only one plugin, called *tbin*. Despite its name, it enables support for both tBIN and tIDE maps.

Note

The tBIN and tIDE formats support 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.

15.7 Other Formats

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

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)

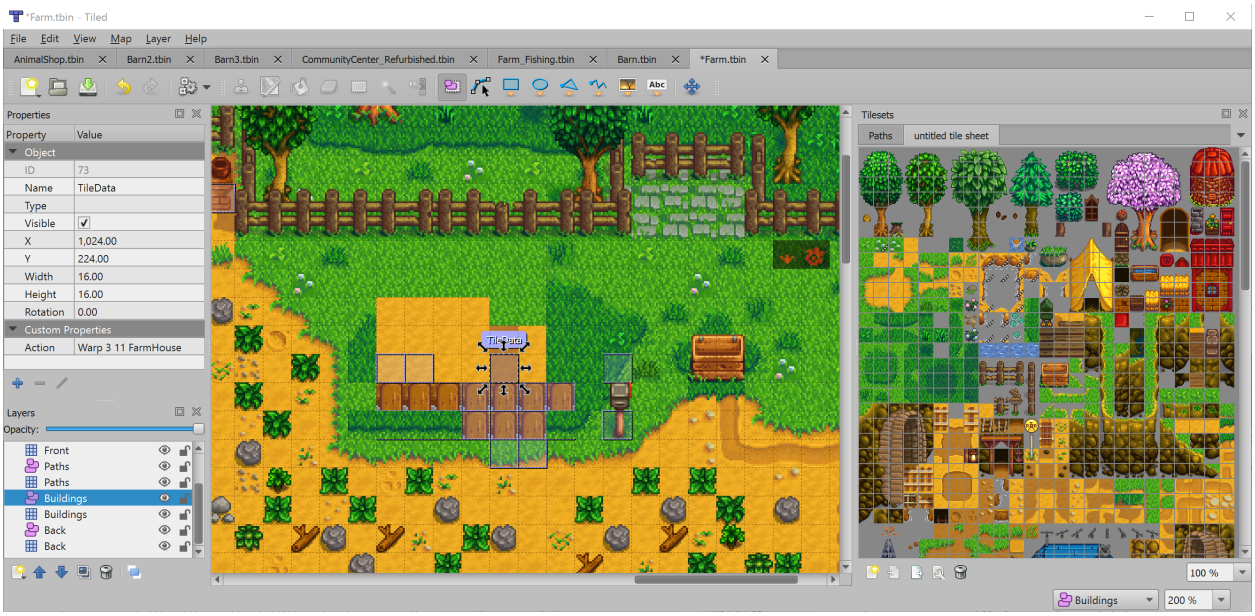


Fig. 1: One of the farm maps from Stardew Valley opened in Tiled.

rpmap
: Adds support for exporting Tiled maps as RpMap (*.rpmap), the format used by MapTool.

  Currently, support is limited to maps using "Image Collection" tilesets since MapTool doesn't support tileset images.

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

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

15.8 Custom Export Formats

Tiled provides several options for extending it with support for additional file formats.

15.8.1 Using JavaScript

Tiled is *extendable using JavaScript* and it is possible to add custom export formats using tiled.registerMapFormat or tiled.registerTilesetFormat.

This is the recommended way to add support for custom map or tileset formats.

15.8.2 Using Python

On some platforms, it is also possible to write *Python scripts* to add support for importing or exporting custom map and tileset formats.

> **Warning**
>
> Python scripting is not supported by the macOS release nor the Tiled snap release for Ubuntu. The plugin is also very specific in the supported Python version. Hence, its use is not recommend.

15.8.3 Using C++

Currently all export options shipping with Tiled are written as C++ Tiled plugins. The API for such plugins is not documented (apart from Doxygen-style comments in the `libtiled` source code), but there are over a dozen examples you can look at.

> **Note**
>
> For binary compatibility reasons, a C++ plugin needs to be compiled for the same platform, by the same compiler and with the same versions of Qt and Tiled that the plugin is supposed to support. Generally, the easiest way to achieve this is by compiling the plugin along with Tiled, which is what all current plugins do. If you write a C++ plugin that could be useful for others, it is recommended you open a pull request to have it shipped with Tiled.

15.9 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 recommended over the Python plugin whenever possible.

Tiled ships with a plugin that enables you to use Python 3 to add support for custom map and tileset formats.

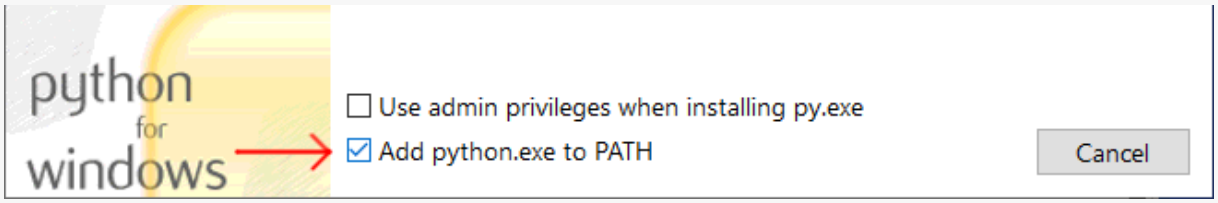
For the scripts to get loaded, they should be placed in `~/.tiled`. Tiled watches this directory for changes, so there is no need to restart Tiled after adding or changing scripts (though the directory needs to exist when you start Tiled).

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**
>
> For the Tiled Python plugin to work you'll need to install a compatible version of Python.
>
> On Windows, get Python from https://www.python.org/. As of Tiled 1.11, the Windows 10+ build requires Python 3.12 whereas the Windows 7-8 build requires Python 3.8. You will also need to check the box "Add python.exe to PATH" in the installer:



On Linux you will need to install the appropriate package. However, currently Linux AppImage builds are done on Ubuntu 22.04 against Python 3.10, and you'd need to install the same version (on Ubuntu likely `libpython3.10` and on Fedora `python3.10-libs`).

The Python plugin is not available for macOS releases, nor in the Ubuntu snap.

15.9.1 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:
```

(continues on next page)

(continued from previous page)

```
                            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.

15.9.2 Tileset Plugins

To write tileset plugins, extend your class from `tiled.TilesetPlugin` instead of `tiled.Plugin`.

15.9.3 Debugging Your Script

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

15.9.4 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.

> **Note**
>
> Any help with maintaining the Python plugin would be very appreciated. See open issues related to Python support

15.10 Export as Image

Maps can be exported as image. Tiled supports most common image formats. Choose *File -> Export as Image...* to open the relevant dialog.

Since exporting a map can in some cases result in a huge image, a *Use current zoom level* option is provided to allow exporting the map at the size it's currently displayed at.

For repeatedly converting a map to an image, manually triggering this export isn't very convenient. For this purpose, a tool called `tmxrasterizer` ships with Tiled, which contrary to its name is able to render any supported map format to an image. It is also able to render *entire worlds* to an image. On Linux this tool can be set up for generating thumbnail previews of maps in the file manager.

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 ...
```

CHAPTER

SIXTEEN

KEYBOARD SHORTCUTS

> **Note**
>
> Most of the below shortcuts can be changed in the *Preferences*.

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

16.1 General

- `Ctrl + N` - Create a new map
- `Ctrl + O` - Open any file or project
- `Ctrl + P` - Open a file in the current project
- `Ctrl + Shift + P` - Search for available actions
- `Ctrl + Shift + T` - Reopen a recently closed file
- `Ctrl + S` - Save current document
- `Ctrl + Shift + S` - Save current document to another file
- `Ctrl + E` - Export current document
- `Ctrl + Shift + E` - Export current document to another file
- `Ctrl + R` - Reload current document
- `Ctrl + W` - Close current document
- `Ctrl + Shift + W` - Close all documents
- `Ctrl + Q` - Quit Tiled
- `Ctrl + MouseWheel` - Zoom in/out on tileset and map
- `Ctrl + Plus/Minus` - Zoom in/out on map
- `Ctrl + 0` - Reset zoom on map
- `Ctrl + /` - Adjust zoom to fit map in view
- `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, objects, properties or layers)
- `Ctrl + G` - Toggle displaying of the tile grid
- `Ctrl + Shift + G` - Go to a tile by its coordinates
- `H` - Toggle highlighting of the current layer
- `Ctrl + M` - Invokes *Automapping*
- `Alt + C` - Copy current position of mouse cursor to clipboard (in tile coordinates)
- `Ctrl + D` - Duplicate selected objects
- `Ctrl + J` - Create a new layer and copy the currently selected objects or tiles to it
- `Ctrl + Shift + J` - Create a new layer and move currently selected objects or tiles to it
- `Ctrl + Shift + D` - Duplicate selected layers
- `F2` - Rename (if applicable in context)
- `Tab` - Hide docks and tool bars
- `Ctrl + PgUp` - Select previous layer (above current layer)
- `Ctrl + PgDown` - Select next layer (below current layer)
- `Ctrl + Shift + Up` - Move selected layers up
- `Ctrl + Shift + Down` - Move selected layers down
- `Ctrl + H` - Show/Hide selected layers
- `Ctrl + L` - Lock/Unlock selected layers
- `Ctrl + Shift + H` - Show/Hide all other layers (only active layer visible / all layers visible)
- `Ctrl + Shift + L` - Lock/Unlock all other layers
- `Ctrl + Tab` / `Alt + Left` - Switch to left document
- `Ctrl + Shift + Tab` / `Alt + Right` - Switch to right document
- `]` - Select next tileset
- `[` - Select previous tileset
- `Ctrl + T` - Force-reload all tilesets used by the current map (mainly useful when not using the automatic reloading)
- `Ctrl + Shift + A` - Clear any object and tile selection

16.2 When a tile layer is selected

- `Right Click on Tile` - Captures the tile under the mouse (drag to capture larger areas).
- `Ctrl + Right Click on Tile` - Selects the layer containing the top-most tile under the mouse.
- `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*
- `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 stamp. When no tile drawing tool is selected, tries to capture the current tile selection (similar to `Ctrl + C`).
- `1-9` - Recall a previously stored tile stamp (similar to `Ctrl + V`)
- `Ctrl + A` - Select the whole layer

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

16.3 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*
- `R` - Activate *Insert Rectangle*
- `I` - Activate *Insert Point*
- `C` - Activate *Insert Ellipse*
- `P` - Activate *Insert Polygon*
  - `Enter` - Finish creating object
  - `Escape` - Cancel creating object
  - `Backspace` - Remove previously added point while creating or extending polygons and polylines
- `T` - Activate *Insert Tile*
- `V` - Activate *Insert Template* (since Tiled 1.1)
- `E` - Activate *Insert Text*

- `Ctrl + A` - Select all objects on the selected layers

16.4 In the Properties dialog

- `Backspace` - Deletes a property

CHAPTER

SEVENTEEN

USER PREFERENCES

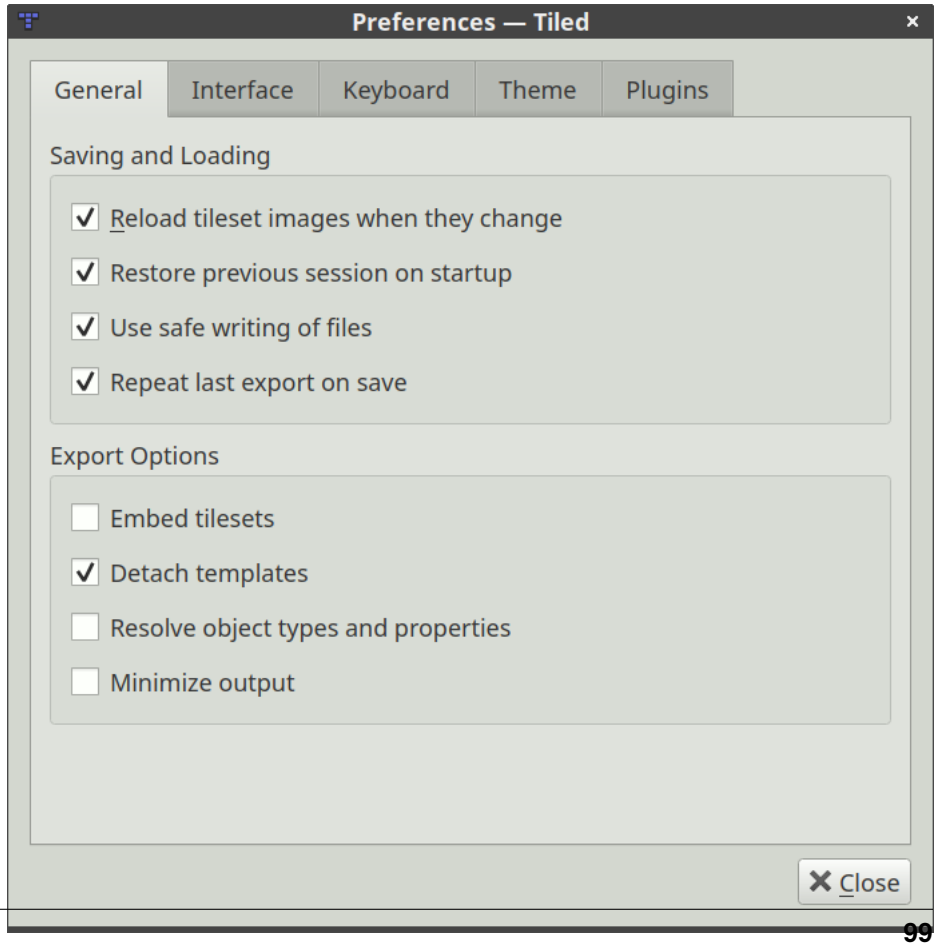
There are only a few options located in the Preferences, accessible though the menu via *Edit* > *Preferences*. Most other options, like whether to draw the grid, what kind of snapping to do or the last used settings when creating a new map are simply remembered persistently.

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`

17.1 General

17.1.1 Saving and Loading



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.

Restore previous session on startup
: When disabled, Tiled always starts with an empty session. This can be useful when you frequently switch projects.

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.

Repeat last export on save
With this feature enabled, any time you save a map or tileset that was previously exported it will automatically be exported again to the same location and format.

17.1.2 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 class and properties with each object. Object properties are inherited from a tile (in case of a tile object) and from the members of their class.

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.

17.2 Interface

17.2.1 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.

Object selection behavior
By default the *Select Objects* tool selects objects from any layer. With this setting, you can make it prefer to select objects from the currently selected layers, or to only pick objects from the selected layers.

When the "Highlight Current Layer" option is enabled, Tiled automatically prefers to select objects from the currently selected layers.

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.

Middle mouse button uses auto-scrolling
With this option enabled, the clicking middle mouse button doesn't drag the map directly but instead controls the speed of a continuous panning movement.

Use smooth scrolling
This option affects the behavior when scrolling with the arrow keys. When disabled, the view scrolls in steps based on key press events. When enabled (the default), the view scrolls continuously while the keys are held down.

17.2.2 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.

17.3 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.

17.4 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.

17.4.1 Custom Interface Font

Normally the application font defaults to the one defined by the system. If you'd like Tiled to use a different font, you can set one here.

17.5 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

EIGHTEEN

SCRIPTING

18.1 Introduction

Tiled can be extended with the use of JavaScript. See the Tiled Scripting API for a reference of all available functionality.

TypeScript definitions of the API are available as the @mapeditor/tiled-api NPM package, which can provide autocompletion in your editor. The API reference is generated based on these definitions.

On startup, Tiled will execute any script files present in *extension folders*. In addition it is possible to run scripts directly from *the console*, as well as to evaluate a script file from the *command-line*. 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

18.1.1 JavaScript Host Environment

Tiled uses the JavaScript engine shipping with Qt's QML module. The QML runtime generally implements the 7th edition of the standard, with some additions. See the JavaScript Host Environment documentation for details.

It may also be helpful to check out the List of JavaScript Objects and Functions that are available.

18.1.2 Scripted Extensions

Extensions can be placed in a system-specific or *project-specific* location.

The system-specific folder can be opened from the Plugins tab in the *Preferences dialog*. The usual location on each supported platform is as follows:

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

The project-specific folder defaults to "extensions", relative to the directory of the `.tiled-project` file, but this can be changed in the *Project Properties*.

> **Warning**
>
> Since Tiled 1.7, project-specific extensions are only enabled by default for new projects you save from Tiled. When opening any other project, a popup will notify you when the project has a scripted extensions directory, allowing you to enable extensions for that project.
>
> Always be careful when enabling extensions on projects you haven't created, since extensions have access to your files and can execute processes.

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

When using the `.mjs` extension, script files are loaded as JavaScript modules. They will then be able to use the import and export statements to split up their functionality over multiple JavaScript files. Such extensions also don't pollute the global scope, avoiding potential name collisions between different extensions.

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.

18.1.3 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.

18.1.4 Command Line

To execute a script (`.js`) or to load a module (`.mjs`) from the command-line, you can pass the `--evaluate` option (or `-e`), followed by the file name. Tiled will quit after executing the script.

The UI will not be instantiated while evaluating scripts on the command-line. This means functions that rely on the UI being present will do nothing and some properties will be `null`. However, scripts are able to load and save maps and tilesets through the available formats (see `tiled.mapFormats` and `tiled.tilesetFormats`), as well as to make any modifications to these assets.

Any additional non-option arguments passed after the script file name are available to the script as `tiled.scriptArguments`.

If you want to evaluate several scripts, use `--evaluate` for each file. Note that evaluating the same JavaScript module (`.mjs`) does not work, since modules are loaded only once.

18.2 API Reference

See the Tiled Scripting API.

The following global variable is currently not documented in the generated documentation, since it conflicts with nodejs types:

__filename
: The file path of the current file being evaluated. Only available during initial evaluation of the file and not when later functions in that file get called. If you need it there, copy the value to local scope.

CHAPTER

NINETEEN

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!

19.1 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.

19.1.1 C

- cute tiled - JSON map loader with examples (zlib/Public Domain).
- libtmj - JSON map and tileset loader with zlib/gzip/zstd support (BSD 2-Clause)
- TMX - TMX map loader with Allegro5 and SDL2 examples (BSD).

19.1.2 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)
- tinytmx A C++17 library to parse maps generated by Tiled Map Editor. Requires no external linking, all dependencies are included.
- Tileson - A Tiled JSON parser for modern C++ (C++17) by Robin Berg Pettersen (BSD)
- tmx++ TMX parser for C++20 or newer, optional compression support, all dependencies are included

19.1.3 C#/.NET

- DotTiled: A fast, memory-efficient, and easy to use .NET library for loading Tiled maps and tilesets with support for both TMX and JSON formats.
- TiledLib: Cross-platform Tiled map parsing utilities.
- MonoGame.Extended has a Tiled map loader and renderer that works with MonoGame on all platforms that support portable class libraries.
- The following projects appear to be no longer maintained, but might still be useful: TiledCS, TiledCSPlus, TiledSharp, NTiled, tmx-mapper-pcl, tiled-xna and TmxCSharp.

19.1.4 Common Lisp

- cl-tiled: TMX/TSX and JSON map/tileset loader.

19.1.5 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.

19.1.6 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.

19.1.7 Dart

- tiled: a library for loading TMX files

19.1.8 Go

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

19.1.9 Haskell

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

19.1.10 Java

- A library for loading TMX files is included with Tiled at util/java/libtiled-java.
- TiledReader is a simple TMX reader that conveys the information in Tiled files via a hand-crafted class structure, but does not load image data.
- 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.

19.1.11 OCaml

- tmx

19.1.12 PHP

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

19.1.13 Pike

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

19.1.14 Processing

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

19.1.15 Python

- Arcade: 2D game library that uses pytiled-parser for easy loading of Tiled maps into a game. Arcade Tiled Examples
- pytiled-parser: Python parser for TMX and JSON maps.
- pytmxlib: library for programmatic manipulation of TMX maps
- pytmxloader: Python library intended to make loading of JSON Tiled maps very easy.
- PyTMX: Python library to read TMX maps.
- ulvl: Simple Python library that can read from, among others, TMX XML files.

19.1.16 Ruby

- tmx gem by erisdiscord

19.1.17 Rust

- tiled, a rust crate for loading TMX maps
- tiled-json-rs, a crate to parse and interact with Tiled editor JSON files

19.1.18 Vala

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

19.2 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.

19.2.1 AndEngine

- AndEngine by Nicolas Gramlich supports rendering TMX maps

19.2.2 Allegro

- allegro_tiled integrates Tiled support with Allegro 5.

19.2.3 Bevy

- bevy_tiled, a plugin for rendering Tiled maps
- bevy_tmx, a plugin that allows you to read .tmx files as scenes
- bevy_ecs_tilemap, a tilemap rendering plugin that makes tiles entities, with support for TMX maps

19.2.4 Castle Game Engine (Object Pascal)

- Castle Game Engine has native support for Tiled maps (see the engine manual about Tiled Maps)

19.2.5 Cell2D

- The Java library Cell2D supports Tiled maps via a pipeline that starts with TiledReader, but currently has more built-in support for orthogonal maps than for other orientations.

19.2.6 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.

19.2.7 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

19.2.8 DragonRuby Game Toolkit

- DRTiled adds support for loading Tiled maps to the DragonRuby Game Toolkit. The maps can be rendered using DRTiled Renderer.

19.2.9 Flame

- flame_tiled is a library for incorporating Tiled maps into the Flame game engine.

19.2.10 Flixel

- Lithander demonstrated his Flash TMX parser combined with Flixel rendering

19.2.11 Game Maker

- Tiled ships with plugins for exporting to *GameMaker: Studio 1.4* and *GameMaker Studio 2.3* room files.
- Tiled2GM Converter by Dmi7ry

19.2.12 Godot

- Tiled ships with a plugin for exporting to *Godot 4* as .tscn scene files.
- Tiled Map Importer imports each map as Godot scene which can be instanced or inherited (forum announcement).
- Godot Tiled importer (Mono version) imports Tiled maps exported to JSON (.tmj) format. Supports all map orientations.
- Tiled To Godot Export is a Tiled *JavaScript extension* for exporting Tilemaps and Tilesets in Godot 3.2 format (forum announcement).

19.2.13 Gosu

- gosu-tiled, a gem for drawing Tiled maps on the Gosu framework.

19.2.14 Grid Engine

- Planimeter's Grid Engine supports Tiled Lua-exported maps.

19.2.15 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

19.2.16 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 TiledMap-Builder.
- Excalibur, an open-source 2D HTML5 game engine, supports loading Tiled maps through the plugin excalibur-tiled.
- 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.

- pixi-tiledmap A loader and renderer for Tiled Maps in Pixi.JS v8+ written in TypeScript. JSON/XML support with no external deps, full layer-type support, typed API and ESM/CJS dual output.
- 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.
- glazeJS A high performance 2D game engine built in Typescript. It supports the TMX format, rendering tile layers on the GPU via WebGL (demo).

19.2.17 indielib-crossplatform

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

19.2.18 Irrlicht

- Irrlicht, a C++ realtime 3D engine, can load TMX files through a 3rd-party library by TheMrCerebro (Zlib).

19.2.19 LibGDX

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

19.2.20 LITIENGINE

- LITIENGINE is an open source Java 2D Game Engine that supports loading, editing, saving, and rendering maps in the .tmx format.

19.2.21 LÖVE

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

19.2.22 MOAI SDK

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

19.2.23 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).

19.2.24 NICO Game Framework

- NICO is a game framework for the Nim programming language that supports loading and saving Tiled maps in JSON format (see the Tilemap API documentation).

19.2.25 Node.js

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

19.2.26 Oak Nut Engine (onut)

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

19.2.27 Orx Portable Game Engine

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

19.2.28 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.

19.2.29 Pyglet

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

19.2.30 PySDL2

- PyTMX by Leif Theden (bitcraft)

19.2.31 RPG Maker MV

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

19.2.32 SDL

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

19.2.33 SFML

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

19.2.34 Slick2D

- Slick2D supports loading TMX maps through TiledMap.

19.2.35 Solar2D (formerly Corona SDK)

- ponytiled is a simple Tiled Map Loader for Solar2D (forum announcement)
- Dusk Engine is a fully featured Tiled map game engine for Solar2D (no longer maintained, but may still be useful)
- Berry is a simple Tiled Map Loader for Solar2D.

- Qiso is an isometric engine for Solar2D that supports loading Tiled maps, and also handles things like path-finding for you.

19.2.36 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.
- JSTileMap is a lightweight SpriteKit implementation of the TMX format supporting iOS 7 and OS X 10.9 and above.

19.2.37 TERRA Engine (Delphi/Pascal)

- TERRA Engine supports loading and rendering of TMX maps.

19.2.38 Unity

- SuperTiled2Unity is a collection of C# Unity scripts that can automatically import Tiled map editor files directly into your Unity projects.
- Tiled TMX Importer, imports into Unity 2017.2's new native Tilemap system.
- 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).
- Tuesday: A generic C# serializer and deserializer plus a set of Unity editor scripts that allow you to drag and drop TMX files into your scene, make edits, and save back out as TMX files. MIT license.
- UniTiled, a native TMX importer for Unity.
- X-UniTMX supports almost all Tiled 0.11 features. Imports TMX/XML files into Sprite Objects or Meshes.
- Orthello Pro (2D framework) offers Tiled map support.

19.2.39 Unreal Engine 4

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

19.2.40 Urho3D

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

19.2.41 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

TWENTY

TMX MAP FORMAT

Version 1.8

TMX and TSX are Tiled's own formats for storing tile maps and tilesets, based on XML. TMX provides 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 and TSX tilesets.

In this document we'll go through each element found in these file formats. 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.

Note

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

For compatibility reasons, it is recommended to ignore unknown elements and attributes (or raise a warning). This makes it easier to add features without breaking backwards compatibility, and allows custom variants and additions to work with existing tools.

20.1 <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)
- **class:** The class of this map (since 1.9, defaults to "").
- **orientation:** Map orientation. Tiled supports "orthogonal", "isometric", "oblique", "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.
- **skewx:** For oblique maps, the pixel offset per tile row.
- **skewy:** For oblique maps, the pixel offset per tile column.
- **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)
- **parallaxoriginx:** X coordinate of the parallax origin in pixels (defaults to 0). (since 1.8)
- **parallaxoriginy:** Y coordinate of the parallax origin in pixels (defaults to 0). (since 1.8)
- **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. The `oblique` orientation uses a skewed grid of parallelogram-shaped tiles.

The tilesets used by the map should always be listed before the layers.

Can contain at most one: *<properties>*, *<editorsettings>* (since 1.3)

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

20.2 <editorsettings>

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

Can contain at most one: *<chunksize>*, *<export>*

20.2.1 <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).

20.2.2 <export>

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

20.3 <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.
- **class:** The class of this tileset (since 1.9, defaults to "").
- **tilewidth:** The width of the tiles in this tileset, which should be at least 1 except in the case of image collection tilesets (in which case it stores the maximum tile width).
- **tileheight:** The height of the tiles in this tileset, which should be at least 1 except in the case of image collection tilesets (in which case it stores the maximum tile height).
- **spacing:** The spacing in pixels between the tiles in this tileset (applies to the tileset image, defaults to 0). Irrelevant for image collection tilesets.
- **margin:** The margin around the tiles in this tileset (applies to the tileset image, defaults to 0). Irrelevant for image collection tilesets.
- **tilecount:** The number of tiles in this tileset (since 0.13). Note that there can be tiles with a higher ID than the tile count, in case the tileset is an image collection from which tiles have been removed.
- **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)
- **tilerendersize:** The size to use when rendering tiles from this tileset on a tile layer. Valid values are `tile` (the default) and `grid`. When set to `grid`, the tile is drawn at the tile grid size of the map. (since 1.9)
- **fillmode:** The fill mode to use when rendering tiles from this tileset. Valid values are `stretch` (the default) and `preserve-aspect-fit`. Only relevant when the tiles are not rendered at their native size, so this applies to resized tile objects or in combination with `tilerendersize` set to `grid`. (since 1.9)

A tileset can be either *based on a single image*, which is cut into tiles based on the given parameters, or a *collection of images*, in which case each tile defines its own image. In the first case there is a single child *<image>* element. In the latter case, each child *<tile>* element contains an *<image>* element.

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.

Can contain at most one: *<image>*, *<tileoffset>*, *<grid>* (since 1.0), *<properties>*, *<terraintypes>*, *<wangsets>* (since 1.1), *<transformations>* (since 1.5)

Can contain any number: *<tile>*

20.3.1 <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.

20.3.2 <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.

20.3.3 <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.
- **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). Including the "#" is optional and Tiled leaves it out for compatibility reasons. (optional)
- **width:** The image width in pixels (optional, used for tile index correction when the image changes)
- **height:** The image height in pixels (optional)

Tiled maps or tilesets with embedded image data can currently only be created using the *JavaScript API*, or in custom tools based on `libtiled` (Qt/C++) or tmxlib (Python).

Can contain at most one: *<data>*

20.3.4 <terraintypes>

Deprecated: This element has been deprecated since Tiled 1.5, in favour of the *<wangsets>* element, which is more flexible. Tilesets containing terrain types are automatically saved with a Wang set instead.

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>

Deprecated: This element has been deprecated since Tiled 1.5, in favour of the *<wangcolor>* element.

- **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>*

20.3.5 <transformations>

This element is used to describe which transformations can be applied to the tiles (e.g. to extend a Wang set by transforming existing tiles).

- **hflip:** Whether the tiles in this set can be flipped horizontally (default 0)
- **vflip:** Whether the tiles in this set can be flipped vertically (default 0)
- **rotate:** Whether the tiles in this set can be rotated in 90 degree increments (default 0)
- **preferuntransformed:** Whether untransformed tiles remain preferred, otherwise transformed tiles are used to produce more variations (default 0)

20.3.6 <tile>

- **id:** The local tile ID within its tileset.
- **type:** The class of the tile. Is inherited by tile objects. (since 1.0, defaults to ""), was saved as `class` in 1.9)
- *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. (deprecated since 1.5 in favour of *<wangtile>*)
- **probability:** A percentage indicating the probability that this tile is chosen when it competes with others while editing with the terrain tool. (defaults to 1)
- **x:** The X position of the sub-rectangle representing this tile (default: 0)
- **y:** The Y position of the sub-rectangle representing this tile (default: 0)
- **width:** The width of the sub-rectangle representing this tile (defaults to the image width)
- **height:** The height of the sub-rectangle representing this tile (defaults to the image height)

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.

20.3.7 <wangsets>

Contains the list of Wang sets defined for this tileset.

Can contain any number: *<wangset>*

<wangset>

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

- **name:** The name of the Wang set.
- **class:** The class of the Wang set (since 1.9, defaults to "").
- **tile:** The tile ID of the tile representing this Wang set.

Can contain at most one: *<properties>*

Can contain up to 254: *<wangcolor>* (255 since Tiled 1.5, 254 since Tiled 1.10.2)

Can contain any number: *<wangtile>*

<wangcolor>

A color that can be used to define the corner and/or edge of a Wang tile.

- **name:** The name of this color.
- **class:** The class of this color (since 1.9, defaults to "").
- **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 1)

Can contain at most one: *<properties>*

<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, since Tiled 1.5 given by a comma-separated list of indexes (0-254) referring to the Wang colors in the Wang set in the order: top, top-right, right, bottom-right, bottom, bottom-left, left, top-left. Index 0 means *unset* and index 1 refers to the first Wang color. Before Tiled 1.5, the Wang ID was saved as a 32-bit unsigned integer stored in the format `0xCECECECE` (where each C is a corner color and each E is an edge color, in reverse order).
- *hflip:* Whether the tile is flipped horizontally (removed in Tiled 1.5).
- *vflip:* Whether the tile is flipped vertically (removed in Tiled 1.5).
- *dflip:* Whether the tile is flipped on its diagonal (removed in Tiled 1.5).

20.4 <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 (defaults to 0, with valid IDs being at least 1). 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 "")
- **class:** The class of the layer (since 1.9, 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.
- **tintcolor:** A *tint color* that is multiplied with any tiles drawn by this layer in `#AARRGGBB` or `#RRGGBB` format (optional).
- **offsetx:** Horizontal offset for this layer in pixels. Defaults to 0. (since 0.14)
- **offsety:** Vertical offset for this layer in pixels. Defaults to 0. (since 0.14)
- **parallaxx:** Horizontal *parallax factor* for this layer. Defaults to 1. (since 1.5)
- **parallaxy:** Vertical *parallax factor* for this layer. Defaults to 1. (since 1.5)
- **mode:** The blend mode to use when rendering the layer. Valid values are `normal`, `add`, `multiply`, `screen`, `overlay`, `darken`, `lighten`, `color-dodge`, `color-burn`, `hard-light`, `soft-light`, `difference` and `exclusion` (since 1.12, defaults to `normal`).

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

20.4.1 <data>

- **encoding:** The encoding used to encode the tile layer data. When used, it can be "base64" and, when used for tile layer data, "csv". (optional)
- **compression:** The compression used to compress the tile layer data. Tiled supports "gzip", "zlib" and (as a compile-time option since Tiled 1.3) "zstd".

This element is usually used as a child of a *<layer>* element, and contains the actual tile layer data. It can also occur as a child of an *<image>* element, where it can store embedded image data.

When no encoding or compression is given, the tiles are stored as individual XML *<tile>* elements, but this option is deprecated. 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 called "global", since they may refer to a tile from any of the tilesets used by the map. The IDs also contain *flipping flags*. The tilesets are always stored with increasing `firstgid`s.

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

20.4.2 <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>*

20.4.3 <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.

20.5 <objectgroup>

- **id:** Unique ID of the layer (defaults to 0, with valid IDs being at least 1). 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 "")
- **class:** The class of the object group (since 1.9, defaults to "").
- **color:** The color used to display the objects in this group. (optional)
- *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)
- **tintcolor:** A color that is multiplied with any tile objects drawn by this layer, in `#AARRGGBB` or `#RRGGBB` format (optional).
- **offsetx:** Horizontal offset for this object group in pixels. (defaults to 0) (since 0.14)
- **offsety:** Vertical offset for this object group in pixels. (defaults to 0) (since 0.14)
- **parallaxx:** Horizontal *parallax factor* for this object group. Defaults to 1. (since 1.5)
- **parallaxy:** Vertical *parallax factor* for this object group. Defaults to 1. (since 1.5)
- **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>*

20.5.1 <object>

- **id:** Unique ID of the object (defaults to 0, with valid IDs being at least 1). 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 class of the object. An arbitrary string. (defaults to "", was saved as `class` in 1.9)
- **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)
- **opacity:** The opacity of the object as a value from 0 to 1. (defaults to 1)
- **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), *<capsule>* (since 1.12), *<point>* (since 1.1), *<polygon>*, *<polyline>*, *<text>* (since 1.0)

20.5.2 <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.

20.5.3 <capsule>

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

20.5.4 <point>

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

20.5.5 <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.

20.5.6 <polyline>

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

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

20.5.7 <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.

20.6 <imagelayer>

- **id:** Unique ID of the layer (defaults to 0, with valid IDs being at least 1). 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 "")
- **class:** The class of the image layer (since 1.9, defaults to "").
- **offsetx:** Horizontal offset of the image layer in pixels. (defaults to 0) (since 0.15)
- **offsety:** Vertical offset of the image layer in pixels. (defaults to 0) (since 0.15)
- **parallaxx:** Horizontal *parallax factor* for this layer. Defaults to 1. (since 1.5)

- **parallaxy:** Vertical *parallax factor* for this layer. Defaults to 1. (since 1.5)
- *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)
- **tintcolor:** A color that is multiplied with the image drawn by this layer in `#AARRGGBB` or `#RRGGBB` format (optional).
- **repeatx:** Whether the image drawn by this layer is repeated along the X axis. (since Tiled 1.8)
- **repeaty:** Whether the image drawn by this layer is repeated along the Y axis. (since Tiled 1.8)

A layer consisting of a single image.

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

20.7 <group>

- **id:** Unique ID of the layer (defaults to 0, with valid IDs being at least 1). 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 "")
- **class:** The class of the group layer (since 1.9, defaults to "").
- **offsetx:** Horizontal offset of the group layer in pixels. (defaults to 0)
- **offsety:** Vertical offset of the group layer in pixels. (defaults to 0)
- **parallaxx:** Horizontal *parallax factor* for this group. Defaults to 1. (since 1.5)
- **parallaxy:** Vertical *parallax factor* for this group. Defaults to 1. (since 1.5)
- **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)
- **tintcolor:** A color that is multiplied with any graphics drawn by any child layers, in `#AARRGGBB` or `#RRGGBB` format (optional).

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

Can contain at most one: *<properties>*

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

20.8 <properties>

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

Can contain any number: *<property>*

20.8.1 <property>

- **name:** The name of the property.
- **type:** The type of the property. Can be `string` (default), `int`, `float`, `bool`, `color`, `file`, `object` or `class` (since 0.16, with `color` and `file` added in 0.17, `object` added in 1.4 and `class` added in 1.8).
- **propertytype:** The name of the *custom property type*, when applicable (since 1.8).
- **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.

Class properties will have their member values stored in a nested *<properties>* element. Only the actually set members are saved. When no members have been set the `properties` element is left out entirely.

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.

Can contain at most one: *<properties>* (since 1.8)

20.9 Template Files

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

20.9.1 <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>
```

Any tileset reference should always come before the object. Embedded tilesets are not supported.

Can contain at most one: *<tileset>*

Should contain exactly one: *<object>*

Fig. 1: Creative Commons License

The **TMX Map Format** by https://www.mapeditor.org is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

CHAPTER

TWENTYONE

TMX CHANGELOG

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

21.1 Tiled 1.12

- Added `mode` attribute on *<layer>* to specific its blend mode.
- Added `oblique` to the supported values for the `orientation` attribute on the *<map>* element, along with the `skewx` and `skewy` attributes for configuring the per-row/column skew in pixels.
- Added capsule object shape. Same parameters as rectangular objects, but marked as capsule with a child element:

```
<object name="..." x="..." y="...">
  <capsule/>
</object>
```

- Added `opacity` attribute to *<object>* to specificy its opacity.

21.2 Tiled 1.10

- Renamed the `class` attribute on *<tile>* and *<object>* back to `type`, to keep compatibility with Tiled 1.8 and earlier. The attribute remains `class` for other elements since it could not be renamed to `type` everywhere.

21.3 Tiled 1.9

- Renamed the `type` attribute on *<tile>* and *<object>* to `class`.
- Added `class` attribute to *<map>*, *<tileset>*, *<layer>*, *<imagelayer>*, *<objectgroup>*, *<group>*, *<wangset>* and *<wangcolor>*.
- Added `x`, `y`, `width` and `height` attributes to the *<tile>* element, which store the sub-rectangle of a tile's image used to represent this tile. By default the entire image is used.
- Added `tilerendersize` and `fillmode` attributes to the *<tileset>* element, which affect the way tiles are rendered.

21.4 Tiled 1.8

- Added support for user-defined custom property types. A reference to the type is saved as the new `propertytype` attribute on the *<property>* element.
- The *<property>* element can now contain a *<properties>* element, in case the property value is a class and at least one member value has been set. The `type` attribute will have the new value `class`.

- Added `parallaxoriginx` and `parallaxoriginy` attributes to the *<map>* element.
- Added `repeatx` and `repeaty` attributes to the *<imagelayer>* element.

21.5 Tiled 1.7

- The *<tile>* elements in a tileset are no longer always saved with increasing IDs. They are now saved in the display order, which can be changed in Tiled.

21.6 Tiled 1.5

- The colors that are part of a *<wangset>* are no longer separated in corner colors and edge colors. Instead, there is now a single *<wangcolor>* element to define a Wang color. This new element also stores *<properties>*.
- The `wangid` attribute on the *<wangtile>* element is now stored as a comma-separated list of values, instead of a 32-bit unsigned integer in hex format. This is because the number of colors supported in a Wang set was increased from 15 to 255.
- Valid transformations of tiles in a set (flipping, rotation) are specified in a *<transformations>* element. The partial support for the `vflip`, `hflip` and `dflip` attributes on the *<wangtile>* element has been removed.
- The *<wangset>* element has replaced the now deprecated *<terraintypes>* element.
- Added `parallaxx` and `parallaxy` attributes to the *<layer>*, *<objectgroup>*, *<imagelayer>* and *<group>* elements.

21.7 Tiled 1.4

- Added the `objectalignment` attribute to the *<tileset>* element, allowing the tileset to control the alignment used for tile objects.
- Added the `tintcolor` attribute to the *<layer>*, *<objectgroup>*, *<imagelayer>* and *<group>* elements, allowing for a number of graphical effects like darkening or coloring a layer.
- Added a new `object` property type, which refers to an *object* by its ID.

21.8 Tiled 1.3

- Added an *<editorsettings>* element, which is used to store editor specific settings that are generally not relevant when loading a map.
- Added support for Zstandard compression for tile layer data (`compression="zstd"` on *<data>* elements).
- Added the `compressionlevel` attribute to the *<map>* element, which stores the compression level to use for compressed tile layer data.

21.9 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.

21.10 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.

21.11 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 *Terrain Sets*. 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.

21.12 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*.

21.13 Tiled 0.18

No file format changes.

21.14 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.

21.15 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`.

21.16 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.

21.17 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.

21.18 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.

21.19 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.

21.20 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.

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

21.22 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 name="Terrain">
  ...
  <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 name="...">
  <image source="..."/>
</imagelayer>
```

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

```
<object name="..." x="..." y="...">
  <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 name="Embedded images">
  ...
  <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>
```

21.23 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

TWENTYTWO

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:

22.1 Map

Field	Type	Description
backgroundcolor	string	Hex-formatted color (#RRGGBB or #AARRGGBB) (optional)
class	string	The class of the map (since 1.9, optional)
compressionlevel	int	The compression level to use for tile layer data (defaults to -1, which means to use the algorithm default)
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`, `oblique`, `staggered` or `hexagonal`
parallaxoriginx	double	X coordinate of the parallax origin in pixels (since 1.8, default: 0)
parallaxoriginy	double	Y coordinate of the parallax origin in pixels (since 1.8, default: 0)
properties	array	Array of *Properties*
renderorder	string	`right-down` (the default), `right-up`, `left-down` or `left-up` (currently only supported for orthogonal maps)
skewx	int	X offset applied per tile row (oblique maps only)
skewy	int	Y offset applied per tile column (oblique 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	string	The JSON format version (previously a number, saved as string since 1.6)
width	int	Number of tile columns

22.1.1 Map Example

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

22.2 Layer

Field	Type	Description
chunks	array	Array of *chunks* (optional). `tilelayer` only.
class	string	The class of the layer (since 1.9, optional)
compression	string	`zlib`, `gzip`, `zstd` (since 1.3) 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. `tilelayer` only.
id	int	Incremental ID - unique across all layers
image	string	Image used by this layer. `imagelayer` only.
imageheight	int	Height of the image used by this layer. `imagelayer` only. (since 1.11.1)
imagewidth	int	Width of the image used by this layer. `imagelayer` only. (since 1.11.1)
layers	array	Array of *layers*. `group` only.
locked	bool	Whether layer is locked in the editor (default: false). (since 1.8.2)
mode	string	The *blend mode* to use when rendering the layer. (since 1.12)
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 (default: 1)
parallaxx	double	Horizontal *parallax factor* for this layer (default: 1). (since 1.5)

continues on next page

Table 1 – continued from previous page

Field	Type	Description
parallaxy	double	Vertical *parallax factor* for this layer (default: 1). (since 1.5)
properties	array	Array of *Properties*
repeatx	bool	Whether the image drawn by this layer is repeated along the X axis. `imagelayer` only. (since 1.8)
repeaty	bool	Whether the image drawn by this layer is repeated along the Y axis. `imagelayer` only. (since 1.8)
startx	int	X coordinate where layer content starts (for infinite maps)
starty	int	Y coordinate where layer content starts (for infinite maps)
tintcolor	string	Hex-formatted *tint color* (#RRGGBB or #AARRGGBB) that is multiplied with any graphics drawn by this layer or any child layers (optional).
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. `tilelayer` only.
x	int	Horizontal layer offset in tiles. Always 0.
y	int	Vertical layer offset in tiles. Always 0.

22.2.1 Tile Layer Example

The data of a tile layer can be stored as a native JSON array or as base64-encoded and optionally compressed binary data, the same as done in the *TMX format*. The tiles are referenced using *Global Tile IDs*.

```
{
  "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
}
```

22.2.2 Object Layer Example

```
{
  "draworder":"topdown",
  "height":0,
  "name":"people",
  "objects":[ ],
  "opacity":1,
  "properties":[
    {
```

(continues on next page)

(continued from previous page)

```
      "name":"layerProp1",
      "type":"string",
      "value":"someStringValue"
    }],
  "type":"objectgroup",
  "visible":true,
  "width":0,
  "x":0,
  "y":0
}
```

22.2.3 Blend Mode

The following values are supported for the `mode` property on *Layer*:

- `normal` (default)
- `add`
- `multiply`
- `screen`
- `overlay`
- `darken`
- `lighten`
- `color-dodge`
- `color-burn`
- `hard-light`
- `soft-light`
- `difference`
- `exclusion`

22.3 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	Row count
width	int	Column count
x	int	X coordinate in tiles
y	int	Y coordinate in tiles

22.3.1 Chunk Example

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

22.4 Object

Field	Type	Description
capsule	bool	Used to mark an object as a capsule
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
opacity	double	The opacity of the object as a value from 0 to 1 (default: 1)
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	The class of the object (was saved as `class` in 1.9, optional)
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

22.4.1 Object Example

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

(continues on next page)

(continued from previous page)

```
  "width":0,
  "x":32,
  "y":32
}
```

22.4.2 Ellipse Example

```
{
  "ellipse":true,
  "height":152,
  "id":13,
  "name":"",
  "rotation":0,
  "opacity":1,
  "type":"",
  "visible":true,
  "width":248,
  "x":560,
  "y":808
}
```

22.4.3 Rectangle Example

```
{
  "height":184,
  "id":14,
  "name":"",
  "rotation":0,
  "opacity":1,
  "type":"",
  "visible":true,
  "width":368,
  "x":576,
  "y":584
}
```

22.4.4 Point Example

```
{
  "height":0,
  "id":20,
  "name":"",
  "point":true,
  "rotation":0,
  "opacity":1,
  "type":"",
  "visible":true,
  "width":0,
  "x":220,
  "y":350
```

(continues on next page)

(continued from previous page)

```
}
```

22.4.5 Polygon Example

```
{
  "height":0,
  "id":15,
  "name":"",
  "polygon":[
  {
    "x":0,
    "y":0
  },
  {
    "x":152,
    "y":88
  },
  {
    "x":136,
    "y":-128
  },
  {
    "x":80,
    "y":-280
  },
  {
    "x":16,
    "y":-288
  }],
  "rotation":0,
  "opacity":1,
  "type":"",
  "visible":true,
  "width":0,
  "x":-176,
  "y":432
}
```

22.4.6 Polyline Example

```
{
  "height":0,
  "id":16,
  "name":"",
  "polyline":[
  {
    "x":0,
    "y":0
  },
  {
    "x":248,
```

(continues on next page)

(continued from previous page)


```
    "y":-32
   },
   {
    "x":376,
    "y":72
   },
   {
    "x":544,
    "y":288
   },
   {
    "x":656,
    "y":120
   },
   {
    "x":512,
    "y":0
   }],
   "rotation":0,
   "opacity":1,
   "type":"",
   "visible":true,
   "width":0,
   "x":240,
   "y":88
}
```


22.4.7 Text Example


```
{
   "height":19,
   "id":15,
   "name":"",
   "text":
   {
    "text":"Hello World",
    "wrap":true
   },
   "rotation":0,
   "opacity":1,
   "type":"",
   "visible":true,
   "width":248,
   "x":48,
   "y":136
}
```

22.5 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`)

22.6 Tileset

Field	Type	Description
backgroundcolor	string	Hex-formatted color (#RRGGBB or #AARRGGBB) (optional)
class	string	The class of the tileset (since 1.9, optional)
columns	int	The number of tile columns in the tileset
fillmode	string	The fill mode to use when rendering tiles from this tileset (`stretch` (default) or `preserve-aspect-fit`) (since 1.9)
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)
tilerendersize	string	The size to use when rendering tiles from this tileset on a tile layer (`tile` (default) or `grid`) (since 1.9)
tiles	array	Array of *Tiles* (optional)
tilewidth	int	Maximum width of tiles in this set
transformations	*Transformations*	Allowed transformations (optional)
transparentcolor	string	Hex-formatted color (#RRGGBB) (optional)
type	string	`tileset` (for tileset files, since 1.0)
version	string	The JSON format version (previously a number, saved as string since 1.6)
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.

22.6.1 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

22.6.2 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)

22.6.3 Transformations

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

Field	Type	Description
hflip	bool	Tiles can be flipped horizontally
vflip	bool	Tiles can be flipped vertically
rotate	bool	Tiles can be rotated in 90-degree increments
preferuntrans-formed	bool	Whether untransformed tiles remain preferred, otherwise transformed tiles are used to produce more variations

22.6.4 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
}
```

22.6.5 Tile (Definition)

Field	Type	Description
animation	array	Array of *Frames*
id	int	Local ID of the tile
image	string	Image representing this tile (optional, used for image collection tilesets)
imageheight	int	Height of the tile image in pixels
imagewidth	int	Width of the tile image in pixels
x	int	The X position of the sub-rectangle representing this tile (default: 0)
y	int	The Y position of the sub-rectangle representing this tile (default: 0)
width	int	The width of the sub-rectangle representing this tile (defaults to the image width)
height	int	The height of the sub-rectangle representing this tile (defaults to the image height)
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 (default: 1)
properties	array	Array of *Properties*
terrain	array	Index of terrain for each corner of tile (optional, replaced by *Wang sets* since 1.5)
type	string	The class of the tile (was saved as `class` in 1.9, optional)

A tileset that associates information with each tile, like its image path, 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:

```
{
  "id":11,
  "properties":[
    {
      "name":"myProperty2",
      "type":"string",
      "value":"myProperty2_value"
    }],
  "terrain":[0, 1, 0, 1]
}
```

22.6.6 Frame

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

22.6.7 Terrain

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

Example:

```
{
  "name":"ground",
  "tile":0
}
```

22.6.8 Wang Set

Field	Type	Description
class	string	The class of the Wang set (since 1.9, optional)
colors	array	Array of *Wang colors* (since 1.5)
name	string	Name of the Wang set
properties	array	Array of *Properties*
tile	int	Local ID of tile representing the Wang set
type	string	`corner`, `edge` or `mixed` (since 1.5)
wangtiles	array	Array of *Wang tiles*

Wang Color

Field	Type	Description
class	string	The class of the Wang color (since 1.9, optional)
color	string	Hex-formatted color (#RRGGBB or #AARRGGBB)
name	string	Name of the Wang color
probability	double	Probability used when randomizing (default: 1)
properties	array	Array of *Properties* (since 1.5)
tile	int	Local ID of tile representing the Wang color

Example:

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

Wang Tile

Field	Type	Description
tileid	int	Local ID of tile
wangid	array	Array of Wang color indexes (`uchar[8]`)

Example:

```
{
  "tileid": 0,
  "wangid": [2, 0, 1, 0, 1, 0, 2, 0]
}
```

22.7 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

22.8 Property

Field	Type	Description
name	string	Name of the property
type	string	Type of the property (`string` (default), `int`, `float`, `bool`, `color`, `file`, `object` or `class` (since 0.16, with `color` and `file` added in 0.17, `object` added in 1.4 and `class` added in 1.8))
propertytype	string	Name of the *custom property type*, when applicable (since 1.8)
value	value	Value of the property

22.9 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

22.10 Changelog

22.10.1 Tiled 1.12

- Added `mode` property to *Layer* to specify the blend mode to use when rendering the layer.

- Added `capsule` property to *Object*.
- Added `opacity` property to *Object*.

22.10.2 Tiled 1.11.1

- Added `imageheight` and `imagewidth` properties to image layers.

22.10.3 Tiled 1.10

- Renamed the `class` property on *Tile (Definition)* and *Object* back to `type`, to keep compatibility with Tiled 1.8 and earlier. The property remains `class` in other places since it could not be renamed to `type` everywhere.

22.10.4 Tiled 1.9

- Renamed the `type` property on *Tile (Definition)* and *Object* to `class`.
- Added `class` property to *Map*, *Tileset*, *Layer*, *Wang Set* and *Wang Color*.
- Added `x`, `y`, `width` and `height` properties to *Tile (Definition)*, which store the sub-rectangle of a tile's image used to represent this tile. By default the entire image is used.
- Added `tilerendersize` and `fillmode` properties to *Tileset*, which affect the way tiles are rendered.

22.10.5 Tiled 1.8

- Added support for user-defined custom property types. A reference to the type is saved as the new `propertytype` property of *Property*.
- The *Property* element can now have an arbitrary JSON object as its `value`, in case the property value is a class. In this case the `type` property is set to the new value `class`.
- Added `parallaxoriginx` and `parallaxoriginy` properties to *Map*.
- Added `repeatx` and `repeaty` properties to *Layer* (applies only to image layers at the moment).

22.10.6 Tiled 1.7

- The *Tile (Definition)* objects in a tileset are no longer always saved with increasing IDs. They are now saved in the display order, which can be changed in Tiled.

22.10.7 Tiled 1.6

- The `version` property is now written as a string ("1.6") instead of a number (1.5).

22.10.8 Tiled 1.5

- Unified `cornercolors` and `edgecolors` properties of *Wang Set* as the new `colors` property and added a `type` field.
- *Wang Color* can now store `properties`.
- Added `transformations` property to *Tileset* (see *Transformations*).
- Removed `dflip`, `hflip` and `vflip` properties from *Wang Tile* (no longer supported).
- Added `parallaxx` and `parallaxy` properties to the *Layer* object.

22.10.9 Tiled 1.4

- Added `objectalignment` to the *Tileset* object.
- Added `tintcolor` to the *Layer* object.
- Added `object` as possible type of *Property*.

22.10.10 Tiled 1.3

- Added an `editorsettings` property to top-level *Map* and *Tileset* objects, which is used to store editor specific settings that are generally not relevant when loading a map or tileset.
- Added support for Zstandard compression for tile layer data (`"compression": "zstd"` on *tile layer objects*).
- Added the `compressionlevel` property to the *Map* object, which stores the compression level to use for compressed tile layer data.

22.10.11 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` properties have been removed.

22.10.12 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 *Terrain Sets*. They are saved in the new *Wang Set* object (since Tiled 1.1.5).

CHAPTER

TWENTYTHREE

GLOBAL TILE IDS

Several of the map formats supported by Tiled, including its native *TMX* and *JSON* map formats, use the same data representation for individual tiles in layers: global tile IDs with flip flags. These GIDs are "global" because they may refer to a tile from any of the tilesets used by the map, rather than being local to a specific tileset. To get at a specific tile from a GID, you will first need to extract and clear the flip flags, then you will need to determine which tileset the tile belongs to, and which tile within the tileset it is.

> **Note**
>
> Despite the "global" name, GIDs are only global within a single map. A given tile may have a different GID in a different map, if that map has different tilesets, or has its tilesets in a different order.

23.1 Tile Flipping

The highest four bits of the 32-bit GID are flip flags, and you will need to read and clear them before you can access the GID itself to identify the tile.

Bit 32 is used for storing whether the tile is horizontally flipped, bit 31 is used for the vertically flipped tiles. In orthogonal and isometric maps, bit 30 indicates whether the tile is flipped (anti) diagonally, which enables tile rotation, and bit 29 can be ignored. In hexagonal maps, bit 30 indicates whether the tile is rotated 60 degrees clockwise, and bit 29 indicates 120 degrees clockwise rotation.

> **Note**
>
> Even if you're parsing a non-hexagonal map, remember to clear bit 29 after you've read the flags. Tiled keeps and outputs that flag even if the map orientation is changed. If not cleared, you may get an invalid tile ID.

When rendering an orthographic or isometric tile, the order of operations matters. The diagonal flip is done first, followed by the horizontal and vertical flips. The diagonal flip should flip the bottom left and top right corners of the tile, and can be thought of as an x/y axis swap. For hexagonal tiles, the order does not matter.

23.2 Mapping a GID to a Local Tile ID

Every tileset has its own, independent local tile IDs, typically (but not always) starting at 0. To avoid conflicts within maps using multiple tilesets, GIDs are assigned in sequence based on the size of each tileset. Each tileset is assigned a `firstgid` within the map, this is the GID that the tile with local ID 0 in the tileset would have.

To figure out which tileset a tile belongs to, find the tileset that has the largest `firstgid` that is smaller than or equal to the tile's GID. Once you have identified the tileset, subtract its `firstgid` from the tile's GID to get the local ID of the tile within the tileset.

> **Note**
>
> The `firstgid` of the first tileset is always 1. A GID of 0 in a layer means that cell is empty.

As an example, here's an excerpt from a TMX file with three tilesets:

```
<tileset firstgid="1" source="TilesetA.tsx"/>
<tileset firstgid="65" source="TilesetB.tsx"/>
<tileset firstgid="115" source="TilesetC.tsx"/>
```

In this map, tiles with GIDs 1-64 would be part of TilesetA, tiles with GIDs 65-114 would be part of TilesetB, and tiles with GIDs 115 and above would be part of tileset C. A tile with GID 72 would be part of TilesetB since TilesetB has the largest `firstgid` that's less than or equal to 72, and its local ID would be 7 (72-65).

23.3 Code example

The following C++ pseudo-code, using TMX as an example, should make it all clear, it deals with flags and deduces the appropriate tileset:

```
// 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;
const unsigned ROTATED_HEXAGONAL_120_FLAG = 0x10000000;

...

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

// If the data is encoded and compressed, decode and decompress:
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) {
    //Read the GID in little-endian byte order:
    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
    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);
    bool rotated_hex120 = (global_tile_id & ROTATED_HEXAGONAL_120_FLAG);
```

(continues on next page)

(continued from previous page)

```
    // Clear all four flags
    global_tile_id &= ~(FLIPPED_HORIZONTALLY_FLAG |
                        FLIPPED_VERTICALLY_FLAG |
                        FLIPPED_DIAGONALLY_FLAG |
                        ROTATED_HEXAGONAL_120_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->get_tile(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!)